Realtime and Embedded Specification for Java
Version 2.0

Draft 66
The Apollo Jupiter Edition for Public Review
7th of December 2017

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

Copyright © 1999–2012 TimeSys
Copyright © 2012–2015 aicas GmbH
All rights reserved
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
Ethan Blanton

Past Expert Group Members

Peter Dibble  TimeSys
David Holmes  Oracle
Table of Contents

Contents i

List of Figures xiv

List of Tables xv

1 Introduction 1
  1.1 Guiding Principles ........................................... 2
    1.1.1 Applicability Across Java Environments ................. 3
    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 ..................................... 6
    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
## TABLE OF CONTENTS

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

2.3.2.1 Methodological Principles ............... 13

2.3.2.2 Expressibility Principles ................. 14

2.3.2.3 Semantic Principles ...................... 14

2.3.2.4 Pragmatic Principles ..................... 15

2.3.3 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 ........................... 19

2.6 Device Access and Raw Memory .................... 19

2.6.1 Raw Memory Access ............................ 19

2.7 System Options ................................ 20

2.8 Exceptions .................................... 20

2.9 Summary .................................... 20

3 General Requirements .............................. 23

3.1 Definitions .................................. 23

3.2 Semantics .................................... 24

3.2.1 Base Requirements ......................... 24

3.2.2 Modules ................................... 25

3.2.2.1 Core Module ........................... 26

3.2.2.2 Alternative Memory Areas Module .......... 28

3.2.2.3 Control Module ......................... 29

3.2.2.4 Device Module .......................... 29

3.2.2.5 POSIX module ........................... 30

3.2.3 Optional Features ............................ 31

3.2.4 Deprecated Classes ......................... 32

3.2.5 Implementation types Allowed ............... 32

3.2.5.1 Realtime Deployment Implementation ........ 33

3.2.5.2 Simulation Implementation ............... 34

3.3 Required Documentation ....................... 35

3.4 Rationale ................................... 37
# TABLE OF CONTENTS

4 Realtime vs Conventional Java 39
4.1 Definitions ................................................. 41
4.2 Semantics .................................................. 41
  4.2.1 Scheduling ................................................. 41
    4.2.1.1 Priority ............................................. 42
    4.2.1.2 Thread Groups ......................................... 43
    4.2.1.3 Current Thread ......................................... 45
  4.2.2 InterruptedException ...................................... 45
  4.2.3 Java Memory Model ........................................ 45
  4.2.4 Memory Management ....................................... 46
    4.2.4.1 Memory Areas .......................................... 46
    4.2.4.2 Garbage Collection ..................................... 46
    4.2.4.3 Realtime Garbage Collections ......................... 47
  4.3 Rationale ................................................. 48

5 Realtime Threads 49
5.1 Definitions ................................................ 50
5.2 Semantics ................................................ 50
  5.2.1 Startup Considerations .................................... 51
  5.3 javax.realtime ............................................ 52
    5.3.1 Enumerations ............................................ 52
      5.3.1.1 PhasingPolicy ....................................... 52
    5.3.2 Classes ................................................. 55
      5.3.2.1 ConfigurationParameters .............................. 55
      5.3.2.2 RealtimeThread ...................................... 59
  5.4 Rationale ................................................. 86

6 Scheduling 89
6.1 Definitions ................................................ 91
6.2 Semantics ................................................ 93
  6.2.1 Schedulers ............................................... 93
    6.2.1.1 Affinity ............................................. 94
    6.2.1.2 Parameter Values ....................................... 94
    6.2.1.3 Release Control ....................................... 96
    6.2.1.4 Dispatching .......................................... 108
    6.2.1.5 Cost Monitoring and Cost Enforcement .................. 110
  6.2.2 Priority Schedulers ..................................... 112
    6.2.2.1 Priorities ........................................... 113
    6.2.2.2 First-In-First-Out-Scheduler ........................ 113
    6.2.2.3 The Round-Robin Scheduler ............................ 114
    6.2.2.4 Parameter Values ..................................... 115
### TABLE OF CONTENTS

6.2.3 Associating Schedulables with Schedulers ........................................ 116
6.2.4 Additional Schedulers .................................................................................. 117
6.2.5 Managing Groups of Schedulables ............................................................... 117
  6.2.5.1 Realtime Thread Groups ........................................................................... 117
  6.2.5.2 Processing Groups .................................................................................. 118
6.3 javax.realtime .................................................................................................. 121
  6.3.1 Interfaces .................................................................................................... 121
    6.3.1.1 BoundSchedulable ................................................................................. 121
    6.3.1.2 Schedulable .......................................................................................... 121
6.3.2 Enumerations ............................................................................................... 133
  6.3.2.1 MinimumInterarrivalPolicy ..................................................................... 133
  6.3.2.2 QueueOverflowPolicy ............................................................................. 136
6.3.3 Classes ......................................................................................................... 139
  6.3.3.1 Affinity ................................................................................................... 139
  6.3.3.2 AperiodicParameters ................................................................................ 147
  6.3.3.3 BackgroundParameters .......................................................................... 153
  6.3.3.4 FirstInFirstOutParameters ................................................................... 155
  6.3.3.5 FirstInFirstOutScheduler ...................................................................... 156
  6.3.3.6 PeriodicParameters ................................................................................ 160
  6.3.3.7 PriorityParameters .................................................................................. 168
  6.3.3.8 PriorityScheduler ................................................................................... 170
  6.3.3.9 ProcessingGroup ..................................................................................... 173
  6.3.3.10 RealtimeThreadGroup .......................................................................... 186
  6.3.3.11 ReleaseParameters ............................................................................... 192
  6.3.3.12 RoundRobinParameters ........................................................................ 204
  6.3.3.13 RoundRobinScheduler .......................................................................... 206
  6.3.3.14 Scheduler .............................................................................................. 210
  6.3.3.15 SchedulingParameters .......................................................................... 214
  6.3.3.16 SporadicParameters .............................................................................. 217
6.4 Rationale ......................................................................................................... 224
  6.4.1 RealtimeThreadGroup and ProcessingGroup ............................................. 225
  6.4.2 Multiprocessor Support .............................................................................. 226
  6.4.3 Impact of Clock Granularity ........................................................................ 227
  6.4.4 Deadline Miss Detection ............................................................................. 227
7 Synchronization .................................................................................................. 229
  7.1 Definitions ...................................................................................................... 229
  7.2 Semantics ....................................................................................................... 230
    7.2.1 Monitor Control ......................................................................................... 230
    7.2.2 Priority Schedulers .................................................................................. 231
    7.2.3 Additional Schedulers .............................................................................. 234
TABLE OF CONTENTS

7.3 `javax.realtime` .......................................................... 235
   7.3.1 Classes .................................................................. 235
      7.3.1.1 `MonitorControl` ........................................... 235
      7.3.1.2 `PriorityCeilingEmulation` .............................. 238
      7.3.1.3 `PriorityInheritance` ...................................... 241
      7.3.1.4 `WaitFreeReadQueue` .................................... 242
      7.3.1.5 `WaitFreeWriteQueue` .................................... 249

7.4 Rationale ................................................................. 256

8 Asynchrony ........................................................................ 257
   8.1 Definitions .............................................................. 258
   8.2 Semantics ................................................................. 259
      8.2.1 Asynchronous Events and their Handlers .............. 259
      8.2.2 Active Events and Dispatching ......................... 261
      8.2.3 Termination ........................................................ 263
   8.3 `javax.realtime` ......................................................... 265
      8.3.1 Interfaces ........................................................ 265
         8.3.1.1 `ActiveEvent` ............................................. 265
         8.3.1.2 `Releasable` .............................................. 268
         8.3.1.3 `Subsumable` ............................................. 268
      8.3.2 Classes ............................................................. 269
         8.3.2.1 `ActiveEventDispatcher` ............................... 269
         8.3.2.2 `AsyncBaseEvent` ...................................... 276
         8.3.2.3 `AsyncBaseEventHandler` ............................ 281
         8.3.2.4 `AsyncEvent` ............................................. 297
         8.3.2.5 `AsyncEventHandler` .................................. 299
         8.3.2.6 `AsyncLongEvent` ...................................... 306
         8.3.2.7 `AsyncLongEventHandler` ............................ 308
         8.3.2.8 `AsyncObjectEvent` .................................... 315
         8.3.2.9 `AsyncObjectEventHandler` ......................... 316
         8.3.2.10 `BlockableReleaseRunner` ......................... 323
         8.3.2.11 `BoundAsyncEventHandler` ......................... 326
         8.3.2.12 `BoundAsyncLongEventHandler` .................. 330
         8.3.2.13 `BoundAsyncObjectEventHandler` ............... 333
         8.3.2.14 `FixedPriorityReleaseRunner` .................... 337
         8.3.2.15 `ReleaseRunner` ...................................... 339
   8.4 Rationale ................................................................. 342

9 Time .............................................................................. 343
   9.1 Definitions ............................................................ 343
   9.2 Semantics ............................................................... 344

RTSJ 2.0 (Draft 66)
### TABLE OF CONTENTS

9.3  javax.realtime .................................................. 347
    9.3.1 Classes .................................................. 347
        9.3.1.1 AbsoluteTime ..................................... 347
        9.3.1.2 HighResolutionTime .................................. 361
        9.3.1.3 RelativeTime ..................................... 371
9.4  Rationale .................................................. 382

10  Clocks and Timers ............................................. 385
    10.1 Definitions .................................................. 386
    10.2 Semantics .................................................. 387
        10.2.1 Clock Model ........................................... 387
        10.2.2 Clocks and Timables ................................... 388
            10.2.2.1 Timable ........................................... 389
            10.2.2.2 Dispatching ..................................... 390
        10.2.3 Modeling Timers ....................................... 392
            10.2.3.1 Counter Model .................................... 392
            10.2.3.2 Comparator Model .................................. 393
            10.2.3.3 Triggering ........................................ 393
            10.2.3.4 Behavior of Timers ................................ 393
            10.2.3.5 Phasing ........................................... 394
    10.3  javax.realtime .............................................. 396
        10.3.1 Interfaces .............................................. 396
            10.3.1.1 AsyncTimable ..................................... 396
            10.3.1.2 Chronograph ...................................... 397
            10.3.1.3 Timable ........................................... 400
        10.3.2 Classes ................................................. 401
            10.3.2.1 Clock .............................................. 401
            10.3.2.2 OneShotTimer ....................................... 408
            10.3.2.3 PeriodicTimer ...................................... 410
            10.3.2.4 TimeDispatcher .................................... 419
            10.3.2.5 Timer .............................................. 422
    10.4  Rationale .................................................. 434

11  Alternative Memory Areas ...................................... 437
    11.1 Definitions .................................................. 439
    11.2 Semantics .................................................. 441
        11.2.1 Allocation Execution Time ................................ 442
        11.2.2 Allocation Context .................................... 442
        11.2.3 Backing Stores ......................................... 443
        11.2.4 The Parent Scope ....................................... 443
        11.2.5 Memory Areas and Schedulables .......................... 444
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.2.6 Scoped Memory Reference Counting</td>
<td>444</td>
</tr>
<tr>
<td>11.2.7 Immortal Memory</td>
<td>446</td>
</tr>
<tr>
<td>11.2.8 Maintaining Referential Integrity</td>
<td>446</td>
</tr>
<tr>
<td>11.2.9 Object Initialization</td>
<td>447</td>
</tr>
<tr>
<td>11.2.10 Maintaining the Scope Stack</td>
<td>447</td>
</tr>
<tr>
<td>11.2.11 The <code>enter</code> Method</td>
<td>449</td>
</tr>
<tr>
<td>11.2.12 The <code>executeInArea</code> or <code>newInstance</code> Methods</td>
<td>449</td>
</tr>
<tr>
<td>11.2.13 Constructor Methods for Schedulables</td>
<td>449</td>
</tr>
<tr>
<td>11.2.14 The Single Parent Rule</td>
<td>450</td>
</tr>
<tr>
<td>11.2.15 Scope Tree Maintenance</td>
<td>450</td>
</tr>
<tr>
<td>11.2.15.1 Pushing a MemoryArea onto the Scope Stack</td>
<td>451</td>
</tr>
<tr>
<td>11.2.15.2 Popping a MemoryArea off the Scope Stack</td>
<td>451</td>
</tr>
<tr>
<td>11.2.15.3 Reservation Management</td>
<td>452</td>
</tr>
<tr>
<td>11.2.16 Physical Memory</td>
<td>452</td>
</tr>
<tr>
<td>11.2.17 Stacked Memory</td>
<td>454</td>
</tr>
<tr>
<td>11.2.17.1 Avoiding Backing Store Fragmentation</td>
<td>454</td>
</tr>
<tr>
<td>11.2.17.2 Enforcing Encapsulation</td>
<td>455</td>
</tr>
<tr>
<td>11.2.17.3 Example</td>
<td>455</td>
</tr>
<tr>
<td>11.2.18 Pinnable Memory</td>
<td>457</td>
</tr>
<tr>
<td>11.2.19 Memory Groups</td>
<td>457</td>
</tr>
<tr>
<td>11.2.20 Startup Considerations</td>
<td>457</td>
</tr>
<tr>
<td>11.3 <code>javax.realtime</code></td>
<td>458</td>
</tr>
<tr>
<td>11.3.1 Enumerations</td>
<td>458</td>
</tr>
<tr>
<td>11.3.1.1 EnclosedType</td>
<td>458</td>
</tr>
<tr>
<td>11.3.2 Classes</td>
<td>460</td>
</tr>
<tr>
<td>11.3.2.1 HeapMemory</td>
<td>460</td>
</tr>
<tr>
<td>11.3.2.2 ImmortalMemory</td>
<td>465</td>
</tr>
<tr>
<td>11.3.2.3 MemoryArea</td>
<td>466</td>
</tr>
<tr>
<td>11.3.2.4 MemoryParameters</td>
<td>481</td>
</tr>
<tr>
<td>11.3.2.5 PerennialMemory</td>
<td>486</td>
</tr>
<tr>
<td>11.3.2.6 SizeEstimator</td>
<td>486</td>
</tr>
<tr>
<td>11.4 <code>javax.realtime.memory</code></td>
<td>493</td>
</tr>
<tr>
<td>11.4.1 Interfaces</td>
<td>493</td>
</tr>
<tr>
<td>11.4.1.1 PhysicalMemoryCharacteristic</td>
<td>493</td>
</tr>
<tr>
<td>11.4.2 Enumerations</td>
<td>493</td>
</tr>
<tr>
<td>11.4.2.1 PhysicalMemorySelector.CachingBehavior</td>
<td>493</td>
</tr>
<tr>
<td>11.4.2.2 PhysicalMemorySelector.PagingBehavior</td>
<td>495</td>
</tr>
<tr>
<td>11.4.3 Classes</td>
<td>497</td>
</tr>
<tr>
<td>11.4.3.1 LTMemory</td>
<td>497</td>
</tr>
<tr>
<td>11.4.3.2 MemoryGroup</td>
<td>500</td>
</tr>
</tbody>
</table>
11.4.3.3 PhysicalMemoryFactory ...................................................... 502  
11.4.3.4 PhysicalMemoryRegion ...................................................... 510  
11.4.3.5 PhysicalMemorySelector .................................................... 511  
11.4.3.6 PinnableMemory ............................................................... 514  
11.4.3.7 ScopedConfigurationParameters ........................................... 535  
11.4.3.8 ScopedMemory ................................................................. 537  
11.4.3.9 ScopedMemoryParameters .................................................... 564  
11.4.3.10 StackedMemory ............................................................... 568  
11.5 The Rationale ................................................................. 585  
11.5.1 Package Separation ............................................................. 585  
11.5.2 The Scoped Memory Model ..................................................... 585  
11.5.3 Reservation Management ...................................................... 587  
11.5.4 Backing Store Management .................................................... 587  
11.5.5 The Physical Memory Model .................................................... 587  
11.5.5.1 The Original Physical Memory Framework ............................... 590  
11.5.5.2 The RTSJ 2.0 Physical Memory Framework ............................... 591  
11.5.5.3 An example ................................................................. 593  

12 Asynchronous Control Flow .......................................................... 595  
12.1 Definitions ................................................................. 596  
12.2 Semantics ................................................................. 598  
12.2.1 Asynchronous Transition Point .............................................. 598  
12.2.2 Asynchronous Transfer of Control ........................................... 599  
12.2.2.1 Extending Conventional Java Interrupts .................................. 602  
12.2.2.2 Nesting AsynchronouslyInterruptedExceptions .................... 602  
12.2.3 Asynchronous Task Termination .............................................. 603  
12.3 javax.realtime.control .......................................................... 605  
12.3.1 Interfaces ................................................................. 605  
12.3.1.1 Interruptible ............................................................. 605  
12.3.2 Classes ................................................................. 606  
12.3.2.1 AsynchronousControlGroup .............................................. 606  
12.3.2.2 AsynchronouslyInterruptedException .............................. 608  
12.3.2.3 Timed ................................................................. 616  
12.4 Rationale ................................................................. 619  
12.4.1 Asynchronous Transfer of Control .......................................... 619  
12.4.2 Asynchronous Task Termination .............................................. 620  

13 Devices and Triggering ............................................................. 621  
13.1 Definitions ................................................................. 622  
13.2 Semantics ................................................................. 623  
13.2.1 Raw Memory ............................................................. 623
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>13.2.1.1</td>
<td>Raw Memory Region</td>
<td>625</td>
</tr>
<tr>
<td>13.2.1.2</td>
<td>Raw Memory Factory</td>
<td>626</td>
</tr>
<tr>
<td>13.2.1.3</td>
<td>Stride</td>
<td>626</td>
</tr>
<tr>
<td>13.2.2</td>
<td>Direct Memory Access Support</td>
<td>626</td>
</tr>
<tr>
<td>13.2.3</td>
<td>External Triggering</td>
<td>627</td>
</tr>
<tr>
<td>13.2.3.1</td>
<td>Happenings</td>
<td>627</td>
</tr>
<tr>
<td>13.2.4</td>
<td>Interrupt Service Routines</td>
<td>629</td>
</tr>
<tr>
<td>13.2.4.1</td>
<td>Synchronization</td>
<td>633</td>
</tr>
<tr>
<td>13.2.4.2</td>
<td>Required Documentation</td>
<td>633</td>
</tr>
<tr>
<td>13.3.1</td>
<td>Interfaces</td>
<td>635</td>
</tr>
<tr>
<td>13.3.1.1</td>
<td>DirectMemoryByteBuffer</td>
<td>635</td>
</tr>
<tr>
<td>13.3.1.2</td>
<td>RawByte</td>
<td>650</td>
</tr>
<tr>
<td>13.3.1.3</td>
<td>RawByteReader</td>
<td>651</td>
</tr>
<tr>
<td>13.3.1.4</td>
<td>RawByteWriter</td>
<td>654</td>
</tr>
<tr>
<td>13.3.1.5</td>
<td>RawDouble</td>
<td>657</td>
</tr>
<tr>
<td>13.3.1.6</td>
<td>RawDoubleReader</td>
<td>657</td>
</tr>
<tr>
<td>13.3.1.7</td>
<td>RawDoubleWriter</td>
<td>661</td>
</tr>
<tr>
<td>13.3.1.8</td>
<td>RawFloat</td>
<td>664</td>
</tr>
<tr>
<td>13.3.1.9</td>
<td>RawFloatReader</td>
<td>664</td>
</tr>
<tr>
<td>13.3.1.10</td>
<td>RawFloatWriter</td>
<td>667</td>
</tr>
<tr>
<td>13.3.1.11</td>
<td>RawInt</td>
<td>670</td>
</tr>
<tr>
<td>13.3.1.12</td>
<td>RawIntReader</td>
<td>671</td>
</tr>
<tr>
<td>13.3.1.13</td>
<td>RawIntWriter</td>
<td>674</td>
</tr>
<tr>
<td>13.3.1.14</td>
<td>RawLong</td>
<td>677</td>
</tr>
<tr>
<td>13.3.1.15</td>
<td>RawLongReader</td>
<td>677</td>
</tr>
<tr>
<td>13.3.1.16</td>
<td>RawLongWriter</td>
<td>680</td>
</tr>
<tr>
<td>13.3.1.17</td>
<td>RawMemory</td>
<td>683</td>
</tr>
<tr>
<td>13.3.1.18</td>
<td>RawMemoryRegionFactory</td>
<td>685</td>
</tr>
<tr>
<td>13.3.1.19</td>
<td>RawShort</td>
<td>705</td>
</tr>
<tr>
<td>13.3.1.20</td>
<td>RawShortReader</td>
<td>705</td>
</tr>
<tr>
<td>13.3.1.21</td>
<td>RawShortWriter</td>
<td>708</td>
</tr>
<tr>
<td>13.3.2</td>
<td>Classes</td>
<td>711</td>
</tr>
<tr>
<td>13.3.2.1</td>
<td>DirectMemoryBufferFactory</td>
<td>711</td>
</tr>
<tr>
<td>13.3.2.2</td>
<td>DirectMemoryRegion</td>
<td>715</td>
</tr>
<tr>
<td>13.3.2.3</td>
<td>Happening</td>
<td>717</td>
</tr>
<tr>
<td>13.3.2.4</td>
<td>HappeningDispatcher</td>
<td>727</td>
</tr>
<tr>
<td>13.3.2.5</td>
<td>InterruptCeilingEmulation</td>
<td>730</td>
</tr>
<tr>
<td>13.3.2.6</td>
<td>InterruptDescriptor</td>
<td>731</td>
</tr>
<tr>
<td>13.3.2.7</td>
<td>InterruptInheritance</td>
<td>734</td>
</tr>
</tbody>
</table>
### TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>13.3.2.8</td>
<td>InterruptMasking</td>
<td>735</td>
</tr>
<tr>
<td>13.3.2.9</td>
<td>InterruptServiceRoutine</td>
<td>736</td>
</tr>
<tr>
<td>13.3.2.10</td>
<td>InterruptUnmaskable</td>
<td>740</td>
</tr>
<tr>
<td>13.3.2.11</td>
<td>RawMemoryFactory</td>
<td>741</td>
</tr>
<tr>
<td>13.3.2.12</td>
<td>RawMemoryRegion</td>
<td>765</td>
</tr>
<tr>
<td>13.4</td>
<td>Rationale</td>
<td>767</td>
</tr>
<tr>
<td>13.4.1</td>
<td>Typed Raw Memory</td>
<td>767</td>
</tr>
<tr>
<td>13.4.2</td>
<td>Direct memory access</td>
<td>769</td>
</tr>
<tr>
<td>13.4.3</td>
<td>Interrupt Handling</td>
<td>770</td>
</tr>
<tr>
<td>14</td>
<td>Interprocess Signaling</td>
<td>773</td>
</tr>
<tr>
<td>14.1</td>
<td>Definitions</td>
<td>773</td>
</tr>
<tr>
<td>14.2</td>
<td>Semantics</td>
<td>773</td>
</tr>
<tr>
<td>14.2.1</td>
<td>POSIX Signals</td>
<td>774</td>
</tr>
<tr>
<td>14.2.2</td>
<td>POSIX Realtime Signals</td>
<td>774</td>
</tr>
<tr>
<td>14.3</td>
<td>javax.realtime.posix</td>
<td>775</td>
</tr>
<tr>
<td>14.3.1</td>
<td>Classes</td>
<td>775</td>
</tr>
<tr>
<td>14.3.1.1</td>
<td>RealtimeSignal</td>
<td>775</td>
</tr>
<tr>
<td>14.3.1.2</td>
<td>RealtimeSignalDispatcher</td>
<td>781</td>
</tr>
<tr>
<td>14.3.1.3</td>
<td>Signal</td>
<td>785</td>
</tr>
<tr>
<td>14.3.1.4</td>
<td>SignalDispatcher</td>
<td>792</td>
</tr>
<tr>
<td>14.4</td>
<td>Rationale</td>
<td>796</td>
</tr>
<tr>
<td>15</td>
<td>System and Options</td>
<td>797</td>
</tr>
<tr>
<td>15.1</td>
<td>Semantics</td>
<td>797</td>
</tr>
<tr>
<td>15.1.1</td>
<td>RealtimeSystem</td>
<td>797</td>
</tr>
<tr>
<td>15.1.2</td>
<td>Realtime Security</td>
<td>798</td>
</tr>
<tr>
<td>15.1.3</td>
<td>GarbageCollection</td>
<td>801</td>
</tr>
<tr>
<td>15.1.4</td>
<td>Compliance Version</td>
<td>801</td>
</tr>
<tr>
<td>15.2</td>
<td>javax.realtime</td>
<td>803</td>
</tr>
<tr>
<td>15.2.1</td>
<td>Enumerations</td>
<td>803</td>
</tr>
<tr>
<td>15.2.1.1</td>
<td>RTSJModule</td>
<td>803</td>
</tr>
<tr>
<td>15.2.2</td>
<td>Classes</td>
<td>806</td>
</tr>
<tr>
<td>15.2.2.1</td>
<td>AffinityPermission</td>
<td>806</td>
</tr>
<tr>
<td>15.2.2.2</td>
<td>CoreMemoryPermission</td>
<td>808</td>
</tr>
<tr>
<td>15.2.2.3</td>
<td>GarbageCollector</td>
<td>812</td>
</tr>
<tr>
<td>15.2.2.4</td>
<td>RealtimePermission</td>
<td>813</td>
</tr>
<tr>
<td>15.2.2.5</td>
<td>RealtimeSystem</td>
<td>817</td>
</tr>
<tr>
<td>15.2.2.6</td>
<td>SchedulingPermission</td>
<td>825</td>
</tr>
<tr>
<td>15.2.2.7</td>
<td>TaskPermission</td>
<td>829</td>
</tr>
<tr>
<td>15.3</td>
<td>javax.realtime.device</td>
<td>833</td>
</tr>
</tbody>
</table>
15.3.1 Classes ........................................ 833
  15.3.1.1 DirectMemoryPermission ..................... 833
  15.3.1.2 HappeningPermission .......................... 836
  15.3.1.3 RawMemoryPermission .......................... 839

15.4 javax.realtime.memory ............................. 843
  15.4.1 Classes ....................................... 843
    15.4.1.1 PhysicalMemoryPermission ................... 843
    15.4.1.2 ScopedMemoryPermission ....................... 846

15.5 javax.realtime.posix .............................. 849
  15.5.1 Classes ....................................... 849
    15.5.1.1 POSIXPermission ............................... 849

15.6 Rationale ........................................ 852

16 Exceptions ........................................ 853
  16.1 Semantics ........................................ 853
  16.2 javax.realtime .................................. 856
    16.2.1 Interfaces .................................... 856
      16.2.1.1 StaticThrowable ............................. 856
    16.2.2 Classes ....................................... 861
      16.2.2.1 AlignmentError .............................. 861
      16.2.2.2 ArrivalTimeQueueOverflowException ......... 862
      16.2.2.3 CeilingViolationException .................. 864
      16.2.2.4 ConstructorCheckedException ................. 866
      16.2.2.5 DeregistrationException ..................... 866
      16.2.2.6 EventQueueOverflowException ................. 867
      16.2.2.7 ForEachTerminationException ................. 868
      16.2.2.8 IllegalAssignmentError ..................... 869
      16.2.2.9 IllegalRealtimeArgumentException .......... 870
      16.2.2.10 IllegalRealtimeStateException ......... 876
      16.2.2.11 IllegalTaskStateException ................. 881
      16.2.2.12 InaccessibleAreaException ................. 887
      16.2.2.13 LateStartException ......................... 889
      16.2.2.14 MITViolationException ...................... 889
      16.2.2.15 MemoryAccessError ......................... 891
      16.2.2.16 MemoryInUseException ...................... 892
      16.2.2.17 MemoryScopeException ...................... 894
      16.2.2.18 MemoryTypeConflictException ............... 895
      16.2.2.19 OffsetOutOfBoundsException ............... 897
      16.2.2.20 OutOfMemoryStaticError ...................(Draft 66) xi
### TABLE OF CONTENTS

16.2.2.23 POSIXInvalidTargetException ........................................... 905
16.2.2.24 POSIXSignalPermissionException ..................................... 906
16.2.2.25 ProcessorAffinityException .......................................... 907
16.2.2.26 RangeOutOfBoundsException ............................................ 908
16.2.2.27 RealtimeSecurityException ........................................... 909
16.2.2.28 RegistrationException ................................................ 914
16.2.2.29 ResourceLimitError .................................................. 915
16.2.2.30 ScopedCycleException ................................................ 917
16.2.2.31 SizeOutOfBoundsException ............................................. 919
16.2.2.32 StaticCheckedException ............................................... 921
16.2.2.33 StaticError ............................................................. 926
16.2.2.34 StaticRuntimeException ............................................... 932
16.2.2.35 StaticThrowableStorage .............................................. 937
16.2.2.36 ThrowBoundaryError .................................................. 943
16.2.2.37 UninitializedStateException ........................................... 945
16.2.2.38 UnsupportedPhysicalMemoryException ............................... 945
16.2.2.39 UnsupportedRawMemoryRegionException ............................ 947
16.2.2.40 UnsupportedRealtimeOperationException ............................ 948
16.3 Rationale ............................................................................. 953

A Bibliography ............................................................................. 955

B Deprecated APIs ....................................................................... 957
B.1 Semantics ............................................................................. 957
B.2 javax.realtime ...................................................................... 958
B.2.1 Interfaces ........................................................................ 958
B.2.1.1 Interruptible ................................................................... 958
B.2.1.2 PhysicalMemoryTypeFilter ........................................... 959
B.2.1.3 Schedulable ................................................................... 966
B.2.2 Classes ............................................................................ 981
B.2.2.1 AbsoluteTime ................................................................ 981
B.2.2.2 AperiodicParameters ..................................................... 985
B.2.2.3 ArrivalTimeQueueOverflowException ............................... 990
B.2.2.4 AsyncEvent .................................................................. 991
B.2.2.5 AsyncEventHandler ...................................................... 994
B.2.2.6 AsynchronouslyInterruptedException ......................... 1011
B.2.2.7 BoundAsyncEventHandler ............................................. 1018
B.2.2.8 DuplicateFilterException ............................................. 1020
B.2.2.9 HighResolutionTime ..................................................... 1021
B.2.2.10 IllegalAssignmentError .............................................. 1024
B.2.2.11 ImmortalPhysicalMemory ............................................. 1024
B.2.2.12 ImportanceParameters ...........................................1033
B.2.2.13 LTMemory .....................................................1036
B.2.2.14 LTPhysicalMemory .............................................1042
B.2.2.15 MemoryAccessException ..........................................1052
B.2.2.16 MemoryParameters .............................................1052
B.2.2.17 MemoryScopeException .........................................1056
B.2.2.18 NoHeapRealtimeThread ..........................................1057
B.2.2.19 OffsetOutOfBoundsException ....................................1061
B.2.2.20 POSIXSignalHandler ..........................................1062
B.2.2.21 PeriodicParameters ...........................................1068
B.2.2.22 PhysicalMemoryManager ........................................1069
B.2.2.23 PriorityParameters ............................................1079
B.2.2.24 PriorityScheduler ...............................................1080
B.2.2.25 ProcessingGroupParameters ....................................1089
B.2.2.26 RawMemoryAccess ...............................................1099
B.2.2.27 RawMemoryFloatAccess .........................................1123
B.2.2.28 RealtimeSecurity ...............................................1135
B.2.2.29 RealtimeSystem .................................................1138
B.2.2.30 RealtimeThread .................................................1139
B.2.2.31 RelativeTime ...................................................1160
B.2.2.32 ReleaseParameters .............................................1165
B.2.2.33 Scheduler .......................................................1166
B.2.2.34 ScopedMemory ..................................................1172
B.2.2.35 SporadicParameters ............................................1187
B.2.2.36 ThrowBoundaryError ...........................................1192
B.2.2.37 Timed ..........................................................1192
B.2.2.38 UnknownHappeningException ..................................1195
B.2.2.39 UnsupportedPhysicalMemoryException .........................1196
B.2.2.40 VTMemory .......................................................1197
B.2.2.41 VTPhysicalMemory ...............................................1203
B.3 Rationale ..........................................................1213

C Indices ............................................................1215
C.1 Class Index ........................................................1215
C.2 Class Index ........................................................1215
C.3 Method Index ........................................................1218
C.4 Method Index ........................................................1218
List of Figures

6.1 Sequence Diagram of Some Example Realtime Thread Releases . . . . . 104
6.2 A State Chart for a Realtime Thread without a Deadline Miss Handler 105
6.3 A State Chart for a Realtime Thread with a Deadline Miss Handler . 106

8.1 The Event Class Hierarchy ........................................... 260
8.2 Releasing an AsyncEventHandler .................................... 262
8.3 States of a Simple AsyncBaseEvent ................................. 262
8.4 States of an ActiveEvent ............................................ 263

10.1 Sequence Diagram for Using a Timer .............................. 390
10.2 Sequence Diagram for Realtime Sleep ............................. 391
10.3 States of a Timer .................................................. 396

11.1 Manipulation of StackedMemory Areas ......................... 456

12.1 Control Flow Change State Diagrams ............................ 599

13.1 Raw Memory Interface .............................................. 624
13.2 Event Classes .................................................... 628
13.3 Happening State Transition Diagram .............................. 629
13.4 Interrupt servicing ................................................ 630
13.5 Creating Raw Memory Accessors .................................. 768
13.6 Flash memory device .............................................. 770
List of Tables

3.1 RTSJ Options ................................................. 31
5.1 PhasingPolicy Effect on First Release of a RealtimeThread with PeriodicParameters ................................................. 53
6.3 AperiodicParameters Default Values ................................................. 148
6.4 FirstInFirstOut Default PriorityParameter Values ................................................. 157
6.5 PeriodicParameter Default Values ................................................. 161
6.6 PriorityScheduler Default PriorityParameter Values ................................................. 171
6.7 ProcessingGroup Default Values ................................................. 175
6.8 ReleaseParameter Default Values ................................................. 193
6.9 SporadicParameters Default Values ................................................. 218
8.1 Event to Handler Matrix ................................................. 258
9.1 Examples of Normalized Times ................................................. 345
9.2 Semantics of Time Conversion ................................................. 346
11.1 Memory Area Referencing Restrictions ................................................. 447
13.1 Properties Array ................................................. 743
B.1 ProcessingGroupParameter Default Values ................................................. 1091
B.2 Properties Array ................................................. 1101
LIST OF TABLES

xvi RTSJ 2.0 (Draft 66)
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 for 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 achieve this goal.

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, imposes additional requirements.

Providing realtime semantics and the additional programming interfaces required
is a core part of this specification. This led the original specification to provided
special memory areas to avoid the use of garbage collection; however, 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 core Java semantics and with 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 periodic 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 a set of programming interfaces requires some
guiding principles around which to organize the RTSJ. The following principles delimit
the scope of the RTSJ and its compatibility requirements with conventional Java.
They ensure that conventional Java code can be run with realtime Java code on a
single Java virtual machine.
1.1.1 Applicability Across 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
1 Introduction

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. These principles are reflected in the following aspects of the realtime refinements and additional classes 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 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, first-in-first-out, scheduler. Since most realtime systems also support round-robin scheduling, a round-robin scheduler shall also be supplied. For compatibility with conventional Java implementations, both schedulers shall use 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 exclusive 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 use 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 should be 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; however, 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 topic—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 `AsyncBaseEventHandler` 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. It also provides a priority-based round-robin (RR) scheduler. Most realtime operating systems (RTOS) are also based on priority preemptive scheduling and support both FIFO and RR scheduling.

The RTSJ defines a number of classes with names of the form <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 and round-robin schedulers, 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
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 Overview

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 mechanism 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. When 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 mechanism, 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.

1. 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.
2. 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.
3. 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.

1. Threads waiting to enter synchronized blocks are granted access to the synchronized block in priority order.
2. A blocked thread that becomes ready to run is given access to a processor in priority order.
3. A thread whose priority is explicitly set by itself or another thread is given access to a processor in priority order.
4. 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.
5. 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, nonblocking, 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 events provide a means of starting computation based on external stimuli. The asynchronous event facility is based on two classes: AsyncBaseEvent and AsyncBaseEventHandler. An AsyncBaseEvent 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 AsyncBaseEventHandler are scheduled and the handleAsyncEvent() methods are invoked, thus the required logic is performed. Also, methods on AsyncBaseEvent are provided to manage the set of instances of AsyncBaseEventHandler associated with the instance of AsyncBaseEvent.

An instance of an AsyncBaseEventHandler 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 AsyncBaseEventHandler from a simple Runnable is that an AsyncBaseEventHandler has associated instances of ReleaseParameters, SchedulingParameters and MemoryParameters that control the actual execution of the handler once the associated AsyncBaseEvent 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 AsyncBaseEvent and AsyncBaseEventHandler (tens of thousands), since the number of fired (in progress) handlers is expected to be much smaller.

There are specialized forms of AsyncBaseEvent: 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 AsyncBaseEventHandler: 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
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 actual 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 only in 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 a signal from another thread or by the expiration of a timer 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 cost and bound of an iteration. 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.

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.

2.3.2.1 Methodological Principles

1. 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).

2. 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.

3. 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`).

### 2.3.2.2 Expressibility Principles

1. 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).

2. 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.

3. 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.

### 2.3.2.3 Semantic Principles

1. When 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.

2. 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. When 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.
### 2.3.2.4 Pragmatic Principles

1. There should be straightforward programming idioms for common cases such as timer handlers and realtime thread termination.
2. When 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.

### 2.3.3 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, e.g., when 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:

1. make all of the application methods of the realtime thread asynchronously interruptible;
2. create an oracle\(^2\) 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;
3. have the handlers call `interrupt()` on each of the realtime threads affected by the change; then
4. 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 but orderly clean up and termination of the realtime thread.

\(^2\)Note, the oracle can comprise as many or as few asynchronous event handlers as appropriate.
2 Overview

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 \texttt{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 \texttt{long} 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 conventional 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 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:
1. 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.

2. Immortal memory represents an area of memory containing objects that may be referenced without exception or garbage collection delay by any schedulable, specifically including realtime threads and asynchronous event handlers configured to not have access to the heap.

3. 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.

4. 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 vice 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 extraheap 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.

For cases where the usage of memory does not follow a stack discipline, in particular code that uses the producer-consumer pattern, a special variant of scoped memory is provided. This variant PinnableMemory has the same semantics as LTMemory except that a task can “pin” the memory, thereby keeping it open, even when no task is in the area. One task can fill the memory, put a reference in its portal, and then pass it on to another task to consume the data therein. Thus one does not have to have a dummy task to hold a pinned area open while it is passed from producer to consumer.

The flexibility provided in choice of scoped memory types enables 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 PhysicalMemoryRegion and PhysicalMemoryFactory classes provide the programmer this flexibility. The
A 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, where `<Size>` is one of `Byte`, `Short`, `Long`, `Float`, or `Double`, define methods that enable the programmer to construct an object that represents a vector of consecutive positions in memory where the `Size` represents a primitive numerical data type, i.e., byte, short, int, long, float, and double respectively. 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. On the other hand, the `PhysicalMemoryRegion` and `PhysicalMemoryFactory` 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 `RawMemory` 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 `RawMemory` interface enables a real-time 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 Overview

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: StaticThrowable. 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 Java exception hierarchy is poorly designed: there is no common base class for checked exceptions, so application 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, where a checked exception is an exception that is not a subclass of RuntimeException and an error is a throwable that is not a subclass of Exception.

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, handling 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

General Requirements

The RTSJ is both an Application Programmer Interface (API) and a refinement of the semantics of the Java virtual machine. Both aspects are necessary to produce a programming environment conducive to programming realtime systems. Most realtime systems require features that go beyond simply being able to react within a defined time bounds, they must also respond to something and take action thereon. Therefore, the ability to interact with the external environment is a necessary part of a realtime specification.

There are many applications that can benefit from the API and semantic refinements of the Java runtime environment that have been described above. Not every application requires all parts, so some flexibility of implementation is necessary. Therefore the RTSJ is divided into a core package and three optional packages. Furthermore, it also provides for different usage modes to support both development and deployment.

Finally, the vast majority of realtime systems are also embedded systems. The constraints of such system must also be considered. The specification begins with the overall requirements of these concerns.

3.1 Definitions

Code — Program text written in the Java programming language.
Java Language — A programming language defined through the Java Community Process.
Heap — An area of memory for allocating data structures (objects) defined by the Java Language.
Extraheap Memory — An area of memory for allocating data structures (objects) other than the heap defined by the Java Language.
Thread — An instance of the java.lang.Thread class.
3 General Requirements

Realtime Thread — An instance of the `javax.realtime.RealtimeThread` class.
Java Thread — An instance of java.lang.Thread class, but does not extend the `javax.realtime.RealtimeThread` class.
Heapless Realtime Thread — An instance of the `javax.realtime.RealtimeThread` class that must not access the heap.
Event Handler — An instance of the `javax.realtime.AsyncBaseEventHandler` class.
Schedulable — Any object that is of type `Schedulable`, and is recognized as a dispatchable entity by the required schedulers. The required schedulers’ set of schedulables comprises instances of `RealtimeThread` and `AsyncBaseEventHandler`. Other schedulers may support a different set of schedulables, but this specification only defines the behavior of the required schedulers so the term schedulable should be understood as “schedulable by the base scheduler.”
Task — Any object that represents computation, including schedulables and Java threads and instances of Schedulable.
Garbage Collection — A processes that reclaims memory on the heap that is no longer reachable by the application program. It may be accomplished through a dedicated set of threads or be distributed throughout the application.

3.2 Semantics

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.

3.2.1 Base Requirements

The base requirements of this specification are as follows.
1. Except as specifically required by this specification, any implementation shall fully conform to a Java platform configuration.
2. Any implementation of this specification shall implement all classes and methods in the base module of this specification.
3. Except as noted in this chapter, all classes and methods in an implemented module shall be implemented.
4. The `javax.realtime` package and its subpackages shall contain no public or protected classes or methods not included in this specification.
5. A realtime JVM implementation shall not be implemented in a way that permits unbounded priority inversion in any scheduling interaction it implements.

6. All methods defined under javax.realtime can safely be used concurrently by multiple threads unless otherwise documented.

7. Static final values, as found in AperiodicParameters, 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 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.

3.2.2 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 related functionality together in a way that promotes maximal adoption for various implementation classes. A conventional JVM may simply implement the Core Module API, without providing any realtime guarantees at all, thereby providing 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 Core Module functionality, but all other modules are optional. The optional modules are the Device Module, the Alternative Memory Areas Module and the POSIX Module. In addition, there are a couple of optional features as well. This give the implementers some choice over which modules and features to include and which not.
3 General Requirements

3.2.2.1 Core Module

The Core 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 all in package `javax.realtime` and are listed below.

- `AbsoluteTime` (Section 9.3.1.1)
- `ActiveEvent` (Section 8.3.1.1)
- `ActiveEventDispatcher` (Section 8.3.2.1)
- `Affinity` (Section 6.3.3.1)
- `AffinityPermission` (Section 15.2.2.1)
- `AperiodicParameters` (Section 6.3.3.2)
- `AsyncBaseEvent` (Section 8.3.2.2)
- `AsyncBaseEventHandler` (Section 8.3.2.3)
- `AsyncEvent` (Section 8.3.2.4)
- `AsyncEventHandler` (Section 8.3.2.5)
- `AsyncLongEvent` (Section 8.3.2.6)
- `AsyncLongEventHandler` (Section 8.3.2.7)
- `AsyncObjectEvent` (Section 8.3.2.8)
- `AsyncObjectEventHandler` (Section 8.3.2.9)
- `BackgroundParameters` (Section 6.3.3.3)
- `BlockableReleaseRunner` (Section 8.3.2.10)
- `BoundAsyncEventDispatcher` (Section 8.3.2.11)
- `BoundAsyncLongEventHandler` (Section 8.3.2.12)
- `BoundAsyncObjectEventHandler` (Section 8.3.2.13)
- `BoundSchedulable` (Section 6.3.1.1)
- `ConfigurationParameters` (Section 5.3.2.1)
- `CoreMemoryPermission` (Section 15.2.2.2)
- `EnclosedType` (Section 11.3.1.1)
- `FirstInFirstOutParameters` (Section 6.3.3.4)
- `FirstInFirstOutScheduler` (Section 6.3.3.5)
- `FixedPriorityReleaseRunner` (Section 8.3.2.14)
- `GarbageCollector` (Section 15.2.2.3)
- `HeapMemory` (Section 11.3.2.1)
- `HighResolutionTime` (Section 9.3.1.2)
- `ImmortalMemory` (Section 11.3.2.2)
- `MemoryArea` (Section 11.3.2.3)
- `MemoryParameters` (Section 11.3.2.4)
• MinimumInterarrivalPolicy (Section 6.3.2.1)
• MonitorControl (Section 7.3.1.1)
• PerennialMemory (Section 11.3.2.5)
• PeriodicParameters (Section 6.3.3.6)
• PhasingPolicy (Section 5.3.1.1)
• PriorityCeilingEmulation (Section 7.3.1.2)
• PriorityInheritance (Section 7.3.1.3)
• PriorityParameters (Section 6.3.3.7)
• PriorityScheduler (Section 6.3.3.8)
• ProcessingGroup (Section 6.3.3.9)
• QueueOverflowPolicy (Section 6.3.2.2)
• RealtimePermission (Section 15.2.2.4)
• RealtimeSecurity (Section B.2.2.28)
• RealtimeSystem (Section 15.2.2.5)
• RealtimeThread (Section 5.3.2.2)
• RealtimeThreadGroup (Section 6.3.3.10)
• RelativeTime (Section 9.3.1.3)
• Releasable (Section 8.3.1.2)
• ReleaseParameters (Section 6.3.3.11)
• ReleaseRunner (Section 8.3.2.15)
• RoundRobinParameters (Section 6.3.3.12)
• RoundRobinScheduler (Section 6.3.3.13)
• RTSJModule (Section 15.2.1.1)
• Schedulable (Section 6.3.1.2)
• Scheduler (Section 6.3.3.14)
• SchedulingParameters (Section 6.3.3.15)
• SchedulingPermission (Section 15.2.2.6)
• SizeEstimator (Section 11.3.2.6)
• SporadicParameters (Section 6.3.3.16)
• Subsumable (Section 8.3.1.3)
• TaskPermission (Section 15.2.2.7)
• WaitFreeReadQueue (Section 7.3.1.4)
• WaitFreeWriteQueue (Section 7.3.1.5)

All throwables defined in the RTSJ are also in the javax.realtime package:

• AlignmentError (Section 16.2.2.1)
• ArrivalTimeQueueOverflowException (Section 16.2.2.2)
• CeilingViolationException (Section 16.2.2.3)
• ConstructorCheckedException (Section 16.2.2.4)
• DeregistrationException (Section 16.2.2.5)
• EventQueueOverflowException (Section 16.2.2.6)
3 General Requirements

- **ForEachTerminationException** (Section 16.2.2.7)
- **IllegalAssignmentError** (Section 16.2.2.8)
- **IllegalRealtimeArgumentException** (Section 16.2.2.9)
- **IllegalRealtimeStateException** (Section 16.2.2.10)
- **IllegalTaskStateException** (Section 16.2.2.11)
- **InaccessibleAreaException** (Section 16.2.2.12)
- **LateStartException** (Section 16.2.2.13)
- **MemoryAccessException** (Section 16.2.2.15)
- **MemoryInUseException** (Section 16.2.2.16)
- **MemoryScopeException** (Section 16.2.2.17)
- **MITViolationException** (Section 16.2.2.14)
- **OffsetOutOfBoundsException** (Section 16.2.2.19)
- **OutOfMemoryStaticError** (Section 16.2.2.20)
- **POSIXException** (Section 16.2.2.21)
- **POSIXInvalidSignalException** (Section 16.2.2.22)
- **POSIXInvalidTargetException** (Section 16.2.2.23)
- **POSIXSignalPermissionException** (Section 16.2.2.24)
- **ProcessorAffinityException** (Section 16.2.2.25)
- **RangeOutOfBoundsException** (Section 16.2.2.26)
- **RealtimeSecurityException** (Section 16.2.2.27)
- **RegistrationException** (Section 16.2.2.28)
- **ResourceLimitError** (Section 16.2.2.29)
- **ScopedCycleException** (Section 16.2.2.30)
- **SizeOutOfBoundsException** (Section 16.2.2.31)
- **StaticCheckedException** (Section 16.2.2.32)
- **StaticError** (Section 16.2.2.33)
- **StaticRuntimeException** (Section 16.2.2.34)
- **StaticThrowable** (Section 16.2.1.1)
- **StaticThrowableStorage** (Section 16.2.2.35)
- **ThrowBoundaryError** (Section 16.2.2.36)
- **UninitializedStateException** (Section 16.2.2.37)
- **UnsupportedPhysicalMemoryException** (Section 16.2.2.38)
- **UnsupportedRawMemoryRegionException** (Section 16.2.2.39)
- **UnsupportedRealtimeOperationException** (Section 16.2.2.40)

3.2.2.2 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 all in package `javax.realtime.memory` and are listed below.

- `LTMemory` (Section 11.4.3.1)
- `MemoryGroup` (Section 11.4.3.2)
- `PhysicalMemoryCharacteristic` (Section 11.4.1.1)
- `PhysicalMemoryFactory` (Section 11.4.3.3)
- `PhysicalMemoryRegion` (Section 11.4.3.4)
- `PhysicalMemorySelector` (Section 11.4.3.5)
- `PinnableMemory` (Section 11.4.3.6)
- `ScopedConfigurationParameters` (Section 11.4.3.7)
- `ScopedMemory` (Section 11.4.3.8)
- `ScopedMemoryParameters` (Section 11.4.3.9)
- `StackedMemory` (Section 11.4.3.10)

### 3.2.2.3 Control Module

Conventional Java provided a single exception for asynchronous control flow change: the `ThreadDeath` error. This is thrown when `Thread.stop()` is called. Unfortunately, throwing `ThreadDeath` is not thread safe, so it has been deprecated. This module provides an alternative that is thread safe. It is optional because support for this module requires significant changes to the VM. The classes required are all in the package `javax.realtime.control` and are listed below.

- `AsynchronousControlGroup` (Section 12.3.2.1)
- `AsynchronouslyInterruptedException` (Section 12.3.2.2)
- `Interruptible` (Section 12.3.1.1)
- `Timed` (Section 12.3.2.3)

### 3.2.2.4 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 all in the package `javax.realtime.device` and are listed below.

- `DirectMemoryBufferFactory` (Section 13.3.2.1)
- `DirectMemoryByteBuffer` (Section 13.3.1.1)
- `DirectMemoryRegion` (Section 13.3.2.2)
- `Happening` (Section 13.3.2.3)
- `HappeningDispatcher` (Section 13.3.2.4)
- `InterruptCeilingEmulation` (Section 13.3.2.5)
3 General Requirements

- InterruptDescriptor (Section 13.3.2.6)
- InterruptInheritance (Section 13.3.2.7)
- InterruptMasking (Section 13.3.2.8)
- InterruptServiceRoutine (Section 13.3.2.9)
- InterruptUnmaskable (Section 13.3.2.10)
- RawByte (Section 13.3.1.2)
- RawByteReader (Section 13.3.1.3)
- RawByteWriter (Section 13.3.1.4)
- RawDouble (Section 13.3.1.5)
- RawDoubleReader (Section 13.3.1.6)
- RawDoubleWriter (Section 13.3.1.7)
- RawFloat (Section 13.3.1.8)
- RawFloatReader (Section 13.3.1.9)
- RawFloatWriter (Section 13.3.1.10)
- RawInt (Section 13.3.1.11)
- RawIntReader (Section 13.3.1.12)
- RawIntWriter (Section 13.3.1.13)
- RawLong (Section 13.3.1.14)
- RawLongReader (Section 13.3.1.15)
- RawLongWriter (Section 13.3.1.16)
- RawMemory (Section 13.3.1.17)
- RawMemoryFactory (Section 13.3.2.11)
- RawMemoryRegion (Section 13.3.2.12)
- RawMemoryRegionFactory (Section 13.3.1.18)
- RawShort (Section 13.3.1.19)
- RawShortReader (Section 13.3.1.20)
- RawShortWriter (Section 13.3.1.21)

3.2.2.5 POSIX module

The POSIX module provides access to functionality particular to POSIX systems. In particular, it addresses POSIX signals and POSIX realtime signals. This module is optional, but an implementation of this standard on a POSIX platform should provide it. Implementations on platforms that are not POSIX compliant may provide it. The classes in this module are in the package \texttt{javax.realtime posix} and are listed below.

- RealtimeSignal (Section 14.3.1.1)
- RealtimeSignalDispatcher (Section 14.3.1.2)
- Signal (Section 14.3.1.3)
- SignalDispatcher (Section 14.3.1.4)
3.2.3 Optional Features

Even with modules, it is difficult to eliminate 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>
<thead>
<tr>
<th>Optional Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hard cost enforcement</td>
<td>Provides an automatic means of controlling the processor usage of a task or group of tasks.</td>
</tr>
<tr>
<td>Processing group deadline less than period</td>
<td>Enables the application to specify a processing group deadline less than the processing group period</td>
</tr>
<tr>
<td>Allocation-rate enforcement on heap allocation</td>
<td>Enables the application to limit the rate at which a schedulable creates objects in the heap.</td>
</tr>
<tr>
<td>Interrupt service routine</td>
<td>Provides first level interrupt processing in Java.</td>
</tr>
</tbody>
</table>

The ProcessingGroup class only intervenes in scheduling on systems that support the hard cost enforcement option. The precision of intervention is limited by the precision of the clock being used to measure time times the number of CPUs involved in the enforcement. When cost enforcement is supported, the precision of enforcement is the drive precision of the clock being used. In any event, cost and deadline overrun handlers are fired with the resolution specified for hard cost enforcement.

In implementations where processing group deadline less than period is not supported, values passed to the constructor for ProcessingGroup 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 InterruptServiceRoutine.register should always throw an UnsupportedOperationException.
3 General Requirements

Extensions to this specification are allowed, but shall not require changes to the public interfaces defined in the `javax.realtime` package tree in particular and the `java` and `javax` package trees in general.

### 3.2.4 Deprecated Classes

Classes and methods that have been deprecated as of this specification are not part of any module, but may be implemented by a full RTSJ implementation. The following classes are deprecated:

- `AsynchronouslyInterruptedException` (Section B.2.2.6)
- `DuplicateFilterException` (Section B.2.2.8)
- `ImmortalPhysicalMemory` (Section B.2.2.11)
- `ImportanceParameters` (Section B.2.2.12)
- `Interruptible` (Section B.2.1.1)
- `LTMemory` (Section B.2.2.13)
- `LTPhysicalMemory` (Section B.2.2.14)
- `NoHeapRealtimeThread` (Section B.2.2.18)
- `PhysicalMemoryManager` (Section B.2.2.22)
- `PhysicalMemoryTypeFilter` (Section B.2.1.2)
- `POSIXSignalHandler` (Section B.2.2.20)
- `ProcessingGroupParameters` (Section B.2.2.25)
- `RawMemoryAccess` (Section B.2.2.26)
- `RawMemoryFloatAccess` (Section B.2.2.27)
- `ScopedMemory` (Section B.2.2.34)
- `Timed` (Section B.2.2.37)
- `UnknownHappeningException` (Section B.2.2.38)
- `VTMemory` (Section B.2.2.40)
- `VTPhysicalMemory` (Section B.2.2.41)

They are documented fully in Chapter B.

### 3.2.5 Implementation types Allowed

As described in Section 3.2.2, the RTSJ now has modules. Every implementation, except one supporting Safety Critical Java, must implement the Core module. Each module provided by an implementation must be provided in full. None of the classes of an unimplemented module should be present. Only an implementation of this specification exclusively used for supporting Safety Critical Java may subset classes and packages herein, but must implement the methods and classes defined in that specification.
3.2.5.1 Realtime Deployment Implementation

A realtime deployment implementation must support all semantics described herein necessary for deterministic programming. In addition to implementing the core module, a realtime deployment implementation must have a realtime garbage collector or implement the alternative memory areas module. All other modules are optional.

The minimum scheduling semantics that must be supported in all implementations of the RTSJ are fixed-priority preemptive scheduling with support for at least 28 unique priority levels\(^1\). Fixed priority means that the system does not change the priority of any Schedulable except, temporarily, for priority inversion avoidance. Priority change is under control of the application.

What the RTSJ precludes by this statement is scheduling algorithms for realtime priorities which change thread priorities according to policies for optimizing throughput. An implementation may not increase the priority of a thread that has been receiving few processor cycles because of higher priority threads (aging) or other so-called fair scheduling algorithms. Fair scheduling operations are also prohibited. These types of algorithms are reserved for conventional Java thread priorities. This does not prohibit an application from implementing other realtime schedulers, such as earliest deadline first, which use underlying OS priorities to support an application meeting its deadlines.

The 28 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 Schedulable behave as though there were at least 28 unique levels. (e.g. if there were 28 RealtimeThreads \(t_1, \ldots, t_{28}\) with priorities \((p_1, \ldots, p_{28})\), 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 \(t_2, \ldots, t_{28}\) and thread \(t_2\) would always execute in preference to threads \(t_3, \ldots, t_{28}\), etc.)

The minimum synchronization semantics that must be supported in all deployment implementations of the RTSJ are detailed in the section on synchronization below 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 schedulables. Both the priority inheritance and the priority ceiling emulation protocols must be implemented, but priority inheritance is the default.

\(^1\)This does not mean that each deployment must have all 28 priorities active
All instances of **Scheduled** waiting to acquire a resource must be queued in priority order. This applies to the processor as well as to synchronized blocks. When schedulables with the same exact priority are possible under the active scheduling policy, schedulables with the same priority are queued in FIFO order. Note that these requirements apply only to the required scheduling policy and hence use the specific term 'priority'. In particular,

1. schedulables waiting to enter synchronized blocks are granted access to the synchronized block in priority order;
2. a blocked schedulable that becomes ready to run is given access to the processor in priority order;
3. a schedulable whose execution eligibility is explicitly set by itself or another schedulable is given access to the processor in priority order;
4. a schedulable that performs a **yield()** will be given access to the processor after all other schedulables waiting at the same priority;
5. however, schedulables that are preempted in favor of a schedulable 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.

Other realtime schedulers must provide and document similar algorithms to expedited schedulables with higher execution eligibility over those with lower execution eligibility.

The **RTSJ** does not require any particular garbage collection algorithm; however, every deployment implementation must either implement the alternate memory area module 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.

Notwithstanding the above, a program that uses the **RTSJ** and is deployed as an executable, so that it does not provide general access to the virtual machine, but solely runs that program code, need only include the **RTSJ** methods and classes needed by the application.

### 3.2.5.2 Simulation Implementation

An implementation that chooses not to provide realtime guarantees, 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.

The following semantics are optional for an **RTSJ** implementation designed and licensed exclusively as a development tool.
1. 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.

2. 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 (AIE) 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.

3. 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.

### 3.3 Required Documentation

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.

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

1. If schedulers other than the required first-in-first-out (FIFO) and round robin (RR) schedulers are available to applications, the behavior of these schedulers and their interaction with each other and the required schedulers as detailed
3 General Requirements

in Chapter 6, Scheduling, shall be documented.

(a) The documentation must define how its order of execution eligibility relates to that of the priority schedulers, where the order of execution eligibility of a priority scheduler is the priority order.
(b) The list of classes whose instances constitute schedulables for the scheduler, unless that list is the same as the list of schedulables for the required schedulers, shall be included.
(c) If there are restrictions on use of the scheduler from a context without heap access, such restrictions shall be documented as well.

2. A scheduler that cannot place a schedulable at the front of the queue for its active priority when it is preempted by a higher-priority schedulable must document such a deviation from the specification.

3. 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.

4. The implementation shall fully document the behavior of any subclasses of GarbageCollector.

5. 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.

6. 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.

7. For any available scheduler other than the required schedulers, an implementation shall document how, if at all, the semantics of synchronization differ from the rules defined for the default PriorityInheritance monitor control policy.

(a) 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.
(b) If there are restrictions on use of the scheduler from an extraheap context, the documentation shall detail the effect of these restrictions for each RTSJ API.

8. The worst-case response interval from the firing of an AsyncEvent, due to a bound happening, to releasing an associated AsyncEventHandler, assuming no higher-priority schedulables are runnable, shall be documented for at least one reference architecture.

9. 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 at least one reference architecture.

10. 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.

11. If hard 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.

12. If the implementation of RealtimeSecurity is more restrictive than the required implementation, or has run-time configuration options, those features shall be documented.

13. 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.

14. 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 extraheap context, the documentation shall detail the consequences of passing the clock, or a time that uses the clock to a heapless schedulable.

### 3.4 Rationale

The embedded market, especially for safety critical applications, is quite sensitive to including code that is not needed by an application. Furthermore, different application domains have differing needs on API. Flexibility is needed to ensure that these diverse domains and requirements are met. Still, it is important to ensure that when a given function is needed, it is included as defined herein. It is also important that an open virtual machine deployment has a well-defined API set. This has required moving a few classes into a new package, so that the resulting modules will be consistent with the rules imposed by the JSR 376, the Java Platform Module System. The above modules and deployment rules provide both this flexibility and standardization.
3 General Requirements
Chapter 4
Realtime vs Conventional Java

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 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. The specification additionally defines both an extension to thread for realtime scheduling and a new type of concurrent activity called an event handler; hence, the meaning of current thread has a different interpretation than in conventional Java. The term `task` is used when referring to any of these three types: conventional Java thread, realtime thread, and event handler.

Behaviors that may be different from conventional Java or may be surprising to developers of conventional Java applications under the RTSJ can be divided into three categories. The first category applies to conventional Java code that was not developed with the RTSJ in mind and does not use RTSJ features but runs under an RTSJ implementation. The second is conventional Java code that was not developed with the RTSJ in mind but is called by code developed for the RTSJ in an RTSJ implementation. The final category is Java code that was developed for the RTSJ and is being used in an RTSJ implementation.

The first category, conventional Java code running on an RTSJ implementation but not using any RTSJ features, may encounter the following behaviors that are not (necessarily) experienced under a conventional Java VM.

- Any object allocated in a static initializer that later becomes garbage may be unable to be collected by the VM. (See Section 11.2.7.)
- Some `Throwables`, in particular those implementing `StaticThrowable`, which includes `OutOfMemoryStaticError`, thrown by an RTSJ VM in preference to `OutOfMemoryError`, have stack trace and message information which is valid
only while the Throwable is in flight and in the thread which originally threw the Throwable. (See Section 16.1.)

The second category, conventional Java code that is running on an RTSJ implementation and in use by code that was developed for the RTSJ, may encounter the following differences in behavior.

- **IllegalAssignmentError** may be thrown in non RTSJ-aware classes when the Alternative Memory Management module (Chapter 11) is in use. (See Section 11.2.8.)
- Tasks in an RTSJ application might not be scheduled by a fair scheduler. The result is that there may be thread starvation unexpected by conventional Java applications. (See Section 6.2.1.)
- A call to Thread.getPriority() may return a priority higher than Thread.MAX_PRIORITY. (See Section 6.3.3.14.2.)
- Methods cannot rely on any thread local information when used in conjunction with asynchronous event handlers. This includes thread local data and calls to Thread.currentThread(). Hence, care must be taken when using thread identifiers to determine the identity of callers. (This is analogous to the use of ThreadPool in conventional Java.) (See Sections 8.2.1 and 8.3.2.5.)

The third and final category comprises behaviors exhibited by code designed for the RTSJ running on an RTSJ implementation that are departures from conventional Java semantics or may be otherwise surprising.

- Finally clauses in asynchronously interruptible methods are not executed during propagation of an AsynchronouslyInterruptedException. However, synchronized code is always ATC-deferred, and therefore monitor locks are released normally. (See Section 12.2.2.)
- Catch clauses that name AsynchronouslyInterruptedException (or its parent classes) will not automatically stop the propagation of AIEs. An AsynchronouslyInterruptedException must be explicitly cleared. (See Section 12.2.2.)
- Exceptions propagating into asynchronously interruptible regions of code will be lost if an AsynchronouslyInterruptibleException is pending. (See Section 12.2.2.)
- Subclasses of AsynchronouslyInterruptibleException indicated in the signature of a method do not indicate that the method is asynchronously interruptible. (See Section 12.2.2.)
- Catch clauses for AsynchronouslyInterruptibleException or its subclasses in asynchronously interruptible methods will not catch an AIE. (See Section 12.2.2.)
- A Throwable crossing a MemoryArea boundary might be transformed into a ThrowBoundaryError, and the original exception may be lost. (See Section 16.2.2.36 and the enter family of methods on MemoryArea.)
4.1 Definitions

**Conventional Java** — The language and runtime as defined by the “Java Language Specification[5]” and “Java Virtual Machine Specification[6],” without any realtime extensions.

**Realtime Java** — Conventional Java extended and refined according to this specification for programming realtime systems.

**Fair Scheduling** — A method of nonrealtime scheduling which tries to ensure that all tasks get a chance to run, thus preventing starvation. Tasks with a higher priority get a notionally larger share of execution time than lower priority tasks. Tasks running at the same priority get notionally equal shares of the processor.

**Happens-Before** — The “Java Language Specification[5]” specifies the happens-before relationship as “If one action happens-before another, then the first is visible to and ordered before the second.” See the specification for the implications of this relationship.

**Priority** — An indication of the relative scheduling eligibility of a task. A task with a higher priority is scheduled before a task with a lower priority. The priority assigned to a task is not necessarily the one used for scheduling, since priority avoidance and cost enforcement mechanisms may transiently override it. See Base Priority in Section 6.1 and Active Priority in Section 7.1.

**Task** — A conventional Java thread or an RTSJ Schedulable.

4.2 Semantics

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, cause unexpected behaviors. This is particularly true when using alternate memory areas, asynchronous transfer of control, and thread local memory in conjunction with unbound asynchronous event handlers.

4.2.1 Scheduling

How tasks are scheduled in a realtime system is quite different from 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
4 Realtime vs Conventional Java

part via task priority.

Tasks 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 underspecified in [5]. This has been resolved in practice to avoid surprising the programmer by providing some sort of fair scheduling for these threads, i.e., scheduling that at least prevents task starvation, but may also try to balance CPU availability across threads. For tasks running in these priorities, an implementation of this specification shall provide some notion of fair scheduling between tasks with priority between one and ten inclusive.

Realtime threads and event handlers need a stronger notion of prioritization than conventional Java threads, so this specification requires the implementation of two priority-preemptive schedulers, one with run to completion (or next suspension point) and one with round-robin semantics. Priorities above the conventional ten priorities are used for these schedulers, and the interactions of the two schedulers are well-defined. 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.

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

Code running at realtime priorities can also starve tasks scheduled on the conventional Java scheduler, possibly indefinitely.

4.2.1.1 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.

4.2.1.1.1 Setting Priority

The setPriority method has the following additional requirements.

1. 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 \( \text{Thread.setPriority()} \) are governed by semantics from Chapter 7 on \textit{Synchronization}.

2. Conventional Java threads may not use \text{setPriority} to apply the expanded range of priorities defined by this specification.

3. When \text{setPriority} is called on a realtime thread, that thread’s \textit{Scheduling-Parameters} are set to \text{null} and the thread is scheduled as if it were a Java thread.

4.2.1.1.2 Getting Priority

The \text{getPriority} method has the following additional requirements.

1. When called on a conventional Java thread, its assigned priority is returned even if it has a higher priority than what would be allowed by conventional Java. It may be higher only when set with an instance of \textit{SchedulingParameters} through a scheduler.

2. When called on a realtime thread with \text{null SchedulingParameters}, a value in the conventional Java priority range is returned.

3. When called on a realtime thread \( (t) \) with \textit{PriorityParameters}, \text{getPriority} behaves effectively as if it included the following code snippet:

   ```
   1. \((\text{PriorityParameters})t.\text{getSchedulingParameters}()).\text{getPriority}()
   ```

4. When the scheduling parameters are of a type other than \textit{PriorityParameters}, a \texttt{ClassCastException} is thrown.

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

4.2.1.2 Thread Groups

Conventional Java provides thread groups as a means of managing groups of threads. Since the RTSJ provides additional classes for encapsulating control flow under the umbrella of \textit{Schedulable}, it makes sense to have facilities for managing groups of these as well. The RTSJ provides an extension of \text{ThreadGroup} for this called \textit{RealtimeThreadGroup}.

Every instance of \text{ThreadGroup} holds a reference to every member thread and every subgroup instance of \text{ThreadGroup}, as well as a reference to its parent group. This is problematic under the RTSJ, since realtime threads may be allocated in scoped memory. Rather than making complicated changes to the semantics of \text{ThreadGroup} (and, in particular, its \texttt{enumerate} methods), the RTSJ requires that no \text{ThreadGroup} or Java thread is allocated in scoped memory, and that no thread allocated in
ScopedMemory is referenced by a ThreadGroup. Instances of RealtimeThreadGroup are instead used for these purposes, and an alternative to enumerate is provided on RealtimeThreadGroup in the form of a visitor.

Realtime thread groups, i.e., instances of RealtimeThreadGroup, a subclass of ThreadGroup, are designed to be able to reference threads, schedulables, and other realtime thread groups, even when they are in scoped memory. These are only reachable using a visitor with a lambda expression. Consequently schedulables and realtime thread groups are not part of any thread group and will hold a realtime thread group reference as their parent thread group. This requires that the thread group of the main thread is also a schedulable group, so that schedulables and schedule groups can be created from the main thread.

In order for this to work in a transparent manner, the following rules must hold.

1. An instance of ThreadGroup that is not an instance of RealtimeThreadGroup cannot contain any instances of Schedulable.
2. In an RTSJ implementation, both the ThreadGroup at the root of the ThreadGroup hierarchy and the ThreadGroup to which the initial thread belongs must be instances of RealtimeThreadGroup.
3. Calls to RealtimeThreadGroup.enumerate(Thread[]) and RealtimeThreadGroup.enumerate(Thread[], boolean) only return Java threads.
4. Calls to RealtimeThreadGroup.enumerate(ThreadGroup[]) and RealtimeThreadGroup.enumerate(ThreadGroup[], boolean) only return threads groups and realtime thread groups allocated in heap and immortal memory.
5. A Java thread (not a realtime thread) that is created from a realtime thread or bound asynchronous event handler without an explicit thread group and that is not assigned a thread group by the security manager, inherits the realtime thread group of its creator, when that group is allocated in heap or immortal memory; otherwise an IllegalAssignmentError is thrown.
6. The thread group of a Java thread that is created from an unbound asynchronous event handler without an explicit thread group and that is not assigned a thread group by the security manager, is assigned to the realtime thread group of the handler’s dispatcher, when that dispatcher’s realtime thread group is allocated in heap or immortal memory; otherwise an IllegalAssignmentError is thrown.
7. A thread group cannot be created in scoped memory. The constructor shall throw an IllegalAssignmentError.
8. Setting a maximum priority on a realtime thread group, either explicitly or via its parent with a thread group specific method, has no influence on the schedulables in that group.
9. Except as specified previously, realtime threads and bound asynchronous event handlers have the same ThreadGroup membership rules as their parent Thread.
4.2.1.3 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 6.1 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 when used with unbound event handlers, because data saved in one release may not be available in the next release.

4.2.2 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 where AIE is an instance of the `AsynchronouslyInterruptedException`. See Chapter 8 on Asynchrony.

4.2.3 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 13.2.1), and `DirectMemoryBufferFactory` fence methods are ordered to be consistent with respect to native code or hardware devices that use platform-native memory coherency protocols to access raw memory or raw byte buffers shared with the virtual machine. In particular, all Java code that precedes a JNI call in the source happens-before the code executed during the JNI call, which happens-before all Java code that follows its return.
Though not specified for conventional Java, most implementations provide explicit fencing for JNI calls.

4.2.4 Memory Management

The specification provides for two means of managing memory: garbage collection and special memory areas. The latter 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 the alternative memory areas module is not present, the conventional Java semantics for allocation prevails.

4.2.4.1 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 when the current allocation context is different from the allocation context in which the object was created; this 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 an object from a scoped memory area will most likely cause an exception. Although using other memory areas, such as scoped memory, is useful for helping to improve 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.

4.2.4.2 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 environment 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 methods it uses to do so.

4.2.4.3 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. Even on a multicore machine, the garbage collector must be incremental, because it must tolerate changes to the heap during garbage collection. Then CPU use is limited by tying the collector to one or more cores.

4.2.4.3.1 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.

4.2.4.3.2 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.

4.2.4.3.3 Alternatives to Garbage Collection

This specification provides 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 Realtime vs Conventional Java

4.2.4.3.4 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 even in a conventional Java implementation.

4.3 Rationale

The threading model of conventional Java was never meant for realtime programming. Refinements to the virtual machine and new APIs are necessary to support the additional requirements of applications, which have tasks that must complete in a fixed amount of time. However, to ensure that any conventional Java program can run on a virtual machine or runtime that implements this specification requires careful consideration of each refinement to the Java programming model. Therefore, conventional Java APIs and semantics have been extended, rather than replaced, to facilitate compatibility with conventional Java runtime implementations.
Chapter 5

Realtime Threads

Conventional Java provides a thread class for its tasking model. Tasks can be run simultaneously by creating multiple threads, but they do not provide realtime scheduling semantics. For this, the specification provides a realtime thread class. This class 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`, and `MemoryParameters` objects that can be passed to the `RealtimeThread` constructor provide the temporal and processor configuration of the thread to be communicated to the scheduler. `ProcessingGroup`, a class derived from `ThreadGroup`, provides cost enforcement on groups of tasks. The `ConfigurationParameters` class defines, among other things, the size of Java’s thread stack. The `PhasingPolicy` class defines the relationship between the threads start time and its first release time when the start time is in the past.

The RTSJ provides two types of objects that 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 functionalities 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; and
- 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 Realtime Threads

5.1 Definitions

Exception — Both a mechanism of nonlocal transfer of control and a Java object which carried information about the cause of the control transfer.

Scheduler — A module that manages the execution of tasks, as well as detects deadline misses and monitoring costs.

5.2 Semantics

Instances of `RealtimeThread` have the same semantics as conventional Java threads except as noted below.

1. Garbage collection executing in the context of a Java thread must not in itself block execution of a schedulable with a higher execution eligibility that may not access the heap; however, application locks work as specified even when the lock causes synchronization between a heap-using thread and a schedulable that may not use the heap.

2. Each schedulable 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, when a call is made to the interrupt method in one of the family of asynchronous event handler classes, and when an asynchronously interrupted exception’s `fire` method is invoked between the time the schedulable has entered that exception’s `doInterruptible` method, and before it has return from `doInterruptible`. (See Chapter 8 on Asynchrony.)

3. A call to `Schedulable.interrupt()` generates the system’s generic `AsynchronouslyInterruptedException`. (See Chapter 8 on Asynchrony.)

4. The `RealtimeThread.waitForNextRelease` method is 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, the method returns when the realtime thread’s next period is due or the next release happens.

5. In the presence of a cost overrun or a deadline miss, the behavior of `waitForNextRelease` 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 and the value of any `PhasingPolicy` parameter passed to it.

7. Instances of `RealtimeThread` may not be created with a thread group which is not an instance of `RealtimeThreadGroup`.

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

9. The scheduling of a realtime thread is governed by its SchedulingParameters
and its Scheduler unless set explicitly with method setPriority(int) in
java.lang.Thread, which causes it to be treated as a conventional java thread
until a new SchedulingParameters object is set.

5.2.1 Startup Considerations

For efficient system startup, it is sometimes necessary that the Java main thread and
all internal system threads have a priority other than Thread.NORM_PRIORITY. The
core module provides a property for this. When javax.realtime.start.priority
is set, the initial and main Java threads start with the given priority; otherwise, the
default of Thread.NORM_PRIORITY is used.
5 Realtime Threads

5.3 javax.realtime

5.3.1 Enumerations

5.3.1.1 PhasingPolicy

public enum PhasingPolicy

Inheritance
java.lang.Object
   java.lang.Enum<PhasingPolicy>
      PhasingPolicy

Description
This class defines a set of constants that specify the supported policies for starting a periodic 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 realtime thread. The effective start time of a periodic timer is similar; where the first firing is equivalent to the first release, and a call to the constructor is equivalent to a call to RealtimeThread.start().

Available since RTSJ 2.0

5.3.1.1.1 Enumeration Constants

ADJUST_IMMEDIATE

public static final PhasingPolicy ADJUST_IMMEDIATE

Description
Indicates that a periodic thread started after the absolute time given for its start time should be released immediately with the next release one period later.

ADJUST_FORWARD

public static final PhasingPolicy ADJUST_FORWARD

52 RTSJ 2.0 (Draft 66)
Table 5.1: PhasingPolicy Effect on First Release of a RealtimeThread with Periodic-Parameters

<table>
<thead>
<tr>
<th>Description</th>
<th>ADJUST_IMMEDIATE</th>
<th>ADJUST_FORWARD</th>
<th>ADJUST_BACKWARD</th>
<th>STRICT_PHASING</th>
</tr>
</thead>
<tbody>
<tr>
<td>RelativeTime</td>
<td>The time of start method invocation plus \texttt{start} time.</td>
<td>The time of start method invocation plus \texttt{start} time.</td>
<td>The time of start method invocation plus \texttt{start} time.</td>
<td>The time of start method invocation plus \texttt{start} time.</td>
</tr>
<tr>
<td>AbsoluteTime, earlier than call to \texttt{start}</td>
<td>Release immediately and set next release time to be at the time the start method was invoked plus \texttt{period}.</td>
<td>All releases before the time \texttt{start} is called are ignored. The first release is at the start time plus the smallest multiple of \texttt{period} whose time is after the time \texttt{start} was called.</td>
<td>The first release occurs immediately and the next release is at the start time plus the smallest multiple of \texttt{period} whose time is after the time \texttt{start} was called.</td>
<td>The \texttt{start} method throws an exception.</td>
</tr>
<tr>
<td>AbsoluteTime, later than call to \texttt{start}</td>
<td>First release is at time passed to \texttt{start}.</td>
<td>First release is at time passed to \texttt{start}.</td>
<td>First release is at time passed to \texttt{start}.</td>
<td>First release is at time passed to \texttt{start}.</td>
</tr>
<tr>
<td>Without Time</td>
<td>First release is at time of start method invocation</td>
<td>First release is at time of start method invocation</td>
<td>First release is at time of start method invocation</td>
<td>First release is at time of start method invocation</td>
</tr>
</tbody>
</table>

Description

Indicates that a periodic thread started after the absolute time given for its start time should be released at the next multiple of its period from its start time.

**ADJUST_BACKWARD**

```java
public static final PhasingPolicy ADJUST_BACKWARD
```
5 Realtime Threads

Description
Indicates that a periodic thread started after the absolute time given for its start time should be released immediately with the next release at the next multiple of its period from its start time.

**STRICT_PHASING**

public static final PhasingPolicy STRICT_PHASING

Description
Indicates that a periodic thread started after the absolute time given for its start time should throw the `LateStartException` exception instead of being released.

5.3.1.1.2 Methods

values

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

Description
Gets all enumeration constants.

valueOf(String)

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

Description
Gets enumeration constants corresponding to name.
5.3.2 Classes
5.3.2.1 ConfigurationParameters

public class ConfigurationParameters

Inheritance
java.lang.Object
   *ConfigurationParameters

Interfaces
Cloneable
Serializable

Description
Configuration parameters provide a way to specify various implementationdependent parameters such as the Java stack and native stack sizes, and to configure the statically allocated ThrowBoundaryError associated with a Schedulable.

Note that these parameters are immutable.

Available since RTSJ 2.0

5.3.2.1.1 Constructors

ConfigurationParameters(int, int, long)

Signature
public
ConfigurationParameters(int messageLength,
   int stackTraceLength,
   long[] sizes)
throws IllegalRealtimeStateException

Description
Creates a parameter object for initializing the state of a Schedulable. The parameters provide the data for this initialization. For RealtimeThread and
bound versions of AsyncBaseEventHandler, the stack and message buffers can be set exactly, but for the unbound event handlers, the system cannot give any guarantees to allow thread sharing.

**Parameters**

**messageLength**—The size of the buffer, in units of char, for storing an exception message used by preallocated exceptions and errors thrown in the context of an instance of Schedulable which was created with this as its configuration parameters. The value 0 indicates that no message should be stored. The value of -1 uses the system default and is the default when an instance of this class is not provided.

**stackTraceLength**—The length of the stack trace buffer, in units of a number of StackTraceElement instances, reserved use by preallocated exceptions and errors thrown in the execution context of the Schedulable object created with these parameters. 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 and is the default when an instance of this class is not provided.

**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. The default is system dependent, and indicated by setting this parameter to null or by not providing an instance of this class.

### ConfigurationParameters(long)

**Signature**

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

**Description**

Same as ConfigurationParameters(int,int,long[]) with arguments (0, 0, sizes).

### 5.3.2.1.2 Methods
setDefault(ConfigurationParameters)

Signature

    public static void
    setDefault(ConfigurationParameters config)

Description

Get the parameters object to be used when none is provided for an instance of Schedulable.

Parameters

    config—the new default parameter object.

getDefault

Signature

    public static javax.realtime.ConfigurationParameters
    getDefault()

Description

Set the parameters object to be used when none is provided for an instance of Schedulable.

Returns

    the default parameter object.

setDefaultRunner(ReleaseRunner)

Signature

    public static void
    setDefaultRunner(ReleaseRunner runner)

Description

Sets the system default release runner.

Parameters

    runner—The runner to be used when none is set. When null, the default release runner is set to the original system default.
5 Realtime Threads

ConfigurationParameters

**getDefaultRunner**

*Signature*

```java
public static javax.realtime.ReleaseRunner getDefaultRunner()
```

*Description*

Gets the system default release runner.

*Returns*

a general runner to be used when none is set.

**getMessageLength**

*Signature*

```java
public int getMessageLength()
```

*Description*

Gets the size of the buffer dedicated to storing the message of the last thrown throwable in the context of instances of `Scheduled` created with these parameters. The value 0 indicates that no message will be stored.

*Returns*

reserved memory size in units of `char`.

**getStackTraceLength**

*Signature*

```java
public int getStackTraceLength()
```

*Description*

Gets the length of the stack trace buffer dedicated to `Scheduled` objects created with these parameters’ preallocated exceptions, measured in number of `StackTraceElement` instances. The amount of space this requires is implementation-specific. The value 0 indicates that no stack trace will be stored.

*Returns*

reserved memory size in implementation-dependent stack frames.
getSizes

**Signature**

```java
class getSizes{
    public long[]
    getSizes()
}
```

**Description**

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

**Returns**

a copy of the array of implementation-specific sizes.

### 5.3.2.2 RealtimeThread

```java
class RealtimeThread{
    Inheritance
    java.lang.Object
    java.lang.Thread
        RealtimeThread

    Interfaces
    javax.realtime.BoundSchedulable
    javax.realtime.AsyncTimable

    Description
    Class `RealtimeThread` extends `Thread` and adds access to realtime services such as advanced scheduling, affinity management, asynchronous transfer of control, and access to scope memory.

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

    **5.3.2.1 Constructors**
```
RealtimeThread(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, ConfigurationParameters, TimeDispatcher, RealtimeThreadGroup, Runnable)

**Signature**

```java
public
    RealtimeThread(SchedulingParameters scheduling, ReleaseParameters release,
                   MemoryParameters memory, MemoryArea area,
                   ConfigurationParameters config, TimeDispatcher dispatcher,
                   RealtimeThreadGroup group,
                   Runnable logic)
```

**Description**

Creates a realtime thread with the given characteristics and a specified Runnable. The realtime thread group of the new thread is inherited from its parent task unless group is set. The newly-created realtime thread is associated with the scheduler in effect during execution of the constructor.

**Available since** RTSJ 2.0

**Parameters**

- **scheduling**—The SchedulingParameters associated with this (And possibly other instances of Schedulable). When scheduling is null and the creator is a schedulable, SchedulingParameters 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 associated with this (and possibly other instances of Schedulable). 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 associated with this (and possibly other instances of Schedulable). 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, and it may access the heap.
area—The initial memory area of this handler.

config—The ConfigurationParameters associated with this (and possibly other instances of Schedulable). When config 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 to use for realtime sleep and determining the period of a periodic thread.

group—The RealtimeThreadGroup of the newly created realtime thread or the parent’s realtime thread group when null.

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

IllegalRealtimeArgumentException—when the parameters are not compatible with the associated scheduler, the current thread group is not a RealtimeThreadGroup and group is null, or the affinity specified is not subsumed by the affinity of its closest threads RealtimeThreadGroup instance.

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 memory parameters, and any of the following is true:

- the initial memory area 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.

ScopedCycleException—when memory is a scoped memory area that has already been entered from a memory area other than the current scope.

RealtimeThread(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, ConfigurationParameters, Runnable)
5 Realtime Threads

RealtimeThread

**Signature**

```java
public RealtimeThread(SchedulingParameters scheduling,
                      ReleaseParameters release,
                      MemoryParameters memory,
                      MemoryArea area,
                      ConfigurationParameters config,
                      Runnable logic)
```  

**Description**

Creates a realtime thread with the given `SchedulingParameters`, `ReleaseParameters`, `MemoryParameters`, `ConfigurationParameters`, a specified `Runnable`, and default values for all other parameters.

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

**Available since RTSJ 2.0**

RealtimeThread(SchedulingParameters, ReleaseParameters, ConfigurationParameters, Runnable)

**Signature**

```java
public RealtimeThread(SchedulingParameters scheduling,
                      ReleaseParameters release,
                      ConfigurationParameters config,
                      Runnable logic)
```  

**Description**

Creates a realtime thread with the given `SchedulingParameters`, `ReleaseParameters`, `MemoryArea` and a specified `Runnable` and default values for all other parameters.

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

**Available since RTSJ 2.0**
RealtimeThread(SchedulingParameters, ReleaseParameters, ConfigurationParameters)

Signature
public
RealtimeThread(SchedulingParameters scheduling,
ReleaseParameters release,
ConfigurationParameters config)

Description
Creates a realtime thread with the given SchedulingParameters, ReleaseParameters and default values for all other parameters.

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

Available since RTSJ 2.0

RealtimeThread(SchedulingParameters, ReleaseParameters, Runnable)

Signature
public
RealtimeThread(SchedulingParameters scheduling,
ReleaseParameters release,
Runnable logic)

Description
Creates a realtime thread with the given SchedulingParameters, ReleaseParameters and a specified Runnable and default values for all other parameters.

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

Available since RTSJ 2.0

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

**Signature**

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

**Description**

Creates a realtime thread with the given `SchedulingParameters` and `ReleaseParameters` and default values for all other parameters. This constructor is equivalent to `RealtimeThread(scheduling, release, null, null, null, null, null, null)`.

RealtimeThread(SchedulingParameters scheduling, TimeDispatcher dispatcher)

**Signature**

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

**Description**

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

**Available since** RTSJ 2.0

RealtimeThread(SchedulingParameters scheduling)

**Signature**

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

**Description**

Creates a realtime thread with the given `SchedulingParameters` and default values for all other parameters.
RealtimeThread

**Signature**

```java
class RealtimeThread
```

**Description**

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

5.3.2.2.2 Methods

**currentRealtimeThread**

**Signature**

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

**Description**

Gets a reference to the current instance of `RealtimeThread`. Calling `currentRealtimeThread` is permissible when control is in an `AsyncEventHandler`. The method will return a reference to the `RealtimeThread` supporting that release of the async event handler.

**Throws**

- `ClassCastException`—when the current execution context is not an instance of `Schedulable`.

**Returns**

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

Signature

```java
public static javax.realtime.RealtimeThread
currentSchedulable()
throws ClassCastException
```

Description

Gets a reference to the current instance of `Scheduled`. It behaves the same when the current thread is an instance of `java.lang.Thread`, but otherwise it produces an instance of `AsyncBaseEventHandler`.

Throws

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

Returns

a reference to the current instance of `Scheduled`.

gGetCurrentReleaseTime

Signature

```java
public static javax.realtime.AbsoluteTime
gGetCurrentReleaseTime()
```

Description

Equivalent to `getLastReleaseTime(null)`

Available since RTSJ 2.0

gGetCurrentReleaseTime(AbsoluteTime)

Signature

```java
public static javax.realtime.AbsoluteTime
gGetCurrentReleaseTime(AbsoluteTime dest)
```

Description

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

Available since RTSJ 2.0

getCurrentConsumption(RelativeTime)

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

Description
Determines the CPU consumption for this release.

Throws
IllegalRealtimeStateException—when the caller is not a Schedulable.

Returns
when dest is null, returns the CPU consumption in a RelativeTime instance created in the current execution context. When dest is not null, returns the CPU consumption in dest

Available since RTSJ 2.0

getCurrentConsumption

Signature
public static javax.realtime.RelativeTime getCurrentConsumption()

Description
Equivalent to getCurrentConsumption(null).

Available since RTSJ 2.0

getCurrentMemoryArea

Signature

RTSJ 2.0 (Draft 66) 67
public static javax.realtime.MemoryArea
getCurrentMemoryArea()

**Description**

Gets a reference to the `MemoryArea` object representing the current allocation context. For a task that is not an instance of `Schedulable`, the result can only be heap or immortal memory.

**Returns**
a reference to the `MemoryArea` object representing the current allocation context.

### sleep(HighResolutionTime)

**Signature**

```java
public static void
sleep(HighResolutionTime<?> time)
throws InterruptedException,
ClassCastException,
IllegalRealtimeArgumentException
```

**Description**

A sleep method that is controlled by a generalized clock. Since the time is expressed as a `HighResolutionTime`, this method is an accurate timer with nanosecond granularity. The actual resolution available for the clock and even the quantity it measures depends on `clock`. The time base is the given `Clock`. The sleep time may be relative or absolute. When relative, then the calling thread is blocked for the amount of time given by `time`, and measured by `clock`. When absolute, then the calling thread is blocked until the indicated value is reached by `clock`. When the given absolute time is less than or equal to the current value of `clock`, the call to sleep returns immediately.

Calling `sleep` is permissible when control is in an `AsyncEventHandler`. The method causes the handler to sleep.

This method must not throw `IllegalAssignmentError`. It must tolerate `time` instances that may not be stored in `this`.

**Parameters**

- `time`—The amount of time to sleep or the point in time at which to awaken.

**Throws**

- `InterruptedException`—when the thread is interrupted by `interrupt()` or `AsynchronouslyInterruptedException.fire()` during the time between calling this method and returning from it.
ClassCastException—when the current execution context is not an instance of Schedulable.
IllegalRealtimeArgumentException—when time is null, when time is a relative time less than zero, or when the Chronograph of time is not a Clock.

`suspend(HighResolutionTime)`

*Signature*

```java
public static void suspend(HighResolutionTime<?> time)
    throws ClassCastException,
            IllegalRealtimeArgumentException
```

*Description*

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

*Parameters*

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

*Throws*

ClassCastException—when the current execution context is not an instance of Schedulable.
IllegalRealtimeArgumentException—when time is null, when time is a relative time less than zero, or when the Chronograph of time is not a Clock.

*Available since* RTSJ 2.0

`spin(HighResolutionTime)`

*Signature*

```java
public static void spin(HighResolutionTime<?> time)
    throws InterruptedException,
            ClassCastException,
            IllegalRealtimeArgumentException
```

*Description*

Similar to `sleep(HighResolutionTime)` except it performs a busy wait by polling on the Chronograph associated with time until time has been reached. Note that interaction with other tasks, scheduling considerations, and other effects

RTSJ 2.0 (Draft 66) 69
may reduce the frequency of polling for long delays, so an application cannot assume that the associated Chronograph will be polled as quickly as possible.

**Parameters**
- time—An absolute or relative time at which to stop spinning.

**Throws**
- InterruptedException—when the thread is interrupted by interrupt() or AsynchronouslyInterruptedException.fire() during the time between calling this method and returning from it.
- ClassCastException—when the current execution context is not an instance of Schedulable.
- IllegalRealtimeArgumentException—when time is null, or when time is a relative time less than zero.

**Available since** RTSJ 2.0

**spin(int)**

**Signature**
```java
public static void spin(int nanos)
throws InterruptedException,
        ClassCastException,
        IllegalRealtimeArgumentException
```

**Description**
The same as calling spin(HighResolutionTime) with a relative time to the default realtime clock, zero milliseconds, and nanos nanoseconds, except no relative time object is necessary.

**Parameters**
- nanos—A relative number of nanoseconds to wait.

**Throws**
- InterruptedException—when the thread is interrupted by interrupt() or AsynchronouslyInterruptedException.fire() during the time between calling this method and returning from it.
- ClassCastException—when the current execution context is not an instance of Schedulable.
- IllegalRealtimeArgumentException—when nanos is less than zero.
Available since RTSJ 2.0

**waitForNextRelease**

*Signature*

```java
public static boolean waitForNextRelease()
throws IllegalRealtimeStateException,
ClassCastException
```

*Description*

Causes the current realtime thread to delay until the next release. (See release().) Used by threads that have a reference to either periodic or aperiodic ReleaseParameters. The first release starts when this thread is released as a consequence of the action of one of the start() family of methods. Each time this method is called it 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.)

*Throws*

- IllegalRealtimeStateException—when this does not have a reference to a ReleaseParameters type of either PeriodicParameters or AperiodicParameters.
- ClassCastException—when the current thread is not an instance of RealtimeThread.

*Returns*

either false when the thread is in a deadline miss condition or true otherwise. When a deadline miss condition occurs is defined by its thread’s scheduler.

Available since RTSJ 2.0

**waitForNextReleaseInterruptible**

*Signature*

```java
public static boolean waitForNextReleaseInterruptible()
```

RTSJ 2.0 (Draft 66)
throws InterruptedException,  
IllegalRealtimeStateException,  
ClassCastException

Description
Same as `waitForNextRelease()` except it can throw an interrupted exception.

Throws
`AsynchronouslyInterruptedException`—when the thread is interrupted by `interrupt()` or `AsynchronouslyInterruptedException.fire()` during the time between calling this method and returning from it and the `ReleaseParameters.isRousable()` on its release parameters returns `true`. 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 timing behavior of the thread is unspecified until the state is reset by altering the thread’s release parameters or the thread is no longer in a deadline miss state.

`IllegalRealtimeStateException`—when `this` does not have a reference to a `ReleaseParameters` type of either `PeriodicParameters` or `AperiodicParameters`.

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

Returns
either `false` when the thread is in a deadline miss condition or `true` otherwise.

When a deadline miss condition occurs is defined by its thread’s scheduler.

Available since RTSJ 2.0

getMemoryArea

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

Description
Obtains 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.

Returns
a reference to the initial memory area for this thread.

Available since RTSJ 1.0.1

getMemoryParameters

Signature

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

Description

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

Returns

a reference to the current `MemoryParameters` object.

getRealtimeThreadGroup

Signature

```java
public javax.realtime.RealtimeThreadGroup
getRealtimeThreadGroup()
```

Description

Gets a reference to the `RealtimeThreadGroup` instance of this `Schedulable`. For instance that are not also instances of `RealtimeThread`, such as instance if `AsyncBaseEventHandler`, the group returned is the group that contains the thread or threads that are used to execute said `Schedulable` instance. For each instance of `Schedulable`, there can be only one directly associated instance of `Schedulable`.

Returns

a reference to the associated `RealtimeThreadGroup` object.

Available since RTSJ 2.0

getConfigurationParameters

Signature

```java
RTSJ 2.0 (Draft 66)
```
public javax.realtime.ConfigurationParameters
getConfigurationParameters()

Description
Gets a reference to the ConfigurationParameters object for this schedulable.

Returns
a reference to the associated ConfigurationParameters object.

Available since RTSJ 2.0

generateReleaseParameters

Signature
public javax.realtime.ReleaseParameters
getReleaseParameters()

Description
Gets a reference to the ReleaseParameters object for this schedulable.

Returns
a reference to the current ReleaseParameters object.

generateScheduler

Signature
public javax.realtime.Scheduler
getScheduler()

Description
Gets a reference to the Scheduler object for this schedulable.

Returns
a reference to the associated Scheduler object.

generateSchedulingParameters

Signature

74 RTSJ 2.0 (Draft 66)
public javax.realtime.SchedulingParameters
getSchedulingParameters()

Description

Gets a reference to the SchedulingParameters object for this schedulable.

Returns

A reference to the current SchedulingParameters object.

release

Signature

public void
release()

Description

Generates a release for this RealtimeThread. The action of this release is
governed by the scheduler. It may, for instance, act immediately, or be queued,
delayed, or discarded. This method does not suspend itself and has a runtime
complexity of $O(1)$.

Throws

IllegalRealtimeStateException—when this does not have a reference to a
ReleaseParameters type of AperiodicParameters.

Available since RTSJ 2.0

interrupt

Signature

public void
interrupt()

Description

In a system where RTSJModule.CONTROL is implemented, it makes the generic
AsynchronouslyInterruptedException pending for this, and sets the inter-
rupted state to true. As with Thread.interrupt(), blocking operations that
are interruptible are interrupted. When this.isRousable() is true a release
happens event if the first release has not yet happened. In any case, `AsynchronouslyInterruptedException` is thrown once a method is entered that implements `AsynchronouslyInterruptedException`.

Otherwise, it behaves as if `Thread.interrupt()` were called.

For instances of `AsyncBaseEventHandler`, it behaves as if `RealtimeThread.interrupt()` was called on the implementation thread underlying it.

Throws

`IllegalTaskStateException`—when this is not currently releasable, i.e., is disabled, not firable, its start method has not been called, or it has terminated.

Available since RTSJ 2.0

**isInterrupted**

*Signature*

```java
class
public boolean isInterrupted()
```

*Description*

Determines whether or not any `AsynchronouslyInterruptedException` is pending.

*Returns*

true when and only when the generic `AsynchronouslyInterruptedException` is pending.

Available since RTSJ 2.0

**deschedule**

*Signature*

```java
class
public void deschedule()
```

*Description*

Performs any deschedule actions specified by this thread’s scheduler, either immediately when in `waitForNextRelease()` or the next time the thread enters `waitForNextRelease()`.

Available since RTSJ 2.0
reschedule

Signature
public void reschedule() throws IllegalTaskStateException

Description
Gets the thread to the blocked-for-next-release state. This causes the next event to release the thread and waitForNextRelease to return. Deadline miss and cost enforcement are re-enabled.

The details of the interaction of this method with deschedule, waitForNextRelease and release are dictated by this thread’s scheduler.

Throws
IllegalTaskStateException—when the configured Scheduler and SchedulingParameters for this RealtimeThread are not compatible.

Available since RTSJ 2.0

startPeriodic(PhasingPolicy)

Signature
public void startPeriodic(PhasingPolicy phasingPolicy) throws LateStartException, IllegalTaskStateException, IllegalRealtimeArgumentException

Description
Starts the thread with the specified phasing policy.

Parameters
phasingPolicy—The phasing policy to be applied when the start time given in the realtime thread’s associated PeriodicParameters 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.
IllegalRealtimeArgumentException—when the thread is not periodic, or when its start time is not absolute.
5 Realtime Threads

RealtimeThread

IllegalTaskStateException—when the configured Scheduler and Scheduling-Parameters for this RealtimeThread are not compatible.

Available since RTSJ 2.0

start

Signature

```java
public void start()
throws IllegalRealtimeStateException
```

Description

Sets up the realtime thread’s environment and starts it. The set up might include delaying it until the assigned start time and initializing the thread’s memory area stack. (See ScopedMemory.)

Throws

IllegalRealtimeStateException—when the configured Scheduler and SchedulingParameters for this RealtimeThread are not compatible.

Available since RTSJ 2.0 adds new exception

getEffectiveStartTime

Signature

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

Description

Equivalent to getEffectiveStartTime(null).

Available since RTSJ 2.0

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

Description

Determines 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.

Returns

the effective start time in dest. When dest is null, returns the effective start time
in an AbsoluteTime instance created in the current memory area.

Available since RTSJ 2.0

getMinConsumption(RelativeTime)

Signature

public javax.realtime.RelativeTime
getMinConsumption(RelativeTime dest)

Description

Gets the minimum CPU consumption measured for any completed release of this
schedulable.

Throws

IllegalRealtimeStateException—when the caller is not a Schedulable.

Returns

the minimum CPU consumption in dest. When dest is null, it returns the
minimum CPU consumption in a RelativeTime instance created in the current
memory area.

Available since RTSJ 2.0
getMinConsumption

Signature

\[
\text{public javax.realtime.RelativeTime getMinConsumption()}
\]

Description
Equivalent to getMinConsumption(null).

Available since RTSJ 2.0

getMaxConsumption(RelativeTime)

Signature

\[
\text{public javax.realtime.RelativeTime getMaxConsumption(RelativeTime dest)}
\]

Description
Gets the maximum CPU consumption measured for any completed release of this schedulable.

Throws
IllegalRealtimeStateException—when the caller is not a Schedulable.

Returns
the maximum CPU consumption in dest. When dest is null, it returns the maximum CPU consumption in a \textbf{RelativeTime} instance created in the current memory area.

Available since RTSJ 2.0

getMaxConsumption

Signature

\[
\text{public javax.realtime.RelativeTime getMaxConsumption()}
\]

Description
Equivalent to getMaxConsumption(null).

Available since RTSJ 2.0
getDispatcher

Signature
   public javax.realtime.TimeDispatcher
getDispatcher()

Description
   Gets the dispatcher responsible for handling sleep requests issued by this thread

See Section Timable.getDispatcher()

Available since RTSJ 2.0

fire

Signature
   public final void
   fire()

Description
   Used by the Clock infrastructure to cause a call to waitForNextRelease to return.

See Section AsyncTimable.fire()

Available since RTSJ 2.0

mayUseHeap

Signature
   public boolean
   mayUseHeap()

Description
   Determines whether or not this schedulable may use the heap.

Returns
   true only when this Schedulable may allocate on the heap and may enter Heap-Memory.
Available since RTSJ 2.0

awaken

Signature

    public final void
    awaken()

Description

Used by the Clock infrastructure to cause a call to sleep to return.

See Section Schedulable.awaken()

Available since RTSJ 2.0

setMemoryParameters(MemoryParameters)

Signature

    public javax.realtime.RealtimeThread
    setMemoryParameters(MemoryParameters memory)

Description

Sets the memory parameters associated with this instance of Schedulable.

This change becomes effective at the next allocation; on multiprocessor
systems, there may be some delay due to synchronization between processors.

Parameters

memory—A MemoryParameters 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.)

Throws

IllegalRealtimeArgumentException—when memory is not compatible with the
schedulable’s scheduler. Also when this schedulable may not use the heap and
memory is located in heap memory.

IllegalAssignmentError—when the schedulable cannot hold a reference to mem-

ory, or when memory cannot hold a reference to this schedulable instance.

Returns

82

RTSJ 2.0 (Draft 66)
RealtimeThread

**Available since** RTSJ 2.0 returns itself

### setReleaseParameters(ReleaseParameters)

**Signature**

```java
class RealtimeThread
{
    public RealtimeThread setReleaseParameters(ReleaseParameters release);
}
```

**Description**

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.

**Parameters**

- `release`—A `ReleaseParameters` 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`.)

**Throws**

- `IllegalRealtimeArgumentException`—when `release` is not compatible with the associated scheduler. Also when this schedulable may not use the 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`.
- `IllegalTaskStateException`—when the task is running and the new release parameters are not compatible with the current scheduler.

**Returns**

- `this`

Available since RTSJ 2.0 returns itself

### setScheduler(Scheduler)

**Signature**

```java
setScheduler(Scheduler);
```
public javax.realtime.RealtimeThread
setScheduler(Scheduler scheduler)

Description
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. If the Schedulable is running, its associated SchedulingParameters (if any) must be compatible with scheduler.

For an instance of RealtimeThread, the Schedulable is running when
RealtimeThread.start() has been called on it and RealtimeThread.join() would block.

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

Throws
IllegalRealtimeArgumentException—when scheduler is null, or the schedulable’s existing parameter values are not compatible with scheduler. Also when this schedulable may not use the heap and scheduler is located in heap memory.

IllegalAssignmentError—when the schedulable cannot hold a reference to scheduler or the current Schedulable is running and its associated SchedulingParameters are incompatible with scheduler.

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

IllegalTaskStateException—when scheduler has scheduling or release parameters that are not compatible with the new scheduler and this schedulable is running.

Returns
this

Available since RTSJ 2.0 returns itself

setScheduler(Scheduler, SchedulingParameters, ReleaseParameters, MemoryParameters)

Signature
public javax.realtime.RealtimeThread
setScheduler(Scheduler scheduler,
SchedulingParameters scheduling,
ReleaseParameters release,
MemoryParameters memoryParameters)

Description

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.

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 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.)

release—A reference to the ReleaseParameters 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.)

memoryParameters—A reference to the MemoryParameters 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.)

Throws

IllegalRealtimeArgumentException—when scheduler is null or the parameter values are not compatible with scheduler. Also thrown when this schedulable may not use the 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.

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

Returns

this

Available since RTSJ 2.0

setSchedulingParameters(SchedulingParameters)

Signature

RTSJ 2.0 (Draft 66)
public javax.realtime.RealtimeThread
setSchedulingParameters(SchedulingParameters scheduling)

Description
Sets the scheduling 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. See the documentation for the scheduler for details.

Parameters
scheduling—A reference to the SchedulingParameters 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.)

Throws
IllegalRealtimeArgumentException—when scheduling is not compatible with the associated scheduler. Also when this schedulable may not use the 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.
IllegalTaskStateException—when the task is active and the new scheduling parameters are not compatible with the current scheduler or the intersection of affinity in scheduling and the affinity of this object’s realtime thread group does not correspond to a valid affinity.

Returns
this

Available since RTSJ 2.0 returns itself

5.4 Rationale
Realtime programming requires a scheduling method radically different than what a conventional Java programmer would expect, but most other aspects of thread behavior are the same. Therefore, it is reasonable to model a realtime thread as an extension to a java.lang.Thread. The main additions needed are for scheduling control such as release control for asynchronous event handling. Here asynchronous includes periodic releases, since release is asynchronous with regards to the executing code.
The RTSJ platform’s priority-preemptive dispatching model is very similar to the dispatching model found in the majority of commercial real-time operating systems. The `ReleaseParameters` and `MemoryParameters` provided to the `RealtimeThread` constructor provide a number of common real-time thread types, including periodic threads. However, conventional Java thread scheduling is supported. The real-time priorities are all above the conventional Java priorities to ensure the real-time threads take precedence over normal tasks.

The `MemoryParameters` class is provided with a may-use-heap option in order to enable time-critical schedulables to execute in preference to the garbage collector given appropriate assignment of execution eligibility when `false`. The memory access and assignment semantics of these heapless schedulables are designed to guarantee that the execution of such threads does not lead to an inconsistent heap state.
Chapter 6
Scheduling

Scheduling is a key differentiator between a conventional Java implementation and a realtime Java implementation. Whereas conventional Java implementations rely on some sort of fair scheduling, a realtime Java implementation must provide a realtime scheduler. In a realtime scheduler, ensuring that critical tasks finish on time is more important than overall throughput or fairness.

The scheduler required by this specification is fixed-priority preemptive with at least 28 unique priority levels. At least 28 must be supported by each implementation, but a deployment need not have all 28 active, when not needed by the application. It is represented by the class FirstInFirstOutScheduler, a subclass of PriorityScheduler, and is called the base scheduler. As the name implies, this scheduler does not time-slice threads at a given priority, but rather runs each to completion, so long as no higher priority thread becomes ready to run and no other processor is available for the higher priority thread. In that case, the current thread is preempted by the higher priority thread.

The schedulables required by this specification are denoted by the Schedulable interface and include the classes RealtimeThread and AsyncBaseEventHandler along with its subclasses. The base scheduler assigns processor resources according to the schedulables’ release characteristics, execution eligibility, and processing group values. Subclasses of these 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, including affinity. 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 waitForNextRelease 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.

The ThreadGroup class has special significance in an RTSJ implementation. As in conventional Java, the maximum priority of a thread is governed in part by its thread group, but the CPU affinity of a thread is also governed by its thread group along with the Affinity class. Furthermore, there are two important subclasses: RealtimeThreadGroup and ProcessingGroup. These classes provide additional means of managing tasks.

An instance of the RealtimeThreadGroup provides scheduling constraints for schedulables similar to how a ThreadGroup does for conventional Java threads. The scheduler and maximum SchedulingParameters can be set. A schedulable can only be created in an instance of RealtimeThreadGroup or its subclass. Therefore the root thread group and the thread group of the initial thread must both be realtime thread groups in an RTSJ implementation.

The ProcessingGroup class is a subclass of RealtimeThreadGroup. An instance of the ProcessingGroup class contains values that define a temporal scope for a processing group. When a schedulable has an associated instance of the ProcessingGroup class, it is said to execute within the temporal scope defined by that instance. A single instance of the ProcessingGroup class can be, and typically is, associated with many schedulables. In an implementation that supports cost enforcement, the combined processor demand of all of the schedulables associated with an instance of the ProcessingGroup 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

- enable the definition of schedulables,
- manage the assignment of execution eligibility to schedulable objects,
- manage the execution of instances of the AsyncBaseEventHandler 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.

6.1 Definitions

Task — 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 schedulers. Instances of these classes are referred to as Schedulables (SO) and provide four principle execution states: executing, eligible-for-execution, blocked, and descheduled.

1. Executing refers to the state where the schedulable is currently running on a processor.
2. 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.
3. Eligible-for-execution refers to the state where the schedulable could be selected to have its state changed to executing.
4. Descheduled refers to the state where the schedulable is ineligible to be released.

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 — The changing of the state of a schedulable from blocked-for-release-event to eligible-for-execution. When the state of a schedulable is blocked-for-release-event and 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.

Completion — 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 — A time before which a schedulable should complete. The $i^{th}$ deadline is associated with the $i^{th}$ release event and a deadline miss occurs when the $i^{th}$ completion would occur after the $i^{th}$ deadline.

Deadline Monitoring — The process by which the implementation responds to deadline misses. When 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** — 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:

1. 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;
2. a schedulable that is not periodic is said to be aperiodic; and
3. 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).

**Cost** — The maximum amount of CPU time that a schedulable is allowed between a release and its associated completion.

**Current CPU Consumption** — The amount of CPU time that the schedulable has consumed since its last release.

**Cost Overrun** — The time at which a schedulable’s current CPU consumption becomes greater than, or equal to, its cost.

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

**Cost Enforcement** — 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.)

**Base Priority** — The priority assigned to a task, either in its associated PriorityParameters object or by Thread.setPriority; the base priority of a Java thread is the priority returned by its getPriority method.

**Enforced Priority** — A priority below the idle priority, which ensures the schedulable has no execution eligibility.

**Active Priority** — The execution eligibility criterion for the priority-based schedulers. It is the maximum of the base (or enforced priority) and any priority a task has acquired due to the action of priority inversion avoidance algorithms (see the Synchronization Chapter).

**Processing Group** — A collection of tasks whose combined execution has further execution time constraints which the scheduler uses to govern the group’s
Semantics 6.2

execution eligibility.

**Base Scheduler** — An instance of the `FirstInFirstOutScheduler` class as defined in this specification. This is the initial default scheduler.

**Round-Robin Scheduler** — An instance of the `RoundRobinScheduler` class as defined in this specification. It is specified to execute in tandem with the base scheduler in a predictable fashion.

**Processor** — 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.

**Affinity** — A set of processors, where a valid affinity is a set on which the global scheduling of a schedulable can be supported.

**Idle Task** — A notional system or VM-provided task that consumes all CPU time not used by other tasks. It may be an actual process or thread, or it may be a power-saving mode that halts or slows the CPU, or it may be an artificial construction. For the purposes of this specification, it has a priority below that of all nonblocked tasks and above that of tasks blocked due to cost overrun. Details of its implementation are not specified here.

### 6.2 Semantics

Scheduling semantics determines when each task runs. Both *The Java Virtual Machine Specification* [6] and *The Java Language Specification* [5] are silent on the semantics for scheduling; only the semantics for synchronization is provided. Since scheduling is central to realtime programming, its detailed semantics, applicable across all available scheduler algorithms, is defined below, along with definitions of the required scheduling algorithms. Semantics that apply to particular classes, constructors, methods, and fields can be found in the class description and the constructor, method, and field detail sections.

#### 6.2.1 Schedulers

There are four basic requirements for schedulers.

1. A scheduler may only change the execution eligibility of the schedulables which it manages and only in accordance with its scheduling algorithm.
2. Each scheduler provided for application code by an RTSJ implementation must have documentation describing its semantics including at least the following: the algorithm used to determine eligibility, what schedulables may be scheduled by it, the subclasses of `Scheduler` and `SchedulingParameters` used to control
the scheduler, and any other classes needed by the scheduler.
3. Every implementation must provide a round-robin scheduler and a first in first out scheduler using priorities above the ten (1–10) conventional Java priorities as documented below.
4. Tasks with a conventional Java priority (1–10) must be scheduled such that when two or more threads run at the same priority, one thread cannot block another indefinitely or violate the requirements dictated by java.lang.Thread.
5. Tasks with a conventional Java priority must be scheduled using some sort of fair scheduler such that higher-priority Java tasks cannot starve lower-priority Java tasks indefinitely.

6.2.1.1 Affinity
For systems that support more than a single processor, one often needs to control what may run on each processor. The Affinity class provides mean of expressing sets of these processors. Affinity has three uses: to inform a scheduler on what processors a task may run, to show what subsets of processors on a system a scheduler may use, and to limit what processors a set of tasks may use. To this end, valid affinities define a distinguished subset of all possible affinities. This set is used to show on what subsets of processors a task may run and only members of this set may be used in the SchedulingParameters given to a task. Others sets may also be used for limited general processor availability.

6.2.1.2 Parameter Values
A 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 all schedulers are as follows, unless otherwise stated.
   (a) Scheduling parameters are copied from the creating schedulable when possible; when the creating schedulable does not have scheduling parameters, the default is an instance of the default parameters for the prevailing scheduler when the schedulable starts.
   (b) The default for release parameters depend on the type of schedulable:
      i. for instance of RealtimeThread, the default is an instance of BackgroundParameters with default values (see AperiodicParameters), and
      ii. for instance of AsyncBaseEventHandler the default is an instance of aperiodic parameters with default values (see AperiodicParameters).
2. Memory parameters default to `null` which signifies that memory allocation by the schedulable is not constrained by the scheduler.

3. The default scheduling parameters are scheduler dependent.

4. All numeric or `RelativeTime` attributes in parameter values must be greater than or equal to zero.

5. Values of period must be greater than zero.

6. Changes to scheduling, release, memory, and processing group parameters, either by methods on the schedulables bound to the parameters or by altering the parameter objects themselves, 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.

7. When changes to a parameter type—scheduling, release, memory, and processing group—take effect depends on the parameter type.

   (a) Changes to scheduling parameters take effect according to rules defined by the associated `Scheduler`.

   (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: when the $i_{th}$ release event occurred at a time $t_i$, then the $t^{th}$ 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 $t_i$. When 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 a Realtime Thread” below.

      iv. Changes to the additional values in `ReleaseParameters` objects and `SporadicParameters` are described, respectively, in “General 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.

   (f) Changes to cost enforcement state, i.e., enabling or disabling cost enforcement on a processing group or release parameters object associated with
one or more schedulables, take effect at the next release of the associated 
ProcessingGroup or associated Schedulable, respectively.

6.2.1.3 Release Control

Schedulables are released in response to the occurrence of events, such as starting 
a realtime thread, calling the release method of a realtime thread, or firing the 
asynchronous event associated with 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 only difference between a 
periodic and an aperiodic event is the regularity of the arrival times.

A scheduler behaves effectively as if it maintained a queue, called the arrival time 
queue, for each 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. When 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. The 
enumeration class QueueOverflowPolicy defines four overflow policies.

<table>
<thead>
<tr>
<th>Policy</th>
<th>Action on Overflow</th>
</tr>
</thead>
<tbody>
<tr>
<td>IGNORE</td>
<td>Silently ignore the arrival. The arrival is not accepted, no release event occurs, and, when 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.</td>
</tr>
<tr>
<td>EXCEPT</td>
<td>Throw an ArrivalTimeQueueOverflowException. The arrival is not accepted, and no release event occurs, but when the arrival was caused programmatically, the caller will have ArrivalTimeQueueOverflowException thrown.</td>
</tr>
</tbody>
</table>
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.

### 6.2.1.3.1 Sporadic Release Control

“Sporadic Release Control” is a special case of “Release Control,” where the arrival time or execution time may be additionally regulated. Sporadic parameters include a minimum interarrival time (MIT) which characterizes the expected frequency of releases. When an arrival is accepted, the 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.

Two mechanisms are specified for enforcing this rule: *arrival-Time regulation* and *release-time regulation*. Arrival-time regulation controls the work-load by considering the time between arrivals. When 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. Release-time regulation controls when events are released. Under this policy all arrivals that can be queued under the current QueueOverflowPolicy are accepted, but the scheduler behaves effectively as if released schedulables were further constrained...
by a scheduling policy that restricts releases to at most one release per MIT. As described in the following tables, three types of arrival-time regulation and one type of release-time regulation are supported.

### Arrival-Time Regulation

<table>
<thead>
<tr>
<th>Policy</th>
<th>Action on Violation</th>
</tr>
</thead>
<tbody>
<tr>
<td>IGNORE</td>
<td>Silently ignore the violating arrival. The arrival is not accepted, no release event occurs, and, when the arrival was caused programmatically (such as by invoking <em>fire</em> on an asynchronous event), the caller is not informed that the arrival has been ignored.</td>
</tr>
<tr>
<td>EXCEPT</td>
<td>Throw a <em>MITViolationException</em>. The arrival is not accepted, and no release event occurs, but when the arrival was caused programmatically, the caller will have <em>MITViolationException</em> thrown.</td>
</tr>
<tr>
<td>REPLACE</td>
<td>The arrival is not accepted and no release event occurs. When 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. When the completion associated with the last release event has occurred, or the deadline has already been missed, the behavior of the REPLACE policy is equivalent to the IGNORE policy.</td>
</tr>
</tbody>
</table>

### Release-Time Regulation

<table>
<thead>
<tr>
<th>Policy</th>
<th>Action on Violation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAVE</td>
<td>The arrival time is delayed until after the current MIT interval. 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 might not schedule the effective release of an asynchronous 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 asynchronous event has not yet reached its effective release. Once an arrival is queued, 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.</td>
</tr>
</tbody>
</table>

---

98 RTSJ 2.0 (Draft 66)
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.

### 6.2.1.3.2 Releasing a Realtime Thread

The repeated release of a realtime thread is achieved by executing in a loop and invoking the `RealtimeThread.waitForNextRelease` methods, or its interruptible equivalent `RealtimeThread.waitForNextReleaseInterruptible`) within that loop. For simplicity, unless otherwise stated, the semantics in this section apply to both forms of this method.

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 `waitForNextRelease`, `schedule`, and `deschedule`;
   (c) the occurrence of deadline misses and whether or not a miss handler is installed; and
   (d) whether the passing of time generates periodic release events or calls to the `release` method generates aperiodic release events.

2. The initial release event depends on the type of release parameters given the realtime thread:
   (a) for a realtime thread with periodic parameters, the initial release event 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`;
   (b) For a realtime thread with aperiodic parameters, the initial release event occurs immediately in response to the invocation of its `start` method.

---

1The method `RealtimeThread.waitForNextPeriod` has been replaced by `RealtimeThread.waitForNextRelease` as of RTSJ 2.0. The same goes for its interruptible equivalent.
3. Changes to the start time in a realtime thread’s `PeriodicParameters` object only have an effect on its initial release time. Consequently, when 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. When subsequent release events occur also depends on the type of release parameters given to the realtime thread:
   (a) for periodic realtime threads, each period (and hence each release) falls due, except as described below (in 6d), at regular intervals such that when the $i^{th}$ release event occurred at a time $t_i$, 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 threads, a release occurs with each call of the release method, except as described below (in 6d); and
   (c) for sporadic realtime threads, a release occurs with each call of the release method, except, as described below (in 6d), when additional regulation is required to enforce MIT as defined in “Sporadic Release Control” below.

5. Each release of an aperiodic realtime thread is an arrival.
   (a) When the thread has release parameters of type `ReleaseParameters`, then the arrival may become a release event for the thread according to the semantics given in “General Release Control” below.
   (b) When 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.

6. The implementation should behave effectively as if the following state variables were added to a realtime thread’s state,
   boolean `deschedule`,
   integer `pendingReleases`,
   integer `missCount`, and
   boolean `lastReturn`;
and manipulated by the actions as described below.
   (a) Initially
      ```
      deschedule = false,
      pendingReleases = 0,
      missCount = 0, and
      lastReturn = true.
      ```
   (b) The function of the `deschedule` method depends on the current state of the realtime thread.
      i. When current state is a blocked state, either blocked-for-release-event or blocked-for-missed-release, it sets the value of `deschedule` to `true` and sets the thread’s state to descheduled.
ii. When the current state is not a blocked state, it just sets the value of \texttt{deschedule} to \texttt{true}.

(c) The function of the \texttt{reschedule} method also depends on the current state of the realtime thread.

i. When the realtime thread is in the descheduled state, it sets the value of \texttt{deschedule} to \texttt{false}, sets the values of \texttt{pendingReleases} and \texttt{missCount} to zero, changes the thread’s state to blocked-for-release-event, and tells the cost monitoring and enforcement system to reset for this thread.

ii. When the realtime thread is \textit{not} in the Descheduled state, it just sets the value of \texttt{deschedule} to \texttt{false}.

(d) A realtime thread that is in the descheduled state will not receive any further release events until after it has been rescheduled by a call to \texttt{reschedule}; this means that no deadline misses can occur.

(e) What happens when a release event occurs depends on the current state.

i. When the state of the realtime thread is descheduled, do nothing.

ii. When the state is blocked-for-release-event, i.e., it is waiting in \texttt{wait-ForNextRelease}, increment the value of \texttt{pendingReleases}, inform cost monitoring and enforcement that the next release event has occurred, and notify the thread to make it eligible for execution;

iii. Otherwise, when the thread is in a release, increment the value of \texttt{pendingReleases}, and inform cost monitoring and enforcement that the next release event has occurred.

(f) On each deadline miss, one of two things happen:

i. when the realtime thread has a deadline miss handler, the value of \texttt{deschedule} is set to \texttt{true}, the handler is atomically released with its \texttt{fireCount} increased by the value of \texttt{missCount} + 1, and zero for \texttt{missCount};

ii. otherwise, one is added to the \texttt{missCount} value.

(g) When the \texttt{waitForNextRelease} method is invoked by the current realtime thread there are three possible behaviors depending on the value of \texttt{missCount} and \texttt{lastReturn}.

i. When \texttt{missCount} is zero, any pending parameter changes are applied, cost monitoring and enforcement are informed of completion, and then the thread waits while \texttt{deschedule} is \texttt{true}, or \texttt{pendingReleases} is zero. Then the \texttt{lastReturn} value is set to \texttt{true}, \texttt{pendingReleases} is decremented, and \texttt{true} is returned.

ii. When \texttt{missCount} is greater than zero and the \texttt{lastReturn} value is \texttt{false}, completion occurs: the \texttt{missCount} value is decremented; then any pending parameter changes are applied, \texttt{pendingReleases} is
decremented, cost monitoring and enforcement is informed that the realtime thread has completed, and false is returned;

iii. Otherwise, when missCount is greater than zero and the lastReturn value is true, the missCount value is decremented and the lastReturn value is set to false and false is returned.

7. An invocation of the `RealtimeThread.waitForNextRelease` method with release parameters, where `ReleaseParameters.isRousable` returns true, behaves as described above with the following differences.

(a) When the invocation commences with 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`. When this occurs, the most recent release has not completed. When the pending instance is the generic AIE instance, then the interrupt state of the realtime thread is cleared.

(b) What happens when an instance of AIE becomes pending on a realtime thread is dependent on the state of the thread.

i. When the thread is descheduled, 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 7c where the time value used as t\text{int} is the time at which the schedulable was rescheduled.

ii. When it is blocked-for-release-event, then this acts as a release event. Execution then continues as in 7c, where the time value used as t\text{int} is the time at which the AIE becomes pending.

(c) 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`. When the pending instance is the generic AIE instance, the interrupt state of the realtime thread is cleared.

iii. The deadline associated with this release is the time t\text{int}+D\text{int}, where D\text{int} is the value of the deadline stored in the realtime thread’s release parameters object at the time t\text{int}.

iv. The next release time for the realtime thread will be t\text{int}+T\text{int}, where T\text{int} is the value of the period stored in the realtime thread’s release parameters object at the time t\text{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 RESTART 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, and
(d) any outstanding deadline misses are cleared.

9. The effect of the change on the thread falls into one of four main cases.
   (a) When the realtime thread is not waiting for the next release event and is not descheduled,
      i. there is no effect until the end of current release, and
      ii. when the change occurs, it is a pseudo restart of the thread, i.e., when the new parameters are aperiodic, the release is immediate and when the parameters are periodic, the periodic start time algorithm is used.
   (b) When the realtime thread is not waiting for the next release event, but there is an outstanding deschedule,
      i. there is an immediate "schedule" of the thread,
      ii. there is no further effect until end of current release, and
      iii. when change occurs, it is a pseudo restart of the thread, i.e., when the new parameters are aperiodic, the release is immediate, and when the new parameters are periodic, the periodic start time algorithm is used.
   (c) When the realtime thread state is blocked-for-release-event, i.e., it is waiting in `waitForNextRelease`, and the release parameter type is changed,
      i. from Periodic to Aperiodic, at the next periodic release event occurs, the thread becomes aperiodic with an immediate release, or
      ii. from Aperiodic to Periodic, there is an immediate pseudo restart of the thread using the periodic start time algorithm.
   (d) When the realtime thread state is descheduled and the of release parameters is changed,
      i. the change is from Periodic to Aperiodic, there is an immediate "schedule" of the thread, and when the next periodic release event occurs, the thread becomes aperiodic with an immediate release, or
      ii. the change is from Aperiodic to Periodic, there is an immediate "schedule" of the thread and there is an immediate pseudo restart of the thread using the periodic start time algorithm.

6.2.1.3.3 UML Diagrams for Realtime Thread Releases

The three UML diagrams in Figures 6.1, 6.2, and 6.3, are provided to illustrate the foregoing rules for releasing realtime threads. The first two figures are for a thread without a deadline miss handler. The first is a UML sequence diagram of some examples of Realtime Thread releases. The second is a UML state chart of the
Figure 6.1: Sequence Diagram of Some Example Realtime Thread Releases
Figure 6.2: A State Chart for a Realtime Thread without a Deadline Miss Handler

```plaintext
Initial

PendingReleases == 0
missCount == 0
deschedule == false

start()

Blocked for Normal Release

missCount == 0
* release
  increments pendingReleases
* deschedule()
  sets deschedule

 endowedNonRealtime()
 called
 [missCount == 0]
 [deschedule == false]
 pendinReleases > 0

[deschedule == true]
 pendingReleases > 0
missCount == 0

Blocked for Missed Release

missCount > 0
* release
  increments pendingReleases
* deschedule()
  sets deschedule
* reschedule()
  clears deschedule

Normal Release

* release
  increments pendingReleases
* deadline miss
  increments missCount
* deschedule()
  sets deschedule
* reschedule()
  clears deschedule

Missed Release

missCount == 0
* release
  increments pendingReleases
* deadline miss
  increments missCount
* deschedule()
  sets deschedule
* reschedule()
  clears deschedule

Descheduled

deschedule == true
in endowedNonRealtime()
* reschedule()
  clears deschedule

Handle Miss

* release
  increments pendingReleases
* deadline miss
  increments missCount
* deschedule()
  sets deschedule
* reschedule()
  clears deschedule
```

RTSJ 2.0 (Draft 66) 105
release process for a realtime thread. The third is a UML state chart of the release process for a realtime thread with a deadline miss handler.

In Figure 6.1, a yellow background marks the execution of a normal release, an orange background marks the execution of a miss handler, and a red background marks the execution of a missed release. Both the miss handler and all missed releases are eligible to run as soon as the previous release is finished. A normal release, which encounters a deadline miss during its execution, is not complete until its miss handler completes.

In the other two figures, a yellow background marks releases and a pink background marks blocked states. There are three release states: normal release, miss handler, and missed release. They can only be left by a call to `waitForNextRelease` or its
equivalent. The miss handler state is part of a normal release that misses its deadline during the release. There are two blocked-for-release-event states: blocked for normal release and blocked for missed release. It is only in these states that descheduling can occur, because only completion occurs upon their entry. In addition, the blocked for missed release is a ephemeral state, since the deadline miss has already occurred before the state is entered, so state is left immediately. It is there to enable all actions that occur on completion.

6.2.1.3.4 Releasing an 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. Unless the handler has release parameters of type SporadicParameters, the arrival becomes a release event for the handler in strict accordance with the semantics given in “General Release Control” above. When the handler has release parameters of type SporadicParameters, the arrival becomes a release event for the handler in strict accordance with the semantics given in “Sporadic Release Control” above.

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 (when 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; and
   (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.
   (a) The getAndDecrementPendingFireCount method decreases the fireCount by one (when it is 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 that
the effective release time honors any restrictions imposed by the MIT violation
policy, when 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`, when the `fireCount`
is now zero, the cost monitoring and enforcement system is told to reset for
this handler.

9. The value of `ReleaseParameters.isRousable` controls whether a call to
`Scheduleable.interrupt` causes a premature release or only affects a running
schedulable.

(a) When interrupt is called on an instance of `Scheduleable` and the schedul-
able is running, the interrupt is made pending and as soon as AI code is
entered, an AIE is thrown.

(b) Depending on the value of the isRousable property, start will prematurely
complete, i.e., start user code, or simply wait for the start time to occur.

(c) Depending on the value of the isRousable property, the next release of
a firable handler, i.e., an enabled instance of `AsyncBaseEventHandler`
which is attached to an instance of `AsyncBaseEvent`, will occur immedi-
ately or not, but in both cases an AIE will be pending until the next AI
method.

6.2.1.4 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 scheduler eligibility 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 in terms of task priority
for priority schedulers, but other schedulers should act similarly with respect to their
own scheduler eligibility levels.

1. A `Scheduleable` can become an executing schedulable only when it is eligible
for execution and one of the processors in its associated affinity is available.

2. When 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 when it is in a ready queue, or when 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. 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.

9. In a multiprocessor system, a schedulable can be on the ready queues of more than one processor. At the extreme, when 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.

10. The dispatching mechanism must enable 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.

11. 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.

12. A realtime thread that performs a `yield()` is placed at the tail of the queue (dictated by its affinity) for its active priority level.

13. 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.

14. A schedulable whose active priority is raised as a result of explicitly setting its base priority (through the `RealtimeThread.setSchedulingParameters()` method, or `Thread.setPriority()` method) is added to the tail of the queues (dictated by its affinity) for its new priority level.

15. Queuing when priorities are adjusted by priority inversion avoidance algorithms is governed by semantics specified in the *Synchronization* chapter.

### 6.2.1.5 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, its 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.

2. The implementation is required to document the granularity at which the current CPU consumption is updated.

3. When a cost overrun is triggered, the cost overrun handler associated with the schedulable, if any, is released. No further action is taken.

4. 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).

When 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.

(a) When the most recent release of the schedulable is the $i^{th}$ release, and the $i + 1$ release event has not yet occurred, the following must hold.

i. When the state of the schedulable is either executing or eligible-for-execution, 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.

ii. 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; when 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.

(b) When the most recent release of the schedulable is the $i^{th}$ release, and 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 be the $i + 1$ release.

2. When the $i^{th}$ release event occurs for a schedulable, the action taken depends on the state of the schedulable.

(a) When 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) When the schedulable is blocked for a reason other than blocked-by-cost-overrun then

i. when 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) When the schedulable is not blocked, no cost monitoring action occurs.

3. 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.
4. Changes to the cost parameter take effect immediately.
   (a) When 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) When the new cost is greater than the current CPU consumption,
       i. in the case that the schedulable is blocked-by-cost-overrun, the schedulable is made eligible for execution;
       ii. in the case that 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, the pending state change is removed;
       iii. in all other cases, no cost monitoring action occurs.

5. When a schedulable changes state to blocked-by-cost-overrun, it must behave as if its base priority has been reduced to the enforced priority. In other words, unless its active priority has been modified by a priority inversion avoidance algorithm as defined in this specification, it should not be scheduled on any CPU. Upon moving out of this state, it will resume execution as if its base priority had been restored to its configured base priority.

6. The state of the cost monitoring system for a schedulable can be reset by the scheduler (see 6.2.1.3.2 in the Release of a Realtime Thread section, below). When 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 \( n > m \)), 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.

### 6.2.2 Priority Schedulers

This specification defines a class of scheduler that are priority preemptive. Their semantics assumes a uniprocessor or shared memory multiprocessor execution environment. Two subclasses are defined: the base scheduler and a round-robin scheduler.

The semantics for the base scheduler is priority preemptive with run to completion semantics, also known as first-in-first-out (FIFO) semantics: \texttt{FirstIn-FirstOutScheduler}. The base scheduler supports the execution of all schedulables. When a schedulable managed by the base scheduler is scheduled, it will run either until it blocks (as on a monitor or for some I/O operation), voluntarily relinquishes the CPU (as for sleep), or is preempted by a higher priority task.

The round-robin scheduler is a fixed-quantum, fixed-priority, priority-preemptive scheduler that interacts predictably with the base scheduler: \texttt{RoundRobinScheduler}. The time at which a quantum expires may be calculated either from the last task switch or on a heartbeat. It uses the \texttt{PriorityParameters} class for the configuration of schedulable priorities. It may not be present on all systems, but if it is present
then it will obey the semantics specified here. When a schedulable managed by the round-robin scheduler is scheduled, it will run no longer than until it blocks (as on a monitor or for some I/O operation), it voluntarily relinquishes the CPU (as for sleep), or it is preempted by a higher priority task, as with the base scheduler, but also yields when its quantum has expired.

The scheduler is not responsible for ensuring that a release, such as an event handler, will complete within the quantum. A release which would run longer than its quantum will be rescheduled at the end of that quantum, when another task with the same priority is ready to run, even if it has not completed. When this is not the desired behavior, the FirstInFirstOutScheduler should be used instead.

Both schedulers share the same base class: PriorityScheduler.

### 6.2.2.1 Priorities

Not only the presence or absence of a time quantum, but also the semantics for scheduling eligibility differ between the base (FIFO) and round-robin schedulers. Both schedulers use a numerical priority value to determine scheduling eligibility. A higher value means a higher scheduler eligibility and a lower one means a lower scheduler eligibility. Although the values themselves have the same relative meaning between the two schedulers, the details of their semantics vary.

### 6.2.2.2 First-In-First-Out-Scheduler

The base scheduler is a priority scheduler with the following requirements.

1. The base scheduler must able to support at least 28 distinct values\(^2\) (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.

2. 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.

3. Higher priority values in an instance of PriorityParameters have a higher execution eligibility.

4. The 10 priorities defined for java.lang.Thread must effectively have lower execution eligibility than the realtime priorities.

5. When the round-robin scheduler is present, the base scheduler must support at least one priority value numerically greater than the maximum allowable round-robin priority.

6. For realtime scheduling, the base priority of each Schedulable under the control of the base scheduler must be from the range of realtime priorities. A

\(^2\)A system may be configured to have fewer, when few are required.
Schedulable with a priority in the java.lang.Thread range will be scheduled as if it were an instance of java.lang.Thread.

7. 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.

8. The base scheduler does not use the importance value in the ImportanceParameters subclass of PriorityParameters.

9. Calling the java.lang.Thread.setPriority on a thread can only be used to set the thread’s priority to a conventional Java priority (1–10).

10. For schedulables managed by the base scheduler, the implementation must not change the execution eligibility for any reason other than
(a) the implementation of a priority inversion avoidance algorithm requires it, or
(b) 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.

11. Use of Thread.setPriority(), or any of the methods defined for schedulables, or any of the methods defined for parameter objects must not affect the correctness of the priority inversion avoidance algorithms controlled by PriorityCeilingEmulation and PriorityInheritance—see Chapter7.

12. When schedulable A, managed by the base scheduler, creates Java thread B, then the initial base priority of B is the minimum of the priority value returned by the getMaxPriority method of B’s java.lang.ThreadGroup object and the priority of A.

13. PriorityScheduler.getNormPriority() shall be set to

<table>
<thead>
<tr>
<th>Line</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>((PriorityScheduler.getMaxPriority() -</td>
</tr>
<tr>
<td>2</td>
<td>PriorityScheduler.getMinPriority()) / 3) +</td>
</tr>
<tr>
<td>3</td>
<td>PriorityScheduler.getMinPriority()</td>
</tr>
</tbody>
</table>

14. Hardware priorities, where supported, have values above the base scheduler’s priority range (see Section 13.2.4).

6.2.2.3 The Round-Robin Scheduler

Priorities in the round-robin scheduler are as in the base scheduler, and priority values are numerically equivalent between the two. Schedulables managed by the round-robin scheduler behave as if they are scheduled from the same FIFO queue as schedulables managed by the base scheduler of the same numeric priority, except
that they will consume no more than one quantum of execution time before being
moved to the tail of the queue. Implementations are permitted to use a single, shared
queue for this purpose.

If the round-robin scheduler is present, its priorities will have the same properties
as the base scheduler, except for the following.

1. The round-robin scheduler must support at least one priority, and may support
an arbitrarily large number of priorities.
2. All round-robin priorities must be greater than 10, and they must include
all integers from the round-robin scheduler’s `getMinPriority()` value to its
`getMaxPriority()` value, inclusive.
3. The round-robin scheduler does not use the `importance` value in the `Import-
tanceParameters` subclass of `PriorityParameters`.
4. `RoundRobinScheduler.getNormPriority()` shall be set to

   ```
   1 ((RoundRobinScheduler.getMaxPriority() - 
   2 RoundRobinScheduler.getMinPriority()) / 3) + 
   3 RoundRobinScheduler.getMinPriority()
   ```

The round-robin scheduler may provide priorities strictly lower than that of the
base scheduler or a set of priorities partially or entirely overlapping with the priorities
provided by the base scheduler.

### 6.2.2.4 Parameter Values

The parameter values used by the priority schedulers shall have the following scheduler-
specific properties.

1. The default scheduling parameter values for parameter objects created by a
schedulable controlled by the priority schedulers are given by the following
table (see `FirstInFirstOutScheduler`).

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Scheduling parameters</strong></td>
<td></td>
</tr>
<tr>
<td>affinity</td>
<td>inherited from creating task</td>
</tr>
<tr>
<td><strong>Priority parameters</strong></td>
<td></td>
</tr>
<tr>
<td>priority</td>
<td>norm priority</td>
</tr>
</tbody>
</table>

2. For a task changing its own `SchedulingParameters`, the change shall take
effect immediately.

3. For a task Changing another task’s `SchedulingParameters`, when the other
task is in the running state, the change shall take effect as soon as practical
and, when the other task is in any other execution state, the change will take
effect immediately.

RTSJ 2.0 (Draft 66) 115
6.2.3 Associating Schedulables with Schedulers

The Scheduler associated with a Schedulable at the time it is started is derived from its scheduler setting and scheduler parameters, as well as the scheduler of the task, instance of Thread or Schedulable, that started it. The start time of a RealtimeThread is the time at which its RealtimeThread.start() method is invoked, and the start time of an event handler is the time at which it is attached to an event with AsyncBaseEvent.addHandler().

For the following discussion, let $s_i$ be the instance of Schedulable being started, $parent$ be the task from which it is started, $ns$ be some arbitrary scheduler, and $sg$ be the RealtimeThreadGroup instance associated with $s_i$. The Scheduler for $s_i$ is determined as follows and in the order stated.

1. When Scheduler.setScheduler($ns$) has been used to explicitly configure a scheduler for $s_i$, that scheduler will be the scheduler associated with $s_i$.
2. When $parent$ is an instance of Schedulable and the scheduler associated with $parent$ is an instance of the class returned by $sg$.getScheduler(), then the scheduler associated with $s_i$ will be the scheduler associated with $parent$.
3. When $parent$ is not an instance of Schedulable (i.e., it is a Java Thread) but is currently scheduled with a realtime Scheduler and that scheduler is an instance of the class returned by $sg$.getScheduler(), then $s_i$ will use the scheduler currently associated with $parent$.
4. When the default scheduler is an instance of the class returned by $sg$.getScheduler(), then $s_i$ will use the default scheduler.
5. When none of these conditions hold, a scheduler cannot be determined for $s_i$ and an IllegalStateException will be thrown.

A task that is eligible for execution must always have a compatible Scheduler and SchedulingParameters. This means that appropriate configuration objects must be passed in at construction time, and that all later changes must be compatible; if both the Scheduler and SchedulingParameters must be changed in such a way that neither is compatible with the current configuration, setScheduler may be called on the Schedulable with both a scheduler and compatible parameters passed at the same time.

In order to change a task’s scheduler to one that is inconsistent with its SchedulingParameter or visa versa, it must be in the the descheduled state. This compatibility must be restored before the task becomes eligible for execution. For RealtimeThread, rescheduling will throw an IllegalTaskStateException when called on a Schedulable with scheduling parameters that are inconsistent with its scheduler. Likewise, since adding a handler to its first event makes it eligible to be fired and hence eligible to be executed, trying to add a handler with SchedulingParameters that do not match its scheduler to an event will also result in an IllegalTaskStateException being thrown.
6.2.4 Additional Schedulers

The RTSJ allows an implementation to support schedulers other than the FirstIn-FirstOut and RoundRobin schedulers. However, it does not define how these implementation-defined schedulers interact with the RTSJ schedulers. All the RTSJ requires is that, for each processor (core), it must be possible to determine the rules that govern which task will be executing on that processor at every instant in time. Furthermore, it requires (in the absence of any execution-eligibility avoidance technique) that this task be one of the most eligible tasks from each scheduler that is active on that processor. In all cases, the servicing of interrupts must be given higher execution eligibility than all schedulables.

6.2.5 Managing Groups of Schedulables

Conventional Java provides the class ThreadGroup to manage groups of threads. Only minimal functionality is provided: limiting priority, setting daemon status, and interrupting a group of threads at once. RTSJ extends this concept in two ways: limiting CPU affinity on an instance of ThreadGroup through the Affinity class and providing subclasses for managing Schedulables.

6.2.5.1 Realtime Thread Groups

The RealtimeThreadGroup subclass of ThreadGroup provides a means of constraining the possible scheduling parameters and scheduler of tasks. The setMaxPriority method on ThreadGroup only pertains to tasks scheduled in the conventional Java range (1–10), and not to tasks scheduled with a realtime scheduler. To ensure that this works and that conventional thread groups must not need to be scope aware, an implementation must enforce several restrictions:

1. only threads in a realtime thread group may use a realtime scheduler,
2. instances of RealtimeThread may only be created in a realtime thread group,
3. the root ThreadGroup instance must be an instance of RealtimeThreadGroup,
4. the ThreadGroup instance of the initial thread must be an instance of RealtimeThreadGroup,
5. an instance of RealtimeThreadGroup may not have a parent that is not an instance of RealtimeThreadGroup, and
6. all groups contained in a RealtimeThreadGroup allocated in a ScopedMemory must be instances of RealtimeThreadGroup.

Furthermore, the enumeration methods on a realtime thread group are not aware of scoped memory and the referential integrity restrictions discussed in Chapter 11, Alternative Memory Areas. The enumeration methods of RealtimeThreadGroup will not return references to any Thread that is also a RealtimeThread and to any
ThreadGroup that is also a RealtimeThreadGroup. This ensures that descendents allocated in a ScopedMemory will not need to be referenced from an enumeration object allocated in an incompatible allocation context. Since the enumeration object is allocated in the current allocation context, only conventional threads are guaranteed to be referencable. For processing RealtimeThreadGroup instances, a visitor should be used.

The maximum priority and scheduler restrictions on RealtimeThreadGroup and ThreadGroup apply only to the base priority of a task belonging to that group. Priority inversion avoidance algorithms (see Chapter 7, Synchronization) may cause a task to temporarily obtain a priority notionally higher than its maximum base priority as specified in its associated instance of ThreadGroup.

A maximum eligibility is provided for capping realtime priorities, but this can only be set once a scheduler type is set that limits the Scheduling-Parameters to a set of comparable types. Changing the maximum eligibility allowed to tasks in a RealtimeThreadGroup via the RealtimeThreadGroup.setMaxEligibility(SchedulingParameters) method takes effect immediately, and will do the following.

1. For any task \( t \) in the affected RealtimeThreadGroup that is associated with a SchedulingParameters not allowable under the new eligibility restriction, set the SchedulingParameters associated with \( t \) to the SchedulingParameters currently being set by setMaxEligibility().
2. For any RealtimeThreadGroup child \( sg \) of the affected RealtimeThreadGroup that has a maximum eligibility not allowed under the new eligibility restriction, set the maximum eligibility of \( sg \) to the SchedulingParameters currently being set by setMaxEligibility(). Note that this will recursively effect the tasks and RealtimeThreadGroup children in \( sg \).

### 6.2.5.2 Processing Groups

A processing group is defined by an instance of the ProcessingGroup, a subclass of RealtimeThreadGroup. Each schedulable that is bound to that object is called a member of that processing group. A processing group instance acts as a proxy for its members, but enforcement does have an effect on the execution of member threads. As a subclass of ThreadGroup and RealtimeThreadGroup, ProcessingGroup instances are members of the thread group hierarchy in the system. Since a RealtimeThreadGroup may have another RealtimeThreadGroup instance as its ancestor, a task might be in more than one realtime thread group, and hence can be in more than one processing group.

1. The deadline of a processing group is defined by the value returned by invoking the getDeadline method of the processing group object.
2. A deadline miss for the processing group is triggered when 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) when the processing group has a miss handler, it is released for execution,
   (b) otherwise, 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 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. Whenever, due to either execution of the members of the processing group or a change in the 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 underrun handler has been set, it is released at the end of any cost period, where the minimal cost has not been consumed by the tasks in the group.

8. When the affinity of the group contains more than one processor, the granularity enforced may be as large as the base granularity times the number of processors in the group’s affinity.

9. When a cost overrun is triggered, the cost overrun handler associated with the processing group, if any, is released.

10. When more than one processing group monitoring a given task or set of tasks reach their limits at the same time, all corresponding handlers are released in an unspecified order.

11. Any group entering enforcement between a given group and the root enforces that group.

12. When cost enforcement is supported, enabled, and triggered, the processing group enters the enforced state. For each member of the processing group:
   (a) the schedulable is placed into the enforced state; and
   (b) when a schedulable is in the enforced state, the base scheduler schedules that schedulable effectively as if it has a base priority lower than that of a notional idle task.

13. When the release event occurs for a processing group, the action taken depends on the state of the processing group.
   (a) When the processing group is not in the enforced state, 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.

14. Changes to the cost, minimum and maximum, take effect immediately.
   (a) When the new cost is less than or equal to the current CPU consumption, and the old cost was greater than the current CPU consumption, a cost overrun is triggered.
   (b) When the new cost is greater than the current CPU consumption there are two case:
      i. when the processing group is enforced, then the processing group behaves as defined in semantic 13;
      ii. otherwise, no cost monitoring and enforcement action occurs.

15. Changes to other parameters take place as follows:
   (a) changes to start have no effect;
   (b) changes to period take effect at each release, so the next period is set based on the current value of the processing group’s period;
   (c) changes to deadline take effect at each release, so the next deadline is set based on the current value of the processing group’s deadline;
   (d) changes to OverrunHandler take effect at each release, so the overrunHandler is set based on the current value of the processing group’s overrunHandler;
   (e) changes to MissHandler take effect at each release, so the missHandler is set based on the current value of the processing group’s missHandler; and
   (f) changes to UnderrunHandler take effect at each release, so the underrunHandler is set based on the current value of the processing group’s underrunHandler.

16. Changes to the membership of the processing group take effect immediately.

17. The start time for the processing group may be relative or absolute.
   (a) When the start time is absolute, the processing group behaves effectively as if the initial release time were the start time.
   (b) When the start time is relative, the initial release time is computed relative to the time that the processing group is constructed.

Note that until a processing group starts (i.e., its start time has been reached) it will perform no cost monitoring or enforcement on the Schedulables that it contains. Once a processing group is started, it behaves effectively as if it runs continuously until the defining ProcessingGroup object is freed. The start time does not affect limits placed on the group that are inherited from ThreadGroup or RealtimeThreadGroup, such as affinity and scheduling parameters.
6.3  javax.realtime

6.3.1  Interfaces

6.3.1.1  BoundSchedulable

public interface BoundSchedulable

Interfaces
javax.realtime.Schedulable

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

Available since RTSJ 2.0

6.3.1.2  Schedulable

public interface Schedulable

Interfaces
Runnable
javax.realtime.Timable

Description
An interface for all types of task defined in this specification. All implementations of Schedulable can be scheduled by any Scheduler defined here. A scheduler uses the information available through this interface to create a suitable context in which to execute the code encapsulated by this object.

Schedulable also represents one of the two subclasses of Timable defined in this specification, together with AsyncTimable. This type represents those objects that can be woken by a clock event, while AsyncTimable represents those that can be released. This functionality is folded into Schedulable because the set of types that can be woken by a clock event and the set of types that can be scheduled on a scheduler are the same, avoiding an extra layer of interface.
6 Scheduling

6.3.1.2.1 Methods

getMemoryParameters

Signature

public javax.realtime.MemoryParameters
getMemoryParameters()

Description

Gets a reference to the MemoryParameters object for this schedulable.

Returns

a reference to the current MemoryParameters object.

setMemoryParameters(MemoryParameters)

Signature

public javax.realtime.Schedulable
setMemoryParameters(MemoryParameters memory)

Description

Sets the memory parameters associated with this instance of Schedulable.

This change becomes effective at the next allocation; on multiprocessor systems, there may be some delay due to synchronization between processors.

Parameters

memory—A MemoryParameters 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.)

Throws

IllegalRealtimeArgumentException—when memory is not compatible with the schedulable's scheduler. Also when this schedulable may not use the 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.

Returns

this
Available since RTSJ 2.0 returns itself

**getReleaseParameters**

*Signature*

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

*Description*

Gets a reference to the `ReleaseParameters` object for this schedulable.

*Returns*

A reference to the current `ReleaseParameters` object.

**setReleaseParameters(ReleaseParameters)**

*Signature*

```java
public javax.realtime.Schedulable setReleaseParameters(ReleaseParameters release)
```

*Description*

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.

*Parameters*

- `release`—A `ReleaseParameters` 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`.)

*Throws*

- `IllegalRealtimeArgumentException`—when `release` is not compatible with the associated scheduler. Also when this schedulable may not use the 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.

IllegalTaskStateException—when the task is running and the new release parameters are not compatible with the current scheduler.

Returns
this

Available since RTSJ 2.0 returns itself

getScheduler

Signature
public javax.realtime.Scheduler getScheduler()

Description
Gets a reference to the Scheduler object for this schedulable.

Returns
a reference to the associated Scheduler object.

setScheduler(Scheduler)

Signature
public javax.realtime.Schedulable setScheduler(Scheduler scheduler) throws RealtimeSecurityException, IllegalTaskStateException

Description
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. If the Schedulable is running, its associated SchedulingParameters (if any) must be compatible with scheduler.

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

Throws

124 RTSJ 2.0 (Draft 66)
IllegalRealtimeArgumentException—when scheduler is null, or the schedulable’s existing parameter values are not compatible with scheduler. Also when this schedulable may not use the heap and scheduler is located in heap memory.

IllegalAssignmentError—when the schedulable cannot hold a reference to scheduler or the current Schedulable is running and its associated SchedulingParameters are incompatible with scheduler.

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

IllegalTaskStateException—when scheduler has scheduling or release parameters that are not compatible with the new scheduler and this schedulable is running.

Returns
this

Available since RTSJ 2.0 returns itself

setScheduler(Scheduler, SchedulingParameters, ReleaseParameters, MemoryParameters)

Signature

dpublic javax.realtime.Schedulable
setScheduler(Scheduler scheduler,
             SchedulingParameters scheduling,
             ReleaseParameters release,
             MemoryParameters memoryParameters)

Description

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.

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 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.)
release—A reference to the `ReleaseParameters` 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`.)

`memoryParameters`—A reference to the `MemoryParameters` 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`.)

Throws

- `IllegalRealtimeArgumentException`—when `scheduler` is null or the parameter values are not compatible with `scheduler`. Also thrown when this schedulable may not use the 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.
- `RealtimeSecurityException`—when the caller is not permitted to set the scheduler for this schedulable.

Returns

this

Available since RTSJ 2.0

### getSchedulingParameters

**Signature**

```java
generic javax.realtime.SchedulingParameters
getSchedulingParameters()
```

**Description**

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

**Returns**

A reference to the current `SchedulingParameters` object.

### setSchedulingParameters(SchedulingParameters)

**Signature**

```java
generic javax.realtime.Schedulable
setSchedulingParameters(SchedulingParameters scheduling)
```
Schedulable

throws IllegalTaskStateException,
   IllegalAssignmentError,
   IllegalRealtimeArgumentException

Description
Sets the scheduling 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. See the documentation
for the scheduler for details.

Parameters
scheduling—A reference to the SchedulingParameters 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.)

Throws
IllegalRealtimeArgumentException—when scheduling is not compatible with
the associated scheduler. Also when this schedulable may not use the 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.

IllegalTaskStateException—when the task is active and the new scheduling
parameters are not compatible with the current scheduler or the intersection of
affinity in scheduling and the affinity of this object’s realtime thread group
does not correspond to a valid affinity.

Returns
this

Available since RTSJ 2.0 returns itself

getRealtimeThreadGroup

Signature
   public javax.realtime.RealtimeThreadGroup
   getRealtimeThreadGroup()

Description
   Gets a reference to the RealtimeThreadGroup instance of this Schedulable.
   For instance that are not also instances of RealtimeThread, such as instance
if AsyncBaseEventHandler, the group returned is the group that contains the thread or threads that are used to execute said Schedulable instance. For each instance of Schedulable, there can be only one directly associated instance of Schedulable.

Returns
a reference to the associated RealtimeThreadGroup object.

Available since RTSJ 2.0

getConfigurationParameters

Signature
public javax.realtime.ConfigurationParameters
getConfigurationParameters()

Description
Gets a reference to the ConfigurationParameters object for this schedulable.

Returns
a reference to the associated ConfigurationParameters object.

Available since RTSJ 2.0

getMinConsumption(RelativeTime)

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

Description
Determines 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, returns the minimum consumption in a RelativeTime instance from the current allocation context. When dest is not null, returns the minimum consumption in dest.

Parameters
dest—When not null, the object in which to return the result.

Returns
the minimum time consumed in any release.

Available since RTSJ 2.0

getMinConsumption

Signature

public javax.realtime.RelativeTime
getMinConsumption()

Description

Equivalent to getMinConsumption(null).

Returns

the minimum time consumed in any release.

Available since RTSJ 2.0

gtetMaxConsumption(RelativeTime)

Signature

public javax.realtime.RelativeTime
getMaxConsumption(RelativeTime dest)

Description

Determines 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, returns the maximum consumption in a RelativeTime instance from the current allocation context. When dest is not null, returns the maximum consumption in dest

Parameters

dest—When not null, the object in which to return the result.

Returns

the maximum time consumed in any release.

Available since RTSJ 2.0
getMaxConsumption

**Signature**

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

**Description**

Equivalent to `getMaxConsumption(null)`.

**Returns**

the maximum time consumed in any release.

**Available since** RTSJ 2.0

setDaemon(boolean)

**Signature**

```java
public void setDaemon(boolean on)
```

**Description**

Marks this schedulable as either a daemon or a user task. A realtime virtual machine exits when the only tasks running are all daemons. This method must be called before the task is attached to any event or started. Once attached or started, it cannot be changed.

**Parameters**

- **on**—When true, marks this event handler as a daemon handler.

**Throws**

- `IllegalThreadStateException`—when this schedulable is active.
- `RealtimeSecurityException`—when the current schedulable cannot modify this event handler.

**Available since** RTSJ 2.0

isDaemon

**Signature**

```java
130 RTSJ 2.0 (Draft 66)
```
public boolean isDaemon()

Description
Tests if this event handler is a daemon handler.

Returns
true when this event handler is a daemon handler; false otherwise.

Available since RTSJ 2.0

mayUseHeap

Signature
public boolean mayUseHeap()

Description
Determines whether or not this schedulable may use the heap.

Returns
true only when this Schedulable may allocate on the heap and may enter the Heap.

Available since RTSJ 2.0

interrupt

Signature
public void interrupt()
throws IllegalTaskStateException

Description
In a system where RTSJModule.CONTROL is implemented, it makes the generic AsynchronouslyInterruptedException pending for this, and sets the interrupted state to true. As with Thread.interrupt(), blocking operations that are interruptible are interrupted. When this.isRousable() is true a release
happens event if the first release has not yet happened. In any case, \texttt{AsynchronouslyInterruptedException} is thrown once a method is entered that implements \texttt{AsynchronouslyInterruptedException}.

Otherwise, it behaves as if \texttt{Thread.interrupt()} were called.

For instances of \texttt{AsyncBaseEventHandler}, it behaves as if \texttt{RealtimeThread.interrupt()} was called on the implementation thread underlying it.

\textit{Throws}

\texttt{IllegalTaskStateException}—when \texttt{this} is not currently releasable, i.e., is disabled, not firable, its start method has not been called, or it has terminated.

\textbf{Available since} RTSJ 2.0

\subsection*{isInterrupted}

\textit{Signature}

\begin{verbatim}
public boolean isInterrupted()
\end{verbatim}

\textit{Description}

Determines whether or not any \texttt{AsynchronouslyInterruptedException} is pending.

\textit{Returns}

\texttt{true} when and only when the generic \texttt{AsynchronouslyInterruptedException} is pending.

\textbf{Available since} RTSJ 2.0

\subsection*{awaken}

\textit{Signature}

\begin{verbatim}
public void awaken()
\end{verbatim}

\textit{Throws} \texttt{IllegalRealtimeStateException}

\textit{Description}

Provides a means for a \texttt{Clock} to end a sleep.

\textit{Throws}

\textbf{132 RTSJ 2.0 (Draft 66)}
IllegalRealtimeStateException—when called from user code.

Available since RTSJ 2.0

6.3.2 Enumerations

6.3.2.1 MinimumInterarrivalPolicy

public enum MinimumInterarrivalPolicy

Inheritance
declaration

Description
Defines the set of policies for handling interarrival time violations in SporadicParameters. Each policy governs every instance of Schedulable which has SporadicParameters with that minimum interarrival time policy.

Available since RTSJ 2.0

6.3.2.1.1 Enumeration Constants

EXCEPT

public static final MinimumInterarrivalPolicy EXCEPT

Description
Represents the "EXCEPT" policy for minimum interarrival time. Under this policy, when an arrival time of a release occurs at a time less than the last release time plus its minimum interarrival time, the fire() method shall throw a preallocated instance of MITViolationException.
6 Scheduling

MinimumInterarrivalPolicy

**IGNORE**

```java
public static final MinimumInterarrivalPolicy IGNORE
```

*Description*

Represents the "IGNORE" policy for minimum interarrival time. Under this policy, when an arrival time of a release occurs at a time less than the last release time plus its minimum interarrival time, the new arrival time is ignored.

**REPLACE**

```java
public static final MinimumInterarrivalPolicy REPLACE
```

*Description*

Represents the "REPLACE" policy for minimum interarrival time. Under this policy, when an arrival time of a release occurs at a time less than the last release time plus its minimum interarrival time, the information for this arrival replaces a previous arrival. For cases when the previous event has already been released or the event queue has a length of zero, the arrival is ignored as with the "IGNORE" policy.

**SAVE**

```java
public static final MinimumInterarrivalPolicy SAVE
```

*Description*

Represents the "SAVE" policy for minimum interarrival time. Under this policy, when an arrival time of a release occurs at a time less than the last release time plus its minimum interarrival time, the new release is queued until the last release time plus its minimum interarrival time is reached.

6.3.2.1.2 Methods

**values**

*Signature*

134 RTSJ 2.0 (Draft 66)
public static javax.realtime.MinimumInterarrivalPolicy[]
values()

Description
Gets all enumeration constants.

valueOf(String)

Signature
public static javax.realtime.MinimumInterarrivalPolicy
valueOf(String name)

Description
Gets enumeration constants corresponding to name.

value

Signature
public java.lang.String
value()

Description
Determines the string corresponding to this value.

Returns
the corresponding string.

value(String)

Signature
public static javax.realtime.MinimumInterarrivalPolicy
value(String value)

Description
Converts a string into a policy type.

Parameters
value—is the string to convert.

Returns
the corresponding policy type.
6 Scheduling

6.3.2.2 QueueOverflowPolicy

public enum QueueOverflowPolicy

Inheritance
java.lang.Object
java.lang.Enum<QueueOverflowPolicy>
QueueOverflowPolicy

Description
Defines the set of policies for handling overflow on event queues used by ReleaseParameters. An event queue holds a number of event arrival times with any respective payload provided with the event. A reference to the event itself is only held when it happens to be the payload, e.g., for an AsyncObjectEvent associated with a Timer.

Available since RTSJ 2.0

6.3.2.2.1 Enumeration Constants

DISABLE

public static final QueueOverflowPolicy DISABLE

Description
Represents the "DISABLE" policy which means, when an arrival occurs, no queuing takes place, thus no overflow can happen. This policy is for instances of ActiveEvent with no payload and instances of RealtimeThread with PeriodicParameters. In contrast to IGNORE, all incoming events increment the pending fire or release count, respectively. For this reason, it may not be used with an event handler that supports an event payload or any instance of Schedulable with SporadicParameters. This policy is also the default for PeriodicParameters. Instances of RealtimeThread with null release parameters have this policy implicitly, as they do not have an event queue either.
EXCEPT

public static final QueueOverflowPolicy EXCEPT

Description
Represents the 'EXCEPT' policy which means, when an arrival occurs and its event time and payload should be queued but the queue already holds a number of event times and payloads equal to the initial queue length, the fire() method shall throw an ArrivalTimeQueueOverflowException. When fire is used within a Timer, the exception is ignored and the fire does nothing, i.e., it acts the same as "IGNORE".

IGNORE

public static final QueueOverflowPolicy IGNORE

Description
Represents the 'IGNORE' policy which means, when an arrival occurs and its event time and payload should be queued, but the queue already holds a number of event times and payloads equal to the initial queue length, the arrival is ignored.

REPLACE

public static final QueueOverflowPolicy REPLACE

Description
Represents the 'REPLACE' policy which means, when an arrival occurs and should be queued but the queue already holds a number of event times and payloads equal to the initial queue length, the information for this arrival replaces a previous arrival. When the queue length is zero, the behavior is the same as the 'IGNORE' policy.

SAVE

public static final QueueOverflowPolicy SAVE

Description
Represents the 'SAVE' policy which means, when an arrival occurs and should be queued but the queue is full, the queue is lengthened and the arrival time and payload are saved. 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. It is also the default for AperiodicParameters.

### 6.3.2.2 Methods

**values**

*Signature*

```java
public static javax.realtime.QueueOverflowPolicy[] values()
```

*Description*

Gets all enumeration constants.

**valueOf(String)**

*Signature*

```java
public static javax.realtime.QueueOverflowPolicy valueOf(String name)
```

*Description*

Gets enumeration constants corresponding to `name`.

**value**

*Signature*

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

*Description*

Determines the string corresponding to this value.

*Returns*

the corresponding string.
value(String)

Signature

    public static javax.realtime.QueueOverflowPolicy
    value(String value)

Description

    Converts a string into a policy type.

Parameters

    value—is the string to convert.

Returns

    the corresponding policy type.

6.3.3 Classes

6.3.3.1 Affinity

public class Affinity

Inheritance

    java.lang.Object
        Affinity

Interfaces

    Cloneable
    Serializable

Description

    This is the API for all processor-affinity-related aspects of the RTSJ. It includes
    a factory that generates Affinity objects. With it, the affinity of every task in
    a JVM can be controlled.

    An affinity is a set of processors that can be associated with certain types of
    tasks. Each task can be associated with an affinity via its SchedulingParameters.
    Groups of these can be assigned an affinity through their RealtimeThreadGroup.

    Each implementation supports an array of predefined affinities. 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 groups of task.
    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.
The processor membership of an affinity is immutable. The tasks associations of an affinity are mutable. The processor affinity of a task can be changed by static methods in this class. 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 factory only generates usable Affinity instances, i.e., affinities that (at least when they are created) can be used with SchedulingParameters, FirstInFirstOutParameters, and RoundRobinParameters. The factory cannot create an affinity with more than one processor member, but such affinities are supported. They may be internally created by the RTSJ runtime at startup time.

The set of affinities created at startup (the predefined set) is visible through the getPredefinedAffinities(Affinity[]) method. The affinity factory may be used to create affinities with a single processor member at any time. This operation only supports processor members that are available to the JVM at the time of 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.

Tasks are subject to both their own processor affinity and that of their thread group. Their processor affinity is governed by the intersection of the thread group’s affinity and the task’s affinity. The intersection of a thread group’s affinity with the schedulable’s affinity must contain at least one entry. Trying to set a task’s affinity outside its thread group always fails. Trying to set the affinity of a thread group that does not intersect with the thread group of its tasks will also fail.

Ordinarily, a task inherits its affinity from its creator unless the creator is an unbound instance of AsyncBaseEventHandler, since such an instance has no affinity. A task created by an unbound instance of AsyncBaseEventHandler inherits its affinity from the thread running the handler. An instance of Schedulable can be given a different affinity, so long as that affinity is subsumed by that of its closest enclosing RealtimeThreadGroup.

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

**Available since** RTSJ 2.0

**Open issue 6.3.1**

When an affinity is provided, must it also be a subset of its realtime thread
getPredefinedAffinitiesCount

Signature
  public static final int
  getPredefinedAffinitiesCount()

Description
  Determines the minimum array size required to store references to all the predefined processor affinities.

Returns
  the minimum array size required to store references to all the predefined affinities.

getPredefinedAffinities

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

Description
  Equivalent to invoking getPredefinedAffinitySets(null).

Returns
  an array of the predefined affinities.

getPredefinedAffinities(Affinity)

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

Description
  Determines what affinities are predefined by the Java runtime.
6 Scheduling

Affinity

Parameters
dest—The destination array, or null.

Throws
IllegalRealtimeArgumentException—when dest is not large enough.

Returns
dest or a newly created array when dest is null, populated with references to the predefined affinities. When dest has excess entries, those entries are filled with null.

isSetAffinitySupported

Signature
public static final boolean
isSetAffinitySupported()

Description
Determines whether or not affinity control is supported.

Returns
ture when more than one affinity set is available.

generate(BitSet)

Signature
public static final javax.realtime.Affinity
generate(BitSet set)

Description
Determines the Affinity corresponding to a BitSet, where each bit in set represents a CPU.

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

Parameters
set—The BitSet to convert into an Affinity.

Throws
NullPointerException—when set is null.

Returns
the resulting Affinity.
getAvailableProcessors

Signature

public static final java.util.BitSet
getAvailableProcessors()

Description

This method is equivalent to getAvailableProcessors(BitSet) with a null argument.

Returns

the set of processors available to the program.

getAvailableProcessors(BitSet)

Signature

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

Description

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 processors that are allocated to the RTSJ runtime and are currently available to execute tasks.

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).

isAffinityChangeNotificationSupported

Signature

public static final boolean
isAffinityChangeNotificationSupported()

Description
Determines whether or not the system can trigger an event for notifying the application when the set of available CPUs changes.

Returns
true when change notification is supported. (See setProcessorAddedEvent(AsyncEvent) and setProcessorRemovedEvent(AsyncEvent).)

getProcessorAddedEvent

Signature
public static javax.realtime.AsyncEvent getProcessorAddedEvent()

Description
Gets the event used for CPU addition notification.

Returns
the async event that will be fired when a processor is added to the set available to the JVM. Returns null when change notification is not supported, or when no async event has been designated.

setProcessorAddedEvent(AsyncEvent)

Signature
public static void setProcessorAddedEvent(AsyncEvent event)
throws UnsupportedRealtimeOperationException,
IllegalRealtimeArgumentException

Description
Sets the AsyncEvent that will be fired when a processor is added to the set available to the JVM.

Parameters
event—The async event to fire in case an added processor is detected, or null to cause no async event to be called in case an added processor is detected.

Throws
UnsupportedRealtimeOperationException—when change notification is not supported.
IllegalRealtimeArgumentException—when event is not in immortal memory.
getProcessorRemovedEvent

Signature

public static javax.realtime.AsyncEvent getProcessorRemovedEvent()

Description

Gets the event used for CPU removal notification.

Returns

the async event that will be fired when a processor is removed from the set available to the JVM. Returns null when change notification is not supported, or when no async event has been designated.

setProcessorRemovedEvent(AsyncEvent)

Signature

public static void setProcessorRemovedEvent(AsyncEvent event)

Description

Sets the AsyncEvent that will be fired when a processor is removed from the set available to the JVM.

Parameters

event—Called when a processor is removed.

Throws

UnsupportedRealtimeOperationException—when change notification is not supported.

IllegalRealtimeArgumentException—when event is not null or in immortal memory.

getProcessors

Signature

public final java.util.BitSet getProcessors()

Description

Obtains a BitSet representing the processor affinity set for this Affinity.
6 Scheduling

Returns
a newly created BitSet representing this Affinity.

getProcessors(BitSet)

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

Description
Determines the set of CPUs representing the processor affinity of this Affinity.

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)

Signature
public final boolean
isProcessorInSet(int processorId)

Description
Asks whether a processor is included in this affinity set.

Parameters
processorId—A number identifying a single CPU in a multiprocessor system.

Returns
true when and only when processorNumber is represented in this affinity set.

getProcessorCount

Signature
public int
getProcessorCount()

Description
Determines the number of CPUs in this affinity

>Returns
the number of CPUs.

**isValid**

**Signature**

```java
public boolean isValid()
```

**Description**

Determine whether or not the affinity can be used for scheduling.

>Returns
true when valid for scheduling and false otherwise.

### 6.3.3.2 AperiodicParameters

```java
public class AperiodicParameters
```

**Inheritance**

java.lang.Object

  ReleaseParameters

  AperiodicParameters

**Description**

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 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 schedulables that use the parameter
object until a setter method on the `AperiodicParameters` object is invoked,
the parameter object is used in `setReleaseParameters()`, or it is used in a
constructor for a schedulable.

The implementation must use modified copy semantics for each `HighResolutionTime`
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 aperiodic task. For an
instance of `RealtimeThread` the arrival time is the time at which the `start()` is
invoked. For other instances of `Schedulable`, the required behavior may force
the implementation to act effectively as if it maintained a queue of arrival times.

When the release parameters for a `RealtimeThread` are set to an instance of
this class or one of its subclasses, the thread does not start executing code until
the `RealtimeThread.release()` method is called.

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

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>cost</td>
<td>new <code>RelativeTime(0,0)</code></td>
</tr>
<tr>
<td>deadline</td>
<td>new <code>RelativeTime(Long.MAX_VALUE, 999999)</code></td>
</tr>
<tr>
<td>overrunHandler</td>
<td>None</td>
</tr>
<tr>
<td>missHandler</td>
<td>None</td>
</tr>
<tr>
<td>rousable</td>
<td>false</td>
</tr>
<tr>
<td>Arrival time queue size</td>
<td>0</td>
</tr>
<tr>
<td>Queue overflow policy</td>
<td>SAVE</td>
</tr>
</tbody>
</table>

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

**AperiodicParameters(RelativeTime, RelativeTime, AsyncEventHandler, AsyncEventHandler, boolean)**

*Signature*

```java
public AperiodicParameters(RelativeTime cost,
                            RelativeTime deadline,
                            AsyncEventHandler overrunHandler,
                            AsyncEventHandler missHandler,
                            boolean rousable)
```

*Description*

Creates an `AperiodicParameters` object.

*Available since RTSJ 2.0*

*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.

- **rousable**—determines whether or not an instance of `Schedulable` can be prematurely released by a thread interrupt.

*Throws*  

RTSJ 2.0 (Draft 66) 149
IllegalRealtimeArgumentException—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.

AperiodicParameters(RelativeTime, RelativeTime, AsyncEventHandler, AsyncEventHandler)

Signature

public AperiodicParameters(RelativeTime cost,
                           RelativeTime deadline,
                           AsyncEventHandler overrunHandler,
                           AsyncEventHandler missHandler)

Description

Equivalent to AperiodicParameters(RelativeTime, RelativeTime, AsyncEventHandler, AsyncEventHandler, boolean) with the argument list (cost, deadline, overrunHandler, missHandler, false).

Parameters

cost—Processing time per invocation. On implementations that support cost enforcement, this value is the maximum amount of time a schedulable receives. On implementations which do not support cost enforcement, 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

IllegalRealtimeArgumentException—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.

AperiodicParameters(RelativeTime, AsyncEventHandler, boolean)

Signature
public
AperiodicParameters(RelativeTime deadline,
AsyncEventHandler missHandler,
boolean rousable)

Description
Equivalent to AperiodicParameters(RelativeTime, RelativeTime, AsyncEventHandler, AsyncEventHandler, boolean) with the argument list (null, deadline, null, missHandler, rousable).

Available since RTSJ 2.0

AperiodicParameters(RelativeTime)

Signature
public
AperiodicParameters(RelativeTime deadline)

Description
Equivalent to AperiodicParameters(RelativeTime, RelativeTime, AsyncEventHandler, AsyncEventHandler, boolean) with the argument list (null, deadline, null, null, false).

Available since RTSJ 2.0

AperiodicParameters

Signature

RTSJ 2.0 (Draft 66) 151
public
AperiodicParameters()

Description
Equivalent to AperiodicParameters(RelativeTime, RelativeTime, AsyncEventHandler, AsyncEventHandler, boolean) with the argument list (null, null, null, null, false).

Available since RTSJ 1.0.1

6.3.3.2.2 Methods

setDeadline(RelativeTime)

Signature
public javax.realtime.AperiodicParameters
  setDeadline(RelativeTime deadline)

Description
Sets the deadline value.

When this parameter object is associated with any schedulable object, either by being passed through the schedulable’s constructor or being set with a method such as RealtimeThread.setReleaseParameters(ReleaseParameters), the deadline of those schedulables takes effect at completion.

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
IllegalRealtimeArgumentException—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—when deadline cannot be stored in this.

Returns
this

Available since RTSJ 2.0 returns itself
6.3.3.3 BackgroundParameters

public class BackgroundParameters

Inheritance
java.lang.Object
  ReleaseParameters
    BackgroundParameters

Description
Parameters for realtime threads that are only released once. A thread using this release parameters may not use `RealtimeThread.waitForNextRelease()` or have its `RealtimeThread.release()` methods called. Calling these methods results in an `IllegalThreadStateException`. Event handlers may not use this type of `ReleaseParameters`.

Available since RTSJ 2.0

6.3.3.3.1 Constructors

BackgroundParameters(RelativeTime, RelativeTime, AsyncEventHandler, AsyncEventHandler)

Signature
public
BackgroundParameters(RelativeTime cost,
  RelativeTime deadline,
  AsyncEventHandler overrunHandler,
  AsyncEventHandler missHandler)

Description
A constructor for both cost and deadline monitoring.

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

Parameters
cost—The maximum cost for executing the run method
FirstInFirstOutParameters

**deadline**—The deadline for the completion of the run method

**overrunHandler**—The handler to call on cost overrun.

**missHandler**—The handler to call on deadline miss.

**Throws**

**IllegalRealtimeArgumentException**—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 chronograph associated with the cost or deadline parameters is not an instance of Clock.

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

---

### BackgroundParameters(RelativeTime, AsyncEventHandler)

**Signature**

```java
public
BackgroundParameters(RelativeTime deadline,
AsyncEventHandler missHandler)
```

**Description**

A constructor for deadline monitoring. Equivalent to BackgroundParameters(null, deadline, null, missHandler)

---

### BackgroundParameters

**Signature**

```java
public
BackgroundParameters()
```

**Description**

A constructor for not having any restrictions on, or monitoring of, scheduling. Equivalent to BackgroundParameters(null, null, null, null, false)
public class FirstInFirstOutParameters

Inheritance
java.lang.Object
   SchedulingParameters
      PriorityParameters
         FirstInFirstOutParameters

Description
Same as PriorityParameters except that it is only valid with the FirstInFirstOutScheduler.

Available since RTSJ 2.0

6.3.3.4.1 Constructors

FirstInFirstOutParameters(int, Affinity)

Signature
public
FirstInFirstOutParameters(int priority,
   Affinity affinity)

Description
Create scheduling parameters restricted to the FIFO scheduler.

Parameters
priority—The priority assigned to schedulables that use this parameter instance.
affinity—The affinity assigned to schedulables that use this parameter instance.

6.3.3.4.2 Methods
isCompatible(Scheduler)

Signature
public boolean isCompatible(Scheduler scheduler)

Description
Determines whether this scheduling parameters can be used by tasks scheduled by scheduler.

Parameters
scheduler—The scheduler to check against

Returns
true when and only when this can be used with scheduler as the scheduler.

Available since RTSJ 2.0

subsumes(SchedulingParameters)

Signature
public boolean subsumes(SchedulingParameters other)

Description
Determine whether this parameters is more eligible than another.

Parameters
other—The other parameters object to be compared with.

Returns
true when and only when this parameters is more eligible than the other parameters.

6.3.3.5 FirstInFirstOutScheduler

public class FirstInFirstOutScheduler

Inheritance
java.lang.Object
Scheduler
PriorityScheduler
FirstInFirstOutScheduler

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

This scheduler, like all schedulers, governs the default values for scheduling-related parameters in its client schedulables. The defaults are as follows:

Table 6.4: FirstInFirstOut Default PriorityParameter Values

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Priority</td>
<td>norm priority</td>
</tr>
</tbody>
</table>

The system contains one instance of the FirstInFirstOutScheduler which is the system’s base scheduler and is returned by FirstInFirstOutScheduler. instance(). The instance returned by the instance() method is the base scheduler and is returned by Scheduler.getDefaultScheduler() unless the default scheduler is reset with Scheduler.setDefaultScheduler(Scheduler).

Available since RTSJ 2.0

6.3.3.5.1 Methods

instance

Signature
public static javax.realtime.FirstInFirstOutScheduler instance()

Description
Obtains a reference to the distinguished instance of PriorityScheduler which is the system’s base scheduler.

Returns
a reference to the distinguished instance PriorityScheduler.

**getMaxPriority**

*Signature*

```java
public int getMaxPriority()
```

*Description*

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

*Returns*

the value of the maximum priority.

**getMinPriority**

*Signature*

```java
public int getMinPriority()
```

*Description*

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

*Returns*

the minimum priority used by this scheduler.

**getNormPriority**

*Signature*

```java
public int getNormPriority()
```

*Description*

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

*Returns*

the value of the normal priority.
getPolicyName

Signature
public java.lang.String
getPolicyName()

Description
Obtains the policy name of this.

Returns
the policy name (Fixed Priority First In First Out) as a string.

reschedule(Thread, SchedulingParameters)

Signature
public void
reschedule(Thread thread,
SchedulingParameters eligibility)

Description
Promotes a java.lang.Thread to realtime priority under this scheduler. The affected thread will be scheduled as if it were a RealtimeThread with the given eligibility. This does not make the affected thread a RealtimeThread, however, and it will not have access to facilities reserved for instances of RealtimeThread. Instances of RealtimeThread will be treated as if their scheduling parameters were set to eligibility.

Parameters
thread—The thread to promote to realtime scheduling.
eligibility—A SchedulingParameters instance such as PriorityParameters for a PriorityScheduler.

Throws
IllegalRealtimeArgumentException—when eligibility is not valid for the scheduler.
IllegalRealtimeStateException—when eligibility specifies parameters that are out of range for the scheduler or the threads state or the intersection of affinity in scheduling and the affinity of realtime thread group associated with thread is empty.
UnsupportedRealtimeOperationException—when thread a normal Java thread and the scheduler does not support promoting normal java threads.
6 Scheduling

PeriodicParameters

Available since RTSJ 2.0

6.3.3.6 PeriodicParameters

public class PeriodicParameters

Inheritance

java.lang.Object
ReleaseParameters
PeriodicParameters

Description

This release parameter indicates that the schedulable is released on a regular basis. For an AsyncEventHandler, this means the handler is either released by a periodic timer or the associated event occurs periodically. For a RealtimeThread, this means the RealtimeThread.waitForNextRelease 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. 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 a change to a PeriodicParameters object caused by methods on that object cause the change to be propagate to all instances of Schedulable using that parameter object. For instance, calling setCost on a PeriodicParameters object will make the change, then notify 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 a RelativeTime object that is the cost for this object changes the cost value but 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, the parameter object is used in setReleaseParameters(), or it is used in a constructor for an SO.

Periodic parameters use HighResolutionTime values for period and start time. Since these times are expressed as a HighResolutionTime 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` 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.

Table 6.5: PeriodicParameter Default Values

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td><code>new RelativeTime(0,0)</code></td>
</tr>
<tr>
<td>period</td>
<td>No default. A value must be supplied</td>
</tr>
<tr>
<td>cost</td>
<td><code>new RelativeTime(0,0)</code></td>
</tr>
<tr>
<td>deadline</td>
<td><code>new RelativeTime(period)</code></td>
</tr>
<tr>
<td>overrunHandler</td>
<td>None</td>
</tr>
<tr>
<td>missHandler</td>
<td>None</td>
</tr>
<tr>
<td>EventQueueOverflowPolicy</td>
<td><code>QueueOverflowPolicy.DISABLE</code></td>
</tr>
</tbody>
</table>

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.

6.3.3.6.1 Constructors

```java
PeriodicParameters(HighResolutionTime, RelativeTime, RelativeTime, RelativeTime, AsyncEventHandler, AsyncEventHandler, boolean)
```
6 Scheduling

Signature

```java
public PeriodicParameters(HighResolutionTime<?> start,
RelativeTime period,
RelativeTime cost,
RelativeTime deadline,
AsyncEventHandler overrunHandler,
AsyncEventHandler missHandler,
boolean rousable)
```

Description

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

Available since RTSJ 2.0

Parameters

- **start**—Time at which the first release begins (i.e. the realtime 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 releases. 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.

- **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.

- **rousable**—When `true`, an interrupt will cause an early release; otherwise not.

Throws

RTSJ 2.0 (Draft 66)
**IllegalRealtimeArgumentException**—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 realtime clock, or when the clocks associated with the `deadline` and `period` parameters are not the same.

**IllegalAssignmentError**—when `start`, `period`, `cost`, `deadline`, `overrunHandler`, or `missHandler` cannot be stored in this.

`PeriodicParameters(HighResolutionTime, RelativeTime, RelativeTime, RelativeTime, AsyncEventHandler, AsyncEventHandler)`

**Signature**

```java
public PeriodicParameters(HighResolutionTime<?> start, RelativeTime period, RelativeTime cost, RelativeTime deadline, AsyncEventHandler overrunHandler, AsyncEventHandler missHandler)
```

**Description**

Equivalent to `PeriodicParameters(HighResolutionTime, RelativeTime, RelativeTime, RelativeTime, AsyncEventHandler, AsyncEventHandler, boolean)` with the argument list `(start, period, cost, deadline, overrunHandler, missHandler, false);

`PeriodicParameters(HighResolutionTime, RelativeTime, RelativeTime, AsyncEventHandler, boolean)`

**Signature**

```java
public PeriodicParameters(HighResolutionTime<?> start, RelativeTime period, RelativeTime deadline, AsyncEventHandler missHandler, boolean rousable)
```
6 Scheduling

PeriodicParameters

**Description**

Equivalent to `PeriodicParameters(HighResolutionTime, RelativeTime, RelativeTime, RelativeTime, AsyncEventHandler, AsyncEventHandler, boolean)` with the argument list `(start, period, deadline, null, null, missHandler, rousable);

Available since RTSJ 2.0

**PeriodicParameters(HighResolutionTime, RelativeTime)**

**Signature**

```java
public PeriodicParameters(HighResolutionTime<?> start,
                          RelativeTime period)
```

**Description**

Equivalent to `PeriodicParameters(HighResolutionTime, RelativeTime, RelativeTime, RelativeTime, AsyncEventHandler, AsyncEventHandler, boolean)` with the argument list `(start, period, null, null, null, null, false);

Available since RTSJ 1.0.1

**PeriodicParameters(RelativeTime)**

**Signature**

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

**Description**

Creates 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, false)`

Available since RTSJ 1.0.1
6.3.3.6.2 Methods

getPeriod

Signature
public javax.realtime.RelativeTime
getPeriod()

Description
Determines the current value of period.

Returns
the object last used to set the period containing the current value of period.

getPeriod(RelativeTime)

Signature
public javax.realtime.RelativeTime
getPeriod(RelativeTime value)

Description
Determines the current value of period.

Returns
value or, when null, the last object used to set the period, set to the current value of period.

Available since RTSJ 2.0

getStart

Signature
public javax.realtime.HighResolutionTime<?>
getStart()

Description
Determines the time used to start an instance of Schedulable, which is not necessarily the time at which it actually started.
setDeadline(RelativeTime)

Signature
public javax.realtime.PeriodicParameters
    setDeadline(RelativeTime deadline)

Description
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)) the deadline of those schedulables takes effect at the next release event.

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
IllegalRealtimeArgumentException—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—when deadline cannot be stored in this.

Returns
this

Available since RTSJ 2.0 returns itself

setPeriod(RelativeTime)

Signature
public javax.realtime.PeriodicParameters
    setPeriod(RelativeTime period)

Description
Sets the period.

Parameters

**PeriodicParameters**

The value to which `period` is set.

*Throws*

- `IllegalRealtimeArgumentException`—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 `AsyncBaseEventHandler` is associated with a `Timer` whose period does not match `period`.

- `IllegalAssignmentError`—when `period` cannot be stored in `this`.

*Returns*

- `this`

Available since RTSJ 2.0 returns itself

**setStart(HighResolutionTime)**

*Signature*

```java
public javax.realtime.PeriodicParameters setStart(HighResolutionTime<?> start)
```

*Description*

Sets the start time.

Changes to the start time in a realtime thread’s `PeriodicParameters` object only have an effect on its initial release time.

Note that 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.

*Parameters*

- `start`—The new start time. When null, the default value is a new instance of `RelativeTime(0,0)`.

*Throws*

- `IllegalRealtimeArgumentException`—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`.

*Returns*

- `this`

Available since RTSJ 2.0 returns itself
6.3.3.7 PriorityParameters

public class PriorityParameters

Inheritance
java.lang.Object
   SchedulingParameters
      PriorityParameters

Description
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 is such a scheduler.

6.3.3.7.1 Constructors

PriorityParameters(int, Affinity)

Signature
public
PriorityParameters(int priority,
   Affinity affinity)

Description
Creates an instance of PriorityParameters with the given features.

Available since RTSJ 2.0

Parameters
priority—The priority assigned to schedulables that use this parameter instance.
affinity—The affinity assigned to schedulables that use this parameter instance.

PriorityParameters(int)
public
PriorityParameters(int priority)

Description
Creates an instance of PriorityParameters with the default affinity.

Parameters
priority—The priority assigned to schedulables that use this parameter instance.

6.3.3.7.2 Methods

getPriority

Signature
public int
getPriority()

Description
Gets the priority value.

Returns
the priority.

isCompatible(Scheduler)

Signature
public boolean
isCompatible(Scheduler scheduler)

Description
Determines whether this scheduling parameters can be used by tasks scheduled by scheduler.

Parameters
scheduler—The scheduler to check against

Returns
true when and only when this can be used with scheduler as the scheduler.

Available since RTSJ 2.0
subsumes(SchedulingParameters)

Signature
public boolean
subsumes(SchedulingParameters other)

Description
Determine whether this parameters is more eligible than another.

Parameters
other—The other parameters object to be compared with.

Returns
ture when and only when this parameters is more eligible than the other parameters.

toString

Signature
public java.lang.String
toString()

Description
Converts the priority value to a string.

Returns
a string representing the value of priority.

6.3.3.8 PriorityScheduler

public abstract class PriorityScheduler

Inheritance
java.lang.Object
Scheduler
PriorityScheduler

Description
Class which represents the required (by the RTSJ) priority-based schedulers. The
default instance is the base scheduler which uses a fixed priority, first-in-first-out,
preemptive scheduling algorithm.
Table 6.6: PriorityScheduler Default PriorityParameter Values

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Priority</td>
<td>norm priority</td>
</tr>
</tbody>
</table>

This scheduler, like all schedulers, governs the default values for scheduling-related parameters in its client schedulables. The defaults are as follows:

Note that the system contains one instance of the `PriorityScheduler` which is the system’s base scheduler and is returned by `FirstInFirstOutScheduler.instance()`. It may, however, contain instances of subclasses of `PriorityScheduler` created through this class’ protected constructor. The instance returned by the `FirstInFirstOutScheduler.instance()` method, the base scheduler, is returned by `Scheduler.getDefaultScheduler()` unless the default scheduler is changed with `Scheduler.setDefaultScheduler(Scheduler)`.

6.3.3.8.1 Constructors

PriorityScheduler

Signature

```java
protected
PriorityScheduler()
```

Description

Constructs an instance of `PriorityScheduler`. Applications will likely not need any instance other than the default instance.

6.3.3.8.2 Methods

getPolicyName

Signature

```java
RTSJ 2.0 (Draft 66)
```
public java.lang.String
getPolicyName()

Description
Gets the policy name of this.

Returns
the policy name (Fixed Priority) as a string.

getMaxPriority

Signature
public abstract int
g getMaxPriority()

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

Returns
the value of the maximum priority.

getMinPriority

Signature
public abstract int
g getMinPriority()

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

Returns
the minimum priority used by this scheduler.

getNormPriority

Signature
public abstract int
g getNormPriority()

Description

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

*Returns*

the value of the normal priority.

### 6.3.3.9 ProcessingGroup

```java
class ProcessingGroup
```

*Inheritance*

java.lang.Object
  java.lang.ThreadGroup
    RealtimeThreadGroup
      ProcessingGroup

*Description*

A descendent class of `ThreadGroup` for handling tasks (instances of `Schedulable` and `java.lang.Thread`) as a group. As with `ThreadGroup` and `RealtimeThreadGroup`, instances of `ProcessingGroup` can be nested. A processing group can contain all group types, i.e., instance of all three classes. The cost of the group, including all tasks in its subgroups, can be both tracked and limited over a given period, by bounding the execution demands of those tasks.

A processing group has an associated affinity. The precision of cost monitoring is dependent on the number of processors in the thread group. In the worst case, it is the base precision times the number of processors in the processing group. The default affinity is that which was inherited from the parent `RealtimeThreadGroup`.

For all tasks with a reference to an instance of `ProcessingGroup p`, no more than `p.cost` will be allocated to the execution of these tasks on the processors 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 tasks will be allowed on any processor other than these processors.

For each running task in a processing group, there must always be at least one processor in the intersection between a task object’s affinity and its processing group’s affinity regardless of the group’s monitoring state.

Logically, a `ProcessingGroup` represents a virtual server. 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 tasks is executable and is the most eligible of the executable tasks.

When the server is logically executable, the associated tasks are executed.

When the cost has been consumed, any \texttt{overrunHandler} is released, and the server is not eligible for logical execution until the period is finished. At this point, its allocated cost (budget) is replenished. When the server is logically executable when its deadline expires, any associated \texttt{missHandler} is released. When the server is logically executable when its next release time occurs, any associated \texttt{underrunHandler} 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 \texttt{HighResolutionTime} values for cost, deadline, period and start time. Since those times are expressed as a \texttt{HighResolutionTime}, 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 \texttt{ProcessingGroup} 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 \texttt{ProcessingGroup} object becomes the processing group parameter object bound to that schedulable object. Changes to the values in the \texttt{ProcessingGroup} object affect that schedulable object. When bound to more than one schedulable then changes to the values in the \texttt{ProcessingGroup} object affect \textit{all} of the associated objects. Note that this is a one-to-many relationship and \textit{not} a many-to-many.

The implementation must use copy semantics for each \texttt{HighResolutionTime} parameter value. The value of each time object should be copied at the time it is passed to the parameter object, and the object reference must not be retained. Only changes to a \texttt{ProcessingGroup} object caused by methods on that object are immediately visible to the scheduler. For instance, invoking \texttt{setPeriod()} on a \texttt{ProcessingGroup} object will make the change, then notify 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 \texttt{RelativeTime} object that affects 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 \texttt{ProcessingGroup} object is invoked or the object is used in a constructor for an SO.

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

\textbf{Caution:} This class is explicitly unsafe in multithreaded situations when it
Table 6.7: ProcessingGroup Default Values

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>new RelativeTime(0,0)</td>
</tr>
<tr>
<td>period</td>
<td>No default. A value must be supplied</td>
</tr>
<tr>
<td>cost</td>
<td>No default. A value must be supplied</td>
</tr>
<tr>
<td>deadline</td>
<td>new RelativeTime(period)</td>
</tr>
<tr>
<td>minimum</td>
<td>null, no minimum</td>
</tr>
<tr>
<td>overrunHandler</td>
<td>None</td>
</tr>
<tr>
<td>missHandler</td>
<td>None</td>
</tr>
<tr>
<td>underrunHandler</td>
<td>None</td>
</tr>
</tbody>
</table>

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.

**Available since** RTSJ 2.0

### 6.3.3.9.1 Constructors

**ProcessingGroup(RealtimeThreadGroup, String, HighResolutionTime, RelativeTime, RelativeTime, AsyncEventHandler, RelativeTime, AsyncEventHandler)**

**Signature**

```java
public ProcessingGroup(RealtimeThreadGroup parent,
                       String name,
                       HighResolutionTime<?> start,
                       RelativeTime period,
                       RelativeTime cost,
                       AsyncEventHandler overrunHandler,
                       AsyncEventHandler missHandler,
                       AsyncEventHandler underrunHandler)
```

RTSJ 2.0 (Draft 66) 175
ProcessingGroup

AsyncEventHandler overrun,
RelativeTime minimum,
AsyncEventHandler underrun)

Description

Creates a ProcessingGroup

Parameters

parent—The parent RealtimeThreadGroup of this ProcessingGroup.
name—A string identifier for this group.
start—The time when monitoring should begin.
period—An amount of time for cost and overrun monitoring and for cost enforcement.
cost—The maximum total execution time of all tasks in the group during a given period.
overrun—It is called when the total execution of all tasks in the group exceeds cost for a given period.
minimum—The least amount of processing time that should be available for all the tasks in this group together.
underrun—A handler to be called at the end of period when the total processing time of all tasks was less than minimum in the last period.

ProcessingGroup(RealtimeThreadGroup, String, HighResolutionTime, RelativeTime, RelativeTime, AsyncEventHandler)

Signature

public

ProcessingGroup(RealtimeThreadGroup parent,
String name,
HighResolutionTime<?> start,
RelativeTime period,
RelativeTime cost,
AsyncEventHandler overrun)

Description

Equivalent to ProcessingGroup(RealtimeThreadGroup, String, HighResolutionTime, RelativeTime, RelativeTime, AsyncEventHandler,
ProcessingGroup

RelativeTime, AsyncEventHandler) with the argument list (parent, name, start, period, cost, overrun, null, null).

ProcessingGroup(String, HighResolutionTime, RelativeTime, RelativeTime, AsyncEventHandler)

Signature

public ProcessingGroup(String name,
            HighResolutionTime<?> start,
            RelativeTime period,
            RelativeTime cost,
            AsyncEventHandler overrun)

Description

Equivalent to ProcessingGroup(RealtimeThreadGroup, String, HighResolutionTime, RelativeTime, RelativeTime, AsyncEventHandler, RelativeTime, AsyncEventHandler) with the argument list (Scheduler.currentSchedulable().getRealtimeThreadGroup(), name, start, period, cost, overrun, null, null).

6.3.3.9.2 Methods

geterEffectiveStart(AbsoluteTime)

Signature

public javax.realtime.AbsoluteTime
geterEffectiveStart(AbsoluteTime dest)

Description

Obtains the actual time of the group’s start as recorded by the system. When the start time is absolute, that is the effective start time; otherwise, the effective start is computed relative to the time that the processing group is constructed.

Parameters

dest—A time value to fill.
Returns
either, a new instance of `AbsoluteTime`, when `dest` is null, or `dest` otherwise. In
either case, its value is the time at which this group actually started.

genericReturn

Signature

```java
class javax.realtime.AbsoluteTime

genericReturn()
```

Description

Obtains the actual time of the group’s start as recorded by the system.
Equivalent to `getEffectiveStart(AbsoluteTime)` where `dest` is set to null.

Returns

a reference to a new instance of `AbsoluteTime` that represents the time at which
this group started.

genericReturn

Signature

```java
class javax.realtime.RelativeTime

genericReturn(RelativeTime dest)
```

Description

Gets the value of `period` in the provided `RelativeTime` object.

Parameters

`dest`—An instance of `RelativeTime` which will be set to the currently configured
`period`. If `dest` is null, a new `RelativeTime` will be created in the current
allocation context.

Returns

a reference to `dest`, or a newly created object if `dest` is null.

genericReturn

Signature

```java
class javax.realtime.RelativeTime

genericReturn()
```

178 RTSJ 2.0 (Draft 66)
Description

Gets the value of period.
Equivalent to getPeriod(null).

Returns

a reference to a newly allocated instance of RelativeTime that represents the value of period.

setPeriod(RelativeTime)

Signature

public javax.realtime.ProcessingGroup
setPeriod(RelativeTime period)
throws IllegalRealtimeArgumentException,
IllegalAssignmentError

Description

Sets the value of period.

Parameters

period—The new value for period. There is no default value. When period is null an exception is thrown.

Throws

IllegalRealtimeArgumentException—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.

Returns

this

getMaximumCost(RelativeTime)

Signature

public javax.realtime.RelativeTime
getMaximumCost(RelativeTime dest)

Description

Gets the value of cost in the provided RelativeTime object.
Parameters
dest—An instance of RelativeTime which will be set to the currently configured cost. If dest is null, a new RelativeTime will be created in the current allocation context.

Returns
a reference to dest, or a newly created object if dest is null.

getMaximumCost

Signature
public javax.realtime.RelativeTime getMaximumCost()

Description
Gets the value of cost.
Equivalent to getMaximumCost(null).

Returns
a reference to a newly allocated object containing the value of cost.

setMaximumCost(RelativeTime)

Signature
public javax.realtime.ProcessingGroup setMaximumCost(RelativeTime cost)
throws IllegalRealtimeArgumentException, IllegalAssignmentError

Description
Sets the value of cost.

Parameters
cost—The new value for cost. When null, an exception is thrown.

Throws
IllegalRealtimeArgumentException—when cost is null or its time value is less than zero.

Returns
this
getMinimumCost(RelativeTime)

Signature
   public javax.realtime.RelativeTime
   getMinimumCost(RelativeTime dest)

Description
   Gets the value of minimum and returns it in the provided RelativeTime object.

Parameters
   dest—An instance of RelativeTime which will be set to the currently configured minimum. If dest is null, a new RelativeTime will be created in the current allocation context.

Returns
   a reference to dest, or a newly created object if dest is null.

getMinimumCost

Signature
   public javax.realtime.RelativeTime
   getMinimumCost()

Description
   Gets the value of minimum and returns it in a newly allocated object.
   Equivalent to getMinimumCost(null).

Returns
   a reference to the value of minimum.

setMinimumCost(RelativeTime)

Signature
   public javax.realtime.ProcessingGroup
   setMinimumCost(RelativeTime cost)
   throws IllegalRealtimeArgumentException, IllegalAssignmentError

Description
   Sets the value of minimum.
Parameters

- **cost**—The new value for **minimum**. When **null**, an exception is thrown.

Throws

- **IllegalRealtimeArgumentException**—when **minimum** is **null** or its time value is less than zero.

Returns

- **this**

**getCostUnderrunHandler**

**Signature**

```java
public javax.realtime.AsyncEventHandler getCostUnderrunHandler()
```

**Description**

Gets the cost underrun handler.

**Returns**

- a reference to an instance of **AsyncEventHandler** that is cost overrun handler of **this**.

**setCostUnderrunHandler(AsyncEventHandler)**

**Signature**

```java
public javax.realtime.ProcessingGroup setCostUnderrunHandler(AsyncEventHandler handler)
```

**Description**

Sets the cost underrun handler.

**Parameters**

- **handler**—This handler is invoked when the **run**() method of the schedulables attempts 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**.

**Returns**

- **this**
**getCostOverrunHandler**

*Signature*

```java
public javax.realtime.AsyncEventHandler getCostOverrunHandler()
```

*Description*

Gets the cost overrun handler.

*Returns*

A reference to an instance of `AsyncEventHandler` that is cost overrun handler of `this`.

**setCostOverrunHandler(AsyncEventHandler)**

*Signature*

```java
public javax.realtime.ProcessingGroup setCostOverrunHandler(AsyncEventHandler handler)
throws IllegalAssignmentError
```

*Description*

Sets the cost overrun handler.

*Parameters*

- `handler`—This handler is invoked when the `run()` method of the schedulables attempts 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`.

*Returns*

`this`

**enforcingCost**

*Signature*

```java
public boolean enforcingCost()
```

*Description*

RTSJ 2.0 (Draft 66) 183
Determines whether or not cost is being enforced for releases.

Returns
true when enforcing code.

enforceCost

Signature
public void enforceCost()
    throws UnsupportedRealtimeOperationException

Description
Starts cost enforcement at next release, when supported. Subsequent invocations have no effect.

Throws
UnsupportedRealtimeOperationException—when cost enforcement is not supported.

getCurrentCost(RelativeTime)

Signature
public javax.realtime.RelativeTime getCurrentCost(RelativeTime dest)

Description
Gets the cost used in the current period so far.

Parameters
dest—The instance to use for returning the time. If dest is null, the result will be returned in a newly allocated object.

Returns
dest containing the cost of the current period

getLastCost(RelativeTime)

Signature

184 RTSJ 2.0 (Draft 66)
public javax.realtime.RelativeTime
getLastCost(RelativeTime dest)

Description
Gets the total cost used in the last period.

Parameters
dest—It is the instance to use for returning the time. If dest is null, the result
will be returned in a newly allocated object.

Returns
dest containing the cost of the last period

calGranularity

Signature
public long
calGranularity()

Description
Determines the measurement granularity of cost monitoring and cost enforcement.

Returns
the granularity in nanoseconds.

See Section setGranularity

setGranularity(long)

Signature
public javax.realtime.ProcessingGroup
setGranularity(long nanos)
throws IllegalRealtimeArgumentException

Description
Sets the measurement granularity of cost monitoring and cost enforcement. The
system provides a lower bound for this. When nanos is below this lower bound,
granularity silently is set to the lower bound. In general, the lower bound is the
precision of the realtime clock.

Note that the granularity applies to a single processor. When a processing group
spans more than one processor, the precision of cost monitoring or enforcement
is this granularity times the number of active processors. This is because more than one task could be running at the same time and cost can be measured at most once per the elapse of this granularity.

Parameters

nanos—the new granularity in nanoseconds.

Throws

IllegalRealtimeArgumentException—when nums is less than one.

Returns

this

6.3.3.10 RealtimeThreadGroup

public class RealtimeThreadGroup

Inheritance
java.lang.Object
java.lang.ThreadGroup
RealtimeThreadGroup

Description
An enhanced ThreadGroup in which a RealtimeThread instance may be started, as well as a convention Thread. Limits for what realtime scheduler and scheduling parameters can be enforced on all tasks in this group. A normal ThreadGroup may not contain an instance of Schedulable, but may contain other instances of RealtimeThreadGroup to form a hierarchy. Every thread is in some instance of ThreadGroup and every instance of RealtimeThread is in some instance of RealtimeThreadGroup.

Available since RTSJ 2.0

6.3.3.10.1 Constructors

RealtimeThreadGroup(RealtimeThreadGroup, String)

Signature

186 RTSJ 2.0 (Draft 66)
public
RealtimeThreadGroup(RealtimeThreadGroup parent,
String name)

Description
Creates a new realtime thread group with its scheduler type inherited from parent.

Parameters
parent—The parent group of the new group
name—The name of the new group

Throws
IllegalRealtimeStateException—when the parent ThreadGroup instance is not an instance of RealtimeThreadGroup.
IllegalAssignmentError—when the parent ThreadGroup instance is not assignable to this.

RealtimeThreadGroup(String)

Signature
public
RealtimeThreadGroup(String name)
throws IllegalRealtimeStateException,
IllegalAssignmentError

Description
Creates a new group with the current ThreadGroup instance as its parent and that parent’s scheduler type for its scheduler type. That parent must be an instance of RealtimeThreadGroup. The primordial realtime thread group has Scheduler.class as its scheduler type.

Parameters
name—The name of the new group

Throws
IllegalRealtimeStateException—when the parent ThreadGroup instance is not an instance of RealtimeThreadGroup.
IllegalAssignmentError—when the parent ThreadGroup instance is not assignable to this.
6.3.3.10.2 Methods

getScheduler

Signature

public java.lang.Class<javax.realtime.Scheduler>
getScheduler()

Description

Finds the type of scheduler tasks in this group may use. The scheduler
of each thread must be an instance of the type returned. The default is
class<Scheduler>, but it may be set to any subtype.

Returns

the scheduler type

getMaxEligibility

Signature

public javax.realtime.SchedulingParameters
getMaxEligibility()

Description

Finds the upper bound on scheduling eligibility that tasks in this group may
have. For example, when it is an instance of PriorityParameters, it gives the
maximum base priority any task in this group.

Returns

the scheduling parameter instance denoting the upper bound on the scheduling
eligibility of threads in this group, null when no such bound has been specified.

setMaxEligibility(SchedulingParameters)

Signature

public javax.realtime.RealtimeThreadGroup
setMaxEligibility(SchedulingParameters parameters)
throws IllegalRealtimeStateException
RealtimeThreadGroup

Description
Sets the upper bound on scheduling eligibility that tasks in this group may have. For example, when it is an instance of PriorityParameters, it sets the maximum base priority any task in this group may have. When a task in the group has a higher eligibility than specified in parameters, the task’s eligibility is silently set to the max specified in parameters.

When the new eligibility is higher than that of any parent’s eligibility, then eligibility is set to the minimum of those priorities. When a child of this RealtimeThreadGroup has a higher max eligibility than specified in parameters, its max eligibility is silently set to the max specified in parameters as if setMaxEligibility were invoked on it recursively.

When a task in this RealtimeThreadGroup or a child of this RealtimeThreadGroup has previously had its maximum eligibility reduced by a call to this method, setting a higher maximum eligibility via this method will not automatically reraise its eligibility.

Parameters
parameters—The SchedulingParameter instance denoting the new upper bound on the scheduling eligibility of threads in this group.

Throws
IllegalRealtimeArgumentException—when parameters are not consistent with the scheduler type. The scheduler specified must be specific enough that only mutually compatible SchedulingParameters could be set. For example, Scheduler is not sufficient to restrict the scheduling parameters to compatible types, but PriorityScheduler does since all PriorityScheduler instances require PriorityParameters.

IllegalRealtimeStateException—when parameters is a higher eligibility than the max eligibility enforced by a SchedulingParameters above this in the hierarchy.

Returns
this

visitConventionalThreads(Consumer, boolean)

Signature
public void
visitConventionalThreads(java.util.function.Consumer<java.lang.Thread> visitor,
boolean recurse)
throws ForEachTerminationException

RTSJ 2.0 (Draft 66) 189
Description

Visit all `java.lang.Thread` instances that are not instances of `Schedulable` contained by this group and optionally all `ThreadGroup` instances contained within recursively.

Parameters

- `visitor`—A consumer of each schedulable instance.
- `recurse`—A boolean to indicated that the visit should be recursive.

Throws

- `ForEachTerminationException`—

`visitConventionalThreads(Consumer)`

Signature

```java
public void visitConventionalThreads(java.util.function.Consumer<java.lang.Thread> visitor)
throws ForEachTerminationException
```

Description

The same as `visitConventionalThreads(Consumer, boolean)` with `recurse` set to `false`.

Parameters

- `visitor`—A consumer of each conventional thread instance

Throws

- `ForEachTerminationException`—when the visitor is prematurely ended.

`visitSchedulables(Consumer, boolean)`

Signature

```java
public void visitSchedulables(java.util.function.Consumer<Schedulable> visitor, boolean recurse)
throws ForEachTerminationException
```

Description

Visit all `Schedulable` instances contained by this group and optionally all `ThreadGroup` instances contained within recursively.

Parameters

- `visitor`—A consumer of each schedulable instance.
- `recurse`—A boolean to indicated that the visit should be recursive.

Throws

- `ForEachTerminationException`—when the visitor is prematurely ended.
visitor—A consumer of each schedulable instance.
recurse—A boolean to indicated that the visit should be recursive.

Throws
ForEachTerminationException—

visitSchedulables(Consumer)

Signature
public void
visitSchedulables(java.util.function.Consumer<Schedulable> visitor)
throws ForEachTerminationException

Description
The same as visitSchedulables(Consumer, boolean) with recurse set to false.

Parameters
visitor—A consumer of each schedulable instance

Throws
ForEachTerminationException—when the visitor is prematurely ended.

visitChildren(Consumer)

Signature
public void
visitChildren(java.util.function.Consumer<java.lang.ThreadGroup> visitor)
throws ForEachTerminationException

Description
Performs some operation on all the children of the current group. The traversal
of the children continues as long as visitor does not throw a ForEachTerminationException. Thus the traversal can be prematurely ended by visitor
throwing this exception, e.g., when a particular element is found.

Parameters
visitor—The function to be called on each child thread group.

Throws
ForEachTerminationException—when the traversal ends prematurely.
visitChildren(Consumer, boolean)

Signature

public void
visitChildren(java.util.function.Consumer<java.lang.ThreadGroup> visitor,
boolean recursive)

Description

Performs some operation on all the children of the current group. The traversal
of the children continues as long as visitor does not throw a ForEachTerminationException. Thus the traversal can be prematurely ended by visitor throwing this exception, e.g., when a particular element is found.

Parameters

visitor—The function to be called on each child thread group.
recursive—A boolean to determine whether or not all subgroups are included,
where true means yes and false means no.

Throws

ForEachTerminationException—when the traversal ends prematurely.

6.3.3.11 ReleaseParameters

public abstract class ReleaseParameters

Inheritance

java.lang.Object
    ReleaseParameters

Interfaces

Cloneable
Serializable

Description

The top-level class for release characteristics used by Schedulable. 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 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, the parameter object is used in `setReleaseParameters()`, or the object is used in a constructor for a schedulable.

Release parameters use `HighResolutionTime` values for cost, and deadline. Since the times are expressed as `HighResolutionTime` values, these values use accurate timers with nanosecond granularity. The actual precision 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` 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.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>cost</td>
<td>new RelativeTime(0,0)</td>
</tr>
<tr>
<td>deadline</td>
<td>no default</td>
</tr>
<tr>
<td>overrunHandler</td>
<td>None</td>
</tr>
<tr>
<td>missHandler</td>
<td>None</td>
</tr>
<tr>
<td>rousable</td>
<td>false</td>
</tr>
<tr>
<td>initial event queue length</td>
<td>0</td>
</tr>
</tbody>
</table>

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

6.3.3.11 Fields

DISABLE_MONITORING

public static final RelativeTime DISABLE_MONITORING

Description
A special value for cost for turning off cost monitoring. This is just a notification
to the VM that the application does not require cost monitoring for a give instance
of Schedulable. What the VM does with it is system dependent; though, when
a cost is so set, the application cannot rely on any cost tracking that involves
said instance.

6.3.3.11.2 Constructors

ReleaseParameters(RelativeTime, RelativeTime, AsyncEventHandler, AsyncEventHandler)

Signature
protected
ReleaseParameters(RelativeTime cost,
    RelativeTime deadline,
    AsyncEventHandler overrunHandler,
    AsyncEventHandler missHandler)

Description
Creates a new instance of ReleaseParameters with the given parameter values.

Parameters
cost—Processing time units per release. On implementations which can measure
the amount of time an instance of schedulable is executed, when null, the
default value is a new instance of RelativeTime(0, 0) meaning that no cost
enforcement will take place. Setting it to DISABLE_MONITORING disables cost
monitoring as well.
**ReleaseParameters** — 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**

- **IllegalRealtimeArgumentException** — 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 chronograph associated with the cost or deadline parameters is not an instance of Clock.
- **IllegalAssignmentError** — when cost, deadline, overrunHandler, or missHandler cannot be stored in this.

---

**ReleaseParameters**

**Signature**

```java
protected ReleaseParameters()
```

**Description**

Equivalent to `ReleaseParameters(RelativeTime, RelativeTime, AsyncEventHandler, AsyncEventHandler)` with the argument list (null, null, null, null).

---

**6.3.3.11.3 Methods**

---

**clone**

**Signature**

```java
public java.lang.Object clone()
```
Description
Obtains 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

getcost

Signature
public javax.realtime.RelativeTime
getcost()

Description
Determines the current value of cost. A value of RelativeTime(0,0) meaning that no cost enforcement will take place; whereas a value of DISABLE_MONITORING means cost monitoring is disabled.

Returns
the object last used to set the cost containing the current value of cost.

getcost(RelativeTime)

Signature
public javax.realtime.RelativeTime
getcost(RelativeTime value)

Description
Determines the current value of cost, where Relative(0,0) means no cost enforcement in being done and DISABLE_MONITORING means cost monitoring is disabled as well.

Parameters
value—The parameter in which to return the cost.

Returns
196 RTSJ 2.0 (Draft 66)
value or, when null, the last object used to set the cost, set to the current value of cost.

Available since RTSJ 2.0

**getCostOverrunHandler**

*Signature*

```java
public javax.realtime.AsyncEventHandler getCostOverrunHandler()
```

*Description*

Gets a reference to the cost overrun handler.

*Returns*

a reference to the associated cost overrun handler.

**getDeadline**

*Signature*

```java
public javax.realtime.RelativeTime getDeadline()
```

*Description*

Determines the current value of deadline.

*Returns*

the object last used to set the deadline containing the current value of deadline.

**getDeadline(RelativeTime)**

*Signature*

```java
public javax.realtime.RelativeTime getDeadline(RelativeTime value)
```

*Description*

Determines the current value of deadline.

*Parameters*

value—The parameter in which to return the deadline.
6 Scheduling

**Returns**
value or, when null, the last object used to set the deadline, set to the current value of deadline.

**Available since** RTSJ 2.0

**getDeadlineMissHandler**

**Signature**
```java
public javax.realtime.AsyncEventHandler
getDeadlineMissHandler()
```

**Description**
Gets a reference to the deadline miss handler.

**Returns**
a reference to the deadline miss handler.

**setCost(RelativeTime)**

**Signature**
```java
public javax.realtime.ReleaseParameters
setCost(RelativeTime cost)
```

**Description**
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)`) the cost of those schedulables takes effect immediately.

**Parameters**
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**

198 RTSJ 2.0 (Draft 66)
IllegalRealtimeArgumentException—when the time value of cost is less than zero, or the clock associated with the cost parameters is not the realtime clock.
IllegalAssignmentError—when cost cannot be stored in this.

Returns
this

Available since RTSJ 2.0 returns itself

setCostOverrunHandler(AsyncEventHandler)

Signature
public javax.realtime.ReleaseParameters
setCostOverrunHandler(AsyncEventHandler handler)
throws IllegalAssignmentError

Description
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)) the cost overrun handler of those schedulables is effective immediately.

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.

Returns
this

Available since RTSJ 2.0 returns itself

setDeadline(RelativeTime)

Signature
public javax.realtime.ReleaseParameters
setDeadline(RelativeTime deadline)
Description

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)`) the deadline of those schedulables take effect at completion.

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

`IllegalRealtimeArgumentException`—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.

Returns

this

Available since RTSJ 2.0 returns itself

setDeadlineMissHandler(AsyncEventHandler)

Signature

public javax.realtime.ReleaseParameters
setDeadlineMissHandler(AsyncEventHandler handler)

Description

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)`) the deadline miss handler of those schedulables take effect at completion.

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.
Returns

this

Available since RTSJ 2.0 returns itself

isRousable

Signature

public boolean isRousable()

Description

Determines whether or not a thread interrupt will cause instances of Schedulable associated with an instance of this class to be prematurely released, i.e., released before the very first release event happens. It has no effect for periodic realtime threads, since the first event of a timing is when start is called. The default value, i.e., before any call to setRousable(boolean), is false.

Note that the rousable state has no effect on instances of RealtimeThread which have an instance of BackgroundParameters for ReleaseParameters or on ordinary event handlers, i.e., those which do not extend ActiveEvent. In the former case, there are no releases to interrupt and, in the case, the handler does not have a ActiveEventDispatcher to release it.

Returns

true when rousable and false when not.

Available since RTSJ 2.0

setRousable(boolean)

Signature

public javax.realtime.ReleaseParameters setRousable(boolean value)

Description

Dictates whether or not a thread interrupt will cause instances of Schedulable associated with an instance of this class to be prematurely released, i.e., released before the very first release event happens.

Parameters
value—When rousable, true and false when not.

Returns
this

Available since RTSJ 2.0

enforcingCost

Signature
public boolean
   enforcingCost()

Description
Determines whether or not cost is being enforced for releases.

Returns
true when enforcing code.

Available since RTSJ 2.0

enforceCost(boolean)

Signature
public void
   enforceCost(boolean value)
   throws UnsupportedRealtimeOperationException

Description
Sets cost enforcement.

Parameters
value—true to enforce code, false to turn off enforcement.

Throws
UnsupportedRealtimeOperationException—when cost enforcement is not supported on this platform.

Available since RTSJ 2.0
getEventQueueOverflowPolicy

Signature
public javax.realtime.QueueOverflowPolicy
getEventQueueOverflowPolicy()

Description
Gets the behavior of the arrival time queue in the event of an overflow.

Returns
the behavior of the arrival time queue.

Available since RTSJ 2.0

setEventQueueOverflowPolicy(QueueOverflowPolicy)

Signature
public javax.realtime.ReleaseParameters
setEventQueueOverflowPolicy(QueueOverflowPolicy policy)

Description
Sets the policy for the arrival time queue for when the insertion of a new element
would make the queue size greater than the initial size given in this.

Parameters
policy—A queue overflow policy to use for handlers associated with this.

Returns
this

Available since RTSJ 2.0

getInitialQueueLength

Signature
public int
getInitialQueueLength()

Description
RTSJ 2.0 (Draft 66)
6 Scheduling

RoundRobinParameters

Gets the initial number of elements the event 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.

**Returns**
the initial length of the queue.

**Available since** RTSJ 2.0 replaces the subclass method `AperiodicParameters.getInitialArrivalTimeQueueLength()`.

**setInitialQueueLength(int)**

**Signature**
```java
public javax.realtime.ReleaseParameters setInitialQueueLength(int initial)
```

**Description**
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. The queue may have a length of zero, i.e., any event, along with its arrival time, received during a previous release is lost.

**Parameters**
- initial—The initial length of the queue.

**Throws**
- `IllegalRealtimeArgumentException`—when initial is less than zero.

**Returns**
this

**Available since** RTSJ 2.0 replaces the subclass method `AperiodicParameters.setInitialArrivalTimeQueueLength(int)`.

6.3.3.12 RoundRobinParameters

public class RoundRobinParameters

**Inheritance**

RTSJ 2.0 (Draft 66)
java.lang.Object
SchedulingParameters
PriorityParameters
RoundRobinParameters

Description
Same as PriorityParameters except that it is only valid with the RoundRobinScheduler.

Available since RTSJ 2.0

6.3.3.12.1 Constructors

RoundRobinParameters(int, Affinity)

Signature
public
RoundRobinParameters(int priority,
Affinity affinity)

Description
Create scheduling parameters restricted to the round robin scheduler.

Parameters
priority—The priority assigned to schedulables that use this parameter instance.
affinity—The affinity assigned to schedulables that use this parameter instance.

6.3.3.12.2 Methods

isCompatible(Scheduler)

Signature
public boolean
isCompatible(Scheduler scheduler)
6 Scheduling

RoundRobinScheduler

Description
Determine whether this scheduling parameters can be used by tasks scheduled by scheduler.

Parameters
scheduler—The scheduler to check against

Returns
true when and only when this can be used with scheduler as the scheduler.

Available since RTSJ 2.0

subsumes(SchedulingParameters)

Signature
public boolean
subsumes(SchedulingParameters other)

Description
Determine whether this parameters is more eligible than another.

Parameters
other—The other parameters object to be compared with.

Returns
ture when and only when this parameters is more eligible than the other parameters.

6.3.3.13 RoundRobinScheduler

public class RoundRobinScheduler

Inheritance
java.lang.Object
Scheduler
PriorityScheduler
RoundRobinScheduler

Description
Class which represents a priority-based round-robin scheduler.
The default instance of this scheduler (returned by instance()) represents the RTSJ-specified round-robin scheduler.

Available since RTSJ 2.0

206 RTSJ 2.0 (Draft 66)
6.3.3.13.1 Methods

instance

Signature
   public static javax.realtime.RoundRobinScheduler
        instance()

Description
   Gets a reference to the distinguished instance of RoundRobinScheduler which is
   the RTSJ-specified round-robin scheduler.

Throws
   UnsupportedRealtimeOperationException—if this platform has no default round-
   robin scheduler.

Returns
   a reference to the distinguished instance of RoundRobinScheduler

setQuantum(RelativeTime)

Signature
   public javax.realtime.RoundRobinScheduler
        setQuantum(RelativeTime quantum)
        throws UnsupportedRealtimeOperationException,
                IllegalRealtimeArgumentException

Description
   Sets the quantum of this instance of RoundRobinScheduler. This takes effect at
   the end of the current quantum.

Parameters
   quantum—The new quantum to use. Copy semantics are used for this argument,
            and future changes to quantum will not affect this scheduler unless it is again
            passed to setQuantum().

Throws
   UnsupportedRealtimeOperationException—if this scheduler’s quantum is not
   configurable at runtime.
IllegalRealtimeArgumentException—if the provided quantum is null, less than zero, or not appropriate for this platform.

Returns
this

goingQuantum

Signature
public javax.realtime.RelativeTime getQuantum()

Description
Gets the quantum of this instance of RoundRobinScheduler.

Returns
a newly-allocated RelativeTime containing the currently-configured quantum of this scheduler.

goingQuantum(RelativeTime)

Signature
public javax.realtime.RelativeTime getQuantum(RelativeTime dest)

Description
Gets the quantum of this instance of RoundRobinScheduler.

Parameters
dest—A time object in which to hold the quantum. When dest is null, a new RelativeTime instance is allocated to hold the returned value.

Returns
the currently-configured quantum of this scheduler.

getMaxPriority

Signature
public int getMaxPriority()
RoundRobinScheduler

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

Returns
the value of the maximum priority.

getMinPriority

Signature
public int
getMinPriority()

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

Returns
the minimum priority used by this scheduler.

getNormPriority

Signature
public int
getNormPriority()

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

Returns
the value of the normal priority.

getPolicyName

Signature
public java.lang.String
getPolicyName()

Description
Gets the policy name of this.

Returns
the policy name (Fixed Priority Round Robin) as a string.

RTSJ 2.0 (Draft 66) 209
reschedule(Thread, SchedulingParameters)

Signature

public void
reschedule(Thread thread,
            SchedulingParameters eligibility)

Description

Promotes a java.lang.Thread to realtime priority under this scheduler. The affected thread will be scheduled as if it were a RealtimeThread with the given eligibility. This does not make the affected thread a RealtimeThread, however, and it will not have access to facilities reserved for instances of RealtimeThread. Instances of RealtimeThread will be treated as if their scheduling parameters were set to eligibility.

Parameters

thread—The thread to promote to realtime scheduling.

eligibility—A SchedulingParameters instance such as PriorityParameters for a PriorityScheduler.

Throws

IllegalRealtimeArgumentException—when eligibility is not valid for the scheduler.

IllegalRealtimeStateException—when eligibility specifies parameters that are out of range for the scheduler or the threads state or the intersection of affinity in scheduling and the affinity of realtime thread group associated with thread is empty.

UnsupportedRealtimeOperationException—when thread a normal Java thread and the scheduler does not support promoting normal java threads.

Available since RTSJ 2.0

6.3.3.14 Scheduler

public abstract class Scheduler

Inheritance

java.lang.Object
    Scheduler

Description
An instance of Scheduler manages the execution of schedulables. 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, e.g., EDFScheduler.

### 6.3.3.14.1 Constructors

Scheduler

Signature

```
protected
Scheduler()
```

Description

Creates an instance of Scheduler.

### 6.3.3.14.2 Methods

getDefaultScheduler

Signature

```
public static javax.realtime.Scheduler
ggetDefaultScheduler()
```

Description

Gets a reference to the default scheduler.

Returns

a reference to the default scheduler.
6 Scheduling

setDefaultScheduler(Scheduler)

Signature

```java
public static void
setDefaultScheduler(Scheduler scheduler)
```

Description

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` at startup.

Parameters

- `scheduler`—The `Scheduler` that becomes the default scheduler assigned to new
  schedulables created by Java threads. When `null` nothing happens.

Throws

- `RealtimeSecurityException`—when the caller is not permitted to set the default
  scheduler.

inSchedulableExecutionContext

Signature

```java
public static boolean
inSchedulableExecutionContext()
```

Description

Determines whether the current calling context is a `Schedulable`: `RealtimeThread` or `AsyncBaseEventHandler`.

Returns

- `true` when yes and `false` otherwise.

Available since RTSJ 2.0

currentSchedulable

Signature

```java
public static javax.realtime.Schedulable
currentSchedulable()
```

Description

212 RTSJ 2.0 (Draft 66)
Scheduler

The current execution context when called from a Schedulable execution context.

Throws

ClassCastException—when the caller is not a Schedulable

Returns

the current Schedulable.

Available since RTSJ 2.0

gPolicyName

Signature

public abstract java.lang.String
gPolicyName()

Description

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 when accessed.

Returns

a String object which is the name of the scheduling policy used by this.

reschedule(Thread, SchedulingParameters)

Signature

public abstract void
reschedule(Thread thread,
SchedulingParameters eligibility)

Description

Promotes a java.lang.Thread to realtime priority under this scheduler. The
affected thread will be scheduled as if it were a RealtimeThread with the given
eligibility. This does not make the affected thread a RealtimeThread, however,
and it will not have access to facilities reserved for instances of RealtimeThread.
Instances of RealtimeThread will be treated as if their scheduling parameters
were set to eligibility.
Parameters

thread—The thread to promote to realtime scheduling.

eligibility—A SchedulingParameters instance such as PriorityParameters for a PriorityScheduler.

Throws

IllegalRealtimeArgumentException—when eligibility is not valid for the scheduler.

IllegalRealtimeStateException—when eligibility specifies parameters that are out of range for the scheduler or the threads state or the intersection of affinity in scheduling and the affinity of realtime thread group associated with thread is empty.

UnsupportedRealtimeOperationException—when thread a normal Java thread and the scheduler does not support promoting normal Java threads.

Available since RTSJ 2.0

6.3.3.15 SchedulingParameters

public class SchedulingParameters

Inheritance

java.lang.Object

SchedulingParameters

Interfaces

Cloneable

Serializable

javax.realtime.Subsumable

Description

Subclasses of SchedulingParameters (PriorityParameters, ImportanceParameters, and any others defined for particular schedulers) provide the parameters to be used by the Scheduler. Changes to the values in a parameters object affects the scheduling behavior of all the Schedulable objects to which it is bound.

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

**SchedulingParameters(Affinity)**

*Signature*

```java
protected SchedulingParameters(Affinity affinity)
```

*Description*

Creates a new instance of `SchedulingParameters`.

*Available since* RTSJ 2.0

*Parameters*

- `affinity`—Sets the affinity for these parameters.

**SchedulingParameters**

*Signature*

```java
protected SchedulingParameters()
```

*Description*

Creates a new instance of `SchedulingParameters`. It is the same as `SchedulingParameters(Affinity.default())`.

*Available since* RTSJ 1.0.1

6.3.3.15.2 Methods

**clone**

*Signature*

```java
RTSJ 2.0 (Draft 66) 215
```
public java.lang.Object clone()

Description

Creates a clone of this. This method should behave effectively as if constructed with a clone of the affinity value used to 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 it affinity value (deep copy).

Available since RTSJ 1.0.1

isCompatible(Scheduler)

Signature

public boolean isCompatible(Scheduler scheduler)

Description

Determines whether this scheduling parameters can be used by tasks scheduled by scheduler.

Parameters

scheduler—The scheduler to check against

Returns

true when and only when this can be used with scheduler as the scheduler.

Available since RTSJ 2.0

subsumes(SchedulingParameters)

Signature

public boolean subsumes(SchedulingParameters other)

Description

Determine whether this parameters is more eligible than another.

Parameters

216 RTSJ 2.0 (Draft 66)
other—The other parameters object to be compared with.

Returns
true when and only when this parameters is more eligible than the other parameters.

getAffinity

Signature
    public javax.realtime.Affinity
    getAffinity()

Description
Determine the affinity set instance associated of these parameters.

Returns
The associated affinity.

Available since RTSJ 2.0

6.3.3.16 SporadicParameters

public class SporadicParameters

Inheritance
java.lang.Object
    ReleaseParameters
    AperiodicParameters
    SporadicParameters

Description
A notice to the scheduler that the associated schedulable 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` 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 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, the parameter object is used in `setReleaseParameters()`, or the object is used in a constructor for an SO.

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

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>minInterarrival time</td>
<td>No default. A value must be supplied</td>
</tr>
<tr>
<td>cost</td>
<td>new <code>RelativeTime(0,0)</code></td>
</tr>
<tr>
<td>deadline</td>
<td>new <code>RelativeTime(mit)</code></td>
</tr>
<tr>
<td>overrunHandler</td>
<td>None</td>
</tr>
<tr>
<td>missHandler</td>
<td>None</td>
</tr>
<tr>
<td>rousable</td>
<td>false</td>
</tr>
<tr>
<td>MIT violation policy</td>
<td>SAVE</td>
</tr>
<tr>
<td>Arrival queue overflow policy</td>
<td>SAVE</td>
</tr>
<tr>
<td>Initial arrival queue length</td>
<td>0</td>
</tr>
</tbody>
</table>

This class enables the application to specify one of four arrival behaviors defined by `MinimumInterarrivalPolicy`. Each behavior indicates what to do when an arrival occurs that is closer in time to the previous arrival than the value given in this class as minimum interarrival time. They also specify what to do when, for any reason, the queue overflows, and what the initial size of the queue should be.

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

**SporadicParameters(RelativeTime, RelativeTime, RelativeTime, AsyncEventHandler, AsyncEventHandler, boolean)**

**Signature**

```java
public
SporadicParameters(RelativeTime minInterarrival,
                    RelativeTime cost,
                    RelativeTime deadline,
                    AsyncEventHandler overrunHandler,
                    AsyncEventHandler missHandler,
                    boolean rousable)
```

**Description**

Creates a `SporadicParameters` object.

**Available since** RTSJ 2.0

**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 affect 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 `minInterarrival: new RelativeTime(minInterarrival)`.

`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
Scheduling

SporadicParameters

handler will be used.

rousable—Determines whether or not an instance of Schedulable can be prematurely released by a thread interrupt.

Throws

IllegalRealtimeArgumentException—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 chronograph associated with deadline and minInterarrival parameters are not identical or not an instance of Clock.

IllegalAssignmentError—when minInterarrival, cost, deadline, overrun-Handler or missHandler cannot be stored in this.

SporadicParameters(RelativeTime, RelativeTime, Relative-Time, AsyncEventHandler, AsyncEventHandler)

Signature

public
SporadicParameters(RelativeTime minInterarrival,
    RelativeTime cost,
    RelativeTime deadline,
    AsyncEventHandler overrunHandler,
    AsyncEventHandler missHandler)

Description

Equivalent to SporadicParameters(RelativeTime, RelativeTime, Rela-
tiveTime, AsyncEventHandler, AsyncEventHandler, boolean) with an argument list of (minInterarrival, cost, deadline, overrunHandler, mis-
shandler, false).

SporadicParameters(RelativeTime, RelativeTime, AsyncEvent-
Handler, boolean)

Signature

public
SporadicParameters(RelativeTime minInterarrival,
    RelativeTime deadline,
    AsyncEventHandler missHandler,
boolean rousable)

**Description**
Equivalent to `SporadicParameters(RelativeTime, RelativeTime, RelativeTime, AsyncEventHandler, AsyncEventHandler, boolean)` with an argument list of `(minInterarrival, null, deadline, null, missHandler, rousable).

**Available since** RTSJ 2.0

### SporadicParameters(RelativeTime)

**Signature**
```java
public SporadicParameters(RelativeTime minInterarrival)
```

**Description**
Equivalent to `SporadicParameters(RelativeTime, RelativeTime, RelativeTime, AsyncEventHandler, AsyncEventHandler, boolean)` with an argument list of `(minInterarrival, null, null, null, null, false).

**Available since** RTSJ 1.0.1

### 6.3.3.16.2 Methods

#### getMinimumInterarrival

**Signature**
```java
public javax.realtime.RelativeTime getMinimumInterarrival()
```

**Description**
Determines the current value of minimal interarrival.

**Returns**
the object last used to set the minimal interarrival containing the current value of minimal interarrival.
getMinimumInterarrival(RelativeTime)

Signature
   public javax.realtime.RelativeTime
   getMinimumInterarrival(RelativeTime value)

Description
   Determines the current value of minimum interarrival.

Returns
   value or, when null, the last object used to set the minimal interarrival, set to
   the current value of minimal interarrival.

Available since RTSJ 2.0

setMinimumInterarrival(RelativeTime)

Signature
   public javax.realtime.SporadicParameters
   setMinimumInterarrival(RelativeTime minimum)

Description
   Sets the minimum interarrival time.

Parameters
   minimum—The release times of the schedulable will occur no closer than this interval.

Throws
   IllegalRealtimeArgumentException—when minimum is null or its time value is
   not greater than zero.

   IllegalAssignmentError—when minimum cannot be stored in this.

Returns
   this

Available since RTSJ 2.0 returns itself

setMinimumInterarrivalPolicy(MinimumInterarrivalPolicy)

Signature

222   RTSJ 2.0 (Draft 66)
public javax.realtime.SporadicParameters
setMinimumInterarrivalPolicy(MinimumInterarrivalPolicy policy)

Description
Sets the policy for handling the arrival time queue when the new arrival time is
closer to the previous arrival time than the minimum interarrival time given in
this.

Parameters
policy—The current policy for MIT violations.
Available since RTSJ 2.0

getMinimumInterarrivalPolicy

Signature
public javax.realtime.MinimumInterarrivalPolicy
getMinimumInterarrivalPolicy()

Description
Gets the arrival time queue policy for handling minimal interarrival time under-
flow.

Returns
the minimum interarrival time violation behavior as a string.
Available since RTSJ 2.0

setEventQueueOverflowPolicy(QueueOverflowPolicy)

Signature
public javax.realtime.SporadicParameters
setEventQueueOverflowPolicy(QueueOverflowPolicy policy)
throws IllegalRealtimeArgumentException

Description
Sets the policy for the arrival time queue for when the insertion of a new element
would make the queue size greater than the initial size given in this.

Parameters
policy—The new overflow policy to use.
6 Scheduling

Throws

IllegalRealtimeArgumentException—when policy is QueueOverflowPolicy.

DISABLE.

Returns
this

6.4 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 as the base scheduler: FirstInFirstOutScheduler. The specification supplies a second scheduler, RoundRobinScheduler, for cases where time slicing behavior is desired. In POSIX terms, SCHED_FIFO meets the RTSJ requirements for the base scheduler, and SCHED_RR meets the requirements for the round-robin scheduler.

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.

### 6.4.1 RealtimeThreadGroup and ProcessingGroup

The `RealtimeThreadGroup` and `ProcessingGroup` classes were added in RTSJ 2.0 both to support the notion of a subsystem constrained by the greater system configuration and to generalize the existing notion of cost monitoring and enforcement for schedulables to groups of schedulables. In addition, they provide a way to enable Java threads to be elevated to realtime scheduling priorities in a controlled fashion.

A combination of security manager policy and the `RealtimeThreadGroup` hierarchy may be used to constrain the maximum priority directly configurable by an entire subsystem. To achieve this, a `RealtimeThreadGroup` with an appropriate maximum priority must be created, the security manager must be configured to disallow threads in that `RealtimeThreadGroup` from accessing their parent `RealtimeThreadGroup`, and all threads for the subsystem must be created in that `RealtimeThreadGroup`. This tactic may even be used recursively. Similar practice can be used with `ProcessingGroup` to constrain the maximum execution time allowable to a subsystem, or other properties configurable in a processing group.

As previously mentioned, a motivation for adding `RealtimeThreadGroup` as a subclass of `ThreadGroup` is to clarify the relationship between Java threads and realtime schedulers. In order to obtain realtime priorities, a Java thread must belong to a `RealtimeThreadGroup`. Its access to realtime scheduling is then restricted (with the exception of priority inversion avoidance protocols, which ignore such restrictions) by the configuration of its `RealtimeThreadGroup`. This enables Java threads to obtain realtime priorities in a controlled and predictable fashion. Likewise, realtime threads (but not necessarily other schedulables) may obtain nonrealtime conventional Java priorities by calling `Thread.setPriority()` on their `RealtimeThread` object. To start a realtime thread with a nonrealtime priority, this call must be made prior to the time at which the realtime thread is started.

A `ProcessingGroup` can also be used to apply cost monitoring and enforcement to a collection of standard Java threads. However, note that placing a Java thread directly in a `ProcessingGroup`, which is an instance of `RealtimeThreadGroup`, may allow it to obtain realtime priorities. This can be avoided by placing the Java threads in a `JavaThreadGroup` which is in turn the child of an appropriately-configured `ProcessingGroup` and applying security manager restrictions.
6 Scheduling

6.4.2 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. Multiprocessor systems have two main variants: Single Instruction, Multiple Data (SIMD) systems and Multiple Instruction, Multiple Data (MIMD) systems. Putting the first class aside, since it does not fit in well with the overall Java programming model\(^3\), there is still a good deal of variation to consider. Though most commercially available systems are symmetric multiprocessors (SMP) systems, nonuniform memory access (NUMA) systems need consideration as well.

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. The range of processors on which global scheduling is possible is dictated by the operating system. For SMP architectures, global scheduling across all 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 be performed by the OS. On these systems, global scheduling across all processors will not be possible.

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.

Being able to support a wide variety of multiprocessor systems has a direct impact on the support for multiprocessing.

1. Since affinity is a widely used concept for controlling multiprocessor systems, the specification adopts this notion.
2. In order to organize the API, affinity is modeled with its own class: Affinity.
3. An instance of Affinity is added to the SchedulingParameters class to avoid support for affinities to be distributed throughout the specification with a proliferation of new constructor methods.
4. The RealtimeThreadGroup has been added to provides affinity support for conventional Java threads without modifying the thread object’s visible API and provide a partitioning mechanism for affinities.
5. Since the set of affinities possible is system dependent, the affinity class provides

\(^3\)The lambda extension provides an intriguing paradigm for extending Java to supporting SIMD coprocessors.
an array of predefined affinities. They can be used either to reflect the scheduling arrangement of the underlying OS. A program is only allowed to dynamically create new affinities with cardinality of one.

6. To support external dynamic control over the set of processors allocated to a RTSJ program, the affinity class provides a means of notifying when the processor set changes.

### 6.4.3 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 \( T + \delta \). \( \delta \) is the difference between \( T \) and the first time the timer clock advances to \( T_0 \), where \( T_0 \geq T \). The upper bound of \( \delta \) 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 \( T_0 \) 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 \( \delta \) for the realtime clock.

### 6.4.4 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 \( D + \delta \). When 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 when a schedulable has not completed the computation associated with its release before its deadline.
This completion event is signaled in the application code by the return of the `handleAsyncEvent` method or a call to `waitForNextRelease` 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 signaled when arguably the application had performed all of its computation.
Chapter 7
Synchronization

One of the strengths of Java is its language support for multithreading. This requires synchronization. In a realtime system, there are additional requirements on this synchronization. Therefore this specification not only tightens the semantics of the synchronization declarations, but it also provides addition classes that specifically manage synchronization.

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 and PriorityCeilingEmulation avoid unbounded priority inversions, which would be unacceptable in realtime systems.

The classes described below provide two main services.
1. They enable the setting of a priority inversion control policy either as the default or for specific objects.
2. They also provide wait-free communication between schedulables (especially instances of Schedulable, whose mayUseHeap is false) and regular Java threads.

These classes 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 schedulables without heap access and schedulables subject to garbage collection delays.

7.1 Definitions

Scheduling Eligibility Inversion — When a more important task is blocked by a less important task. This is usually caused by synchronization, where a
more important task must wait for a less important task to release a required resource, which can in turn be blocked by a task of intermediate importance. The classical example is priority inversion in a system with a priority-based scheduler.

**Governed by** — An object \( A \) that has been assigned (either by default or via an explicit method call) to the `MonitorControlPolicy` \( \alpha \) is said to be *governed by* \( \alpha \).

**Active Priority** — The priority of a task used for scheduling at any given time. It is the maximum of the tasks’s current base priority and any priority boosting due to priority inversion avoidance mechanisms. The base priority can be temporarily reduced by cost enforcement.

### 7.2 Semantics

Synchronization semantics has two main aspects: monitor control and scheduling. The first determines which inversion avoidance is to use. The second determines how it is done. Since only priority-based schedulers are defined in the RTSJ, the semantics is only completely defined for priority-based schedulers.

#### 7.2.1 Monitor Control

The specification provides for two monitor control policies with the following semantics.

1. The initial default monitor control policy shall be `PriorityInheritance`. The default policy can be altered by using the `setMonitorControl()` method.
2. 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`.
3. The `PriorityCeilingEmulation` monitor control policy is also required.
4. An implementation that provides any additional `MonitorControl` subclasses must document their effects, particularly with respect to priority inversion control.
5. An object’s monitor control policy affects *each* task that attempts to lock the object; i.e., regular Java threads as well as schedulables.
6. When a task enters synchronized code, the target object’s monitor control policy must be supported by the thread schedulable’s scheduler; otherwise an `IllegalTaskStateException` is thrown. An implementation that defines a new `MonitorControl` subclass must document which schedulers, if any, do not support this policy.
7. Since priorities in the interrupt priority range must be implemented by masking hardware interrupts, a thread which enters a monitor with an interrupt priority as its ceiling will cause the corresponding hardware interrupts to be masked until the monitor is exited.

7.2.2 Priority Schedulers

The two schedulers provided by the RTSJ must both handle synchronization in the same way. All tasks governed by these schedulers are subject to the following semantics when they synchronize on objects governed by monitor control policies defined in this section.

1. Each task has a base priority and an active priority. A task that holds a lock on a PCE-governed object also has a ceiling priority.
2. The base priority for a task is limited by the maximum priority of its realtime thread groups’ maximum scheduling parameters.
3. The active priority for a task is independent of its realtime thread groups.
4. The base priority for a task \( t \) 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:
   (a) \( pparam.setPriority(prio) \), where \( t \) is a schedulable with \( pparams \) as its SchedulingParameters and \( pparams \) is an instance of PriorityParameters or one of its subclasses, where the new base priority is \( prio \);
   (b) \( t.setSchedulingParameters(pparams) \), where \( t \) is a schedulable and \( pparams \) is an instance of PriorityParameters, where the new base priority is \( pparams.getPriority() \);
   (c) \( t.setPriority(prio) \), where \( t \) is a schedulable object the new base priority is \( prio \), and when it is a Java thread the new base priority is the lesser of \( prio \) and the maximum priority for \( t \)’s thread group; and
   (d) \( sg.setMaxEligibility(pparams) \), where \( sg \) is in \( t \)’s RealtimeThreadGroup hierarchy and the priority of \( pparams \) is less than the current base priority of \( t \), where the new base priority is the priority specified in \( pparams \) as a result of setting the task’s scheduling parameters to \( pparams \).
5. When the task \( t \) 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 the new priority is higher than the old priority, and at the beginning otherwise.
6. When task \( t \) 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
The priority sources resulting from the monitor control policies defined in this section, and their associated priorities for a schedulable t, are as follows:

(a) **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).

(b) **Source** Each object locked by t and governed by a PriorityCeilingEmulation policy
**Associated Priority** The maximum value ceil, where ceil is the ceiling of a PriorityCeilingEmulation policy governing an object locked by t.
**Note** This value is also referred to as the ceiling priority for t.

(c) **Source** Each task 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.

(d) **Source** Each task 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.

7. The addition of a priority source for t either leaves t’s active priority unchanged, or increases it. When t’s active priority is unchanged, t’s status in its relevant queue(s), e.g., blocked waiting for some object, is not affected. When t’s active priority is increased, t is placed at the tail of the relevant queue(s) at its new active priority level.

8. The removal of a priority source for t either leaves t’s active priority unchanged, or decreases it. When t’s active priority is unchanged, then t’s status in its
relevant queue, e.g., blocked waiting for some object, is not affected. When \( t \)'s active priority is decreased and \( t \) is either ready or running, then \( t \) must be placed at the head of the ready queue at its new active priority level. When \( t \)'s active priority is decreased and \( t \) is blocked, then \( t \) is queued at the end of the queue for the new priority when it becomes unblocked.

The above rules have four main consequences.

1. A thread or schedulable \( t \)'s priority sources from 6b are added and removed synchronously; i.e., they are established based on \( t \)'s entering or leaving synchronized code. However, priority sources from 6a, 6c, and 6d may be added and removed asynchronously, as an effect of actions by other threads or schedulables.

2. A task holding only one lock, when it releases this lock, has its active priority set to its base priority.

3. A task’s active priority is never less than its base priority.

4. When a task blocks at a call of \texttt{obj.wait()}, it releases the lock on \texttt{obj} and hence relinquishes the priority source(s) based on \texttt{obj}'s monitor control policy. The task will be queued at a new active priority that reflects the loss of these priority sources.

When modifying the active priority of a task, the active priority may exceed the priority range of the task’s scheduler. For example, a thread scheduled on the standard Java scheduler may be assigned a priority greater than 10, or a thread scheduled on the round robin scheduler may be assigned a priority greater than the round robin maximum priority but within the default scheduler priority range. In both cases, the task will be rescheduled on the default scheduler until its active priority is once again within the range schedulable on its associated scheduler. A task scheduled on the round robin scheduler, however, need not be moved to the default scheduler while its active priority remains within the allowable range for the round robin scheduler. Any scheduler not defined in this standard must specify the behavior of tasks associated with it with respect to these priority-based monitor control policies.

Since base priorities may be shared (i.e., the same \texttt{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 \texttt{obj} governed by a \texttt{PriorityCeilingEmulation} policy with ceiling \texttt{ceil}, then \( t \)'s active priority may exceed \texttt{ceil} but \( t \)'s base priority must not. In contrast, once \( t \) has successfully synchronized on \texttt{obj}, then \( t \)'s base priority may also exceed \texttt{obj}'s monitor control policy's ceiling. Note that either or both of \( t \)'s base priority and \texttt{obj}'s monitor control policy may have been dynamically modified.
7 Synchronization

7.2.3 Additional Schedulers

Schedulers based on criteria other than priority, for example, deadline in a deadline first scheduler, must consider how synchronization is handled to avoid scheduling eligibility inversion. Such a scheduler must conform to the following semantics for tasks managed by that scheduler when they synchronize on objects with the monitor control policies defined above.

1. An implementation that defines a new Scheduler subclass must document which (if any) monitor control policies the new scheduler does not support.
2. An implementation must document how, if at all, the semantics of synchronization differ from the rules defined for the default PriorityInheritance instance and for the PriorityCeilingEmulation policy. It must supply documentation for the behavior of the new scheduler with priority inheritance and priority ceiling emulation protocol equivalent to the semantics for the default priority scheduler found in the previous section.
3. An implementation must also document the appropriate monitor control policy for use when objects are shared between tasks under the control of different schedulers.
4. The new Scheduler subclass must conform to the semantics for parameter values, release control, dispatching, and cost monitoring described in Section 6.2.1.
public abstract class MonitorControl

Inheritance

java.lang.Object
    MonitorControl

Description

Abstract superclass for all monitor control policies.

7.3.1.1.1 Constructors

MonitorControl

Signature

protected MonitorControl()

Description

Invoked from subclass constructors.

7.3.1.1.2 Methods

getMonitorControl(Object)

Signature

public static javax.realtime.MonitorControl getMonitorControl(Object obj)
7 Synchronization

MonitorControl

Description
Gets the monitor control policy of the given instance of Object.

Parameters
obj—The object being queried.

Throws
IllegalRealtimeArgumentException—when obj is null.

Returns
the monitor control policy of the obj parameter.

getMonitorControl

Signature
public static javax.realtime.MonitorControl getMonitorControl()

Description
Gets the current default monitor control policy.

Returns
the default monitor control policy object.

setMonitorControl(MonitorControl)

Signature
public static javax.realtime.MonitorControl setMonitorControl(MonitorControl policy)
throws IllegalRealtimeArgumentException,
UnsupportedRealtimeOperationException,
IllegalRealtimeStateException

Description
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 whose creation happens after the method completes, until either
1. a new “per-object” policy is set for that object, thereby altering the monitor control policy for a single object without changing the default policy, or
2. a new default policy is set.
Like the per-object method (see `setMonitorControl(Object, MonitorControl)`), the setting of the default monitor control policy occurs immediately, but may not be visible on all processors of a multicore system simultaneously.

**Parameters**
- `policy`—The new monitor control policy. When `null`, the default `MonitorControl` policy is not changed.

**Throws**
- `RealtimeSecurityException`—when the caller is not permitted to alter the default monitor control policy.
- `IllegalRealtimeArgumentException`—when `policy` is not in immortal memory.
- `UnsupportedRealtimeOperationException`—when `policy` is not a supported monitor control policy.

**Returns**
- the default `MonitorControl` policy in effect on completion.

Available since RTSJ 1.0.1 The return type is changed from void to `MonitorControl`.

### setMonitorControl(Object, MonitorControl)

**Signature**

```
public static javax.realtime.MonitorControl setMonitorControl(Object obj,  
                  MonitorControl policy)
```

**Description**

Immediately sets `policy` as the monitor control policy for `obj`.

Monitor control policy changes on a monitor that is actively contended may lead to queued or enqueuing tasks following either the old or new policy in an unpredictable fashion. Tasks enqueued after the monitor is released after a policy change will follow the new policy.

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`.

**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

IllegalRealtimeArgumentException—when obj is null or policy is not in immortal memory.

UnsupportedRealtimeOperationException—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.

Available since RTSJ 1.0.1 The return type has been changed from void to MonitorControl.

7.3.1.2 PriorityCeilingEmulation

public class PriorityCeilingEmulation

Inheritance

java.lang.Object
    MonitorControl
        PriorityCeilingEmulation

Description

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 \texttt{PriorityCeilingEmulation} policy with ceiling \( \text{ceil} \):

- \( t \)'s base priority does not exceed \( \text{ceil} \)
- \( t \)'s ceiling priority (when \( t \) is holding any other \texttt{PriorityCeilingEmulation} locks) does not exceed \( \text{ceil} \).

Thus for any object \( \text{targetObj} \) that will be governed by priority ceiling emulation, the programmer needs to provide (via \texttt{MonitorControl.setMonitorControl(Object, MonitorControl)} a \texttt{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 \( \text{targetObj} \)
- the maximum ceiling priority value that any task may have when it attempts to synchronize on \( \text{targetObj} \).

More formally,

- when a thread or schedulable \( t \), whose base priority is \( p_1 \), attempts to synchronize on an object governed by a \texttt{PriorityCeilingEmulation} policy with ceiling \( p_2 \), where \( p_1 > p_2 \), then a \texttt{CeilingViolationException} is thrown in \( t \); likewise, a \texttt{CeilingViolationException} is thrown in \( t \) when \( t \) is holding a \texttt{PriorityCeilingEmulation} lock and has a ceiling priority exceeding \( p_2 \).

The values of \( p_1 \) and \( p_2 \) are passed to the constructor for the exception and may be queried by an exception handler.

A consequence of the above rule is that a thread or schedulable may nest synchronizations on \texttt{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 \texttt{PriorityInheritance} and \texttt{PriorityCeilingEmulation} policies requires one other piece of behavior in order to avoid unbounded priority inversions. When a thread or schedulable holds a \texttt{PriorityInheritance} lock, then any \texttt{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 \texttt{PriorityCeilingEmulation}-governed object with ceiling \( \text{ceil} \), then \( \text{ceil} \) must be within the priority range allowed by \( t \)'s scheduler; otherwise, an \texttt{IllegalTaskStateException} is thrown. Note that this does not prevent a regular Java thread from synchronizing on an object governed by a \texttt{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 PriorityInheritance`, and `CeilingViolationException`.

### 7.3.1.2.1 Methods

#### instance(int)

**Signature**

```
public static javax.realtime.PriorityCeilingEmulation instance(int ceiling)
```

**Description**

Creates 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.

**Parameters**

- `ceiling`—Priority ceiling value.

**Throws**

- `IllegalRealtimeArgumentException`—when `ceiling` is out of the range of permitted priority values (e.g., less than `PriorityScheduler.instance().getMinPriority()` or greater than `PriorityScheduler.instance().getMaxPriority()` for the base scheduler).

**Available since** RTSJ 1.0.1

#### getCeiling

**Signature**

```
public int getCeiling()
```

**Description**

Gets the priority ceiling for this `PriorityCeilingEmulation` object.
Returns
the priority ceiling.

Available since RTSJ 1.0.1

getMaxCeiling

Signature
public static javax.realtime.PriorityCeilingEmulation getMaxCeiling()

Description
Gets a PriorityCeilingEmulation object whose ceiling is PriorityScheduler.
instance().getMaxPriority(). 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.

Returns
a PriorityCeilingEmulation object whose ceiling is PriorityScheduler.
instance().getMaxPriority().

Available since RTSJ 1.0.1

7.3.1.3 PriorityInheritance

public class PriorityInheritance

Inheritance
java.lang.Object
    MonitorControl
        PriorityInheritance

Description
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 \texttt{t2} when either \texttt{t2} releases the lock on \texttt{obj}, or \texttt{t1} ceases attempting to synchronize on \texttt{obj} (e.g., when \texttt{t1} incurs an ATC).

2. Otherwise (i.e., \texttt{t1}'s active priority exceeds the maximum priority allowed by \texttt{t2}'s scheduler), an \texttt{IllegalTaskStateException} is thrown in \texttt{t1}.

Note on the second rule, throwing the exception in \texttt{t1}, rather than in \texttt{t2}, ensures that the exception is synchronous.

See also \texttt{MonitorControl} and \texttt{PriorityCeilingEmulation}

### 7.3.1.3.1 Methods

**instance**

**Signature**

```java
public static javax.realtime.PriorityInheritance instance()
```

**Description**

Obtains a reference to the singleton \texttt{PriorityInheritance}.

- This is the default \texttt{MonitorControl} policy in effect at system startup.
- The \texttt{PriorityInheritance} instance shall be allocated in \texttt{ImmortalMemory}.

### 7.3.1.4 WaitFreeReadQueue

**public class** \texttt{WaitFreeReadQueue<T>}

**Inheritance**

\texttt{java.lang.Object \rightarrow WaitFreeReadQueue<T>}

**Description**

A queue that can be non-blocking for consumers. The \texttt{WaitFreeReadQueue} class is intended for single-reader multiple-writer communication, although it may also be used (with care) for multiple readers. A \texttt{reader} is generally an instance of \texttt{Schedulable} which may not use the heap, and the \texttt{writers} are generally regular Java threads or heap-using instances of \texttt{Schedulable}. 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.

• 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 enabling instances of `Schedulable` that may not use the heap and conventional Java threads to synchronize on an object without the risk of that Schedulable instance 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.IllegalArgumentException`,
- `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.

### 7.3.1.4.1 Constructors

**WaitFreeReadQueue(Runnable, Runnable, int, Memory-Area, boolean)**

*Signature*

```java
public
WaitFreeReadQueue(Runnable writer,
                    Runnable reader,
                    int maximum,
```

RTSJ 2.0 (Draft 66) 243
MemoryArea memory,  
boolean notify)  
throws IllegalRealtimeArgumentException,  
MemoryScopeException,  
InaccessibleAreaException

Description

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 instances of Schedule 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.

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 in which internal elements are allocated.
notify—A flag that establishes whether a reader is notified when the queue becomes non-empty.

Throws

IllegalRealtimeArgumentException—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)**

**Signature**

```java
definition WaitFreeReadQueue(Runnable writer,
Runnable reader,
int maximum,
MemoryArea memory)
throws IllegalRealtimeArgumentException,
MemoryScopeException,
InaccessibleAreaException
```

**Description**

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.

Equivalent to `WaitFreeReadQueue(writer, reader, maximum, memory, false)`

**WaitFreeReadQueue(int, MemoryArea, boolean)**

**Signature**

```java
definition WaitFreeReadQueue(int maximum,
MemoryArea memory,
boolean notify)
throws IllegalRealtimeArgumentException,
InaccessibleAreaException
```

**Description**

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.
7 Synchronization

WaitFreeReadQueue

Equivalent to WaitFreeReadQueue(null, null, maximum, memory, notify)

Available since RTSJ 1.0.1

WaitFreeReadQueue(int, boolean)

Signature

public
WaitFreeReadQueue(int maximum,
   boolean notify)
   throws IllegalRealtimeArgumentException

Description

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.

Equivalent to WaitFreeReadQueue(null, null, maximum, null, notify)

Available since RTSJ 1.0.1

7.3.1.4.2 Methods

clear

Signature

public void
clear()

Description

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.
isNullOrEmpty

Signature
   public boolean
   isEmpty()

Description
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.

Returns
true when this is empty; false when this is not empty.

isFull

Signature
   public boolean
   isFull()

Description
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.

Returns
true when this is full; false when this is not full.

read

Signature
   public T
   read()

Description
   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.

Returns
   the instance of T read, or else null when this is empty.
size

Signature

```java
public int size()
```

Description

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.

Returns

the number of positions in this occupied by elements that have been written but not yet read.

waitForData

Signature

```java
public void waitForData()
```  

throws UnsupportedRealtimeOperationException, InterruptedException

Description

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).

Throws

UnsupportedRealtimeOperationException—when this has not been constructed with notify set to true.

InterruptedException—when the thread is interrupted by interrupt() or AsynchronouslyInterruptedException.fire() during the time between calling this method and returning from it.

Available since RTSJ 1.0.1 InterruptedException was added to the throws clause.
**WaitFreeWriteQueue**

### write(T)

**Signature**

```java
class WaitFreeWriteQueue<T> {
    public synchronized void write(T value)
        throws MemoryScopeException,
                InterruptedException
```

**Description**

A synchronized and blocking write. This call blocks on queue full and will wait until there is space in the queue.

**Parameters**

- `value` — The `java.lang.Object` that is placed in the queue.

**Throws**

- `InterruptedException` — when the thread is interrupted by `interrupt()` or `AsynchronouslyInterruptedException.fire()` 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.

**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.

---

### 7.3.1.5 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 `Schedulable` which may not use the heap, and the readers are generally conventional Java threads or instances of `Schedulable` which use the heap. 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 enabling schedulables which may not use the heap and regular Java threads to synchronize on an object without the risk of the schedulable 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.

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.

7.3.1.5.1 Constructors

WaitFreeWriteQueue(Runnable, Runnable, int, MemoryArea)

Signature

public
WaitFreeWriteQueue(Runnable writer,
    Runnable reader,
    int maximum,
    MemoryArea memory)
throws IllegalRealtimeArgumentException,
    MemoryScopeException,
    InaccessibleAreaException

250 RTSJ 2.0 (Draft 66)
Description

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 R unnables 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, 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.

Parameters

writer—An instance of Schedulable or null.
reader—An instance of Schedulable or null.
maximum—The maximum number of elements in the queue.
memory—The MemoryArea in which this and internal elements are allocated.

Throws

IllegalRealtimeArgumentException—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)

Signature

public

RTSJ 2.0 (Draft 66) 251
WaitFreeWriteQueue(int maximum, MemoryArea memory) throws IllegalRealtimeArgumentException, InaccessibleAreaException

Description
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.
Equivalent to WaitFreeWriteQueue(null, null, maximum, memory)

Available since RTSJ 1.0.1

WaitFreeWriteQueue(int)

Signature
public
WaitFreeWriteQueue(int maximum)
throws IllegalRealtimeArgumentException

Description
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.
Equivalent to WaitFreeWriteQueue(null, null, maximum, null)

Available since RTSJ 1.0.1

7.3.1.5.2 Methods

clear

Signature
public void
clear()
WaitFreeWriteQueue

**Description**

Sets this to empty.

**isEmpty**

**Signature**

```java
public boolean isEmpty()
```

**Description**

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.

**Returns**

true, when this is empty; false, when this is not empty.

**isFull**

**Signature**

```java
public boolean isFull()
```

**Description**

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.

**Returns**

true, when this is full; false, when this is not full.

**read**

**Signature**

```java
public synchronized T read()
```

**Description**

RTSJ 2.0 (Draft 66) 253
A synchronized and possibly blocking operation on the queue.

**Throws**

InterruptedException—when the thread is interrupted by interrupt() or AsynchronouslyInterruptedException.fire() during the time between calling this method and returning from it.

**Returns**

the T least recently written to the queue. When this is empty, the calling schedulable blocks until an element is inserted; when it is resumed, read removes and returns the element.

Available since RTSJ 1.0.1 Throws InterruptedException

### size

**Signature**

```java
public int size()
```

**Description**

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.

**Returns**

the number of positions in this occupied by elements that have been written but not yet read.

### force(T)

**Signature**

```java
public boolean force(T value)
```

**Description**

Unconditionally inserts value 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 `value` was inserted was vacant (`false`) or occupied (`true`).

**Parameters**

`value`—An instance of `T` to insert.

**Throws**

- `MemoryScopeException`—when a memory access error or illegal assignment error would occur while storing `value` in the queue.
- `IllegalRealtimeArgumentException`—when `value` is `null`.

**Returns**

- `true` when `value` 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(T)`

**Signature**

```java
public boolean write(T value)
throws MemoryScopeException,
IllegalRealtimeArgumentException
```

**Description**

Inserts `value` into `this` when `this` is non-full and otherwise has no effect on `this`; the `boolean` result reflects whether `value` 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).

**Parameters**

`value`—An instance of `T` to insert.

**Throws**

- `MemoryScopeException`—when a memory access error or illegal assignment error would occur while storing `value` in the queue.
- `IllegalRealtimeArgumentException`—when `value` is `null`.

**Returns**

- `true` when the queue was non-full; `false` otherwise.
7 Synchronization

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 the initial default mechanism in this specification.

Priority ceiling emulation is also a useful protocol. It is necessary for blocking out interrupts in interrupt service routines and simplifies scheduling analysis for single core systems. Since it can easily be implemented in user space, it is required as well.

The interaction of priority-based monitor policies such as priority inheritance and priority ceiling emulation with alternative schedulers that may not themselves be priority-based is complicated and contains many corner cases that are dependent on the scheduling protocol being implemented. Therefore, this specification does not define these interactions, but requires that the alternative scheduler implementation document the specifics of its interaction with the base priority-based schedulers defined here.

Since the same object may be accessed from synchronized code by both a schedulable which may not use the heap and an arbitrary thread or schedulable which may, 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

One of the most important aspects of this specification is its support for reacting to asynchronous events. This specifications provides a mechanism to bind the execution of program logic to the occurrence of internal and external events. This is provided by asynchronous event handling. Using this, an application can define some computation that is executed every time an event is “fired,” either from a clock or from some signal.

Asynchronous event handling is represented by the classes AsyncBaseEvent (AE) and AsyncBaseEventHandler (AEH), along with their subclasses. 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 event 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 the abstract class ActiveEventDispatcher to provide execution context for dispatching.

By convention, 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 method with an argument and others do not. The types of ActiveEvent supported are described in subsequent chapters.
Any variety of AEH may be associated with any variety of AE. The event actually delivered depends on the combination of the two. The table 8.1 illustrates this.

<table>
<thead>
<tr>
<th>Types</th>
<th>AsyncEvent</th>
<th>AsyncLongEvent</th>
<th>AsyncObjectEvent</th>
</tr>
</thead>
<tbody>
<tr>
<td>AsyncEventHandler</td>
<td>Nothing</td>
<td>Nothing</td>
<td>Nothing</td>
</tr>
<tr>
<td>AsyncLongEventHandler</td>
<td>Event Id</td>
<td>Payload</td>
<td>Event Id</td>
</tr>
<tr>
<td>AsyncObjectEventHandler</td>
<td>Event Object</td>
<td>Event Object</td>
<td>Payload</td>
</tr>
</tbody>
</table>

Memory assignment rules apply to the payload passed to AsyncObjectEventHandler.

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 waitForNextRelease() method. 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 handleAsyncEvent() method. The implication of this is that calls to Thread.currentThread(), RealtimeThread.currentRealtimeThread(), and access to thread-local storage may have unpredictable results from release to release.

The default manner in which the implementation selects a realtime thread to release a given AEH at a given release is defined by BlockableReleaseRunner, but the user can override this default by defining a new subclass of its abstract superclass, ReleaseRunner. The interface BoundSchedulable is used to mark subclasses of AsyncBaseEventHandler, such as BoundAsyncEventHandler, which have a dedicated realtime server thread. Such a server thread is associated with one and only one bound AEH for the lifetime of that AEH.

### 8.1 Definitions

**Asynchronous Event (AE)** — An instance of one of the subclasses of the javax.realtime.AsyncBaseEvent class.

**Asynchronous Event Handler (AEH)** — An instance of one of the subclasses of the AsyncBaseEventHandler class.

**Bound Asynchronous Event Handler (Bound AEH)** — An instance of a subclasses of the AsyncBaseEventHandler class that also implements BoundSchedulable.

**Bounded Execution Time** — As a particular task or schedulable may not be scheduled on a CPU for an arbitrarily long period of time, bounds on the responsiveness of a given task or schedulable are defined in terms of execution time.
time during which that task is scheduled on a CPU and executing. Time during which a task is blocked, either voluntarily, pending acquisition of a resource, or due to a higher-priority task executing on the CPUs available to it, is not considered execution time.

**Firable Asynchronous Event Handler** — An instance of `AsyncBaseEventHandler` is *firable* whenever there is an agent that can release it. This includes cases when the `AsyncBaseEventHandler` is

1. a miss handler or overrun handler of a `RealtimeThread` instance that has been started but not yet terminated;
2. a handler associated with an `AsyncBaseEvent` that can be fired; or
3. a miss handler or overrun handler for an instance of `AsyncBaseEventHandler` that is firable.

**Lexical Scope** — The textual region within programming block, such as a 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.

### 8.2 Semantics

Basic event types are passive: they are not directly associated with a thread of control. They are intended to be fired programmatically. Handling external events, such as clocks (see Chapter 10) and happenings (see Chapter 13), requires an execution context. The `ActiveEvent` interface is provided to mark these and provide additional execution semantics. Figure 8.1 illustrates the event hierarchy.

### 8.2.1 Asynchronous Events and their Handlers

This following points give the basic semantics for asynchronous events and their handlers. Semantics that apply to particular classes, constructors, methods, and fields are provided in the class description and the constructor, method, and field specifications.

1. When an asynchronous event occurs, either by program logic or by the triggering of a happening, and the event is enabled, its attached handlers, i.e., all AEHs that have been added to the AE by the execution of `addHandler()`, are released for execution.
   (a) Every occurrence of an event increments the `fireCount` in each attached handler.
   (b) Handlers may elect to execute logic for each occurrence of the event or not.
2. When `interrupt` is called on an AEH whose runnable state is `true`, i.e., its release parameters `isRousable` method returns `true`, that AEH will be released independently of all other AEH attached to any common AE.

3. The release of attached handlers occurs in execution eligibility order, i.e., 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.

4. The release of attached handlers is an atomic operation with respect to adding and removing handlers.

5. The logical release of an attached handler may occur before the previous release has completed.

6. Releasing an AEH is accomplished through the handler’s instance of `ReleaseRunner` as depicted in Figure 8.2.

7. Each handler has an application configurable, handler type dependent queue for holding events that have been released before a previous release has completed.

8. The overflow policy of a handler’s queue is also application configurable.
9. A deadline may be associated with each logical release of an attached handler. The deadline is relative to the occurrence of the associated event.

10. AEs and AEHs may be created and used by any program logic within the constraints of the memory assignment rules.

11. 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.

12. The same AEH may be added to more than one AE.

13. By default all AEHs are daemons: the daemon status is 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.

14. 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.

15. System-related termination activity (such as execution of finalizers for scoped objects in scopes that become unreferenced) triggered when an AEH becomes unfirable is not subject to cost enforcement or deadline miss detection.

16. 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 unfirable.

`AsyncBaseEvent` 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 8.3 illustrates this state space.

### 8.2.2 Active Events and Dispatching

Active events refine the semantics of `AsyncBaseEventHandler` 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 for it, or at least that context is limited out of necessity, and often needs to have a very short duration of execution. Thus dispatching an unlimited number of handlers in that context is not acceptable. This dispatching requires 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. Each class implementing `ActiveEvent` must provide its own `trigger` method for initiating the handler release by releasing another execution context. Since the `trigger` methods may vary in the number of their arguments depending on the type of event, they are not provided by the `ActiveEvent` class. Each `trigger` 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 a
Figure 8.2: Releasing an AsyncEventHandler

Figure 8.3: States of a Simple AsyncBaseEvent

Note:
fire count is incremented at release and decremented just before handleAsyncEvent

separate execution context.

This extra execution context is exposed to the user as an ActiveEventDispatcher. There is an 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 determines the priority and the affinity of a dispatcher, as well as 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 extension of the state space for an AsyncBaseEvent depicted in Figure 8.3. ActiveEvent adds the notion of active and inactive on top of enabled and disabled, as depicted in Figure 8.4. Note that the enabled-disabled distinction only splits the active state. The inactive state is by definition disabled.

Figure 8.4: States of an ActiveEvent

8.2.3 Termination

An RTSJ program terminates when and only when

1. all nondacon threads, either regular Java threads or realtime threads, are terminated;
2. the fireCounts of all nondacon instances of AsyncBaseEventHandler are zero and all of their releases are completed; and
3. there are no non-daemon instances of AsyncBaseEventHandler attached to a firable instance of ActiveEvent.

Bound and unbound AEH are treated alike. As with conventional Java, daemon tasks, including service threads such as a dispatcher’s thread or the threads used to run unbound AEH, do not hinder termination.
8.3 javax.realtime

8.3.1 Interfaces

8.3.1.1 ActiveEvent

public interface ActiveEvent<T extends Releasable<T, D>, D extends ActiveEventDispatcher<D, T>>

Interfaces
c javax.realtime.Releasable

Description
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.

An instance of this class holds a reference to its dispatcher. When ActiveEvent.isActive is true, the dispatcher must also hold a reference to the instance. For this reason, whenever an active event instance is active, it is also an execution context, so that this reference can be safely held during this time. Only the active event instance must be assignable to its dispatcher instance under the memory assignment rules, but not visa versa.

Available since RTSJ 2.0

8.3.1.1.1 Methods

isActive

Signature
public boolean isActive()

Description
Determines the activation state of this event, i.e., it has been started but not yet stopped again.

**Returns**

*true* when active, *false* otherwise.

### isRunning

**Signature**

```java
public boolean isRunning()
```

**Description**

Determines the running state of this event, i.e., it is both active and enabled.

**Returns**

*true* when active and enabled, *false* otherwise.

### start

**Signature**

```java
public void start()
```

**Throws**

`IllegalRealtimeStateException`—when this event has already been started.

**Description**

Starts this active event.

**Signature**

```java
public void start(boolean disabled)
```

**Description**

Starts this active event.
Parameters
disabled—True for starting in a disabled state.

Throws
IllegalRealtimeStateException—when this event has already been started.

stop

Signature
public boolean stop()
throws IllegalRealtimeStateException

Description
Stops this active event.

Throws
IllegalRealtimeStateException—when this event is not running.

Returns
the previous enabled state.

enable

Signature
public void enable()

Description
Changes the state of the event so that associated handlers are released on fire.
Each subclass provides a fire method as a means of dispatching its handlers when requested. This method enables that request mechanism.

disable

Signature
public void disable()

Description
Changes the state of the event so that associated handlers are skipped on fire. Each subclass provides a fire method as a means of dispatching its handlers when requested. This method disables that request mechanism.

### 8.3.1.2 Releasable

```java
public interface Releasable<T extends Releasable<T, D>, D extends ActiveEventDispatcher<D, T>>
```

**Description**

A base interface for everything that has a dispatcher.

**Available since** RTSJ 2.0

#### 8.3.1.2.1 Methods

**getDispatcher**

**Signature**

```java
public D extends javax.realtime.ActiveEventDispatcher<D, T> getDispatcher()
```

**Description**

Obtains the dispatcher for this.

**Returns**

that dispatcher.

### 8.3.1.3 Subsumable

```java
public interface Subsumable<T>
```

**Description**

RTSJ 2.0 (Draft 66)
A partial ordering relationship. One object subsumes another if and only if the set represented by other is a subset of this object. Objects which represent disjoint set or sets whose intersection is less than either of the objects sets are mutually not subsumable.

**Available since** RTSJ 2.0

### 8.3.1.3.1 Methods

**subsumes(T)**

**Signature**

```java
public boolean subsumes(T other)
```

**Description**

Indicates that some set represented by other is subsumed by the set represented by this object.

**Parameters**

- `other`—The object to be compared with.

**Returns**

true when and only when the set represented by other is subsumed by the set represented by this object.

### 8.3.2 Classes

#### 8.3.2.1 ActiveEventDispatcher

```java
public abstract class ActiveEventDispatcher<D extends ActiveEventDispatcher<D, T>, T extends Releasable<T, D>>
```

**Inheritance**

- java.lang.Object
  - ActiveEventDispatcher<D extends ActiveEventDispatcher<D, T>, T extends Releasable<T, D>>

**Description**

RTSJ 2.0 (Draft 66) 269
Provides a means of dispatching a set of ActiveEvents. It acts as if it contains a daemon 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 realtime priority on the base scheduler. Dispatchers do not maintain a queue of pending event.

Application code cannot extend this class.

Available since RTSJ 2.0

### 8.3.2.1.1 Constructors

ActiveEventDispatcher(SchedulingParameters, RealtimeThreadGroup)

**Signature**

```java
protected ActiveEventDispatcher(SchedulingParameters schedule, RealtimeThreadGroup group) throws IllegalRealtimeStateException
```

**Description**

Creates a new dispatcher.

**Parameters**

- *schedule*—Provides scheduling information to the new object.
- *group*—The RealtimeThreadGroup of the thread of this dispatcher.

**Throws**

- `IllegalRealtimeStateException`—when the intersection of affinity in `schedule` and the affinity of `group` does not correspond to a valid affinity.

ActiveEventDispatcher(SchedulingParameters)

**Signature**

```java
protected ActiveEventDispatcher(SchedulingParameters schedule)
```

270 RTSJ 2.0 (Draft 66)
ActiveEventDispatcher

javax.realtime 8.3

throws IllegalRealtimeStateException

Description

Creates a new dispatcher.

Parameters

schedule—Provides scheduling information to the new object.

Throws

IllegalRealtimeStateException—when the intersection of affinity in schedule and the affinity of the current thread group does not correspond to a valid affinity.

8.3.2.1.2 Methods

getScheduler

Signature

public javax.realtime.Scheduler
getScheduler()

Description

Gets a reference to the Scheduler object for this schedulable.

Returns

a reference to the associated Scheduler object.

setScheduler(Scheduler)

Signature

public javax.realtime.ActiveEventDispatcher<D, T>
setScheduler(Scheduler scheduler)
throws RealtimeSecurityException,
IllegalTaskStateException

Description

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. If the Schedulable is running, its associated SchedulingParameters (if any) must be compatible with scheduler.

Parameters

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

Throws

IllegalRealtimeArgumentException—when scheduler is null, or the schedulable’s existing parameter values are not compatible with scheduler. Also when this schedulable may not use the heap and scheduler is located in heap memory.

IllegalAssignmentError—when the schedulable cannot hold a reference to scheduler or the current Schedulable is running and its associated SchedulingParameters are incompatible with scheduler.

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

IllegalTaskStateException—when scheduler has scheduling or release parameters that are not compatible with the new scheduler and this schedulable is running.

Returns

this

setScheduler(Scheduler, SchedulingParameters)

Signature

public javax.realtime.ActiveEventDispatcher<D, T> setScheduler(Scheduler scheduler, SchedulingParameters scheduling)

Description

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.

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 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.)
Throws

`IllegalRealtimeArgumentException`—when `scheduler` is `null` or the parameter values are not compatible with `scheduler`. Also thrown when this schedulable may not use the 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.

`RealtimeSecurityException`—when the caller is not permitted to set the scheduler for this schedulable.

Returns

this

getsSchedulingParameters

Signature

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

Description

Determines how the thread associated with this dispatcher is scheduled.

Returns

the scheduling parameters of the dispatcher thread.

setsSchedulingParameters(SchedulingParameters)

Signature

```java
public javax.realtime.ActiveEventDispatcher<D, T>
setSchedulingParameters(SchedulingParameters scheduling)
throws IllegalTaskStateException,
IllegalAssignmentError,
IllegalRealtimeArgumentException
```

Description

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

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.
Parameters

**scheduling**—A reference to the *SchedulingParameters* 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*.)

**Throws**

- **IllegalRealtimeArgumentException**—when **scheduling** is not compatible with the associated scheduler. Also when this schedulable may not use the 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.
- **IllegalTaskStateException**—when the task is active and the new scheduling parameters are not compatible with the current scheduler or the intersection of affinity in **scheduling** and the affinity of this object’s realtime thread group is not an valid affinity.

**Returns**

this

**getRealtimeThreadGroup**

**Signature**

```java
public javax.realtime.RealtimeThreadGroup getRealtimeThreadGroup()
```

**Description**

Determines in which group the thread associated with this dispatcher is.

**Returns**

the realtime thread group of the dispatcher thread.

**register(T)**

**Signature**

```java
public abstract void register(T event)
```

**Description**

274 RTSJ 2.0 (Draft 66)
**ActiveEventDispatcher**

Registers an active event with this dispatcher.

*Parameters*

- **event**—The event to register

*Throws*

- **RegistrationException**—when *event* is already registered.
- **IllegalRealtimeStateException**—when this object has been destroyed or its scheduling parameters are not compatible with the current scheduler.
- **IllegalRealtimeArgumentException**—when *event* is not stopped.

**deregister(T)**

*Signature*

```java
public abstract void
deregister(T event)
throws DeregistrationException,
        IllegalRealtimeStateException,
        IllegalRealtimeArgumentException
```

*Description*

Deregisters an active event from this dispatcher.

*Parameters*

- **event**—The event to deregister

*Throws*

- **DeregistrationException**—when *event* is already registered.
- **IllegalRealtimeStateException**—when this object has been destroyed.
- **IllegalRealtimeArgumentException**—when *event* is not stopped.

**destroy**

*Signature*

```java
public abstract void
destroy()
throws IllegalRealtimeStateException
```

*Description*

Makes the dispatcher unusable.

*Throws*
IllegalRealtimeStateException—when called on a dispatcher that has one or more registered objects.

8.3.2.2 AsyncBaseEvent

public abstract class AsyncBaseEvent

Inheritance
java.lang.Object
  AsyncBaseEvent

Description
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 with AsyncLongEvent and AsyncObjectEvent.

Note that when this class is collected, all its handlers are automatically removed as if setHandler was called with a null parameter.

Available since RTSJ 2.0

8.3.2.2.1 Methods

isRunning

Signature
public boolean
isRunning()

Description
Determines the firing state (releasing or skipping) of this event, i.e., whether it is enabled or disabled.

Returns
true when releasing, false when skipping.

Available since RTSJ 2.0 Inherited by AsyncEvent
handledBy(AsyncBaseEventHandler)

Signature

public boolean handledBy(AsyncBaseEventHandler handler)

Description

Determines whether or not the handler given as the parameter is associated with this.

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.

Available since RTSJ 2.0 Inherited by AsyncEvent

enable

Signature

public void enable()

Description

Changes the state of the event so that associated handlers are released on fire. Each subclass provides a fire method as means of dispatching its handlers when requested. This method enables that request mechanism.

Available since RTSJ 2.0 Inherited by AsyncEvent

disable

Signature

public void disable()

Description
Changes 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.

**Available since** RTSJ 2.0 Inherited by AsyncEvent

### addHandler(AsyncBaseEventHandler)

**Signature**
```
public void
addHandler(AsyncBaseEventHandler handler)
```

**Description**
Adds a handler to the set of handlers associated with this event. An instance of `AsyncBaseEvent` 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.

Note that there is an implicit reference to the handler stored in `this`. The assignment must be valid under any applicable memory assignment rules.

**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**
- `IllegalRealtimeArgumentException`—when `handler` is null or the handler has `PeriodicParameters`. Only the subclass `PeriodicTimer` is allowed to have handlers with `PeriodicParameters`.
- `IllegalAssignmentError`—when this `AsyncBaseEvent` cannot hold a reference to `handler`.
- `IllegalRealtimeStateException`—when the configured `Scheduler` and scheduling parameters for `handler` are not compatible with one another.
- `ScopedCycleException`—when `handler` has an explicit initial scoped memory area that has already been entered from a memory area other than the area where `handler` was allocated.

**Available since** RTSJ 2.0 Inherited by AsyncEvent
setHandler(AsyncBaseEventHandler)

Signature

public void
setHandler(AsyncBaseEventHandler handler)

Description

Associates a new handler with this event and removes all existing handlers. The execution of this method is atomic with respect to the execution of the fire() method.

Parameters

handler—The instance of AsyncBaseEventHandler to be associated with this. When handler is null then no handler will be associated with this, i.e., it behaves effectively as if setHandler(null) invokes removeHandler(AsyncBaseEventHandler) for each associated handler.

Throws

IllegalRealtimeArgumentException—when handler has PeriodicParameters. Only the subclass PeriodicTimer is allowed to have handlers with PeriodicParameters.

IllegalAssignmentError—when this AsyncBaseEvent cannot hold a reference to handler.

Available since RTSJ 2.0 Inherited by AsyncEvent

removeHandler(AsyncBaseEventHandler)

Signature

public void
removeHandler(AsyncBaseEventHandler handler)

Description

Removed 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 unfirable, this method shall not return until all related finalization has completed.

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. **Available since** RTSJ 2.0 Inherited by AsyncEvent

**hasHandlers**

*Signature*

```java
class AsyncBaseEvent {
    public boolean hasHandlers();
}
```

*Description*

Determines whether or not this event has any handlers.

*Returns*

`true` when and only when at least one handler is associated with this event.

**Available since** RTSJ 2.0 Inherited by AsyncEvent

**createReleaseParameters**

*Signature*

```java
class AsyncBaseEvent {
    public javax.realtime.ReleaseParameters createReleaseParameters();
}
```

*Description*

Creates a `ReleaseParameters` object appropriate to the release characteristics of this event. The default is the most pessimistic: `AperiodicParameters`. 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` 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`, there is no requirement for an implementation to check that the handler is released periodically.

*Returns*

a new `ReleaseParameters` object.
8.3.2.3 AsyncBaseEventHandler

public abstract class AsyncBaseEventHandler

Inheritance
java.lang.Object
    AsyncBaseEventHandler

Interfaces
javax.realtime.Schedulable

Description
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 AsyncEventHandler with AsyncLongEventHandler and AsyncObjectEventHandler.

Available since RTSJ 2.0

8.3.2.3.1 Methods

gCurrentReleaseTime

Signature
public static javax.realtime.AbsoluteTime getCurrentReleaseTime()

Description
Gets the last release time of this timer.

Throws
IllegalRealtimeStateException—when this timer has not been released since it was last started.

Returns
a reference to a newly-created AbsoluteTime object representing this timer’s last release time. When the timer has not been released since it was last started, throws an exception.

Available since RTSJ 2.0
getCurrentReleaseTime(AbsoluteTime)

**Signature**
```java
public static javax.realtime.AbsoluteTime
getCurrentReleaseTime(AbsoluteTime dest)
```

**Description**
Gets the last release time of this timer in the object provided.

**Parameters**
dest—An object used to return the results

**Returns**
When dest is null, returns a reference to a newly-created AbsoluteTime object representing this timer’s last release time. When dest is not null, sets dest to this timer’s last release time. When the timer has not been released, returns null.

**Available since** RTSJ 2.0

getConsumption(RelativeTime)

**Signature**
```java
public static javax.realtime.RelativeTime
getConsumption(RelativeTime dest)
```

**Description**
Determines the CPU consumption for this release. When dest is null, returns the CPU consumption in an otherwise unused RelativeTime instance in the current execution context. Otherwise, when dest is not null, returns the CPU consumption in dest

**Parameters**
dest—When not null, the object in which to return the result.

Throws
IllegalRealtimeStateException—when the caller is not a Schedulable.

Returns
the time consumed in the current release.

**Available since** RTSJ 2.0 Inherited by AsyncEventHandler
getCurrentConsumption

Signature
public static javax.realtime.RelativeTime getCurrentConsumption()

Description
Equivalent to getCurrentConsumption(null).

Returns
the time consumed in the current release.

Available since RTSJ 2.0 Inherited by AsyncEventHandler

getPendingFireCount

Signature
protected abstract int getPendingFireCount()

Description
This is an accessor method for fireCount. The fireCount field nominally holds the number of times associated instances of AsyncEvent 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.

Returns
the value held by fireCount.

getAndClearPendingFireCount

Signature
protected abstract int getAndClearPendingFireCount()

Description
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
Asynchrony

AsyncBaseEventHandler may be used by handlers for which the logic can accommodate multiple releases in a single execution.

The general form for using this is

```java
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.

Returns
the value held by `fireCount` prior to setting the value to zero.

getAndDecrementPendingFireCount

Signature
protected abstract int
getAndDecrementPendingFireCount()

Description
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:

```java
public void handleAsyncEvent()
{
    <setup>
    do
    {
        <handle the event>
    }
    while(getAndDecrementPendingFireCount() > 0);
}
```
This construction is necessary only in cases 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.

**Returns**

the value held by `fireCount` prior to decrementing it by one.

### getMemoryArea

**Signature**

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

**Description**

This is an accessor method for the initial instance of `MemoryArea` associated with `this`.

**Returns**

the instance of `MemoryArea` which was passed as the `area` parameter when `this` was created (or the default value when `area` was allowed to default.

### getMemoryParameters

**Signature**

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

**Description**

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

**Returns**

a reference to the current `MemoryParameters` object.

### getReleaseParameters

**Signature**

```java
RTSJ 2.0 (Draft 66) 285
```
public javax.realtime.ReleaseParameters
getReleaseParameters()

Description
Gets a reference to the ReleaseParameters object for this schedulable.

Returns
a reference to the current ReleaseParameters object.

getscheduler

Signature
public javax.realtime.Scheduler
getscheduler()

Description
Gets a reference to the Scheduler object for this schedulable.

Returns
a reference to the associated Scheduler object.

getschedulingparameters

Signature
public javax.realtime.SchedulingParameters
getschedulingparameters()

Description
Gets a reference to the SchedulingParameters object for this schedulable.

Returns
A reference to the current SchedulingParameters object.

getrealtimethreadgroup

Signature
public javax.realtime.RealtimeThreadGroup
getrealtimethreadgroup()

Description

286 RTSJ 2.0 (Draft 66)
Gets a reference to the `RealtimeThreadGroup` instance of this `Schedulable`. For instance that are not also instances of `RealtimeThread`, such as instance if `AsyncBaseEventHandler`, the group returned is the group that contains the thread or threads that are used to execute said `Schedulable` instance. For each instance of `Schedulable`, there can be only one directly associated instance of `Schedulable`.

Retuns
a reference to the associated `RealtimeThreadGroup` object.

Available since RTSJ 2.0

getConfigurationParameters

Signature

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

Description

Gets a reference to the `ConfigurationParameters` object for this schedulable.

Retuns

the `ConfigurationParameters` instance of its `ReleaseRunner` instance, i.e.,

```java
getReleaseRunner().getConfigurationParameters()
```

setMemoryParameters(MemoryParameters)

Signature

```java
public javax.realtime.AsyncBaseEventHandler
setMemoryParameters(MemoryParameters memory)
```

Description

Sets the memory parameters associated with this instance of `Schedulable`. This change becomes effective at the next allocation; on multiprocessor systems, there may be some delay due to synchronization between processors.

Parameters

memory—A `MemoryParameters` 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`.)
8 Asynchrony

AsyncBaseEventHandler

Throws

**IllegalRealtimeArgumentException**—when memory is not compatible with the schedulable’s scheduler. Also when this schedulable may not use the 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.

Returns

this

Available since RTSJ 2.0 returns itself

**setReleaseParameters(ReleaseParameters)**

Signature

```java
public javax.realtime.AsyncBaseEventHandler
setReleaseParameters(ReleaseParameters release)
```

Description

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.

Parameters

release—A ReleaseParameters 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.)

Throws

**IllegalRealtimeArgumentException**—when release is not compatible with the associated scheduler. Also when this schedulable may not use the 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.

**IllegalTaskStateException**—when the task is running and the new release parameters are not compatible with the current scheduler.
Returns
this
Available since RTSJ 2.0 returns itself

setScheduler(Scheduler)

Signature
public javax.realtime.AsyncBaseEventHandler
setScheduler(Scheduler scheduler)

Description
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. If the Schedulable is running, its associated SchedulingParameters (if any) must be compatible with scheduler.

For an instance of AsyncBaseEventHandler, the Schedulable is running for the purpose of setting the scheduler when it is attached to an AsyncEvent, even when AsyncBaseEvent.isRunning() would return false for that event.

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

Throws
IllegalRealtimeArgumentException—when scheduler is null, or the schedulable’s existing parameter values are not compatible with scheduler. Also when this schedulable may not use the heap and scheduler is located in heap memory.

IllegalAssignmentError—when the schedulable cannot hold a reference to scheduler or the current Schedulable is running and its associated SchedulingParameters are incompatible with scheduler.

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

IllegalTaskStateException—when scheduler has scheduling or release parameters that are not compatible with the new scheduler and this schedulable is running.

Returns
this
Available since RTSJ 2.0 returns itself
setScheduler(Scheduler, SchedulingParameters, ReleaseParameters, MemoryParameters)

Signature

public javax.realtime.AsyncBaseEventHandler
setScheduler(Scheduler scheduler,
               SchedulingParameters scheduling,
               ReleaseParameters release,
               MemoryParameters memoryParameters)

Description

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.

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 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.)

release — A reference to the ReleaseParameters 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.)

memoryParameters — A reference to the MemoryParameters 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.)

Throws

IllegalRealtimeArgumentException — when scheduler is null or the parameter
                                    values are not compatible with scheduler. Also thrown when this schedulable
                                    may not use the 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.

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

Returns

this

Available since RTSJ 2.0
setSchedulingParameters(SchedulingParameters)

**Signature**

```java
public javax.realtime.AsyncBaseEventHandler
setSchedulingParameters(SchedulingParameters scheduling)
```

**Description**

Sets the scheduling 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. See the documentation for the scheduler for details.

**Parameters**

- `scheduling`—A reference to the `SchedulingParameters` 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`.)

**Throws**

- `IllegalRealtimeArgumentException`—when `scheduling` is not compatible with the associated scheduler. Also when this schedulable may not use the 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.
- `IllegalTaskStateException`—when the task is active and the new scheduling parameters are not compatible with the current scheduler or the intersection of affinity in `scheduling` and the affinity of this object’s realtime thread group does not correspond to a valid affinity.

**Returns**

- `this`

**Available since** RTSJ 2.0 returns itself

setDaemon(boolean)

**Signature**

```java
public final void
setDaemon(boolean on)
```

**Description**

RTSJ 2.0 (Draft 66)
Marks this schedulable as either a daemon or a user task. A realtime virtual machine exits when the only tasks running are all daemons. This method must be called before the task is attached to any event or started. Once attached or started, it cannot be changed.

Parameters
on—When true, marks this event handler as a daemon handler.

Throws
IllegalThreadStateException—when this schedulable is active.
RealtimeSecurityException—when the current schedulable cannot modify this event handler.

Available since RTSJ 2.0

isDaemon

Signature
public final boolean isDaemon()

Description
Tests if this event handler is a daemon handler.

Returns
true when this event handler is a daemon handler; false otherwise.

Available since RTSJ 2.0

getDispatcher

Signature
public javax.realtime.TimeDispatcher getDispatcher()

Description
Gets the dispatcher associated with this Timable.

See Section Timable.getDispatcher()
**getQueueLength**

*Signature*

```java
public int getQueueLength()
```

*Description*

Finds the current length of the event queue. The event queue holds the time and payload of all released events that are still outstanding. The queue may have a length of zero.

*Returns*

the queue length.

**Available since** RTSJ 2.0 Inherited by AsyncEventHandler

**getMinConsumption(RelativeTime)**

*Signature*

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

*Description*

Determines the minimum CPU consumption of all completed releases. When `dest` is `null`, returns the CPU consumption in an otherwise unused `RelativeTime` instance in the current execution context. Otherwise, when `dest` is not `null`, returns the CPU consumption in `dest`.

*Parameters*

`dest`—When not `null`, the object in which to return the result.

*Returns*

the minimum time consumed in any release.

**getMinConsumption**

*Signature*

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

*Description*
Same as $\text{getMinConsumption}(\text{RelativeTime})$ with a null argument.

Returns
the minimum time consumed in any release.

\textbf{getMaxConsumption}(\text{RelativeTime})

\textbf{Signature}
\begin{verbatim}
public javax.realtime.RelativeTime getMinConsumption(RelativeTime dest)
\end{verbatim}

\textbf{Description}
Determines the maximum CPU consumption of all completed releases. When $\text{dest}$ is \texttt{null}, returns the CPU consumption in an otherwise unused $\text{RelativeTime}$ instance in the current execution context. Otherwise, when $\text{dest}$ is not \texttt{null}, returns the CPU consumption in $\text{dest}$.

\textbf{Parameters}
$\text{dest}$—When not \texttt{null}, the object in which to return the result.

\textbf{Returns}
the maximum time consumed in any release.

\textbf{getMaxConsumption}

\textbf{Signature}
\begin{verbatim}
public javax.realtime.RelativeTime getMaxConsumption()
\end{verbatim}

\textbf{Description}
Same as $\text{getMaxConsumption}(\text{RelativeTime})$ with a null argument.

\textbf{Returns}
the maximum time consumed in any release.

\textbf{mayUseHeap}

\textbf{Signature}
\begin{verbatim}
public boolean mayUseHeap()
\end{verbatim}
**isInterrupted**

*Signature*

```java
class AsyncBaseEventHandler {
    public boolean isInterrupted()
}
```

*Description*

Determines whether or not any `AsynchronouslyInterruptedException` is pending.

*Returns*

`true` when and only when the generic `AsynchronouslyInterruptedException` is pending.

*Available since* RTSJ 2.0

**interrupt**

*Signature*

```java
class AsyncBaseEventHandler {
    public void interrupt()
}
```

*Description*

In a system where `RTSJModule.CONTROL` is implemented, it makes the generic `AsynchronouslyInterruptedException` pending for this, and sets the interrupted state to `true`. As with `Thread.interrupt()`, blocking operations that are interruptible are interrupted. When this `isRousable()` is `true` a release happens event if the first release has not yet happened. In any case, `AsynchronouslyInterruptedException` is thrown once a method is entered that implements `AsynchronouslyInterruptedException`.

Otherwise, it behaves as if `Thread.interrupt()` were called.

For instances of `AsyncBaseEventHandler`, it behaves as if `RealtimeThread.interrupt()` was called on the implementation thread underlying it.
Asynchrony

AsyncBaseEventHandler

Throws

`IllegalTaskStateException`—when `this` is not currently releasable, i.e., is disabled, not firable, its start method has not been called, or it has terminated.

Available since RTSJ 2.0

isRousable

Signature

```java
public boolean isRousable()
```

Description

Determines whether or not it is possible for an interruptible to prematurely release the handler.

Returns

*true* when it is possible, otherwise *false*.

Available since RTSJ 2.0 Inherited by AsyncEventHandler

setRousable(boolean)

Signature

```java
public javax.realtime.AsyncBaseEventHandler setRousable(boolean value)
```

Description

Sets a state indicating whether or not a interrupt can prematurely release this handler.

Parameters

`value`—The new value of the wake by interrupt state.

Returns

`this`

Available since RTSJ 2.0 Inherited by AsyncEventHandler

296 RTSJ 2.0 (Draft 66)
awaken

Signature
public final void
awaken()

Description
Use internally to indicate a sleep period has ended.

See Section Schedulable.awaken()

run

Signature
public abstract void
run()

Description
This method is only to be used by the infrastructure, and should not be called by the application.

The handleAsyncEvent() family of methods provides the equivalent functionality to Runnable.run() for asynchronous event handlers, including execution of the logic argument passed to this object’s constructor. Applications should override that method or provide a logic object for the default implementation to invoke.

8.3.2.4 AsyncEvent

public class AsyncEvent

Inheritance
java.lang.Object
    AsyncBaseEvent
    AsyncEvent

Description
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).
Available since RTSJ 2.0 extends AsyncBaseEvent

8.3.2.4.1 Constructors

AsyncEvent

Signature
    public
    AsyncEvent()

Description
    Creates a new AsyncEvent object.

8.3.2.4.2 Methods

fire

Signature
    public void
    fire()

Description
    When enabled, release the asynchronous event handlers associated with this instance of AsyncEvent. When this object is disabled the method does nothing, i.e., it skips the release. When no handlers are attached, the release is ignored. This method does not suspend itself and has a runtime complexity of $O(n)$, where $n$ is the number of attached handlers. For an instance of AsyncEvent that has more than one instance of AsyncEventHandler,
    - when one of these handlers throws an exception, all instances of AsyncEventHandler not affected by the exception must be released normally before the exception is propagated, and
AsyncEventHandler

- when more than one of these handlers throws an exception, the propagation of MITViolationException has precedence over ArrivalTimeQueueOverflowException, which has precedence over all others. The later case can only occur when more than one of the handlers has a release parameters instance of type SporadicParameters, since only them can MITViolationException and ArrivalTimeQueueOverflowException be thrown.

Throws
MITViolationException—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 interarrival 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 release information, arrival time and payload, overflows. Only the handlers which do not cause this exception to be thrown are released in this situation. When fire is called from the infrastructure, such as for an ActiveEvent, this exception is ignored.

8.3.2.5 AsyncEventHandler

public class AsyncEventHandler

Inheritance
java.lang.Object
AsyncBaseEventHandler
AsyncEventHandler

Description
An asynchronous event handler encapsulates code that is released after an instance of AsyncEvent 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() method. Control over the number of calls to AsyncEventHandler.handleAsyncEvent() is given by methods which manipulate a fireCount. These may be called by the application via sub-classing and overriding AsyncEventHandler.handleAsyncEvent().

Instances of AsyncEventHandler with a release parameter of type SporadicParameters or AperiodicParameters have a list of release times which correspond to the occurrence times of instances of AsyncEvent to which they are
attached. The minimum interarrival time specified in SporadicParameters 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() 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, 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.waitForNextRelease method.

Normally, handlers are bound to an execution context dynamically when the instances of AsyncEvents to which they are bound occur. This can introduce a (small) time penalty. For critical handlers that cannot afford the expense, and where this penalty is a problem, BoundAsyncEventHandler can be used.

The scheduler for an asynchronous event handler is inherited from the task that created it. When created from a task that is not an instance of Schedulable, 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.

Available since RTSJ 2.0 extends AsyncBaseEventHandler

8.3.2.5.1 Constructors

AsyncEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, ReleaseRunner, Runnable)

Signature

public
AsyncEventHandler(SchedulingParameters scheduling,
AsyncEventHandler(javax.realtime 8.3)

ReleaseParameters release,
MemoryParameters memory,
MemoryArea area,
ReleaseRunner runner,
Runnable logic)

Description

Creates a handler with the given scheduling, release, memory, group, and configuration parameters to run the given logic.

Available since RTSJ 2.0

Parameters

scheduling—Parameter for scheduling the new handler (and possibly other instances of Schedulable). When scheduling is null and the creator is an instance of Schedulable, SchedulingParameters 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 task that is not an instance of Schedulable, the contents and type of the new SchedulingParameters object are governed by the associated scheduler.

release—Parameter for scheduling the new handler (and possibly other instances of Schedulable). When release is null the new AsyncEventHandler will use a clone of the default ReleaseParameters for the associated scheduler created in the memory area that contains the AsyncEventHandler object.

memory—Parameter for scheduling the new handler (and possibly other instances of Schedulable). 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.

area—The initial memory area of this handler.

runner—A pool of realtime threads to provide an execution context for this handler.

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, MemoryParameters, MemoryArea, Runnable)

Signature

public

RTSJ 2.0 (Draft 66) 301
AsyncEventHandler(SchedulingParameters scheduling, ReleaseParameters release, MemoryParameters memory, MemoryArea area, Runnable logic)

Description
Calling this constructor is equivalent to calling AsyncEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, ReleaseRunner, Runnable) with arguments (scheduling, release, memory, area, null, logic).

Available since RTSJ 2.0

AsyncEventHandler(SchedulingParameters, ReleaseParameters, Runnable)

Signature
public
AsyncEventHandler(SchedulingParameters scheduling, ReleaseParameters release, Runnable logic)

Description
Calling this constructor is equivalent to calling AsyncEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, ReleaseRunner, Runnable) with arguments (scheduling, release, null, null, null, logic).

Available since RTSJ 2.0

AsyncEventHandler(SchedulingParameters, ReleaseParameters)
AsyncEventHandler

ReleaseParameters release)

Description
Calling this constructor is equivalent to calling AsyncEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, ReleaseRunner, Runnable) with arguments (scheduling, release, null, null, null, null)

Available since RTSJ 2.0

AsyncEventHandler(Runnable)

Signature
public
AsyncEventHandler(Runnable logic)

Description
Calling this constructor is equivalent to calling AsyncEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, ReleaseRunner, Runnable) with arguments (null, null, null, null, null, logic).

AsyncEventHandler

Signature
public
AsyncEventHandler()

Description
Creates an instance of AsyncEventHandler with default values for all parameters.

See Section AsyncEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, ReleaseRunner, Runnable)

8.3.2.5.2 Methods
handleAsyncEvent

Signature
  public void
  handleAsyncEvent()

Description
  This method holds the logic which is to be executed when any AsyncEvent 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.

run

Signature
  public final void
  run()

Description
  This method is only to be used by the infrastructure, and should not be called by the application.
  The handleAsyncEvent() family of methods provides the equivalent functionality to Runnable.run() for asynchronous event handlers, including execution of the logic argument passed to this object’s constructor. Applications should override that method or provide a logic object for the default implementation to invoke.

getPendingFireCount

Signature
  protected int
  getPendingFireCount()
Description
This is an accessor method for fireCount. The fireCount field nominally holds the number of times associated instances of AsyncEvent 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.

Returns
the value held by fireCount.

getAndDecrementPendingFireCount

Signature
protected int
getAndDecrementPendingFireCount()

Description
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:

```java
public void handleAsyncEvent()
{
    <setup>
    do
    {
        <handle the event>
    }
    while(getAndDecrementPendingFireCount() > 0);
}
```

This construction is necessary only in cases 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.
Returns
the value held by fireCount prior to decrementing it by one.

getAndClearPendingFireCount

Signature
protected int
getAndClearPendingFireCount()

Description
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 be used by handlers for which the logic can accommodate multiple releases in a single execution.

The general form for using this is

```java
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.

Returns
the value held by fireCount prior to setting the value to zero.

8.3.2.6 AsyncLongEvent

public class AsyncLongEvent

Inheritance
java.lang.Object
    AsyncBaseEvent
        AsyncLongEvent

Description
A new type of event that carries a long as a payload.
See Section AsyncEvent

Available since RTSJ 2.0

8.3.2.6.1 Constructors

AsyncLongEvent

Signature
public
AsyncLongEvent()

Description
Creates a new AsyncLongEvent object.

8.3.2.6.2 Methods

fire(long)

Signature
public void
fire(long value)
throws MITViolationException,
EventQueueOverflowException

Description
When enabled, releases the handlers associated with this instance of AsyncLongEvent with the long passed by fire(long). When no handlers are attached or this object is disabled the method does nothing, i.e., it skips the release. For an instance of AsyncEvent that has more than one instance of AsyncEventHandler, • when one of these handlers throws an exception, all instances of AsyncEventHandler not affected by the exception must be released normally before the exception is propagated, and
• when more than one of these handlers throws an exception, the propagation of MITViolationException has precedence over ArrivalTimeQueueOverflowException, which has precedence over all others.

The later case can only occur when more than one of the handlers has a release parameters instance of type SporadicParameters, since only them can MITViolationException and ArrivalTimeQueueOverflowException be thrown.

Parameters

value—The payload passed to the event.

Throws

MITViolationException—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.

EventQueueOverflowException—when the queue of release information, arrival time and payload, overflows. Only the handlers which do not cause this exception to be thrown are released in this situation. When fire is called from the infrastructure, such as for an ActiveEvent, this exception is ignored.

8.3.2.7 AsyncLongEventHandler

public class AsyncLongEventHandler

Inheritance

java.lang.Object
  AsyncBaseEventHandler
  AsyncLongEventHandler

Description

A version of AsyncBaseEventHandler that carries a long value as payload.

Available since RTSJ 2.0

8.3.2.7.1 Constructors

308 RTSJ 2.0 (Draft 66)
AsyncLongEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, ReleaseRunner, LongConsumer)

**Signature**

```java
class AsyncLongEventHandler {
    public AsyncLongEventHandler(SchedulingParameters scheduling,
                                  ReleaseParameters release,
                                  MemoryParameters memory,
                                  MemoryArea area,
                                  ReleaseRunner runner,
                                  LongConsumer logic)
        throws IllegalRealtimeArgumentException
}
```

**Description**

Creates an asynchronous event handler that receives a `Long` payload with each fire.

**Parameters**

- `scheduling`—Parameter for scheduling the new handler (and possibly other instances of `Schedulable`). When `scheduling` is `null` and the creator is an instance of `Schedulable`, `SchedulingParameters` 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 task that is not an instance of `Schedulable`, the contents and type of the new `SchedulingParameters` object are governed by the associated scheduler.

- `release`—Parameter for scheduling the new handler (and possibly other instances of `Schedulable`). When `release` is `null` the new `AsyncEventHandler` will use a clone of the default `ReleaseParameters` for the associated scheduler created in the memory area that contains the `AsyncEventHandler` object.

- `memory`—Parameter for scheduling the new handler (and possibly other instances of `Schedulable`). 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.

- `area`—The initial memory area of this handler.

- `runner`—Logic to be executed by `handleAsyncEvent`

- `logic`—The logic to run for each fire. When `logic` is `null`, the `handleAsyncEvent()` method in the new object will serve as its logic.

**Throws**

RTSJ 2.0 (Draft 66) 309
IllegalRealtimeArgumentException—when the event queue overflow policy is QueueOverflowPolicy.DISABLE.

AsyncLongEventHandler(SchedulingParameters, ReleaseParameters, LongConsumer)

Signature
public
AsyncLongEventHandler(SchedulingParameters scheduling,
ReleaseParameters release,
LongConsumer logic)
throws IllegalRealtimeArgumentException

Description
Calling this constructor is equivalent to calling AsyncLongEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, ReleaseRunner, LongConsumer) with arguments (scheduling, release, null, null, null, logic).

AsyncLongEventHandler(SchedulingParameters, ReleaseParameters)

Signature
public
AsyncLongEventHandler(SchedulingParameters scheduling,
ReleaseParameters release)
throws IllegalRealtimeArgumentException

Description
Calling this constructor is equivalent to calling AsyncLongEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, ReleaseRunner, LongConsumer) with arguments (scheduling, release, null, null, null, null)
AsyncLongEventHandler(LongConsumer)

**Signature**
```java
public
AsyncLongEventHandler(LongConsumer logic)
```

**Description**
Calling this constructor is equivalent to calling `AsyncLongEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, ReleaseRunner, LongConsumer)` with arguments (null, null, null, null, null, logic).

AsyncLongEventHandler

**Signature**
```java
public
AsyncLongEventHandler()
```

**Description**
Creates an instance of `AsyncLongEventHandler` (ALEH) with default values for all parameters.

See Section `AsyncLongEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, ReleaseRunner, LongConsumer)`

8.3.2.7.2 Methods

handleAsyncEvent(long)

**Signature**
```java
public void
handleAsyncEvent(long payload)
```

**Description**
This method holds the logic which is to be executed when any `AsyncEvent` 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.

**Parameters**

`payload`—It is the long value associated with a fire.

**peekPending**

**Signature**

```java
public long peekPending()
throws IllegalRealtimeStateException
```

**Description**

Determines the next value queued for handling.

**Throws**

`IllegalRealtimeStateException`—when the fire count is zero.

**Returns**

the long value at the head of the queue of longs to be passed to `handleAsyncEvent(long)`.

**run**

**Signature**

```java
public final void run()
```

**Description**

This method is only to be used by the infrastructure, and should not be called by the application.

The `handleAsyncEvent()` family of methods provides the equivalent functionality to `Runnable.run()` for asynchronous event handlers, including execution of the `logic` argument passed to this object’s constructor. Applications should
override that method or provide a logic object for the default implementation to invoke.

**getPendingFireCount**

*Signature*

```java
protected int getPendingFireCount()
```

*Description*

This is an accessor method for `fireCount`. The `fireCount` field nominally holds the number of times associated instances of `AsyncEvent` 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.

*Returns*

the value held by `fireCount`.

**getAndDecrementPendingFireCount**

*Signature*

```java
protected int getAndDecrementPendingFireCount()
```

*Description*

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:

```java
public void handleAsyncEvent()
{
    <setup>
    do
    {
        <handle the event>
    }
```

RTSJ 2.0 (Draft 66) 313
8 Asynchrony

AsyncLongEventHandler

```java
while(getAndDecrementPendingFireCount() > 0);
}
```

This construction is necessary only in cases 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.

**Returns**
the value held by fireCount prior to decrementing it by one.

**getAndClearPendingFireCount**

**Signature**
```java
protected int
getAndClearPendingFireCount()
```

**Description**
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 be used by handlers for which the logic can accommodate multiple releases
in a single execution.

The general form for using this is

```java
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.

**Returns**
the value held by fireCount prior to setting the value to zero.
AsyncObjectEvent

8.3.2.8 AsyncObjectEvent

public class AsyncObjectEvent<P>

Inheritance
java.lang.Object
   AsyncBaseEvent
      AsyncObjectEvent<P>

Description
A new type of event that carries an object as a payload.

See Section AsyncEvent

Available since RTSJ 2.0

8.3.2.8.1 Constructors

AsyncObjectEvent

Signature
public
AsyncObjectEvent()

Description
Creates a new AsyncObjectEvent instance.

8.3.2.8.2 Methods

fire(P)

Signature
public void fire(P value)
throws MITViolationException,
EventQueueOverflowException,
IllegalAssignmentError

Description
When enabled, fires this instance of AsyncObjectEvent. The asynchronous event handlers associated with this event will be released with the object passed by fire. When no handlers are attached or this object is disabled the method does nothing, i.e., it skips the release. For an instance of AsyncEvent that has more than one instance of AsyncEventHandler,

- when one of these handlers throws an exception, all instances of AsyncEventHandler not affected by the exception must be released normally before the exception is propagated, and
- when more than one of these handlers throws an exception, the propagation of MITViolationException has precedence over ArrivalTimeQueueOverflowException, which has precedence over all others.

The later case can only occur when more than one of the handlers has a release parameters instance of type SporadicParameters, since only them can MITViolationException and ArrivalTimeQueueOverflowException be thrown.

Parameters
value—The payload passed to the event.

Throws
MITViolationException—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 releases information, arrival time and payload, overflows. Only the handlers which do not cause this exception to be thrown are released in this situation. When fire is called from the infrastructure, such as for an ActiveEvent, this exception is ignored.

IllegalAssignmentError—when P is not assignable the event queue of one of the associated handlers.

8.3.2.9 AsyncObjectEventHandler
public class AsyncObjectEventHandler<P>

Inheritance
java.lang.Object
   AsyncBaseEventHandler
   AsyncObjectEventHandler<P>

Description
A version of AsyncBaseEventHandler that carries an Object value as payload.

Available since RTSJ 2.0

8.3.2.9.1 Constructors

AsyncObjectEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, ReleaseRunner, Consumer)

Signature
public
AsyncObjectEventHandler(SchedulingParameters scheduling,
   ReleaseParameters release,
   MemoryParameters memory,
   MemoryArea area,
   ReleaseRunner runner,
   java.util.function.Consumer<P> logic)
   throws IllegalRealtimeArgumentException

Description
Creates an asynchronous event handler that receives a Long payload with each fire.

Parameters
scheduling—Parameter for scheduling the new handler (and possibly other instances of Schedulable). When scheduling is null and the creator is an instance of Schedulable, SchedulingParameters 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 task that is not an instance of Schedulable, the contents and
type of the new `SchedulingParameters` object are governed by the associated scheduler.

**release**—Parameter for scheduling the new handler (and possibly other instances of `Schedulable`). When `release` is `null` the new `AsyncEventHandler` will use a clone of the default `ReleaseParameters` for the associated scheduler created in the memory area that contains the `AsyncEventHandler` object.

**memory**—Parameter for scheduling the new handler (and possibly other instances of `Schedulable`). 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.

**area**—The initial memory area of this handler.

**runner**—Logic to be executed by `handleAsyncEvent`

**logic**—The logic to run for each fire. When `logic` is `null`, the `handleAsyncEvent` method in the new object will serve as its logic.

**Throws**

`IllegalRealtimeArgumentException`—when the event queue overflow policy is `QueueOverflowPolicy.DISABLE`.

### AsyncObjectEventHandler(SchedulingParameters, ReleaseParameters, Consumer)

**Signature**

```java
public AsyncObjectEventHandler(SchedulingParameters scheduling,
                               ReleaseParameters release,
                               java.util.function.Consumer<P> logic)
    throws IllegalRealtimeArgumentException
```

**Description**

Calling this constructor is equivalent to calling `AsyncObjectEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, ReleaseRunner, Consumer)` with arguments `(scheduling, release, null, null, null, logic)`. 

### AsyncObjectEventHandler(SchedulingParameters, ReleaseParameters)
AsyncObjectEventHandler

**Signature**

```java
public AsyncObjectEventHandler(SchedulingParameters scheduling,
                               ReleaseParameters release)
    throws IllegalRealtimeArgumentException
```

**Description**

Calling this constructor is equivalent to calling `AsyncObjectEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, ReleaseRunner, Consumer)` with arguments `(scheduling, release, null, null, null, null)`.

**AsyncObjectEventHandler(Consumer)**

**Signature**

```java
public AsyncObjectEventHandler(java.util.function.Consumer<P> logic)
```

**Description**

Calling this constructor is equivalent to calling `AsyncObjectEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, ReleaseRunner, Consumer)` with arguments `(null, null, null, null, null, null, logic)`.

**Parameters**

- `logic`—It is the function to call on the object received.

**AsyncObjectEventHandler**

**Signature**

```java
public AsyncObjectEventHandler()
```

**Description**

Creates an instance of `AsyncObjectEventHandler` (AOEH) with default values for all parameters.
8 Asynchrony

AsyncObjectEventHandler

See Section AsyncObjectEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, ReleaseRunner, Consumer)

8.3.2.9.2 Methods

handleAsyncEvent(P)

Signature
public void handleAsyncEvent(P value)

Description
This method holds the logic which is to be executed when any AsyncEvent 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 AOEH is a source of reference for its initial memory area while this AOEH is released.

All throwables from (or propagated through) handleAsyncEvent(P) are caught, a stack trace is printed and execution continues as if handleAsyncEvent(P) had returned normally.

peekPending

Signature
public P peekPending()
throws IllegalRealtimeStateException

Description
Determines the next value queued for handling.

Throws
IllegalRealtimeStateException—when the fire count is zero.

Returns
the object reference at the head of the queue of object references to be passed to `handleAsyncEvent`).

**run**

*Signature*

```java
public final void run()
```

*Description*

This method is only to be used by the infrastructure, and should not be called by the application.

The `handleAsyncEvent()` family of methods provides the equivalent functionality to `Runnable.run()` for asynchronous event handlers, including execution of the `logic` argument passed to this object's constructor. Applications should override that method or provide a `logic` object for the default implementation to invoke.

**getPendingFireCount**

*Signature*

```java
protected int getPendingFireCount()
```

*Description*

This is an accessor method for `fireCount`. The `fireCount` field nominally holds the number of times associated instances of `AsyncEvent` 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.

*Returns*

the value held by `fireCount`.

**getAndDecrementPendingFireCount**

*Signature*
protected int
getAndDecrementPendingFireCount()

Description
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 cases 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.

Returns
the value held by fireCount prior to decrementing it by one.

getAndClearPendingFireCount

Signature
protected int
getAndClearPendingFireCount()

Description
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 be used by handlers for which the logic can accommodate multiple releases in a single execution.

The general form for using this is

```java
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.

**Returns**

the value held by `fireCount` prior to setting the value to zero.

### 8.3.2.10 BlockableReleaseRunner

**public class BlockableReleaseRunner**

**Inheritance**

java.lang.Object
    ReleaseRunner
        BlockableReleaseRunner

**Description**

The default `ReleaseRunner` that can manage handlers that suspend themselves, e.g., by calling the `Object.wait()` method. It maintains a pool of threads to run releases. It guarantees that at least \( \min(\text{no. handlers}, \text{no. available CPUs} + \text{fraction of no. handlers}) \) are available for running releases, where the default fraction is defined by the constant `DEFAULT_LOADING_FACTOR`. For systems with many `AsyncBaseEventHandler` instances, there can be significantly fewer threads to run releases of those handlers in the system.

**Available since** RTSJ 2.0

### 8.3.2.10.1 Fields
8 Asynchrony

**DEFAULT_LOADING_FACTOR**

```java
public static final float DEFAULT_LOADING_FACTOR
```

*Description*

The value of *loading* used by the argumentless constructor.

### 8.3.2.10.2 Constructors

**BlockableReleaseRunner(ConfigurationParameters, RealtimeThreadGroup, float)**

*Signature*

```java
public BlockableReleaseRunner(ConfigurationParameters config,
                               RealtimeThreadGroup group,
                               float loading)
```

*Description*

Create a release runner which maintains a pool of threads to run releases of `AsyncBaseEventHandler` instances. The threads in the pool all run in a given `RealtimeThreadGroup` instance. The thread pool size is determined by the number of CPUs available to `group`, and the `loading`, such that at least `min( no. attached handlers, no. available CPUs + (loading * no. attached handlers)` are available for running releases.

*Parameters*

- `config`—the `ConfigurationParameters` object to use for all handler run from this pool, which means for each thread in the pool.
- `group`—for the pool threads.
- `loading`—a factor between 0.0 and 1.0 for deciding the number of threads in the pool.

**BlockableReleaseRunner(ConfigurationParameters)**

*Signature*

```java

```
public BlockableReleaseRunner(ConfigurationParameters config)

Description
Same as BlockableReleaseRunner(ConfigurationParameters, Real-timeThreadGroup, float) with arguments {null, null, 0.3f}.

8.3.2.10.3 Methods

getConfigurationParameters

Signature
public javax.realtime.ConfigurationParameters getConfigurationParameters()

Description
Get the ConfigurationParameters object used for all threads provided by this release runner.

Returns
those parameters.

release(AsyncBaseEventHandler)

Signature
protected final void release(AsyncBaseEventHandler handler)

Description
Finds a thread and has it call the AsyncBaseEventHandler.run() method. Care should be exercised when implementing this method, since it adds to both the latency and jitter of releasing events.
This method should only be called from the infrastructure.

Parameters
handler—The handler to be released.
attach(AsyncBaseEventHandler)

**Signature**

```java
protected final void
attach(AsyncBaseEventHandler handler)
throws IllegalRealtimeStateException
```

**Description**

Attach a handler from this runner, so it will be released. Adjusts the number of threads for running handlers accordingly.

**Parameters**

- `handler`—to be removed.

detach(AsyncBaseEventHandler)

**Signature**

```java
protected final void
detach(AsyncBaseEventHandler handler)
throws IllegalRealtimeStateException
```

**Description**

Detach a handler from this runner, so it will no longer be released. Adjusts the number of threads for running handlers accordingly.

**Parameters**

- `handler`—to be detached.

8.3.2.11 BoundAsyncEventHandler

```java
public class BoundAsyncEventHandler
```

**Inheritance**

- `java.lang.Object`
  - `AsyncBaseEventHandler`
    - `AsyncEventHandler`
      - `BoundAsyncEventHandler`

**Interfaces**

- `javax.realtime.BoundSchedulable`

**Description**

326 RTSJ 2.0 (Draft 66)
A bound asynchronous event handler is an instance of `AsyncEventHandler` 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.

### 8.3.2.11.1 Constructors

**BoundAsyncEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, RealtimeThreadGroup, ConfigurationParameters, Runnable)**

#### Signature

```
public
BoundAsyncEventHandler(SchedulingParameters scheduling, 
ReleaseParameters release, 
MemoryParameters memory, 
MemoryArea area, 
RealtimeThreadGroup group, 
ConfigurationParameters config, 
Runnable logic)
```

#### Description

Creates an instance of `BoundAsyncEventHandler` (BAEH) with the specified parameters. The newly-created handler inherits the affinity of its creator.

**Available since** RTSJ 2.0

**Parameters**

- `scheduling`—A `SchedulingParameters` object which will be associated with the constructed instance. When `null`, and the creator is not an instance of `Runnable`, a `SchedulingParameters` object is created which has the default `SchedulingParameters` for the scheduler associated with the current thread. When `null`, and the creator is an instance of `Runnable`, the `SchedulingParameters` are inherited from the current schedulable (a new `SchedulingParameters` object is cloned).
release—A ReleaseParameters object which will be associated with the constructed instance. When null, this will have default ReleaseParameters for the BAEH’s scheduler.

memory—A MemoryParameters object which will be associated with the constructed instance. When null, this will have no MemoryParameters and the handler can access the heap.

area—The MemoryArea for this. When null, the memory area will be that of the current thread/schedulable.

group—A RealtimeThreadGroup object which will be associated with the constructed instance. When null, this will not be associated with any realtime thread group.

config—The ConfigurationParameters associated with this (and possibly other instances of Schedulable). When config is null, this BoundAsyncEventHandler 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 is executed by AsyncEventHandler.handleAsyncEvent(). When null, the default handleAsyncEvent() method invokes nothing.

Throws
IllegalRealtimeArgumentException—when config is of type javax.realtime.memory.ScopedConfigurationParameters and logic, any parameter object, or this is in heap memory.

IllegalAssignmentError—when the new AsyncEventHandler instance cannot hold a reference to any value assigned to one of the scheduling, release, memory, or group parameters, or when those parameters cannot hold a reference to the new AsyncEventHandler. Also when the new AsyncEventHandler instance cannot hold a reference to values assigned to area or logic.

BoundAsyncEventHandler(SchedulingParameters, ReleaseParameters, Runnable)

Signature
public
BoundAsyncEventHandler(SchedulingParameters scheduling,
ReleaseParameters release,
Runnable logic)

Description

328 RTSJ 2.0 (Draft 66)
Creates an instance of `BoundAsyncEventHandler` with the specified parameters. The newly-created handler inherits the affinity of its creator.

Equivalent to `BoundAsyncEventHandler(scheduling, release, null, null, null, null, null)`

Available since RTSJ 2.0

**BoundAsyncEventHandler(SchedulingParameters, ReleaseParameters)**

**Signature**

```java
public BoundAsyncEventHandler(SchedulingParameters scheduling,
                            ReleaseParameters release)
```

**Description**

Creates an instance of `BoundAsyncEventHandler` with the specified parameters. The newly-created handler inherits the affinity of its creator.

Equivalent to `BoundAsyncEventHandler(scheduling, release, null, null, null, null, null)`

Available since RTSJ 2.0

**BoundAsyncEventHandler(Runnable)**

**Signature**

```java
public BoundAsyncEventHandler(Runnable logic)
```

**Description**

Creates an instance of `BoundAsyncEventHandler` with the specified parameters. The newly-created handler inherits the affinity of its creator.

Equivalent to `BoundAsyncEventHandler(null, null, null, null, null, null, logic)`

Available since RTSJ 2.0
BoundAsyncEventHandler

**Signature**

```java
public BoundAsyncEventHandler()
```

**Description**

Creates an instance of `BoundAsyncEventHandler`. The newly-created handler inherits the affinity of its creator.

Equivalent to `BoundAsyncEventHandler(null, null, null, null, null, null, null)`

---

### 8.3.2.12 BoundAsyncLongEventHandler

```java
public class BoundAsyncLongEventHandler
```

**Inheritance**

- `java.lang.Object`
  - `AsyncBaseEventHandler`
    - `AsyncLongEventHandler`
      - `BoundAsyncLongEventHandler`

**Interfaces**

- `javax.realtime.BoundSchedulable`

**Description**

A bound asynchronous event handler is an instance of `AsyncLongEventHandler` that is permanently bound to a dedicated realtime thread. Bound asynchronous long 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.

**Available since** RTSJ 2.0

---

### 8.3.2.12.1 Constructors

330 RTSJ 2.0 (Draft 66)
BoundAsyncLongEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, RealtimeThreadGroup, ConfigurationParameters, LongConsumer)

**Signature**

```
public
BoundAsyncLongEventHandler(SchedulingParameters scheduling,
                          ReleaseParameters release,
                          MemoryParameters memory,
                          MemoryArea area,
                          RealtimeThreadGroup group,
                          ConfigurationParameters config,
                          LongConsumer logic)
```

**Description**

Creates an instance of `BoundAsyncLongEventHandler` which specifies all possible parameters. The newly-created handler inherits the affinity of its creator.

**Parameters**

- **scheduling**—A `SchedulingParameters` object which will be associated with the constructed instance. When `null`, and the creator is not an instance of `Schedulable`, a `SchedulingParameters` object is created which has the default `SchedulingParameters` for the scheduler associated with the current thread. When `null`, and the creator is an instance of `Schedulable`, the `SchedulingParameters` are inherited from the current schedulable (a new `SchedulingParameters` object is cloned).
- **release**—A `ReleaseParameters` object which will be associated with the constructed instance. When `null`, this will have default `ReleaseParameters` for the BAEH’s scheduler.
- **memory**—A `MemoryParameters` object which will be associated with the constructed instance. When `null`, this will have no `MemoryParameters` and the handler can access the heap.
- **area**—The `MemoryArea` for this. When `null`, the memory area will be that of the current thread/schedulable.
- **group**—A `RealtimeThreadGroup` object which will be associated with the constructed instance. When `null`, this will not be associated with any realtime thread group.
- **config**—The `ConfigurationParameters` associated with this, and possibly other
instances of Schedulable. When config is null, this BoundAsyncEvent-
tHandler will reserve no space for preallocated exceptions and implementation-
specific values will be set to their implementation-defined defaults.

logic—The LongConsumer object whose accept() method is executed by
AsyncLongEventHandler.handleAsyncEvent(long). When null, the default
handleAsyncEvent(long) method invokes nothing.

Throws
IllegalRealtimeArgumentException—when config is of type javax.realtime.
memory.ScopedConfigurationParameters and logic, any parameter object,
or this is in heap memory.

IllegalAssignmentError—when the new AsyncEventHandler instance cannot
hold a reference to any value assigned to one of the scheduling, release,
memory, or group parameters, or when those parameters cannot hold a reference
to the new AsyncEventHandler. Also when the new AsyncEventHandler
instance cannot hold a reference to values assigned to area or logic.

BoundAsyncLongEventHandler(SchedulingParameters, ReleaseParameters, LongConsumer)

Signature
public
BoundAsyncLongEventHandler(SchedulingParameters scheduling,
ReleaseParameters release,
LongConsumer logic)

Description
Creates an instance of BoundAsyncLongEventHandler. This constructor is equivalent to
BoundAsyncLongEventHandler(scheduling, release, null, null, null, null, logic)

BoundAsyncLongEventHandler(SchedulingParameters, ReleaseParameters)

Signature
public
BoundAsyncLongEventHandler(SchedulingParameters scheduling,
ReleaseParameters release)
**Description**

Creates an instance of `BoundAsyncLongEventHandler`. Calling this constructor is equivalent to calling `BoundAsyncLongEventHandler(scheduling, release, null, null, null, null, null)`

**BoundAsyncLongEventHandler(LongConsumer)**

**Signature**

```
public BoundAsyncLongEventHandler(LongConsumer logic)
```

**Description**

Creates an instance of `BoundAsyncLongEventHandler`. Calling this constructor is equivalent to calling `BoundAsyncLongEventHandler(null, null, null, null, null, null, logic)`

**BoundAsyncLongEventHandler**

**Signature**

```
public BoundAsyncLongEventHandler()
```

**Description**

Creates an instance of `BoundAsyncLongEventHandler` using default values. Calling this constructor is equivalent to calling `BoundAsyncLongEventHandler(null, null, null, null, null, null)`

---

**8.3.2.13  BoundAsyncObjectEventHandler**

```java
public class BoundAsyncObjectEventHandler<P>

Inheritance
java.lang.Object
  AsyncBaseEventHandler
```

RTSJ 2.0 (Draft 66) 333
AsyncObjectEventHandler<P>
BoundAsyncObjectEventHandler<P>

Interfaces
javax.realtime.BoundSchedulable

Description
A bound asynchronous event handler is an instance of AsyncObjectEventHandler that is permanently bound to a dedicated realtime thread. Bound asynchronous object 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.

Available since RTSJ 2.0

8.3.2.13.1 Constructors

BoundAsyncObjectEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, ProcessingGroup, ConfigurationParameters, Consumer)

Signature
public
BoundAsyncObjectEventHandler(SchedulingParameters scheduling,
ReleaseParameters release,
MemoryParameters memory,
MemoryArea area,
ProcessingGroup group,
ConfigurationParameters config,
java.util.function.Consumer<P> logic)

Description
Creates an instance of BoundAsyncObjectEventHandler which specifies all possible parameters. The newly-created handler inherits the affinity of its creator.

Parameters
scheduling—A **SchedulingParameters** object which will be associated with the constructed instance. When null, and the creator is not an instance of **Scheduled**, a **SchedulingParameters** object is created which has the default **SchedulingParameters** for the scheduler associated with the current thread. When null, and the creator is an instance of **Scheduled**, the **SchedulingParameters** are inherited from the current schedulable (a new **SchedulingParameters** object is cloned).

release—A **ReleaseParameters** object which will be associated with the constructed instance. When null, this will have default **ReleaseParameters** for the BAEH’s scheduler.

memory—A **MemoryParameters** object which will be associated with the constructed instance. When null, this will have no **MemoryParameters** and the handler can access the heap.

area—The **MemoryArea** for this. When null, the memory area will be that of the current thread/schedulable.

group—A **RealtimeThreadGroup** object which will be associated with the constructed instance. When null, this will not be associated with any realtime thread group.

config—The **ConfigurationParameters** associated with this, and possibly other instances of **Scheduled**. When config is null, this **BoundAsyncEventHandler** will reserve no space for preallocated exceptions and implementation-specific values will be set to their implementation-defined defaults.

logic—The **Consumer** object whose `accept()` method is executed by **AsyncObjectEventHandler.handleAsyncEvent**. When null, the default `handleAsyncEvent` method invokes nothing.

**Throws**

**IllegalRealtimeArgumentException**—when config is of type `javax.realtime.memory.ScopedConfigurationParameters` and logic, any parameter object, or this is in heap memory.

**IllegalAssignmentError**—when the new AsyncEventHandler instance cannot hold a reference to any value assigned to one of the scheduling, release, memory, or group parameters, or when those parameters cannot hold a reference to the new AsyncEventHandler. Also when the new AsyncEventHandler instance cannot hold a reference to values assigned to area or logic.

**BoundAsyncObjectEventHandler**(SchedulingParameters, ReleaseParameters, Consumer)
**Signature**

```java
public BoundAsyncObjectEventHandler(SchedulingParameters scheduling,
ReleaseParameters release,
java.util.function.Consumer<P> logic)
```

**Description**

Creates an instance of `BoundAsyncObjectEventHandler`. This constructor is equivalent to `BoundAsyncObjectEventHandler(scheduling, release, null, null, null, null, logic)`

---

**BoundAsyncObjectEventHandler(SchedulingParameters, ReleaseParameters)**

**Signature**

```java
public BoundAsyncObjectEventHandler(SchedulingParameters scheduling,
ReleaseParameters release)
```

**Description**

Creates an instance of `BoundAsyncObjectEventHandler`. Calling this constructor is equivalent to calling `BoundAsyncObjectEventHandler(scheduling, release, null, null, null, null, null)`

---

**BoundAsyncObjectEventHandler(Consumer)**

**Signature**

```java
public BoundAsyncObjectEventHandler(java.util.function.Consumer<P> logic)
```

**Description**

Creates an instance of `BoundAsyncObjectEventHandler`. Calling this constructor is equivalent to calling `BoundAsyncObjectEventHandler(null, null, null, null, null, null, logic)`

---

336 RTSJ 2.0 (Draft 66)
**BoundAsyncObjectEventHandler**

*Signature*

```
public BoundAsyncObjectEventHandler()
```

*Description*

Creates an instance of `BoundAsyncObjectEventHandler` using default values. This constructor is equivalent to `BoundAsyncObjectEventHandler(null, null, null, null, null, null, null)`

### 8.3.2.14 FixedPriorityReleaseRunner

**public class** `FixedPriorityReleaseRunner`

**Inheritance**

`java.lang.Object`

`ReleaseRunner`

`FixedPriorityReleaseRunner`

*Description*

A release runner that has only one thread per priority. It provides faster dispatching, but should not be used with handlers that can be blocked. All handlers must also have the same affinity.

**Available since** RTSJ 2.0

### 8.3.2.14.1 Methods

**getConfigurationParameters**

*Signature*

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

*Description*
Get the ConfigurationParameters object used for all threads provided by this release runner.

Returns
those parameters.

release(AsyncBaseEventHandler)

Signature
protected final void
release(AsyncBaseEventHandler handler)

Description
Finds a thread and has it call the AsyncBaseEventHandler.run() method. Care should be exercised when implementing this method, since it adds to both the latency and jitter of releasing events.
This method should only be called from the infrastructure.

Parameters
handler—The handler to be released.

attach(AsyncBaseEventHandler)

Signature
protected final synchronized void
attach(AsyncBaseEventHandler handler)

Description
Notifies this runner that the handler is now associated with it.
This method should only be called from the infrastructure.

Parameters
handler—The handler to be attached

Throws
IllegalRealtimeStateException—when handler is already attached.

detach(AsyncBaseEventHandler)

Signature

338 RTSJ 2.0 (Draft 66)
protected final synchronized void
detach(AsyncBaseEventHandler handler)

Description

Notifies this runner that the handler is no longer associated with it.
This method should only be called from the infrastructure.

Parameters

- handler: The handler to be removed

Throws

- IllegalRealtimeStateException: when handler is not attached.

8.3.2.15 ReleaseRunner

public abstract class ReleaseRunner

Inheritance

java.lang.Object
  ReleaseRunner

Description

Manages a pool of threads to execute asynchronous event handler releases. The implementer is responsible for maintaining the pool of threads and ensuring they all have at least the desired ConfigurationParameters, RealtimeThreadGroup, and Affinity.

The other parameters for instances of Schedulable can either be set for each release or be configurable for the pool. In the latter case, one should not be able to associate a handler with the runner that has an incompatible parameter set. These other parameters are SchedulingParameters, ReleaseParameters, and MemoryParameters, as well as the MemoryArea in which the release should take place.

The default release runner, BlockableReleaseRunner, sets these other parameters on the releasing thread at each release. Since there may be a performance penalty for doing this, an application can define its own release runners for commonly occurring cases of these parameters. It is then up to the application to ensure that handlers are matched to the correct release runner.

Available since RTSJ 2.0
8 Asynchrony

8.3.2.15.1 Constructors

ReleaseRunner(RealtimeThreadGroup)

Signature
protected
ReleaseRunner(RealtimeThreadGroup group)

Description
Enables creating a subclass of this class.

8.3.2.15.2 Methods

getRealtimeThreadGroup

Signature
protected javarrealtime.RealtimeThreadGroup
getRealtimeThreadGroup()

Description
Determine the RealtimeThreadGroup instance used.

Returns
the RealtimeThreadGroup instance used by all threads used for running releases.

getConfigurationParameters

Signature
public abstract javarrealtime.ConfigurationParameters
getConfigurationParameters()

Description
Get the ConfigurationParameters object used for all threads provided by this release runner.
Returns
those parameters.

**release(AsyncBaseEventHandler)**

**Signature**

```java
protected abstract void
release(AsyncBaseEventHandler handler)
```

**Description**

Finds a thread and has it call the `AsyncBaseEventHandler.run()` method. Care should be exercised when implementing this method, since it adds to both the latency and jitter of releasing events.

This method should only be called from the infrastructure.

**Parameters**

- **handler**—The handler to be released.

**attach(AsyncBaseEventHandler)**

**Signature**

```java
protected abstract void
attach(AsyncBaseEventHandler handler)
```

**Description**

Notifies this runner that the handler is now associated with it.

This method should only be called from the infrastructure.

**Parameters**

- **handler**—The handler to be attached

**Throws**

- **IllegalRealtimeStateException**—when `handler` is already attached.

**detach(AsyncBaseEventHandler)**

**Signature**

```java
protected abstract void
detach(AsyncBaseEventHandler handler)
```
8 Asynchrony

    throws IllegalRealtimeStateException

Description

Notifies this runner that the handler is no longer associated with it.

This method should only be called from the infrastructure.

Parameters

    handler—The handler to be removed

Throws

    IllegalRealtimeStateException—when handler is not attached.

8.4 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 AsyncBaseEventHandler) or alternatively as bound to a dedicated realtime
thread (a instance of BoundSchedulable). The RTSJ does not define at what point
an unbound 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.
Chapter 9

Time

Realtime systems must be able to handle both very short time durations and very long ones. They also need 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. Nor do they provide 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 may not be created, as the class exists to provide a common parent type for 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, which can be used to describe a time duration.

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 or a time duration, and
- provide simple arithmetic operations for using them.

All time handling is based on these classes.

9.1 Definitions

Time Object — An instance of AbsoluteTime or RelativeTime. A time object is always associated with some Chronograph. By default, it is associated with
the realtime clock.

**Universal epoch** — The time at which the universal clock began ticking, defined by fiat as January 1, 1970 00:00:00 UTC.

**Epoch** — The date and time relative to which times on a RTSJ Chronograph \( c \) are determined. The epoch for a chronograph is defined in terms of the Universal epoch, and is represented as the time elapsed on the Universal clock since the Universal epoch at the time that \( c \) would have returned a time stamp of 0 ms and 0 ns.

**Time Value Representation** — 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`.

**Normalized (Canonical) Time Value** — Unique 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 four constraints:

1. when both millisecond and nanosecond components are nonzero, they have the same sign;
2. the algebraic time values of the time object is the algebraic sum of the two components;
3. the millisecond component represents the algebraic number of milliseconds in the time object, within a range of \([-2^{63}, 2^{63} - 1]\); and
4. 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, \((\text{MAX_LONG} \text{ milliseconds}, \text{MAX_INT} \text{ nanoseconds})\) cannot be normalized and is an illegal value.

The following table has examples of normalized representations.

### 9.2 Semantics

The points below define the general semantics of the time classes. Semantics specific to particular classes, constructors, methods, and fields are in the class description and the constructor, method, and field detail sections.

1. All time objects must maintain nanosecond precision and report their values in terms of millisecond and nanosecond constituents.
2. Time objects can 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 `Chronograph` interface.
Table 9.1: Examples of Normalized Times

<table>
<thead>
<tr>
<th>time in ns</th>
<th>millis</th>
<th>nanos</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000000</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>1999999</td>
<td>1</td>
<td>999999</td>
</tr>
<tr>
<td>1000001</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>-999999</td>
<td>0</td>
<td>-999999</td>
</tr>
<tr>
<td>-1000000</td>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>-1000001</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

3. Time objects maintain and report time values in normalized form, but the normalized form is not required for input parameter values. This enables computation to be performed individually with the constituent time parts, using the full signed range and restrictions of the underlying type.

(a) 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.

i. 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.

ii. 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.

iii. The resulting components are then represented, and reported when necessary, with the above computed sign.

(b) 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.

(c) 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.
(d) 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.

4. 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.

5. A time object is always associated with a Chronograph. By default it is associated with the realtime clock. Chronographs are involved both in the setting as well as the usage of time objects, for example in comparisons.

6. Methods are provided to facilitate the handling of time objects generically via the HighResolutionTime class. These methods enable converting, according to a Chronograph, between AbsoluteTime objects and RelativeTime objects. These methods also enable changing the Chronograph 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:

<table>
<thead>
<tr>
<th>Chronograph association &amp; conversion</th>
<th>returned/updated object</th>
</tr>
</thead>
<tbody>
<tr>
<td>an_absolute.absolute(chronograph_a)</td>
<td>chronograph_a</td>
</tr>
<tr>
<td></td>
<td>ms,ns</td>
</tr>
<tr>
<td>an_absolute.absolute(chronograph_b)</td>
<td>chronograph_b</td>
</tr>
<tr>
<td></td>
<td>ms,ns</td>
</tr>
<tr>
<td>an_absolute.absolute(null)</td>
<td>realtime_clock</td>
</tr>
<tr>
<td></td>
<td>ms,ns</td>
</tr>
<tr>
<td>an_absolute.relative(chronograph_a)</td>
<td>chronograph_a</td>
</tr>
<tr>
<td></td>
<td>chronograph_a.getTime().subtract(ms,ns)</td>
</tr>
<tr>
<td>an_absolute.relative(chronograph_b)</td>
<td>chronograph_b</td>
</tr>
<tr>
<td></td>
<td>chronograph_b.getTime().subtract(ms,ns)</td>
</tr>
<tr>
<td>an_absolute.relative(null)</td>
<td>realtime_clock</td>
</tr>
<tr>
<td></td>
<td>realtime_clock.getTime().subtract(ms,ns)</td>
</tr>
<tr>
<td>a_relative.relative(chronograph_a)</td>
<td>chronograph_a</td>
</tr>
<tr>
<td></td>
<td>ms,ns</td>
</tr>
<tr>
<td>a_relative.relative(chronograph_b)</td>
<td>chronograph_b</td>
</tr>
<tr>
<td></td>
<td>ms,ns</td>
</tr>
<tr>
<td>a_relative.relative(null)</td>
<td>realtime_clock</td>
</tr>
<tr>
<td></td>
<td>ms,ns</td>
</tr>
<tr>
<td>a_relative.absolute(chronograph_a)</td>
<td>chronograph_a</td>
</tr>
<tr>
<td></td>
<td>chronograph_a.getTime().add(ms,ns)</td>
</tr>
<tr>
<td>a_relative.absolute(clock_b)</td>
<td>chronograph_b</td>
</tr>
<tr>
<td></td>
<td>chronograph_b.getTime().add(ms,ns)</td>
</tr>
<tr>
<td>a_relative.absolute(null)</td>
<td>realtime_clock</td>
</tr>
<tr>
<td></td>
<td>realtime_clock.getTime().add(ms,ns)</td>
</tr>
</tbody>
</table>

7. Time objects must implement the Comparable interface.
9.3  javax.realtime

9.3.1  Classes

9.3.1.1  AbsoluteTime

public class AbsoluteTime

Inheritance
java.lang.Object
    HighResolutionTime<AbsoluteTime>
    AbsoluteTime

Description
An object that represents a specific point in time given by milliseconds plus nanoseconds past some point in time fixed by its Chronograph. For the universal 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.

9.3.1.1.1  Constructors

AbsoluteTime(long, int, Chronograph)

Signature
public
    AbsoluteTime(long millis,
               int nanos,
               Chronograph chronograph)
throws IllegalRealtimeArgumentException

Description

Constructs 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 that 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 calendar clock, but other classes of chronograph may define other epochs.

Available since RTSJ 2.0

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—Provides the time reference for the newly constructed object. The realtime clock is used when this argument is null.

Throws

IllegalRealtimeArgumentException—when there is an overflow in the millisecond component when normalizing.

AbsoluteTime(long, int)

Signature

public
    AbsoluteTime(long millis, int nanos)
    throws IllegalRealtimeArgumentException

Description

Equivalent to AbsoluteTime(long, int, Chronograph) with the argument list (millis, nanos, null)
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

- `IllegalRealtimeArgumentException`—when there is an overflow in the millisecond component when normalizing.

AbsoluteTime(Date, Chronograph)

Signature

```java
public AbsoluteTime(Date date,
                   Chronograph chronograph)
    throws IllegalRealtimeArgumentException
```

Description

Equivalent to `AbsoluteTime(long, int, Chronograph)` with the argument list `(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 time reference is given by the `Chronograph` parameter. When `Chronograph` is `null` the association is made with the default realtime clock.

Available since RTSJ 2.0

Parameters

- `date`—The `java.util.Date` representation of the time past the `epoch`.
- `chronograph`—Provides the time reference for the newly constructed object.

Throws

- `IllegalRealtimeArgumentException`—when the `date` parameter is `null`.

AbsoluteTime(Date)

Signature

```java
RTSJ 2.0 (Draft 66) 349
```
public AbsoluteTime(Date date)
    throws IllegalRealtimeArgumentException

Description
Equivalent to AbsoluteTime(long, int, Chronograph) with the argument list
(date.getTime(), 0, null).

Parameters
date—The java.util.Date representation of the time past the epoch.

Throws
IllegalRealtimeArgumentException—when the date parameter is null.

AbsoluteTime(AbsoluteTime)

Signature
public AbsoluteTime(AbsoluteTime time)
    throws IllegalRealtimeArgumentException

Description
Equivalent to AbsoluteTime(long, int, Chronograph) with the argument list
(time.getMilliseconds(), time.getNanoseconds(), time.getChronograph()).

Parameters
time—The AbsoluteTime object which is the source for the copy.

Throws
IllegalRealtimeArgumentException—when the time parameter is null.

AbsoluteTime(Chronograph)

Signature
public AbsoluteTime(Chronograph chronograph)

Description
AbsoluteTime

Equivalent to \texttt{AbsoluteTime(long, int, Chronograph)} with the argument list \((0, 0, \text{chronograph})\).

Available since RTSJ 2.0

Parameters

\texttt{chronograph}—Provides the time reference for the newly constructed object.

\textbf{AbsoluteTime}

\textit{Signature}

\begin{verbatim}
    public AbsoluteTime()
\end{verbatim}

\textit{Description}

Equivalent to \texttt{AbsoluteTime(long, int, Chronograph)} with the argument list \((0, 0, \text{null})\).

\section*{9.3.1.1.2 Methods}

\textbf{absolute(Chronograph)}

\textit{Signature}

\begin{verbatim}
    public javax.realtime.AbsoluteTime absolute(Chronograph chronograph)
\end{verbatim}

\textit{Description}

Creates a copy of \texttt{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 \texttt{abstract} method of the \texttt{HighResolutionTime} base class. No conversion into \texttt{AbsoluteTime} is needed in this case. The result is associated with the \texttt{Chronograph} passed as a parameter. When \texttt{Chronograph} is \texttt{null}, the association is made with the default realtime clock.

\textit{Parameters}

\texttt{chronograph}—It is used only as the new time reference associated with the result, since no conversion is needed.
Returns
The copy of this in a newly allocated AbsoluteTime object, associated with the Chronograph parameter.

Available since RTSJ 2.0

absolute(Chronograph, AbsoluteTime)

Signature
public javax.realtime.AbsoluteTime absolute(Chronograph chronograph,
                                          AbsoluteTime dest)

Description
Copies this into dest, when necessary modified 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 result is associated with the Chronograph passed as a parameter. When Chronograph is null, the association is made with the default realtime clock.

Parameters
chronograph—It is used only as the new time reference associated with the result, since no conversion is needed.
dest—the instance to fill.

Returns
The copy of this in a newly allocated AbsoluteTime object, associated with the Chronograph parameter.

Available since RTSJ 2.0

relative(Chronograph)

Signature
public javax.realtime.RelativeTime relative(Chronograph chronograph)

Description

352      RTSJ 2.0 (Draft 66)
Converts the time of this to a relative time, using the given instance of Chronograph to determine the current time. The calculation is the current time indicated by the given instance of Chronograph 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 time reference of the result is given by the Chronograph passed as a parameter.

Parameters
chronograph—The instance of Chronograph used to convert the time of this into relative time, and the new chronograph 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.

Available since RTSJ 2.0

relative(Chronograph, RelativeTime)

Signature
public javax.realtime.RelativeTime
relative(Chronograph chronograph,
    RelativeTime dest)

Description
Converts the time of this to a relative time, using the given instance of Chronograph to determine the current time. The calculation is the current time indicated by the given instance of Chronograph 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 in it and returned. Otherwise, a new object is allocated for the result. The time reference of the result is given by the Chronograph passed as a parameter.

Parameters
chronograph—The instance of Chronograph 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 in it and returned.

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, associated with the Chronograph
parameter.

add(long, int)

Signature
public javax.realtime.AbsoluteTime
add(long millis,
    int nanos)
    throws ArithmeticException

Description
Creates 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
chronograph association as this.

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)

Signature
public javax.realtime.AbsoluteTime
add(long millis,
    int nanos,
    AbsoluteTime dest)
    throws ArithmeticException

Description
Returns 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 in it and returned. Otherwise, a new object is allocated for the result. The result will have the same chronograph association as this, and the chronograph association with dest is ignored.

**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 in it and returned.

**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)

**Signature**

```java
def public javax.realtime.AbsoluteTime add(RelativeTime time) throws ArithmeticException, IllegalRealtimeArgumentException
```

**Description**

Creates a new instance of AbsoluteTime representing the result of adding time to the value of this and normalizing the result. The Chronograph associated with this and the Chronograph associated with the time parameter must be the same, and such association is used for the result.

**Parameters**

- **time**—The time to add to this.

**Throws**

- IllegalRealtimeArgumentException—when the Chronograph associated with this and the Chronograph 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)

Signature

```java
generic javax.realtime.AbsoluteTime
add(RelativeTime time,
    AbsoluteTime dest)
throws ArithmeticException,
       IllegalRealtimeArgumentException
```

Description

Returns 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 in it and returned. Otherwise, a new object is allocated for the result. The Chronograph associated with this and the Chronograph associated with the time parameter must be the same, and such association is used for the result. The Chronograph associated with the dest parameter is ignored.

Parameters

time—The time to add to this.
dest—When dest is not null, the result is placed in it and returned.

Throws

- `IllegalRealtimeArgumentException`—when the Chronograph associated with this and the Chronograph 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.

gDate

Signature

```java
public java.util.Date
gDate()
throws UnsupportedRealtimeOperationException
```

Description

Converts the time given by this to a Date format. Note that Date represents time as milliseconds so the nanoseconds of this will be lost.
**Throws**

UnsupportedRealtimeOperationException—when the Chronograph 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)**

**Signature**

```java
public javax.realtime.AbsoluteTime set(Date date) throws IllegalRealtimeArgumentException
```

**Description**

Changes 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 chronograph association is implicitly made with the default realtime clock.

**Parameters**

date—A reference to a Date which will become the time represented by this after the completion of this method.

**Throws**

IllegalRealtimeArgumentException—when the parameter date is null.

**Returns**

this

Available since RTSJ 2.0 returns itself

**subtract(AbsoluteTime)**

**Signature**

```java
public javax.realtime.RelativeTime subtract(AbsoluteTime time) throws IllegalRealtimeArgumentException, ArithmeticException
```

**Description**

RTSJ 2.0 (Draft 66) 357
Creates a new instance of `RelativeTime` representing the result of subtracting `time` from the value of `this` and normalizing the result. The `Chronograph` associated with `this` and the `Chronograph` associated with the `time` parameter must be the same, and such association is used for the result.

**Parameters**
- `time`—The time to subtract from `this`.

**Throws**
- `IllegalRealtimeArgumentException`—when the `Chronograph` associated with `this` and the `Chronograph` 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)

**Signature**
```java
public javax.realtime.RelativeTime subtract(AbsoluteTime time,
                                           RelativeTime dest)
throw IllegalRealtimeArgumentException,
       ArithmeticException
```

**Description**
Returns 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 `Chronograph` associated with `this` and the `Chronograph` associated with the `time` parameter must be the same, and such association is used for the result. The `Chronograph` associated with the `dest` parameter is ignored.

**Parameters**
- `time`—The time to subtract from `this`.
- `dest`—When `dest` is not `null`, the result is placed in it and returned.

**Throws**
- `IllegalRealtimeArgumentException`—when the `Chronograph` associated with `this` and the `Chronograph` 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)

Signature
public javax.realtime.AbsoluteTime subtract(RelativeTime time)
   throws IllegalRealtimeArgumentException, ArithmeticException

Description
Creates a new instance of AbsoluteTime representing the result of subtracting time from the value of this and normalizing the result. The Chronograph associated with this and the Chronograph associated with the time parameter must be the same, and such association is used for the result.

Parameters
time—The time to subtract from this.

Throws
IllegalRealtimeArgumentException—when the Chronograph associated with this and the Chronograph 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)

Signature
public javax.realtime.AbsoluteTime subtract(RelativeTime time,
   AbsoluteTime dest)
9 Time

```java
throws IllegalRealtimeArgumentException,
ArithmeticException
```

**Description**

Returns 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 `Chronograph` associated with `this` and the `Chronograph` associated with the `time` parameter must be the same, and such association is used for the result. The `Chronograph` associated with the `dest` parameter is ignored.

**Parameters**
- `time`—The time to subtract from `this`.
- `dest`—When `dest` is not `null`, the result is placed there and returned.

**Throws**
- `IllegalRealtimeArgumentException`—when the `Chronograph` associated with `this` and the `Chronograph` 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**

**Signature**

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

**Description**

Creates 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)`

**Returns**
a String object converted from the time given by `this`.  

360 RTSJ 2.0 (Draft 66)
9.3.1.2 HighResolutionTime

public abstract class HighResolutionTime<T extends HighResolutionTime<T>>

Inheritance
java.lang.Object
   HighResolutionTime<T extends HighResolutionTime<T>>

Interfaces
Comparable
Cloneable

Description
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 or RelativeTime 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.

9.3.1.2.1 Methods

waitForObject(Object, HighResolutionTime)

Signature
public static boolean
waitForObject(Object target,
   HighResolutionTime<?> time)
throws InterruptedException,
   IllegalMonitorStateException,
   IllegalRealtimeArgumentException,
   UnsupportedRealtimeOperationException

Description
Behaves like target.wait() but with the enhancement that it waits with a precision of HighResolutionTime and returns true when the associated notify
was 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`.

**Parameters**

- `target`—The object for 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` or `AsynchronouslyInterruptedException.fire` while it is waiting.
- `IllegalRealtimeArgumentException`—when `time` represents a relative time less than zero.
- `IllegalMonitorStateException`—when `target` is not locked by the caller.
- `UnsupportedRealtimeOperationException`—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.

**Available since** RTSJ 2.0 updated to add a return value.

**equals(T)**

**Signature**

```java
public boolean equals(T time)
```

**Description**

Proves if the argument `time` has the same type and values as `this`.

Equality includes `Chronograph` association.

**Parameters**

- `time`—Value to be compared with `this`.

---

362 RTSJ 2.0 (Draft 66)
Returns
true when the parameter time is of the same type and has the same values as this, as well as the same Chronograph association.

Available since RTSJ 2.0

getchronograph

Signature
public final javax.realtime.Clock getClock()
throws UnsupportedRealtimeOperationException

Description
Gets the reference to the clock associated with this.

Throws
UnsupportedRealtimeOperationException—when the time is based on a Chronograph that is not a Clock.

Returns
a reference to the clock associated with this.

Available since RTSJ 1.0.1

getchronograph

Signature
public final javax.realtime.Chronograph getChronograph()
getMilliseconds

Signature
   public final long
getMilliseconds()

Description
   Gets the milliseconds component of this.

Returns
   the milliseconds component of the time represented by this.

getNanoseconds

Signature
   public final int
getNanoseconds()

Description
   Gets the nanoseconds component of this.

Returns
   the nanoseconds component of the time represented by this.

set(T)

Signature
   public T extends javax.realtime.HighResolutionTime<T>
set(T time)

Description
   Changes the value represented by this to that of the given time. The Chronograph associated with this is set to be the Chronograph associated with the time parameter.

Parameters
   time—The new value for this.

Throws
   IllegalRealtimeArgumentException—when the parameter time is null.
ClassCastException—when the type of this and the type of the parameter time are not the same.

Returns
this

Available since RTSJ 1.0.1 The description of the method in 1.0 was erroneous.

Available since RTSJ 2.0 returns itself

set(Chronograph, long, int)

Signature
public T extends javax.realtime.HighResolutionTime<T>
    set(Chronograph chronograph,
       long millis,
       int nanos)
    throws IllegalRealtimeArgumentException

Description
Sets the all components of this. The setting is subject to parameter normalization. When after normalization the time is negative, the time represented by this is set to a negative value, but note, negative times are not supported everywhere. For instance, a negative relative time is an invalid value for a periodic thread’s period.

Parameters
chronograph—The time reference for the other components of this set during the call call.
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
IllegalRealtimeArgumentException—when there is an overflow in the millisecond component while normalizing.

Returns
this

Available since RTSJ 2.0 returns itself
set(long, int)

Signature
public T extends javax.realtime.HighResolutionTime<T>
    set(long millis,
        int nanos)
    throws IllegalRealtimeArgumentException

Description
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.

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
IllegalRealtimeArgumentException—when there is an overflow in the millisecond component while normalizing.

Returns
this

Available since RTSJ 2.0 returns itself

set(long)

Signature
public T extends javax.realtime.HighResolutionTime<T>
    set(long millis)

Description
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).

Parameters
HighResolutionTime

**millis**—This value shall be the value of the millisecond component of this at the completion of the call.

*Returns*

this

*Available since* RTSJ 2.0 returns itself

**hashCode**

*Signature*

```java
public int hashCode()
```

*Description*

Returns a hash code for this object in accordance with the general contract of `Object.hashCode`. Time objects that are equal, as defined by `equals`, have the same hash code.

*Returns*

the hashcode value for this instance.

**clone**

*Signature*

```java
public java.lang.Object clone()
```

*Description*

Returns 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

**compareTo(T)**

*Signature*

RTSJ 2.0 (Draft 66) 367
public int compareTo(T time)

Description

Compares this HighResolutionTime with the specified HighResolutionTime time.

Parameters

time—To be compared 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 chronograph 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.

Available since RTSJ 2.0

equals(Object)

Signature

public boolean equals(Object object)

Description

Determined whether or not the argument object has the same type and values as this.

Equality includes Chronograph association.

Parameters

object—Value to be compared with this.

Returns

ture when the parameter object is of the same type and has the same values as this, as well as the same Chronograph association.

absolute(Chronograph, AbsoluteTime)

Signature

368 RTSJ 2.0 (Draft 66)
public abstract javax.realtime.AbsoluteTime
absolute(Chronograph chronograph,
    AbsoluteTime dest)

Description
Converts the time of this to an absolute time, using the given instance of
Chronograph to determine the current time when necessary. When Chronograph
is null the default realtime clock is assumed. When dest is not null, the result is
placed in it 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.

Parameters
chronograph—The instance of Chronograph 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 in it and returned.

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.

absolute(Chronograph)

Signature
public abstract javax.realtime.AbsoluteTime
absolute(Chronograph chronograph)

Description
Converts the time of this to an absolute time, using the given instance of
Chronograph 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 chronograph
association of the result is the Chronograph passed as a parameter. See the
subclass comments for more specific information.

Parameters
chronograph—The instance of Chronograph 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
Chronograph parameter.
relative(Chronograph, RelativeTime)

Signature

```
public abstract javax.realtime.RelativeTime
relative(Chronograph chronograph,
         RelativeTime dest)
```

Description

Converts the time of this to a relative time, using the given instance of Chronograph to determine the current time when necessary. When Chronograph is null the 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 chronograph association of the result is the Chronograph passed as a parameter. See the subclass comments for more specific information.

Parameters

- **chronograph**—The instance of Chronograph 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 in it and returned.

Returns

the RelativeTime conversion in dest when dest is not null, otherwise the result is returned in a newly allocated object.

Available since RTSJ 2.0

relative(Chronograph)

Signature

```
public abstract javax.realtime.RelativeTime
relative(Chronograph chronograph)
```

Description

Converts the time of this to a relative time, using the given instance of Chronograph 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 chronograph association of the result is the Chronograph passed as a parameter. See the subclass comments for more specific information.

Parameters

- **chronograph**—The instance of Chronograph 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.

Available since RTSJ 2.0

9.3.1.3 RelativeTime

```java
public class RelativeTime
```

Inheritance

```
java.lang.Object
  HighResolutionTime<RelativeTime>
  RelativeTime
```

Description

An object that represents a time interval milliseconds/10³ + nanoseconds/10⁹
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 $[-2^{63}\text{ milliseconds } + (-10^6+1) \text{ nanoseconds}]$ to $[(2^{63}-1)\text{ milliseconds } + (10^6-1) \text{ nanoseconds}]$.

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.

9.3.1.3.1 Constructors

RelativeTime(long, int, Chronograph)

Signature

```java
public RelativeTime(long millis,
                     int nanos,
                     Chronograph chronograph)
```
RelativeTime

Throws IllegalRealtimeArgumentException

Description

Constructs a RelativeTime object representing an interval based on the parameter millis plus the parameter nanos. The construction is subject to millis and nanos parameter normalization. When there is an overflow in the millisecond component when normalizing then an IllegalRealtimeArgumentException 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

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 time reference of the newly constructed object. Defaults to the realtime clock when null.

Throws

IllegalRealtimeArgumentException—when there is an overflow in the millisecond component when normalizing.

RelativeTime(long, int)

Signature

public
RelativeTime(long millis,
int nanos)
throws IllegalRealtimeArgumentException

Description

Equivalent to RelativeTime(long, int, Chronograph) with argument list (millis, nanos, null).

Parameters

millis—The desired value for the millisecond component of this. The actual value is the result of parameter normalization.
RelativeTime

nanos—The desired value for the nanosecond component of this. The actual value is the result of parameter normalization.

Throws
IllegalRealtimeArgumentException—when there is an overflow in the millisecond component when normalizing.

RelativeTime(RelativeTime)

Signature
public RelativeTime(RelativeTime time)

Description
Equivalent to RelativeTime(long, int, Chronograph) with argument list (time.getMilliseconds(), time.getNanoseconds(), time.getChronograph()).

Parameters
time—The RelativeTime object which is the source for the copy.

RelativeTime(Chronograph)

Signature
public RelativeTime(Chronograph chronograph)

Description
Equivalent to RelativeTime(long, int, Chronograph) with argument list (0, 0, chronograph).

Available since RTSJ 2.0

Parameters
chronograph—The time reference for the newly constructed object.
public RelativeTime()

Description

Equivalent to RelativeTime(long, int, Chronograph) with argument list (0, 0, null).

9.3.1.3.2 Methods

absolute(Chronograph)

Signature

public javax.realtime.AbsoluteTime absolute(Chronograph chronograph)

Description

Converts the time of this to an absolute time, using the given instance of Chronograph 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 chronograph association of the result is the Chronograph passed as a parameter. See the subclass comments for more specific information.

Available since RTSJ 2.0

See Section HighResolutionTime.absolute(Chronograph)

absolute(Chronograph, AbsoluteTime)

Signature

public javax.realtime.AbsoluteTime absolute(Chronograph chronograph, AbsoluteTime dest)

Description

RTSJ 2.0 (Draft 66)
Converts the time of this to an absolute time, using the given instance of `Chronograph` to determine the current time when necessary. When `Chronograph` is `null` the default realtime clock is assumed. When `dest` is not `null`, the result is placed in it 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.

**Available since** RTSJ 2.0

See Section `HighResolutionTime.absolute(Chronograph, AbsoluteTime)`

### relative(Chronograph)

**Signature**

```java
public javax.realtime.RelativeTime relative(Chronograph chronograph)
```

**Description**

Converts the time of this to a relative time, using the given instance of `Chronograph` 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 chronograph association of the result is the `Chronograph` passed as a parameter. See the subclass comments for more specific information.

**Available since** RTSJ 2.0

See Section `HighResolutionTime.relative(Chronograph)`

### relative(Chronograph, RelativeTime)

**Signature**

```java
public javax.realtime.RelativeTime relative(Chronograph chronograph,
                                          RelativeTime dest)
```

**Description**

Converts the time of this to a relative time, using the given instance of `Chronograph` to determine the current time when necessary. When `Chronograph` is `null` the 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 chronograph association of the result is the Chronograph passed as a parameter. See the subclass comments for more specific information.

Available since RTSJ 2.0

See Section HighResolutionTime.relative(Chronograph, RelativeTime)

add(long, int)

Signature
public javax.realtime.RelativeTime
add(long millis,
    int nanos)
throws ArithmeticException

Description
Creates 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 chronograph association as this. An ArithmeticException is when the result does not fit in the normalized format.

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)

Signature
public javax.realtime.RelativeTime
add(long millis,
    int nanos,
    RelativeTime dest)
throws ArithmeticException

Description
Returns 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 chronograph association as this, and the chronograph association with dest is ignored.

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.

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)

Signature
public javax.realtime.RelativeTime
add(RelativeTime time)
throws IllegalRealtimeArgumentException, ArithmeticException

Description
Creates a new instance of RelativeTime representing the result of adding time to the value of this and normalizing the result.

The chronograph 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.

Parameters
time—The time to add to this.

Throws
IllegalRealtimeArgumentException—when the Chronograph associated with this and the Chronograph associated with the time parameter are different, or when the time parameter is null.
Addition

**add(RelativeTime, RelativeTime)**

**Signature**

```java
public javax.realtime.RelativeTime add(RelativeTime time, RelativeTime dest)
throws IllegalRealtimeArgumentException, ArithmeticException
```

**Description**

Returns 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 `Chronograph` associated with `this` and the `Chronograph` associated with the `time` parameter are expected to be the same, and such association is used for the result.

The `Chronograph` associated with the `dest` parameter is ignored.

**Parameters**

- `time`—The time to add to `this`.
- `dest`—When `dest` is not `null`, the result is placed there and returned.

**Throws**

- `IllegalRealtimeArgumentException`—when the `Chronograph` associated with `this` and the `Chronograph` 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.

**Subtraction**

**subtract(RelativeTime)**

**Signature**

```java
```
public javax.realtime.RelativeTime subtract(RelativeTime time) throws IllegalRealtimeArgumentException, ArithmeticException

Description

Creates a new instance of RelativeTime representing the result of subtracting time from the value of this and normalizing the result.

The Chronograph associated with this and the Chronograph associated with the time parameter are expected to be the same, and such association is used for the result.

Parameters

time—The time to subtract from this.

Throws

IllegalRealtimeArgumentException—when the Chronograph associated with this and the Chronograph 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.

subtract(RelativeTime, RelativeTime)

Signature

public javax.realtime.RelativeTime subtract(RelativeTime time, RelativeTime dest) throws IllegalRealtimeArgumentException, ArithmeticException

Description

Returns 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 Chronograph associated with this and the Chronograph associated with the time parameter are expected to be the same, and such association is used for the result.
The **Chronograph** associated with the `dest` parameter is ignored.

**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**

- `IllegalRealtimeArgumentException`—when the **Chronograph** associated with `this` and the **Chronograph** 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)

**Signature**

```java
public javax.realtime.RelativeTime
scale(int factor)
```

**Description**

Changes the length of this relative time by multiplying it by `factor`.

**Parameters**

- `factor`—Value by which to increase the time interval.

**Returns**

- a new object with value of this scaled by `factor`.

**Available since RTSJ 2.0**

### scale(int, RelativeTime)

**Signature**

```java
public javax.realtime.RelativeTime
scale(int factor,
     RelativeTime time)
```
**RelativeTime**

*Description*
Sets `time` to the value of `this` time multiplied by `factor`.

*Parameters*
- `factor` — Value by which to increase the time in `this`.
- `time` — Where to store the result.

*Returns*
`time` with the value of `this` scaled by `factor`.

Available since RTSJ 2.0

**compareToZero**

*Signature*
```java
public int compareToZero()
```

*Description*
Compares `this` to relative time zero returning the result of the comparison. Equivalent to `constantZero.compareTo(this)`.

*Returns*
negative when `this` is less than zero, 0, when it is equal to zero and a positive when `this` is greater than zero.

Available since RTSJ 2.0

**compareTo(RelativeTime)**

*Signature*
```java
public int compareTo(RelativeTime time)
```

*Description*
Compares `this` HighResolutionTime with the specified HighResolutionTime `time`.

*Parameters*
- `time` — To be compared with the time of `this`.

Throws

RTSJ 2.0 (Draft 66) 381
ClassCastException—when the time parameter is not of the same class as this.
IllegalRealtimeArgumentException—when the time parameter is not associated with the same chronograph 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.

Available since RTSJ 2.0

toString

Signature
  public java.lang.String toString()

Description
  Creates 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)'

Returns
a String object converted from the time given by this.

9.4 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.
9 Time
Chapter 10

Clocks and Timers

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 10.2.2).

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 AsyncBaseEventHandler 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 Clocks and Timers

10.1 Definitions

Timing Mechanism — Something capable of representing and following the progress of time, by means of time values.

Chronograph — A passive timing mechanism, which can only provide the current time.

Clock — An active timing mechanism, which can both provide the current time and cause some action when a particular time is reached. All clocks are, by definition, chronographs, but not necessarily vice versa.

Monotonically Increasing Timing Mechanism — A timing mechanism whose time values never decrease. Monotonicity is a Boolean property, while time synchronization, uniformity, and accuracy are characteristics that depend on agreed tolerances. All monotonic clocks referenced in this specification are monotonically increasing timing mechanisms.

Time Synchronization — 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 — The agreement between a chronograph and the true value that it measures, e.g., absolute wall clock time.

Resolution — The minimal time value interval that can be represented by the clock model.

Precision — The smallest tick size that a particular chronograph will observe.

Uniformity — In this context, 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 — 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. Jitter might be caused by short-term and noncumulative small time variation due to noise sources, such as thermal noise.

Stability — The resistance to jitter, in this case temporal jitter. Lack of stability can account for large and often cumulative variations, due to such occurrences such as supply voltage and temperature change.

Drift — The rate of change of the cumulative variation between two timing mechanisms.

Counting Time — The time accumulated by a Timer, while active, when created or rescheduled using a RelativeTime to specify the initial firing or skipping time. Counting Time is zeroed at the beginning of an activation and when rescheduled, while active, before the initial firing or skipping of an activation.
10.2 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.

10.2.1 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.

1. 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.

2. 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.

3. The precision of RTSJ definable clock and chronograph (and, by proxy, the precision of the timers associated with clocks) are defined in terms of nanoseconds per observable tick, and provided to the application programmer via the various precision setters on `Clock` and `Chronograph`.

4. The realtime clock shall be monotonically increasing, and other clocks and chronographs should be monotonically increasing as well. Where the universal clock needs to be resynchronized with the external environment, this should be at the expense of its uniformity rather than its monotonicity.

5. 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.

6. 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 universal 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.

The RTSJ Clock model is designed for maximum utility and predictability on monotonically increasing timing mechanisms. Clocks that do not have this property may display certain inconsistencies in the event of reverse discontinuities. In particular, any the following may occur.

1. When a `OneShotTimer` is set on a nonmonotonic `Clock`, that clock experiences a reverse discontinuity, and that timer has already fired, but the reverse discontinuity would cause its expiry time to occur again, the timer will not fire again.

2. When a `PeriodicTimer` is set on a nonmonotonic `Clock`, that clock experiences a reverse discontinuity, and that timer has already fired for time $T$ but the reverse discontinuity would cause time $T$ to occur again, the handler for time $T$ will not be released again for time $T$. This may mean that the elapsed wall clock time between two firings of the `PeriodicTimer` exceeds the period without the release of an associated miss handler or other detection, as the next firing will happen when the clock reaches $T + P$, where $P$ is the period of the timer.

3. When a `Timer` is set on a nonmonotonic `Clock` and that clock experiences a reverse discontinuity while that timer is scheduled for release at time $T$, but the reverse discontinuity causes time $T$ to be “pushed back” with respect to wall clock time, the `Timer` will not fire until time $T$ is reached on the clock and the elapsed wall clock time to $T$ will be of longer duration than it was when the timer was set.

Forward or reverse discontinuities on a Clock may cause races for Timer releases occurring very close to the time of the discontinuity. Therefore, the default realtime clock should increment as consistently as possible under the design constraints of the system.

### 10.2.2 Clocks and Timables

A `Clock` is the basic mechanism of measuring time and triggering events based on the passage of time. Both a Timer and a RealtimeThread with PeriodicParameters can request a signal from the clock when a given time is reached. That signal should come as close to the actual time requested as possible. A schedulable uses a clock to implement the realtime sleep methods. `HighResolutionTime` also defines `waitForObject` with a timeout with a clock. Each clock instance shall be capable of reporting the achievable resolution of timers based on that clock. Each
implementation shall have a default realtime clock that is used whenever no other clock is specified. An application can also define additional clocks, including a UTC.

10.2.2.1 Timable

A Timer uses a clock to measure time, which informs the timer’s 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 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 an ActiveEvent. This means that it 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 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 causes 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, i.e., is active but disabled 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.

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 RealtimeThread with PeriodicParameters acts analogously to a PeriodicTimer with a single AsyncEventHandler. A Clock is to signal its TimeDispatcher to release the thread from its waitForNextRelease method. When a release happens before the thread reaches its call to waitForNextRelease, the thread simply continues, otherwise it waits for its release.

Both Timer and RealtimeThread have a fire method for this signaling. They both implement the AsyncTimable interface, a child interface of Timable. Timable is used for implementing application-defined clocks.

A Clock can also be used to provide pauses in execution of any Schedulable through a realtime sleep method, hence they are also classified as timables under
the `Timable` interface. A `Schedulable` has a `TimeDispatcher` to manage sleeping.

Since `RealtimeThread` supports both sleep, as a `Schedulable`, and release behavior, as an `AsyncTimable`, a single interface is insufficient. Instead, both `Schedulable` and `AsyncTimable` are children of `Timable`. Note that how the time out on `waitForObject` is handle is unspecified, since it is not visible to the application programer.

### 10.2.2.2 Dispatching

At any given time, a `Timable`, `AsyncTimer` or `Schedulable`, has at most one clock associated with it, on which the measurement of time for blocking is 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 an application-defined clock and Figure 10.2 depicts the same for using realtime sleep in a schedulable.

Figure 10.1: Sequence Diagram for Using a Timer
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, as shown in step three in both diagrams, when the timable is created and it represents the next alarm that the timable should receive: either 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 starts 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. When 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.

When the new alarm is the next alarm to be triggered, the clock arranges to
10 Clocks and Timers

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 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 as many 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.

10.2.3 Modeling 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.

10.2.3.1 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).
1. **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.

2. In both cases, a delay is introduced.

3. An RTSJ Timer, when using the counter model, resets the count when it is restarted after being stopped.

4. 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.

### 10.2.3.2 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).

1. 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.

2. In this case, no delay is introduced.

3. 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.

### 10.2.3.3 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 its 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.

### 10.2.3.4 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:

1. 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
2. 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 $n = 0, 1, 2, \ldots$ 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 8.4. 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.

10.2.3.5 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 $S$:

1. $S$ is the next multiple of $A + nT$, when phasing is `ADJUST_FORWARD`,
2. $S$ is the most recent multiple of $A + nT$, when phasing is `ADJUST_BACKWARD`,
3. $S$ is “now,” when phasing is `ADJUST_TO_START`, and
4. $S$ is undefined and an exception it thrown when phasing is `STRICT_PHASING`. 
The default phasing is `ADJUST_TO_START`.
10.3  `javax.realtime`

10.3.1  Interfaces

10.3.1.1  AsyncTimable

```java
public interface AsyncTimable

    Interfaces
    `javax.realtime.Timable`

    Description
```

1\footnote{Note that the semantics of the fire transition differ among the subclasses of `Timer`.}
A common type for Timer and RealtimeThread to indicate that they can be associated with a Clock and released by time events on that clock.

Available since RTSJ 2.0

10.3.1.1.1 Methods

fire

Signature
    public void
    fire()

Description
    Called by the dispatcher associated with this to indicate that a time event has occurred.

10.3.1.2 Chronograph

public interface Chronograph

Description
    The interface for all devices that support the measurement of time with great accuracy.

Available since RTSJ 2.0

10.3.1.2.1 Methods
getEpochOffset

Signature

    public javax.realtime.RelativeTime
    getEpochOffset()
    throws UnsupportedRealtimeOperationException,
           UninitializedStateException

Description

Determines the difference between the epoch of this clock from the Epoch. For
the UTC, the result is always a RelativeTime value equal to zero. For other
clocks, it is a value representing the difference between zero on that clock and
zero on the UTC measured on the UTC.

Throws

UnsupportedRealtimeOperationException—when the chronograph does not have
the concept of date.

UninitializedStateException—when UTC time is not yet available.

Returns

a newly allocated RelativeTime object in the current execution context with the
UTC as its chronograph and containing the time when this chronograph was
zero.

g getTime

Signature

    public javax.realtime.AbsoluteTime
    getTime()

Description

Determines the current time. This method returns an absolute time value
representing the chronograph’s notion of absolute time. For chronographs that
do not measure calendar time, this absolute time may not represent a wall clock
time.

Returns

a newly allocated instance of AbsoluteTime in the current allocation context,
representing the current time. The returned object has the chronograph from
this.
**getTime(AbsoluteTime)**

*Signature*

```java
class Chronograph {
    public javax.realtime.AbsoluteTime getTime(AbsoluteTime dest) {
        // Implementation
    }
}
```

*Description*

Obtains the current time. The time represented by the given `AbsoluteTime` is changed at some time between the invocation of the method and the return of the method. This method will return an absolute time value that represents this chronographs’s notion of the absolute time. For chronographs that do not measure calendar time, this absolute time may not represent a wall clock time.

*Parameters*

- `dest`—The instance of `AbsoluteTime` object which will be updated in place.

*Returns*

the instance of `AbsoluteTime` passed as parameter, or a new object when `dest` is `null`. The returned object represents the current time and is associated with `this` chronograph.

**getQueryPrecision**

*Signature*

```java
class Chronograph {
    public javax.realtime.RelativeTime getQueryPrecision() {
        // Implementation
    }
}
```

*Description*

Obtains the precision with which time can be read, i.e., the nominal interval between ticks. It is the same as calling `getQueryPrecision(RelativeTime)` with `null` as an argument.

*Returns*

a newly allocated time value holding the read precision.

**getQueryPrecision(RelativeTime)**

*Signature*

```java
class Chronograph {
    public javax.realtime.RelativeTime getQueryPrecision(RelativeTime dest) {
        // Implementation
    }
}
```
Description

Obtains the precision with which time can be read, i.e., the nominal interval between ticks.

Parameters

dest—The time object in which to return the results.

Returns

the read precision in dest, when dest is not null, or in a newly created object otherwise.

10.3.1.3 Timable

public interface Timable

Interfaces

javax.realtime.Releasable

Description

A type for all classes that can use a Clock for timing, either for being woken, as from a sleep or timeout, or for being released at a given time on the clock.

Available since RTSJ 2.0

10.3.1.3.1 Methods

getDispatcher

Signature

public javax.realtime.TimeDispatcher getDispatcher()

Description

Gets the dispatcher associated with this Timable.
10.3.2 Classes

10.3.2.1 Clock

public abstract class Clock

Inheritance
java.lang.Object
  Clock

Interfaces
javax.realtime.Chronograph

Description
A clock marks the passing of time. It has a concept of now that can be queried through Clock.getTime(), and it can have events queued on it which will be fired when their appointed time is reached.

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.

10.3.2.1.1 Constructors

Clock

Signature
public
Clock()

Description
Constructor for the abstract class.
10 Clocks and Timers

10.3.2.1.2 Methods

**getRealtimeClock**

*Signature*

```java
public static javax.realtime.Clock
getRealtimeClock()
```

*Description*

There is always at least one clock object available: the system realtime clock. This clock is monotonically increasing and does not need to start at the Epoch. On a POSIX system, it is equivalent to `CLOCK_MONOTONIC`. It is the default `Clock`.

*Returns*

the singleton instance of the default `Clock`

**setRealtimeClock(Clock)**

*Signature*

```java
public static void
setRealtimeClock(Clock clock)
```

*Description*

Sets the system default realtime clock.

*Parameters*

clock—To be used for the realtime clock. When `null`, the default realtime clock is set to the original system default.

**getUniversalClock**

*Signature*

```java
public static javax.realtime.Clock
getUniversalClock()
```

*Description*

402 RTSJ 2.0 (Draft 66)
A means of obtaining the Universal Time, which has no summer or winter time. Local time can be obtained by adding the appropriate time zone offset. Such a time source is not available on all systems and may take a while to set up on some systems which support it. It is not guaranteed to be monotonic.

*Throws*

- `UnsupportedRealtimeOperationException`—when the system does not support UTC.
- `UninitializedStateException`—when UTC time is not yet available.

*Returns*

a Clock that tracks UTC, such as the POSIX `CLOCK_REALTIME`, when the timezone is set to UTC.

**Available since** RTSJ 2.0

### `setUniversalClock(Clock)`

**Signature**

```java
public static void setUniversalClock(Clock clock)
```

**Description**

Sets the system default universal clock.

**Parameters**

clock—To be used for the universal clock. When `null`, the default universal clock is set to the original system default.

**Available since** RTSJ 2.0

### `getEpochOffset`

**Signature**

```java
public final javax.realtime.RelativeTime getEpochOffset()
```

**Throws**

- `UnsupportedRealtimeOperationException`
- `UninitializedStateException`

**Description**

RTSJ 2.0 (Draft 66) 403
Determines the difference between the epoch of this clock from the Epoch. For the UTC, the result is always a RelativeTime value equal to zero. For other clocks, it is a value representing the difference between zero on that clock and zero on the UTC measured on the UTC.

**Throws**

UnsupportedRealtimeOperationException—when the chronograph does not have the concept of date.

UninitializedStateException—when UTC time is not yet available.

**Available since** RTSJ 1.0.1

### getTime

**Signature**

```java
public final javax.realtime.AbsoluteTime getTime()
```

**Description**

Determines the current time. This method returns an absolute time value representing the chronograph’s notion of absolute time. For chronographs that do not measure calendar time, this absolute time may not represent a wall clock time.

**Returns**

a newly allocated instance of AbsoluteTime in the current allocation context, representing the current time. The returned object has the chronograph from this.

### getTime(AbsoluteTime)

**Signature**

```java
public abstract javax.realtime.AbsoluteTime getTime(AbsoluteTime dest)
```

**Description**

Obtains the current time. The time represented by the given AbsoluteTime is changed at some time between the invocation of the method and the return of the method. This method will return an absolute time value that represents this chronographs’s notion of the absolute time. For chronographs that do not measure calendar time, this absolute time may not represent a wall clock time.
**Parameters**

`dest`—The instance of `AbsoluteTime` object which will be updated in place.

**Returns**

the instance of `AbsoluteTime` passed as parameter, or a new object when `dest` is `null`. The returned object represents the current time and is associated with `this` chronograph.

**Available since** RTSJ 1.0.1 The return value is updated from `void` to `AbsoluteTime`.

**Available since** RTSJ 2.0 When `dest` is `null`, a new object is allocated, when not chronograph is overwritten with `this`.

**getQueryPrecision**

**Signature**

```java
public abstract javax.realtime.RelativeTimegetQueryPrecision()
```

**Description**

Obtains the precision with which time can be read, i.e., the nominal interval between ticks. It is the same as calling `getQueryPrecision(RelativeTime)` with `null` as an argument.

**Returns**

a newly allocated time value holding the read precision.

**Available since** RTSJ 2.0

**getQueryPrecision(RelativeTime)**

**Signature**

```java
public abstract javax.realtime.RelativeTimegetQueryPrecision(RelativeTime dest)
```

**Description**

Obtains the precision with which time can be read, i.e., the nominal interval between ticks.

**Parameters**

`dest`—The time object in which to return the results.
Returns
the read precision in dest, when dest is not null, or in a newly created object otherwise.

Available since RTSJ 2.0

getDrivePrecision

Signature
public abstract javax.realtime.RelativeTime getDrivePrecision()

Description
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) with null as its argument.

Returns
a value representing the drive precision.

Available since RTSJ 2.0

getDrivePrecision(RelativeTime)

Signature
public abstract javax.realtime.RelativeTime getDrivePrecision(RelativeTime dest)

Description
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).

Parameters
dest—To return the relative time value in dest. When dest is null, it allocates a new RelativeTime instance to hold the returned value.

Returns
dest set to values representing the drive precision.

Available since RTSJ 2.0

406 RTSJ 2.0 (Draft 66)
triggerAlarm

Signature

protected final void
triggerAlarm()

Description

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, which in turn will cause a AsyncTimable.fire() or Schedulable.awaken() to be called depending on the kind of Timable associated with the alarm.

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

setAlarm(long, int)

Signature

protected abstract void
setAlarm(long milliseconds,
int nanoseconds)

Description

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.

Parameters

milliseconds—The millisecond part of an absolute time for the next alarm.
nanoseconds—The nanosecond part of an absolute time for the next alarm.

**Available since** RTSJ 2.0

clearAlarm

Signature

RTSJ 2.0 (Draft 66)
10 Clocks and Timers

protected abstract void
clearAlarm()

Description

Implemented by subclasses to cancel the current outstanding alarm.

Available since RTSJ 2.0

10.3.2.2 OneShotTimer

public class OneShotTimer

Inheritance

java.lang.Object
   AsyncBaseEvent
      AsyncEvent
         Timer
            OneShotTimer

Description

A timed AsyncEvent that is driven by a Clock. 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 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.

10.3.2.2.1 Constructors

OneShotTimer(HighResolutionTime, TimeDispatcher)

Signature

408 RTSJ 2.0 (Draft 66)
OneShotTimer

`public OneShotTimer(HighResolutionTime<?> time, TimeDispatcher dispatcher) throws IllegalRealtimeArgumentException, UnsupportedRealtimeOperationException, IllegalAssignmentError`

**Description**

Creates an instance of `OneShotTimer`, based on the given `clock`, that will execute its `fire` method according to the given time. The `Clock` association of the parameter `time` is ignored.

**Available since** RTSJ 2.0

**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()`.
- **dispatcher**—The `dispatcher` used to interface between this `timer` and its associated `clock`. When `null`, the system default `dispatcher` is used.

**Throws**

- `IllegalRealtimeArgumentException`—when `time` is a `RelativeTime` instance less than zero.
- `UnsupportedRealtimeOperationException`—when the `Chronograph` associated with `time` is not a `Clock`.
- `IllegalAssignmentError`—when this `OneShotTimer` cannot hold references to `time`, `handler`, or `clock`.

**OneShotTimer(HighResolutionTime, AsyncEventHandler)**

**Signature**

`public OneShotTimer(HighResolutionTime<?> time, AsyncEventHandler handler)`

**Description**

The equivalent of calling `OneShotTimer(HighResolutionTime, TimeDispatcher)` with arguments `time`, `null` followed by a call to `setHandler(handler)`.
Parameters

- **time**—Time to release its handlers.
- **handler**—Handler to be released.

### 10.3.2.2.2 Methods

#### fire

**Signature**

```java
public void fire()
```

**Description**

This should not be called for application code, except for emulation. The fire method is reserved for the use of the system. When this is enabled, it releases all handlers and then calls `Timer.stop()`. When disabled, but active, it only calls `Timer.stop()`. Otherwise it does nothing.

**Available since** RTSJ 2.0 moved here from Timer, since `OneShotTimer` and `PeriodicTimer` have slightly different semantics.

### 10.3.2.3 PeriodicTimer

```java
public class PeriodicTimer
```

**Inheritance**

- `java.lang.Object`
- `AsyncBaseEvent`
- `AsyncEvent`
- `Timer`
- `PeriodicTimer`

**Description**

An `AsyncEvent` 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.

RTSJ 2.0 (Draft 66)
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.

Semantics details are described in the Timer pseudo-code 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.

10.3.2.3.1 Constructors

PeriodicTimer(HighResolutionTime, RelativeTime, TimeDispatcher)

Signature
public
PeriodicTimer(HighResolutionTime<?,?> start,
RelativeTime interval,
TimeDispatcher dispatcher)
throws IllegalRealtimeArgumentException,
IllegalAssignmentError,
UnsupportedRealtimeOperationException

Description
Creates a timer that executes its fire method periodically.

Available since RTSJ 2.0

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 to 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.

dispatcher—The dispatcher to use for triggering this event.

Throws

IllegalRealtimeArgumentException—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.

UnsupportedRealtimeOperationException—when the Chronograph associated with time is not a Clock.

PeriodicTimer(HighResolutionTime, RelativeTime, AsyncEventHandler)

Signature

public PeriodicTimer(HighResolutionTime<?> start, RelativeTime interval, AsyncEventHandler handler)

throws IllegalRealtimeArgumentException, IllegalAssignmentError

Description

Creates a timer that executes its fire method periodically. Equivalent to PeriodicTimer(start, interval, handler, null).

10.3.2.3.2 Methods

addHandler(AsyncBaseEventHandler)

Signature

public void addHandler(AsyncBaseEventHandler handler)
throws IllegalRealtimeArgumentException, IllegalAssignmentError

**Description**

Add a handler to the set of handlers associated with this event. It overrides the method in AsyncBaseEvent to allow the use of handlers with PeriodicParameters, but these parameters must match the period of this timer, otherwise IllegalRealtimeArgumentException is thrown.

**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**

IllegalRealtimeArgumentException—when `handler` is null or the handler has PeriodicParameters with a period that does not match the period of this.

IllegalAssignmentError—when this AsyncEvent cannot hold a reference to handler.

Available since RTSJ 2.0

**setHandler(AsyncBaseEventHandler)**

**Signature**

```java
public void setHandler(AsyncBaseEventHandler handler)
throws IllegalRealtimeArgumentException, IllegalAssignmentError
```

**Description**

Associates a new handler with this event and removes all existing handlers. It overrides the method in AsyncBaseEvent to allow the use of handlers with PeriodicParameters, but these parameters must match the period of this timer, otherwise IllegalRealtimeArgumentException is thrown.

**Parameters**

`handler`—The instance of AsyncBaseEventHandler to be associated with this. When `handler` is null, no handler will be associated with this, i.e., behave effectivelthe effect is as when setHandler(null) invokes removeHandler(AsyncBaseEventHandler) for each associated handler.
**Throws**

1. **IllegalRealtimeArgumentException**—when handler has **PeriodicParameters** with a period that does not match the period of this.
2. **IllegalAssignmentError**—when this AsyncEvent cannot hold a reference to handler.

**Available since RTSJ 2.0**

### start(PhasingPolicy)

**Signature**

```java
public void start(PhasingPolicy phasingPolicy)
throws LateStartException,
        IllegalRealtimeArgumentException
```

**Description**

Starts the timer with the specified **PhasingPolicy**.

**Parameters**

- **phasingPolicy**—Determines what happens when the start is too late.

**Throws**

1. **LateStartException**—when this method is called after its absolute start time and the phasingPolicy is **PhasingPolicy.STRICT_PHASING**.
2. **IllegalRealtimeArgumentException**—when the start time of this timer is not an absolute time, or phasingPolicy is null.

**Available since RTSJ 2.0**

### start(boolean, PhasingPolicy)

**Signature**

```java
public void start(boolean disabled,
                  PhasingPolicy phasingPolicy)
throws LateStartException,
        IllegalRealtimeArgumentException
```

**Description**

414 RTSJ 2.0 (Draft 66)
Starts the timer with the specified `PhasingPolicy` and the specified disabled state.

**Parameters**
- `disabled`—It determines the mode of start: `true` for enabled and `false` for disabled for consistency with `Timer.start(boolean)`.
- `phasingPolicy`—It 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`.
- `IllegalRealtimeArgumentException`—when the start time of this timer is not an absolute time, or `phasingPolicy` is `null`.

**Available since** RTSJ 2.0

### getClock

**Signature**
```java
class PeriodicTimer {
    public javax.realtime.Clock
clock() throws IllegalRealtimeStateException
}
```

**Description**
Each instance can only be associated with a single clock, which this method can obtain.

**Throws**
- `IllegalRealtimeStateException`—when `this` has been destroyed.

**Returns**
the instance of `Clock` that is associated with `this`.

**Available since** RTSJ 1.0.1

### createReleaseParameters

**Signature**
```java
class PeriodicTimer {
    public javax.realtime.ReleaseParameters
createReleaseParameters() {
}
```

RTSJ 2.0 (Draft 66) 415
10 Clocks and Timers

Description

Creates a release parameters object with new objects containing copies of the values corresponding to this timer. When the PeriodicTimer interval is greater than 0, creates a PeriodicParameters object with a start time and period that correspond to the next firing (or skipping) time, and interval, of this timer. When the interval is 0, creates an AperiodicParameters object, since in this case the timer behaves like a OneShotTimer.

When this timer is active, then the start time is the next firing (or skipping) time returned as an AbsoluteTime. Otherwise, the start time is the initial firing (or skipping) time, as set by the last call to Timer.reschedule, or when there was no such call, by the constructor of this timer.

Throws

IllegalRealtimeStateException—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, returns a new instance of PeriodicParameters. When the interval is zero returns a new instance of AperiodicParameters.

getFireTime

Signature

public javax.realtime.AbsoluteTime
getFireTime()
throws ArithmeticException,
IllegalRealtimeStateException

Description

Gets 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 or firing. When the PeriodicTimer is not-active it throws IllegalRealtimeStateException.

Throws

ArithmeticException—when the result does not fit in the normalized format.

IllegalRealtimeStateException—when this Timer has been destroyed, or when it is not-active.

Returns
the absolute time at which this is next expected to fire or to skip, in a newly allocated \texttt{AbsoluteTime} object. When the timer has been created or re-scheduled (see \texttt{Timer.reschedule(HighResolutionTime)}) using an instance of \texttt{RelativeTime} for its time parameter, then it will return the sum of the current time and the \texttt{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.

\textbf{getFireTime(AbsoluteTime)}

\textit{Signature}

\hspace*{1em} \texttt{public javax.realtime.AbsoluteTime getFireTime(AbsoluteTime dest)}

\textit{Description}

Gets the time at which this \texttt{PeriodicTimer} is next expected to fire or to skip. When the \texttt{PeriodicTimer} is disabled, the returned time is that of the skipping. When the \texttt{PeriodicTimer} is not-active it throws \texttt{IllegalRealtimeStateException}.

\textit{Parameters}

\hspace*{1em} \texttt{dest}—The instance of \texttt{AbsoluteTime} which will be updated in place and returned.

\hspace*{2em} The clock association of the \texttt{dest} parameter is ignored. When \texttt{dest} is null, a new object is allocated for the result.

\textit{Throws}

\hspace*{1em} \texttt{ArithmeticException}—when the result does not fit in the normalized format.

\hspace*{1em} \texttt{IllegalRealtimeStateException}—when this \texttt{Timer} has been destroyed, or when it is not-active.

\textit{Returns}

the instance of \texttt{AbsoluteTime} passed as parameter, with time values representing the absolute time at which \texttt{this} is expected to fire or to skip. When the \texttt{dest} parameter is null, the result is returned in a newly allocated object. When the timer has been created or re-scheduled (see \texttt{Timer.reschedule(HighResolutionTime)}) using an instance of \texttt{RelativeTime} for its time parameter then it will return the sum of the current time and the \texttt{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.
Available since RTSJ 1.0.1

getInterval

Signature

public javax.realtime.RelativeTime
getInterval()

Description

Gets the interval of this Timer.

Throws

IllegalRealtimeStateException—when this Timer has been destroyed.

Returns

the RelativeTime instance assigned as this periodic timer’s interval by the constructor or setInterval(RelativeTime).

setInterval(RelativeTime)

Signature

public javax.realtime.PeriodicTimer
setInterval(RelativeTime interval)

Description

Resets the interval value of this.

Parameters

interval—A RelativeTime 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 affects future fire (or skip) times after the next.

Throws

IllegalRealtimeArgumentException—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.

IllegalRealtimeStateException—when this Timer has been destroyed.

Returns
this

fire

Signature
public void
fire()

Description
This should not be called for application code, except for emulation. The fire method is reserved for the use of the system. When this is enabled, it releases all handlers and then reschedules itself for the next period without changing state. When disabled, but active, it simply reschedules itself. Otherwise it does nothing.

Available since RTSJ 2.0 moved here from Timer, since OneShotTimer and PeriodicTimer have slightly different semantics.

10.3.2.4 TimeDispatcher

public class TimeDispatcher

Inheritance
java.lang.Object

ActiveEventDispatcher<TimeDispatcher, Timable>

TimeDispatcher

Description
A dispatcher for time events: Timer and RealtimeThread.sleep.

Available since RTSJ 2.0

10.3.2.4.1 Constructors
TimeDispatcher(SchedulingParameters, RealtimeThreadGroup)

Signature

```java
public TimeDispatcher(SchedulingParameters schedule,
                      RealtimeThreadGroup group)
    throws IllegalRealtimeStateException
```

Description

Creates a new dispatcher, whose dispatching thread runs with the given scheduling parameters.

Parameters

schedule—It gives the parameters for scheduling this dispatcher

Throws

IllegalRealtimeStateException—when the intersection of affinity in schedule and the affinity of group does not correspond to a valid affinity.

TimeDispatcher(SchedulingParameters)

Signature

```java
public TimeDispatcher(SchedulingParameters schedule)
    throws IllegalRealtimeStateException
```

Description

Creates a new dispatcher, whose dispatching thread runs with the given scheduling parameters.

Parameters

schedule—It gives the parameters for scheduling this dispatcher

Throws

IllegalRealtimeStateException—when the intersection of affinity in schedule and the affinity of the current thread group does not correspond to a valid affinity.
10.3.2.4.2 Methods

setDefaultDispatcher(TimeDispatcher)

Signature
public static void
setDefaultDispatcher(TimeDispatcher dispatcher)

Description
Sets the system default time dispatcher.

Parameters
dispatcher—To be used when no dispatcher is provided. When null, the default
time dispatcher is set to the original system default.

register(Timable)

Signature
public void
register(Timable target)
throws RegistrationException,
IllegalRealtimeStateException,
IllegalRealtimeArgumentException

Description
Registers a Timable with this dispatcher.

Parameters
target—To be registered

Throws
RegistrationException—when target is already registered,
IllegalRealtimeStateException—when this object has been destroyed.
IllegalRealtimeArgumentException—when target is not stopped.

deregister(Timable)

Signature
public void
deregister(Timable target)
throws DeregistrationException,
    IllegalRealtimeStateException,
    IllegalRealtimeArgumentException

Description
Deregisters a Timable from this dispatcher.

Parameters
target—to be deregistered

Throws
DeregistrationException—when target is not already registered.
IllegalRealtimeStateException—when this object has been destroyed.
IllegalRealtimeArgumentException—when target is not stopped.

destroy

Signature
public void
destroy()
throws IllegalRealtimeStateException

Description
Releases all resources thereby making the dispatcher unusable.

Throws
IllegalRealtimeStateException—when called on a dispatcher that has one or
more registered Timable objects.

10.3.2.5 Timer

public abstract class Timer

Inheritance
java.lang.Object
    AsyncBaseEvent
    AsyncEvent
    Timer
Interfaces
java.realtime.AsyncTimable
java.realtime.ActiveEvent

Description
A timer is a timed event that measures time according to a given Clock. This class defines basic functionality available to all timers. Applications will generally use either PeriodicTimer to create an event that is fired repeatedly at regular intervals, or OneShotTimer for an event that just fires once at a specific time. A timer is always associated with at least one Clock, 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.

10.3.2.5.1 Constructors

Timer(HighResolutionTime, TimeDispatcher)

Signature
protected
Timer(HighResolutionTime<?> time,
     TimeDispatcher dispatcher)
throws IllegalRealtimeArgumentException,
     UnsupportedRealtimeOperationException,
     IllegalAssignmentError

Description
Creates a timer that fires according to the given time based on the Clock associated with time and is dispatched by the specified dispatcher.

Available since RTSJ 2.0

Parameters
time—The parameter 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().
dispatcher—The object used to interface between this timer and its associated clock. When null, the system default dispatcher is used.
10 Clocks and Timers

Timer

Throws

IllegalRealtimeArgumentException—when time is a negative RelativeTime value.

UnsupportedRealtimeOperationException—when time has a Chronograph is not a clock.

IllegalAssignmentError—when this Timer cannot hold references to handler and clock.

Timer(HighResolutionTime)

Signature

protected
Timer(HighResolutionTime<?> time)
throws IllegalRealtimeArgumentException, UnsupportedRealtimeOperationException, IllegalAssignmentError

Description

Creates a timer that fires according to the given time based on the Clock associated with time and is dispatched by the system default dispatcher.

This is equivalent to Timer(time, null).

Available since RTSJ 2.0

Parameters

time—The parameter 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().

Throws

IllegalRealtimeArgumentException—when time is a negative RelativeTime value.

UnsupportedRealtimeOperationException—when time has a Chronograph is not a clock.

IllegalAssignmentError—when this Timer cannot hold references to handler and clock.

Timer(HighResolutionTime, Clock, AsyncEventHandler)
Signature

protected Timer(HighResolutionTime<?> time, Clock clock, AsyncEventHandler handler)
throws IllegalRealtimeArgumentException, UnsupportedRealtimeOperationException, IllegalAssignmentError

Description

Creates a timer that fires according to the given time, which must be based on the supplied Clock clock (if any), and is handled by the specified AsyncEventHandler handler. The system default dispatcher will be used.

This constructor is slated for deprecation in a future release, and a constructor that does not receive a Clock argument should be used in preference.

Parameters

time—The parameter 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().
clock—The clock on which to base this timer. When null, the clock associated with time is used.
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.

Throws

IllegalRealtimeArgumentException—when time is a negative RelativeTime value or the supplied clock is not the Clock associated with time.
UnsupportedRealtimeOperationException—when time has a Chronograph that is not an instance of Clock.
IllegalAssignmentError—when this Timer cannot hold references to handler and clock.

10.3.2.5.2 Methods

getchClock
10 Clocks and Timers

Signature

```java
public javax.realtime.Clock
clock()
throws IllegalRealtimeStateException
```

Description

Obtains the instance of `Clock` on which this `Timer` is based.

Throws

`IllegalRealtimeStateException`—when this `Timer` has been destroyed.

Returns

the instance of `Clock` associated with this `Timer`.

getStart

Signature

```java
public javax.realtime.HighResolutionTime<?>
start()
```

Description

Gets 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 affect the start time of this `Timer`.

Returns

a reference to the time (or start) parameter used when constructing this `Timer`, ensuring the content has the original values.

Available since RTSJ 2.0

getEffectiveStartTime

Signature

```java
public javax.realtime.AbsoluteTime
effectiveStart()
throws IllegalRealtimeStateException,
ArithmeticException
```

Description

426 RTSJ 2.0 (Draft 66)
Returns a newly-created time representing the time when the timer actually started, or when the timer has been rescheduled, the effective start time after the reschedule.

Throws

- `IllegalRealtimeStateException`—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.

Available since RTSJ 2.0

generateEffectiveStartTime(AbsoluteTime)

Signature

```java
public javax.realtime.AbsoluteTime
generateEffectiveStartTime(AbsoluteTime dest)
throws IllegalRealtimeStateException,
ArithmeticException
```

Description

Updates `dest` to represent the time when the timer actually started, or when the timer has been rescheduled, the effective start time after the reschedule. When `dest` is null, behaves as if `generateEffectiveStartTime()` had been called.

Parameters

dest—An object used to store the time `this` actually started.

Throws

- `IllegalRealtimeStateException`—when the timer is not active or has been destroyed.
- `ArithmeticException`—when the result does not fit in the normalized format.

Returns

the time when the timer actually started, or when it has been rescheduled, the effective start time after the reschedule.

Available since RTSJ 2.0
getFireTime

Signature

```java
public javax.realtime.AbsoluteTime
getFireTime()
throws IllegalRealtimeStateException,
        ArithmeticException
```

Description

Gets the time at which this Timer is expected to fire. When the Timer is disabled, the returned time is that of the skipping or the firing. When the Timer is not-active, it throws IllegalRealtimeStateException.

Throws

- ArithmeticException—when the result does not fit in the normalized format.
- IllegalRealtimeStateException—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 object. When the timer has been created or re-scheduled (see Timer.reschedule) 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)

Signature

```java
public javax.realtime.AbsoluteTime
getFireTime(AbsoluteTime dest)
throws IllegalRealtimeStateException,
        ArithmeticException
```

Description

Gets the time at which this Timer is expected to fire. When the Timer is disabled, the returned time is that of the skipping or the firing. When the Timer is not-active it throws IllegalRealtimeStateException.

Parameters

428 RTSJ 2.0 (Draft 66)
**dest**—The instance of `AbsoluteTime` which will be updated in place and returned. The clock association of the `dest` parameter is ignored. When `dest` is `null`, a new object is allocated for the result.

*Throws*
- `ArithmeticException`—when the result does not fit in the normalized format.
- `IllegalRealtimeStateException`—when this `Timer` has been `destroyed`, or when it is `not-active`.

*Returns*
the instance of `AbsoluteTime` 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 rescheduled (see `Timer.reschedule`) 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.

**Available since** RTSJ 1.0.1

### getDispatcher

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

*Description*
Gets the dispatcher associated with this Timable.

*Returns*
null

**Available since** RTSJ 2.0

### isActive

*Signature*
public `boolean` isActive()

**Available since** RTSJ 2.0
Description
Determines the activation state of this happening, i.e., it has been started.

Returns
true when active, false otherwise.

Available since RTSJ 2.0

isRunning

Signature
public boolean
isRunning()
throws IllegalRealtimeStateException

Description
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 answers the question 'Is firing expected?'.

Throws
IllegalRealtimeStateException—when this Timer has been destroyed.

Returns
true when the timer is active and enabled; otherwise 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)

Signature
public boolean
handledBy(AsyncEventHandler handler)
throws IllegalRealtimeStateException

Description
Replaced by AsyncBaseEvent.handledBy(AsyncBaseEventHandler)

Parameters
handler—An event handler to be added to the Timer

Throws

430 RTSJ 2.0 (Draft 66)
IllegalRealtimeStateException—when this Timer has been destroyed.

Returns
true when handler is associated with this, otherwise false.

Available since RTSJ 1.0.1

createReleaseParameters

Signature
public javax.realtime.ReleaseParameters
createReleaseParameters()
throws IllegalRealtimeStateException

Description
Creates a ReleaseParameters object appropriate to the timing characteristics of this event. The default is the most pessimistic: AperiodicParameters. 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.

Throws
IllegalRealtimeStateException—when this Timer has been destroyed.

Returns
a newly created ReleaseParameters object.

enable

Signature
public void
enable()
throws IllegalRealtimeStateException

Description
Re-enables this timer after it has been disabled. (See Timer.disable().) When the Timer is already enabled, this method does nothing. When the Timer is not-active, this method does nothing.

Throws
IllegalRealtimeStateException—when this Timer has been destroyed.
**disable**

*Signature*

```java
class Timer {
    public void disable() throws IllegalRealtimeStateException {
        // Implementation
    }
}
```

*Description*

Disables 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 or 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.

*Throws*

- `IllegalRealtimeStateException`—when this `Timer` has been destroyed.

**start**

*Signature*

```java
class Timer {
    public void start() throws IllegalRealtimeStateException {
        // Implementation
    }
}
```

*Description*

Starts this timer. A timer starts measuring time from when it is started; this method makes the timer active and enabled.

*Throws*

- `IllegalRealtimeStateException`—when this `Timer` has been destroyed, or when this timer is already active.
start(boolean)

Signature
public void
start(boolean disabled)
throws IllegalRealtimeStateException

Description
Starts this timer. A timer starts measuring time from when it is started. When
disabled is true starts the timer making it active in a disabled state. When
disabled is false this method behaves like the start() method.

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
IllegalRealtimeStateException—when this Timer has been destroyed, or when
this timer is active.

Available since RTSJ 1.0.1

stop

Signature
public boolean
stop()
throws IllegalRealtimeStateException

Description
Stops a timer when it is active and changes its state to inactive and disabled.

Throws
IllegalRealtimeStateException—when this Timer has been destroyed.

Returns
ture when this was enabled and false otherwise.

reschedule(HighResolutionTime)

Signature
### Description

Changes 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 that 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.

### 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.

### Throws

- `IllegalRealtimeArgumentException`—when `time` is a negative `RelativeTime` value.
- `IllegalRealtimeStateException`—when this `Timer` has been `destroyed`.

### 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 strange, but it enables the developer to accommodate systems with different resources. For instance, most systems have an on board clock, which is provided as the default clock through the operation system. This clock is the natural clock to use for the RTSJ default clock, but this clock may not be stable or accurate enough for a given application. The clock API can be used to provide a second realtime clock that is based on an external clock source which can provide the needed accuracy and stability. For example, this
could be taken from an external board with a hardware oscillator, a timing circuit that can generate an interrupt, and a small battery. A more exotic example would be to associate a clock with an object that rotates, where one degree is a second, a minute, or an hour depending on the rotation speed and accuracy needed, so long as the clock can trigger something at some fraction of a turn. Without a triggering mechanism, it could still be a chronograph.

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 precision for different purposes. For example, a clock for interacting with humans need much less precision than for controlling the opening 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: getDrivePrecision() andgetQueryPrecision inherited from Chronograph. Clocks should behave as if their tick (setAlarm()) precision is the same as returned by getResolution().
Chapter 11

Alternative Memory Areas

Conventional Java uses a single heap for storing all objects. The thread stacks hold only primitive objects and references to objects. This is fine for desktop and server systems, where there are no realtime, locality, or isolation requirements. Even for most realtime systems, a single heap is usually also sufficient when used in conjunction with a deterministic garbage collector. For all other situations, this specification defines classes directly related to memory and memory management. These classes provide a more generalized means of memory management than is 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 allocation area, 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 manages the allocation area used when the thread or 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. Unreachable 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; objects are collected in scoped memory when there are no schedulables currently active in that area and it is not pinned. 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 the scope is not pinned and there is no schedulable active in its associated allocation area, 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 memory assignment rules, an exception is thrown.

Physical Memory

In embedded systems it is often the case that multiple directly addressable memory types are available to the application. For example, SRAM, DRAM, and Flash memory may all fall within the processor address space. Moreover, as the JVM implementer may require the VM to be portable between systems within the same processor family, the VM itself may not have detailed knowledge of the underlying memory architecture. 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. These physical memory regions can be allocated as either immortal or scoped memory areas.

Pinnable Memory

RTSJ 2.0 adds a new type of scoped memory called pinnable memory. This memory area is designed to support the producer consumer pattern. Here one set of tasks is responsible for producing some data to be consumed by another set of tasks. Usually, the second set cannot enter the area until the first is finished. However, with normal scopes, this would result in the data being lost. A developer must have a special thread, referred to as a wedge thread, to keep the scope alive. With pinnable memory, a wedge thread is no longer necessary. A pinnable scope can simply be pinned. The area is not collected until no threads are in the scope and the scope is unpinned.

Stacked Memory

RTSJ 2.0 also adds a new type of scoped memory called stacked memory. Stacked memory enables systems to maintain predictable memory performance over a long period of time while still releasing memory at runtime. The older scoped memory interfaces left sufficient ambiguity in the specification that the user may not have been able to sufficiently characterize internal and external fragmentation upon creating or
destroying scoped memory areas. The \texttt{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 guarantees are provided by constraining the order in which an application may enter \texttt{StackedMemory} areas, as well as the manner in which they may be arranged on the scope stack. These constraints are enforced by the implementation.

\section*{Memory Groups}

Analogous to \texttt{ProcessingGroups}, memory groups provide a means of partitioning memory between groups. Each group may have its use of backing store limited. Currently, only backing store is considered, but future version of the specification may do more.

\section*{Summary}

In summary, the classes and interfaces defined in this chapter enable
1. the definition of regions of memory outside of the conventional Java heap;
2. the definition of regions of scoped memory, that is, memory regions with a limited lifetime;
3. the definition of regions of memory containing objects whose lifetime matches that of the application;
4. the definition of regions of memory mapped to specific physical addresses with specific virtual memory characteristics;
5. the specification of maximum memory area consumption and maximum allocation rates for individual schedulables;
6. the programmer to query information characterizing the behavior of the garbage collection algorithm, and to some limited ability, alter the behavior of that algorithm.

\section*{11.1 Definitions}

\textbf{Allocation Context} — An abstraction representing memory from which a new object can be allocated. In conventional Java, this is the Java heap. The \texttt{MemoryArea} class is the base class representing all allocation contexts in the RTSJ, of which the heap (represented by \texttt{HeapMemory}) is just one type.

\textbf{Current Allocation Context} — The memory area which will be used when object allocation is requested in the currently active thread of control.

\textbf{Allocation Area} — The area of memory that is managed by a \texttt{MemoryArea} from which objects are allocated. The allocation area for an extraheap memory area
11 Alternative Memory Areas

is logically and physically separate from the Java heap.

**Backing Store** — A range of memory addresses from which the allocation area of a MemoryArea is drawn. There are several backing stores: the global backing store for general scoped memory, the host backing store used for stacked memory, which is usually taken from the global backing store, the backing store used for physical memory areas, and the backing stores for the default heap and immortal memory areas.

**Explicit Initial Memory Area** — A memory area given to a constructor of a Schedulable type, when it is created.

**Execution Context** — A memory area upon which execution is dependent. This includes areas in which a Schedulable or ActiveEvent is allocated. In order to prevent references from becoming invalid, the memory associated with an execution context may not be reclaimed. The following conditions cause a memory area to be an execution context:

1. it contains a Thread instance that has been started but have not terminated (including the RealtimeThread instances contained by ActiveEventDispatcher instances),
2. it contains an ActiveEvent instance that is active,
3. it contains a firable asynchronous event handler,
4. it is on the scope stack inherited by one of the schedulable or event types listed above from the schedulable that created it, or
5. it is on the scope stack of an active schedulable beyond its inherited stack.

**Default Initial Memory Area** — The initial memory area for a schedulable is default when it is the memory area in which the schedulable was created.

**Inner Memory Area** — A memory area on a scope stack that is closer to the current allocation context of the stack’s Schedulable than another memory area. It can also refers to a scoped memory with a greater nesting count in a scoped memory tree.

**Memory Assignment Rules** — The rules for when a reference to an object may be saved in another object. In general, an object created in a memory area may only be stored in the current memory area or a more deeply nested memory area (scoped memory). For these rules, instances of `HeapMemory` and `ImmortalMemory` are equivalent.

**Outer Memory Area** — A memory area on a scope stack that is further from the current allocation context of the stack’s Schedulable than another memory area. It can also refers to a scoped memory with a greater nesting count in a scoped memory tree.

---

1The backing store for the default heap and immortal memory areas are identical to their allocation contexts.

2Defined in Section 8.1
Portal — A location for storing a reference to an object allocated in an instance of `ScopedMemory` settable on that instance. A portal can be used to pass information between instances of `Schedulable` executing in a given area.

Perennial Memory — all memory areas whose contents can be unexceptionally referenced. In other words, any memory area can store a reference to an object stored in one of these areas. This includes all concrete memory areas in the core package. Only memory areas of this type can be a root for a scope stack.

Primordial Scope — An imaginary scope used as the parent for an instance of `ScopedMemory` which is in use and has been entered directly from a perennial memory. It is used distinguish between instances of `ScopedMemory` area that has no parent because it is not active on any scope stack and one that has no parent because it is the first `ScopedMemory` area on a scope stack. Hence, there is no memory area that corresponds to the primordial scope.

Root StackedMemory — A `StackedMemory` that has a parent which is not of type `StackedMemory`.

Scope Stack — A sequence of the memory areas an instance of `Schedulable` has entered, in order of entry, where the first entered is the bottom of the stack and the last entered is the top. Not all memory areas on a scope stack are scoped memories.

Scope Tree — A set of scoped memories with the same root scoped memory. The transitive closure of the parents of any scope in the tree contains that root scoped memory.

Realtime Task — Any task except instances of `java.lang.Thread`. Instances of `Realtime`, `AsyncBaseEventHandler`, `InterruptServiceRoutine`, and `Dispatcher` are all realtime tasks.

### 11.2 Semantics

The classes `MemoryArea`, `HeapMemory`, and `ImmortalMemory` are part of the base module and the semantics below that apply to those modules must be fulfilled by all RTSJ implementations. The rest of the features described here belong to the Alternative Memory Areas Module introduced in Section 3.2.2.2 and are only required for implementations that include that module. The following lists define the general semantics of the classes of this section. Semantics of particular classes, constructors, methods, and fields are detailed further on, in the sections describing those classes, constructors, methods, and fields.
11.2.1 Allocation Execution Time

The following two requirements apply to allocation in any memory area, including the heap.

1. All nondeprecated MemoryArea classes are required to have allocation times linear in the size of the object being allocated. Ignoring performance variations due to hardware caches or similar optimizations and ignoring the execution time of any static initializers, the linear time attribute requires the execution time of new to 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. The execution time of object constructors and time spent in class loading and static initialization are not governed by the 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.

11.2.2 Allocation Context

The following requirements apply to the allocation context represented by a memory area.

1. 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.

2. 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.

3. 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 AsyncBaseEventHandler, 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.

4. 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 on `MemoryArea`.

5. 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 schedulable, 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.

6. The `executeInArea`, `newInstance`, and `newArray` methods, when invoked on an instance of `ScopedMemory` require that instance to be on the current schedulable’s scope stack.

7. An instance of `ScopedMemory` is said to be in use when it has a positive reference count as defined by semantic 1 below.

### 11.2.3 Backing Stores

Every `MemoryArea` has a notional backing store, which is a range of memory addresses from which its allocation area, i.e., the memory range from which objects in it are allocated, is drawn. In the case of Physical Memory, these addresses may be explicitly managed by the programmer, while in other cases they are hidden by the virtual machine and RTSJ implementation. The RTSJ gives the programmer tools to manage the backing store as a limited system resource via the `ScopedMemory` class and its descendants and the `ScopedMemoryParameters` class.

Besides the backing stores for the memory areas provided by the core module, heap and immortal, whose backing stores are fixed or at least predefined at startup, two types of backing store may be available for allocating to newly created memory areas: global backing store and physical backing store. All memory area created via the physical memory factory use the physical memory backing store. All other instances of `ScopedMemory` other than of type `StackedMemory` and root instance of `StackedMemory` take their backing stores from the global backing store that represents system memory available for use as RTSJ memory area allocation areas.

Access to this finite global resource by individual instances of `Schedulable` is controlled by their associated `ScopedMemoryParameters`. Once a given `Schedulable`’s budget of global backing store has been allocated, further attempts to allocate from the global backing store will result in an exception.

### 11.2.4 The Parent Scope

The following requirements apply to a scope’s parent.

1. Instances of `ScopedMemory` have special semantics, including a definition of `parent`. If a `ScopedMemory` object is neither in use nor the initial memory area
for a schedulable, it has no parent scope.

(a) 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.

(b) 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.

2. Instances of ScopedMemory must satisfy the single parent rule, which requires that each scoped memory has a unique parent as defined in semantic 1.

11.2.5 Memory Areas and Schedulables

The following requirements govern the relationship between memory and execution.

1. Pushing a scoped memory onto a scope stack is always subject to the single parent rule.

2. Each schedulable has a default 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 a schedulable may supply constructors with an explicit initial memory area that override the default.

3. 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.

4. A Java thread may use executeInArea, and the newInstance and newArray methods from the ImmortalMemory and HeapMemory classes. These methods enable it to execute with an immortal current allocation context, but semantic of item 3 above applies even during execution of these methods.

11.2.6 Scoped Memory Reference Counting

The following requirements apply to references to scoped memory.

1. Each instance of the class ScopedMemory, or its subclasses, must maintain a reference count which is greater than zero when and only when it is an execution context or more exactly, the reference count is the number of causes for a given memory area to be an execution context.
2. 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).

3. 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 finalization.
   
   (a) When 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.
   
   (b) 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 asynchronous event in the scope is no longer treated as a source of fireability for asynchronous event handlers.
   
   (c) When that action causes object creation in the scope, the finalization process resumes from the beginning;
   
   (d) When the reference count is no longer ready to be decremented to zero, the finalization process terminates.
   
   (e) Otherwise, the reference count is decremented to zero and the memory scope is emptied of all objects.
   
   (f) 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.

4. 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 3 above applies.

5. The `RTSJ` implementation must behave effectively as if during the finalization process the schedulable executing the finalization of a scope holds a synchronized lock that must also be acquired
   
   (a) to increase the reference count when entering the scope,
   
   (b) to increase the reference count during startup for a thread with the finalizing scope as its explicit initial memory area, and
   
   (c) to increase the reference count while making firable an asynchronous event handler with the scope as its explicit initial memory area.
6. Although the steps in scope finalization are ordered, no order is specified for finalization of objects or for disarming fireability of asynchronous event handlers. 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.

7. 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 unfirable, that realtime thread or `AsyncEventHandler` is considered in control of the scope and must execute the finalizers.

8. From the time objects in a scope are deleted until the portal on the scope is successfully set to a reference value (not null) with `setPortal`, the value returned by `getPortal` on that scoped memory object must be `null`.

### 11.2.7 Immortal Memory

The following requirements apply to immortal memory.

1. Objects created in any immortal memory area are unexceptionally referencable from all Java threads, and all schedulables, and the allocation and use of objects in immortal memory is never subject to garbage collection delays.

2. 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.

3. Class objects, the associated static memory, and interned Strings behave effectively as if they were allocated in immortal memory with respect to memory reference and assignment rules, and preemption delays by schedulables which may not access the heap.

4. 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.

### 11.2.8 Maintaining Referential Integrity

The following rules apply to references to objects in scoped memory.
1. Memory 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. Both Immortal-

<table>
<thead>
<tr>
<th>Stored in Area</th>
<th>Reference to Object in Heap</th>
<th>Reference to Object in Immortal</th>
<th>Reference to Object in Scoped</th>
<th>null</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perennial-Memory</td>
<td>Permit</td>
<td>Permit</td>
<td>Forbid</td>
<td>Permit</td>
</tr>
<tr>
<td>Scoped-Memory</td>
<td>Permit</td>
<td>Permit</td>
<td>Permit from same or less deeply nested scope</td>
<td>Permit</td>
</tr>
<tr>
<td>Local Variable</td>
<td>Permit</td>
<td>Permit</td>
<td>Permit</td>
<td>Permit</td>
</tr>
<tr>
<td>Heap Memory</td>
<td>Permit</td>
<td>Permit</td>
<td></td>
<td>Permit</td>
</tr>
</tbody>
</table>

Memory and HeapMemory are types of PerennialMemory. All subclasses of ScopedMemory and PerennialMemory are equivalent to their respective base class for the purposes of this table.

2. 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.

### 11.2.9 Object Initialization

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`.

### 11.2.10 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 memory assignment rules, and the stack is explicitly visible through the `visitRootScopes(Consumer<ScopedMemory>)` and `visitNestedScopes(Consumer<ScopedMemory>)` methods on ScopedMemory.
11 Alternative Memory Areas

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`.

1. The memory area at the top of a schedulable object’s scope stack is the schedulable’s current allocation context.

2. For an instance of `Schedulable`, $n_4$, created by task $t$, the scope stack of $n_t$ is determined by both $t$ and $n_t$:
   (a) when $n_t$ is created in a heap or immortal memory area, $n_t$ is created with a scope stack containing only that heap or immortal memory area,
   (b) when the memory area of $t$ is a `ScopedMemory` instance, $n_t$ 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 $n_t$; and
   (c) when $n_t$ has an explicit initial memory area, $ima$, then $ima$ is pushed on $n_t$’s newly-created scope stack, e.g., a task executing with the scope stack $A \rightarrow B \rightarrow C$ creates a new `Scheduled` instance $s$ with initial memory area $D$ which is not currently in use, $s$ gets the scope stack $A \rightarrow B \rightarrow C \rightarrow D$.

3. When a memory area, $ma$, is entered by calling a `ma.enter` method, $ma$ is pushed onto the scope stack of the current schedulable and becomes its allocation context. When control returns from the `enter` method, the allocation context is popped from the scope stack.

4. When a memory area, $ma$, is entered by calling $ma$’s `executeInArea` method or one of the `ma.newInstance` methods, the scope stack before the method call is preserved and replaced with a scope stack constructed as follows:
   (a) 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
   (b) 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.

1. 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.

2. 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.
11.2.11 The enter Method

For ma.enter(logic):

```plaintext
1. push ma on the scope stack belonging to the current schedulable
2. -- which may throw ScopedCycleException
3. execute logic.run method
4. pop ma from the scope stack
```

11.2.12 The executeInArea or newInstance Methods

For ma.executeInArea(logic), ma.newInstance(), or ma.newInstance():

```plaintext
1. when ma is an instance of PerennialMemory, 
   2. start a new scope stack containing only ma.
   3. make the new scope stack the scope stack for the current schedulable.
   4. else if ma is in the scope stack for the current schedulable, 
   5. start a new scope stack containing ma and all scopes below ma on the scope stack.
   6. make the new scope stack the scope stack for the current schedulable.
   7. else 
   8. throw InaccessibleAreaException, execute logic.run, 
   9. or construct the object.
   10. restore the previous scope stack for the current schedulable.
   11. discard the new scope stack.
12. end
```

11.2.13 Constructor Methods for Schedulables

For construction of a schedulable in memory area cma with initial memory area of ima:

```plaintext
1. if cma is an instance of PerennialMemory, 
   2. create a new scope stack containing cma.
   3. else 
   4. start a new scope stack containing the entire current scope stack.
   5. 6. 
   7. if ima != cma 
   8. push ima on the new scope stack 
   9. -- 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.

11.2.14 The Single Parent Rule

Every push of a scoped memory type on a scope stack must obey 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:
1. when the memory area is not currently on any scope stack, it has no parent;
2. when the memory area is the first scoped memory area on a scope stack, i.e., it was entered from an instance of a `PerennialMemory`, its parent is the primordial scope,
3. 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.

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 `null`, i.e., “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.

11.2.15 Scope Tree Maintenance

The single parent rule is enforced effectively as if there were a tree with the primordial scope at its root, and other nodes corresponding to every scoped memory area currently on any schedulable’s or interrupt service routine’s memory area 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.

When a scoped memory area is the explicit initial memory area of a schedulable or an interrupt service routine that has not terminated, it is referred to as `reserved`. A reserved area with a reference and pin count of zero does not have any objects allocated in it, but it is in a scope stack as long as the schedulable or interrupt service routine is active. Since it is possible for more than one schedulable to have the same explicit initial memory area, the memory area must behave as if a reference count for reservation is also maintained.
11.2.15.1 Pushing a MemoryArea onto the Scope Stack

The following procedure could be used to maintain the scope tree and ensure that push operations on a schedulable’s or ISR’s memory area stack does not violate the single parent rule.

preconditions

ma.parent is set to the correct parent (either a scoped memory area or the primordial scope) or to null (no parent).

Action

if ma is scoped,
    parent = findFirstScope(t.scopeStack).
if ma.parent == null
    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.
}

11.2.15.2 Popping a MemoryArea off the Scope Stack

ma = t.scopeStack.pop.
if ma is scoped
    if !(ma.in_use || (ma.reserve_count > 0))
        ma.parent = null.
11.2.15.3 Reservation Management

Reservation management is separate from managing the scope stack for a schedulable or ISR. Reservations are maintained in order to constrain the code locations at which an exception may be thrown due to violations of the single parent rule with respect to explicit initial scoped memory areas (EISMAs). With the RTSJ reservation management semantics, such exceptions will always be thrown at the explicit invocation of configuration methods. When a realtime thread with an EISMA is created, an ASEH with an EISMA is added to an ASE, or an InterruptServiceRoutine is registered with an interrupt, the following happens atomically with respect to other tasks in the VM:

```java
1 ma = t.eisma // explicit initial scoped memory area
2
3 if (ma.parent == null),
4   ma.parent = findFirstScope(t.scopeStack)
5 ma.reserve_count++. // should now be equal one
6 else if (ma == findFirstScope(t.scopeStack)),
7   ma.reserve_count++. // should now be greater than zero
8 else
9   throw ScopedCycleException.
```

When a realtime thread with an EISMA terminates, an ASEH is removed from an ASE, or an InterruptServiceRoutine is unregistered, the following happens atomically with respect to other tasks in the VM:

```java
1 ma = t.eisma // explicit initial scoped memory area
2
3 ma.reserve_count--.
4 if ((ma.reserve_count == 0) &&
5   (ma.enter_count == 0) &&
6   (ma.pin_count == 0))
7 ma.parent = null.
```

11.2.16 Physical Memory

Physical memory provides a means of allocating Java objects in specific areas of a system’s physical address space. This is accomplished by creating a memory area that resides in the desired address range. The memory area can be any of the memory areas defined by this specification other than heap. A physical memory area is not type distinct from a normal memory area; it is just created by a different means.

1. Physical immortal memory—an immortal memory area that can be created by the application such that the associated allocation 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.

2. Physical scoped memory—a scoped memory area, that can be created by the application such that the associated backing store has specified physical and virtual memory characteristics.

This physical memory model is based on two constraints.

1. Java objects can only be allocated in a memory area when the physical allocation area 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.
   (a) 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.
   (b) Similarly, nonvolatile 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 13.2.1).

2. 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 or from some property file read at program initialization time.

The RTSJ defines a physical memory factory, which maintains a mapping between physical memory characteristics and the associated physical addresses of memory that support those characteristics. The physical memory factory 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 associated with a physical memory range by the application. The physical memory factory 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

---

3For 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/](http://www.acpi.info/)
memory with certain characteristics and a virtual memory type (for example, SRAM that is permanently resident in memory).

11.2.17 Stacked Memory

A StackedMemory area represents both an memory area providing ScopedMemory semantics and an explicit backing store from which its allocation area is drawn. The backing store may be further subdivided into additional allocation areas and backing stores. Such divisions behave as if new allocation areas are allocated contiguously from the bottom of the container, while new backing stores are allocated contiguously from the top, with allocation areas and backing stores meeting when the outer backing store is completely occupied.

11.2.17.1 Avoiding Backing Store Fragmentation

StackedMemory backing stores 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 store, or a guest, which draws its allocation area directly from its parent’s backing store. When a StackedMemory object is created in an allocation context other than StackedMemory, it is necessarily a host and is called a root StackedMemory. In this case, its backing store is drawn from the global backing store or the physical backing store. A root StackedMemory’s backing store will be freed under the same conditions as other host StackedMemory backing stores, but applications should not assume that the implementation provides any guarantees with respect to fragmentation, should this occurs. 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 store is drawn from its parent area’s backing store, and its allocation area is created in the newly-divided backing store. When it is created as a guest, its allocation area is created in its parent’s backing store.

Object lifetimes for objects allocated in StackedMemory allocation contexts are the same as those in ScopedMemory allocation contexts. When a StackedMemory object itself is finalized, its allocation area is returned to the backing store from which it was drawn, and in the case of host StackedMemory areas, the associated backing store is also returned to the parent’s backing store. Additionally, the allocation area of a StackedMemory can be resized under certain conditions. These semantics allow the memory represented by a root StackedMemory backing store 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 by the infrastructure. Those rules are
1. a nonroot StackedMemory area can only be entered by a schedulable when its allocation context is the same as the allocation context in which that StackedMemory area’s object was created;
2. a StackedMemory area may have at most one direct child in the scope stack that is a guest StackedMemory area;
3. a guest StackedMemory area may not have a direct child area that is a host StackedMemory area;
4. a host StackedMemory object cannot be created from another StackedMemory allocation context unless its backing store is allocated from that area’s backing store; and
5. a StackedMemory’s allocation area cannot be resized if there are unfinalized guest StackedMemory allocation areas placed after it in the same backing store.

11.2.17.2 Enforcing Encapsulation

Access to backing store memory for creation of memory areas can also be controlled in a fine-grained fashion by creating a Schedulable with a StackedMemory as its explicit initial scoped memory area. Such a Schedulable may allocate StackedMemory instances only from the backing store of its initial allocation area and its children in the scope stack. This means that a Schedulable thus configured

1. may not construct a root StackedMemory, and
2. may only construct a StackedMemory if its explicit initial scoped memory area is on its current scope stack.

Used in conjunction with the limits provided by ScopedMemoryParameters, this allows a Schedulable’s access to global memory resources to be tightly constrained.

11.2.17.3 Example

Figure 11.1 graphically depicts the behavior of StackedMemory backing stores and allocation areas 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. An implementation may require a constant amount of overhead, drawn from the backing store, for each StackedMemory area created in the store.

```java
1 // Create a StackedMemory with a 10 kB backing store and
2 // 2 kB allocation area
3 rootArea = new StackedMemory(2048, 10240); // (a)
4 rootArea.enter(new Runnable())
5 {
```
Commented points (a), (b), and (c) correspond to their respective subfigures in Figure 11.1. At point (a), a root \texttt{StackedMemory} has been created with its 10 kB backing store drawn from the global store. It contains a 2 kB allocation area, which is then entered. With that allocation area as the current allocation context, a new host \texttt{StackedMemory} is created at (b), reserving 6 kB of the root \texttt{StackedMemory}'s backing store for its own use and creating a second 2 kB allocation area within that reservation. A new guest \texttt{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 backing store in the root area. At this point, the root area’s backing store is almost entirely occupied, with one 2 kB allocation area, one 1.5 kB store, and a 6 kB host area backing store reservation, and 512 B of free backing store in between. The host \texttt{StackedMemory} created at (b) has 4 kB of its backing store remaining unoccupied in its reservation, which could be allocated to additional host or guest \texttt{StackedMemory} areas beneath it in the stack.
11.2.18 Pinnable Memory

As with LTMemory, PinnableMemory class is a subset of ScopedMemory. The only difference between PinnableMemory and LTMemory is that PinnableMemory supports the concept of pinning. Pinning adds some additional requirements to reference counting, as described in Section 11.2.6. To minimize the chance for race conditions, when and from what context a PinnableMemory can be pinned and unpinned is restricted. More details are given in the documentation for PinnableMemory.pin and PinnableMemory.unpin.

11.2.19 Memory Groups

The class MemoryGroup is a subclass of RealtimeThreadGroup for managing memory. It is similar to ProcessingGroup except it targets memory consumption instead of CPU consumption. Currently, only a limit on backing store can be set, but future version of the specification may support more general restrictions. Instance can be used to partition backing store between parts of an application. By nesting instances of MemoryGroup and ProcessingGroup, both memory and CPU can be controlled for arbitrary groups of threads.

11.2.20 Startup Considerations

Normally, a realtime Java program starts with a conventional Java thread. The Alternative Memory Module provide a means of changing this for better interaction with scoped memory. Both thread type and memory area can be set for startup.

By setting the property javax.realtime.start.realtime to true, the Java main thread and all internal system threads are created as realtime threads, instead of conventional Java threads. Though a conventional Java thread may have a realtime priority, it may not enter a scoped memory. This property removes the need to create an extra thread for entering scoped memory.

Furthermore, setting the property javax.realtime.start.immortal to true configures the system to not use a heap at all. Instead, the initial memory area is the instance of ImmortalMemory. An implementation that cannot run without a heap should not start up when this property is set; where possible, the process should return a documented nonzero error code.

The default value for each of these properties is false. Setting javax.realtime.start.immortal to true without setting javax.realtime.start.realtime as well is an error condition. Where possible, the process should return another documented nonzero error code for this error.
11.3 javax.realtime

11.3.1 Enumerations

11.3.1.1 EnclosedType

```java
public enum EnclosedType

Inheritance
    java.lang.Object
    java.lang.Enum<EnclosedType>
    EnclosedType

Description
    Represents type size classes for deciding how large a lambda is. This size is
dependent on what variables the lambda expression contains in its closure, i.e., it
encloses. It is used by the reserveLambda methods in SizeEstimator.

Available since RTSJ 2.0

11.3.1.1.1 Enumeration Constants

BOOLEAN

    public static final EnclosedType BOOLEAN

Description
    Represents a Java boolean.

BYTE

    public static final EnclosedType BYTE

Description
    Represents a Java byte.
```
CHAR

public static final EnclosedType CHAR

Description
Represents a Java char.

SHORT

public static final EnclosedType SHORT

Description
Represents a Java short.

INT

public static final EnclosedType INT

Description
Represents a Java int.

FLOAT

public static final EnclosedType FLOAT

Description
Represents a Java float.

LONG

public static final EnclosedType LONG

Description
Represents a Java long.

DOUBLE

public static final EnclosedType DOUBLE

Description
Represents a Java double.
11 Alternative Memory Areas

REFERENCE

public static final EnclosedType REFERENCE

Description
Represents a reference to any object.

11.3.1.1.2 Methods

values

Signature
public static javax.realtime.EnclosedType[] values()

Description
Gets all enumeration constants.

valueOf(String)

Signature
public static javax.realtime.EnclosedType valueOf(String name)

Description
Gets enumeration constants corresponding to name.

11.3.2 Classes

11.3.2.1 HeapMemory

public class HeapMemory

Inheritance
java.lang.Object
  MemoryArea

460 RTSJ 2.0 (Draft 66)
HeapMemory

Description
The HeapMemory class is a singleton object that allows logic with a non-heap allocation context to allocate objects in the Java heap.

11.3.2.1.1 Methods

enter

Signature
public void enter()

Description
Associates 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) or the enter method exits.

Throws
IllegalTaskStateException—when the caller context in not an instance of Schedulable.
IllegalRealtimeArgumentException—when the caller is a schedulable and a null value for logic was supplied when the memory area was constructed.
MemoryAccessException—when caller is a schedulable which may not use the heap.

enter(Runnable)

Signature
public void enter(Runnable logic)

Description
Associates 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) or the enter method exits.

Parameters

logic—The Runnable object whose run() method should be invoked.

Throws

MemoryAccessError—when caller is a schedulable which may not use the heap.

IllegalTaskStateException—when the caller context is not an instance of Schedulable.

IllegalRealtimeArgumentException—when the caller is a schedulable and logic is null.

instance

Signature

public static javax.realtime.HeapMemory instance()

Description

Returns a reference to the singleton instance of HeapMemory representing the Java heap. The singleton instance of this class shall be allocated in the ImmortalMemory area.

Returns

the singleton HeapMemory object.

executeInArea(Runnable)

Signature

public void executeInArea(Runnable logic)

Description

Executes 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.
Parameters
logic—The runnable object whose run() method should be executed.

Throws
IllegalRealtimeArgumentException—when logic is null.
MemoryAccessError—when caller is a schedulable which may not use the heap.

newArray(Class, int)

Signature
public java.lang.Object
newArray(java.lang.Class<?> type,
    int number)

Description
Allocates an array of the given type in this memory area. This method may be concurrently used by multiple threads.

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
MemoryAccessError—when caller is a schedulable which may not use the heap.
IllegalRealtimeArgumentException—when number is less than zero, type is null, or type is java.lang.Void.TYPE.
OutOfMemoryStaticError—when space in the memory area is exhausted.

Returns
a new array of class type, of number elements.

newInstance(Class)

Signature
public T
newInstance(java.lang.Class<T> type)
    throws IllegalAccessException,
    InstantiationException

Description
RTSJ 2.0 (Draft 66)
Allocates an object in this memory area. This method may be concurrently used by multiple threads.

**Parameters**
- `type`—The class of which to create a new instance.

**Throws**
- `MemoryAccessError`—when caller is a schedulable which may not use the heap.
- `IllegalAccessException`—The class or initializer is inaccessible.
- `IllegalRealtimeArgumentException`—when `type` is null.
- `ExceptionInInitializerError`—when an unexpected exception has occurred in a static initializer.
- `OutOfMemoryStaticError`—when space in the memory area is exhausted.
- `InstantiationException`—when the specified class object could not be instantiated. Possible causes are it is an interface, it is abstract, or it is an array.

**Returns**
a new instance of class `type`.

### `newInstance(Constructor, Object)`

#### Signature
```java
public T newInstance(java.lang.reflect.Constructor<T> c,
                     java.lang.Object[] args)
throws IllegalAccessException,
        InstantiationException,
        InvocationTargetException
```

#### Description
Allocates an object in this memory area. This method may be concurrently used by multiple threads.

**Parameters**
- `c`—The constructor for the new instance.
- `args`—An array of arguments to pass to the constructor.

**Throws**
- `MemoryAccessError`—when caller is a schedulable which may not use the heap.
- `IllegalAccessException`—when the class or initializer is inaccessible under Java access control.
InstantiationException—when the specified class object could not be instantiated. Possible causes are it is an interface, it is abstract, it is an array.

OutOfMemoryStaticError—when space in the memory area is exhausted.

IllegalRealtimeArgumentException—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.

InvocationTargetException—when the underlying constructor throws an exception.

Returns
a new instance of the object constructed by c.

11.3.2.2 ImmortalMemory

public class ImmortalMemory

Inheritance
java.lang.Object
  MemoryArea
    PerennialMemory
      ImmortalMemory

Description

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., javax.realtime.memory.LTMemory, javax.realtime.memory.StackedMemory, or javax.realtime.memory.PinnableMemory.

Objects in immortal memory 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.

11.3.2.2.1 Methods
11 Alternative Memory Areas

instance

Signature

public static javax.realtime.ImmortalMemory instance()

Description

Returns a pointer to the singleton ImmortalMemory object.

Returns

The singleton ImmortalMemory object.

executeInArea(Runnable)

Signature

public void executeInArea(Runnable logic)

Description

Executes 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.

Parameters

logic—The runnable object whose run() method should be executed.

Throws

IllegalRealtimeArgumentException—when logic is null.

11.3.2.3 MemoryArea

public abstract class MemoryArea

Inheritance

java.lang.Object

   MemoryArea

Description

466 RTSJ 2.0 (Draft 66)
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.

11.3.2.3.1 Constructors

MemoryArea(long, Runnable)

Signature
protected
MemoryArea(long size,
Runnable logic)
throws IllegalRealtimeArgumentException,
OutOfMemoryStaticError,
IllegalAssignmentError

Description
Creates an instance of MemoryArea.

Parameters
size—The size of MemoryArea to allocate, in bytes.
logic—A runnable, whose run() method will be called whenever enter() is called.

When logic is null, this constructor is equivalent to MemoryArea(long size).

Throws
IllegalRealtimeArgumentException—when the size parameter is less than zero.
OutOfMemoryStaticError—when there is insufficient memory for the MemoryArea object or for its allocation area in its backing store.
IllegalAssignmentError—when storing logic in this would violate the assignment rules.

MemoryArea(SizeEstimator, Runnable)
protected
MemoryArea(SizeEstimator size,
    Runnable logic)
throws IllegalRealtimeArgumentException,
    OutOfMemoryStaticError,
    IllegalAssignmentError

Description
Equivalent to MemoryArea(long, Runnable) with the argument list (size.
    getEstimate(), logic).

Parameters
size—A SizeEstimator object which indicates the amount of memory required
by this MemoryArea.
logic—A runnable, whose run() method will be called whenever enter() is
called. When logic is null, this constructor is equivalent to Memory-
Area(SizeEstimator size).

Throws
IllegalRealtimeArgumentException—when size is null or size.
    getEstimate() is negative.
OutOfMemoryStaticError—when there is insufficient memory for the MemoryArea
    object or for its allocation area in its backing store.
IllegalAssignmentError—when storing logic in this would violate the assign-
    ment rules.

MemoryArea(long)

Signature
protected
MemoryArea(long size)
throws IllegalRealtimeArgumentException,
    OutOfMemoryStaticError

Description
Equivalent to MemoryArea(long, Runnable) with the argument list (size,
    null).

Parameters
size—The size of MemoryArea to allocate, in bytes.
MemoryArea

java.realtime 11.3

Throws
IllegalRealtimeArgumentException—when size is less than zero.
OutOfMemoryStaticError—when there is insufficient memory for the MemoryArea object or for its allocation area in its backing store.

MemoryArea(SizeEstimator)

Signature
protected
MemoryArea(SizeEstimator size)
throws IllegalRealtimeArgumentException,
OutOfMemoryStaticError

Description
Equivalent to MemoryArea(long, Runnable) with the argument list (size. getEstimate(), null).

Parameters
size—A SizeEstimator object which indicates the amount of memory required by this MemoryArea.

Throws
IllegalRealtimeArgumentException—when the size parameter is null, or size. getEstimate() is negative.
OutOfMemoryStaticError—when there is insufficient memory for the MemoryArea object or for its allocation area in its backing store.

11.3.2.3.2 Methods

enter

Signature
public void
enter()
throws IllegalRealtimeArgumentException,
OutOfMemoryStaticError,
IllegalAssignmentError,
MemoryAccessError

Description
Associates 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`) or the `enter` method exits.

Throws
- `IllegalTaskStateException`—when the caller context is not an instance of `Scheduleable`.
- `IllegalRealtimeArgumentException`—when the caller is a schedulable and a 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`, so the JVM cannot be permitted to deliver the exception. The `ThrowBoundaryError` instance is preallocated by the VM to avoid cascading creation of `ThrowBoundaryError`.
- `MemoryAccessError`—when caller is a schedulable that may not use the heap and this memory area’s logic value is allocated in heap memory.

`enter(Runnable)`

Signature
```java
public void enter(Runnable logic)
```

Description
Associates 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`) or the `enter` method exits.

Parameters
- `logic`—The Runnable object whose `run()` method should be invoked.

Throws
- `470 RTSJ 2.0 (Draft 66)`
IllegalStateException—when the caller context is not an instance of Sched-
ulable.
IllegalRealtimeArgumentException—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, so
the JVM cannot be permitted to deliver the exception. The ThrowBound-
aryError instance is preallocated by the VM to avoid cascading creation of
ThrowBoundaryError.

enter(Supplier)

Signature
public T
    enter(java.util.function.Supplier<T> logic)

Description
Same as enter(Runnable) except that the executed method is called get and an
object is returned. The Supplier.get() method must ensure that the returned
object is allocated outside the area, when the area is not a PerennialMemory.

Parameters
logic—The object whose get method will be executed.

Throws
IllegalAssignmentError—when the return value allocated in area and area is not
a PerennialMemory.

Returns
a result from the computation.

Available since RTSJ 2.0

enter(BooleanSupplier)

Signature
public boolean
    enter(BooleanSupplier logic)

Description

RTSJ 2.0 (Draft 66) 471
Same as `enter(Runnable)` except that the executed method is called `get` and a `boolean` is returned.

**Parameters**
- `logic`—The object whose get method will be executed.

**Returns**
a result from the computation.

**Available since** RTSJ 2.0

---

**`enter(IntSupplier)`**

**Signature**
```
public int enter(IntSupplier logic)
```

**Description**
Same as `enter(Runnable)` except that the executed method is called `get` and an `int` is returned.

**Parameters**
- `logic`—The object whose get method will be executed.

**Returns**
a result from the computation.

**Available since** RTSJ 2.0

---

**`enter(LongSupplier)`**

**Signature**
```
public long enter(LongSupplier logic)
```

**Description**
Same as `enter(Runnable)` except that the executed method is called `get` and a `long` is returned.

**Parameters**
- `logic`—The object whose get method will be executed.

**Returns**
a result from the computation.

**Available since** RTSJ 2.0
a result from the computation.

**Available since** RTSJ 2.0

**enter(DoubleSupplier)**

**Signature**

```java
public double enter(DoubleSupplier logic)
```

**Description**

Same as **enter(Runnable)** except that the executed method is called `get` and a `double` is returned.

**Parameters**

- `logic`—The object whose get method will be executed.

**Returns**

a result from the computation.

**Available since** RTSJ 2.0

**getMemoryArea(Object)**

**Signature**

```java
public static javax.realtime.MemoryArea getMemoryArea(Object object)
```

**Description**

Gets the **MemoryArea** in which the given object is located.

**Throws**

`IllegalRealtimeArgumentException`—when the value of `object` is `null`.

**Returns**

the instance of **MemoryArea** from which `object` was allocated.

**memoryConsumed**

**Signature**

RTSJ 2.0 (Draft 66) 473
public long memoryConsumed()

Description
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.

Returns
the amount of memory consumed in bytes.

memoryRemaining

Signature
public long memoryRemaining()

Description
An approximation of the total amount of memory currently available for future
allocated objects, measured in bytes.

Returns
the amount of remaining memory in bytes.

c newArray(Class, int)

Signature
public java.lang.Object newArray(java.lang.Class<?> type,
int number)
throws IllegalRealtimeArgumentException,
OutOfMemoryStaticError,
RealtimeSecurityException

Description
Allocates an array of the given type in this memory area. This method may be
concurrently used by multiple threads.

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`.

`OutOfMemoryStaticError`—when space in the memory area is exhausted.

`RealtimeSecurityException`—when the caller does not have permission to create
a new instance.

Returns

a new array of class type, of number elements.

newArrayInArea(Object, Class, int)

Signature

```
public java.lang.Object
newArrayInArea(Object object,
      java.lang.Class<?> type, int size)
```

Description

A helper method to create an array of type `type` in the memory area containing
`object`.

Parameters

object—is the reference for determining the area in which to allocate the array.

type—is the type of the array element for the returned array.

size—is the size of the array to return.

Returns

a new array of element `type` with `size` elements.

Available since RTSJ 2.0

newInstance(Class)

Signature

RTSJ 2.0 (Draft 66) 475
public T
newInstance(java.lang.Class<T> type)
throws IllegalAccessException,
    IllegalRealtimeArgumentException,
    InstantiationException,
    OutOfMemoryStaticError,
    ExceptionInInitializerError,
    RealtimeSecurityException

Description
Allocates an object in this memory area. This method may be concurrently used
by multiple threads.

Parameters
type—The class of which to create a new instance.

Throws
IllegalAccessException—The class or initializer is inaccessible.
IllegalRealtimeArgumentException—when type is null.
InstantiationException—when the specified class object could not be instanti-
ated. Possible causes are it is an interface, it is abstract, or it is an array.
ConstructorCheckedException—a checked exception was thrown by the construc-
tor.
OutOfMemoryStaticError—when space in the memory area is exhausted.
ExceptionInInitializerError—when an unexpected exception has occurred in
a static initializer.
RealtimeSecurityException—when the caller does not have permission to create
a new instance.

Returns
a new instance of class type.

newInstance(Constructor, Object)

Signature
public T
newInstance(java.lang.reflect.Constructor<T> c,
            java.lang.Object[] args)
throws ExceptionInInitializerError,
    IllegalAccessException,
    IllegalRealtimeArgumentException,
InstantiationException,
InvocationTargetException,
OutOfMemoryStaticError,
RealtimeSecurityException

Description

Allocates an object in this memory area. This method may be concurrently used by multiple threads.

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.

IllegalRealtimeArgumentException—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.

OutOfMemoryStaticError—when space in the memory area is exhausted.

RealtimeSecurityException—when the caller does not have permission to create a new instance.

Returns

a new instance of the object constructed by c.

size

Signature

public long
size()

Description
Queries 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.

Returns
the size of the memory area in bytes.

evaluateInArea(Runnable)

Signature
public void
evaluateInArea(Runnable logic)
throws IllegalRealtimeArgumentException

Description
Executes the run() method from the logic parameter using this memory area as the current allocation context. The effect of evaluateInArea on the scope stack is specified in the subclasses of MemoryArea.

Parameters
logic—The runnable object whose run() method should be executed.

Throws
IllegalRealtimeArgumentException—when logic is null.

evaluateInArea(Supplier)

Signature
public T
evaluateInArea(java.util.function.Supplier<T> logic)

Description
Same as evaluateInArea(Runnable) except that the executed method is called get and an object is returned. For a memory area that is not a PerennialMemory, care must be taken that the returned value is assignable to an object allocated in the current area.

Parameters
logic—The object whose get method will be executed.

Throws
IllegalAssignmentError—when the return value is not assignable to an object allocated in the current area.
Returns
a result from the computation.

Available since RTSJ 2.0

executeInArea(BooleanSupplier)

Signature
public boolean
executeInArea(BooleanSupplier logic)

Description
Same as executeInArea(Runnable) except that the executed method is called get and a boolean is returned.

Parameters
logic—The object whose get method will be executed.

Returns
a result from the computation.

Available since RTSJ 2.0

executeInArea(IntSupplier)

Signature
public int
executeInArea(IntSupplier logic)

Description
Same as executeInArea(Runnable) except that the executed method is called get and an int is returned.

Parameters
logic—the object whose get method will be executed.

Returns
a result from the computation.

Available since RTSJ 2.0
executeInArea(LongSupplier)

**Signature**
public long executeInArea(LongSupplier logic)

**Description**
Same as `executeInArea(Runnable)` except that the executed method is called `get` and a `long` is returned.

**Parameters**
- `logic`—The object whose get method will be executed.

**Returns**
a result from the computation.

**Available since** RTSJ 2.0

executeInArea(DoubleSupplier)

**Signature**
public double executeInArea(DoubleSupplier logic)

**Description**
Same as `executeInArea(Runnable)` except that the executed method is called `get` and a `double` is returned.

**Parameters**
- `logic`—The object whose get method will be executed.

**Returns**
a result from the computation.

**Available since** RTSJ 2.0

mayHoldReferenceTo

**Signature**
public boolean mayHoldReferenceTo()
Description
Determines 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.

Returns
ture when \( B \) can be assigned to a field of \( A \), otherwise false.

Available since RTSJ 2.0

mayHoldReferenceTo(Object)

Signature
\[
\text{public boolean mayHoldReferenceTo(Object value)}
\]

Description
Determines whether an object \( A \) allocated in the memory area represented by this can hold a reference to the object \( value \).

Parameters
value—The object to test.

Returns
ture when \( value \) can be assigned to a field of \( A \), otherwise false.

Available since RTSJ 2.0

11.3.2.4 MemoryParameters

public class MemoryParameters

Inheritance
java.lang.Object
\[
\text{MemoryParameters}
\]

Interfaces
Cloneable
 Serializable

Description

RTSJ 2.0 (Draft 66) 481
Memory parameters can be given on the constructor of any `Scheduled`. They provide limits on allocation. For garbage-collected objects, they provide the rate of allocation, and for Immortal, the overall amount of allocation.

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 failed object allocation throws an `OutOfMemoryError`.

A `MemoryParameters` object may be bound to more than one schedulable, 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.

As of RTSJ 2.0, instances of `MemoryParameters` are immutable.

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

### 11.3.2.4.1 Fields

#### UNLIMITED

<table>
<thead>
<tr>
<th>public static final long UNLIMITED</th>
</tr>
</thead>
</table>

*Description*

Specifies no maximum limit.

*Available since RTSJ 2.0*

### 11.3.2.4.2 Constructors

#### MemoryParameters(long, long, long)

*Signature*

```java
public
MemoryParameters(long maxInitialArea,
long maxImmortal,
```

**RTSJ 2.0 (Draft 66)**
long allocationRate)
throws IllegalRealtimeArgumentException

Description
Creates a MemoryParameters object with the given values.

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

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

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 UNLIMITED. 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 positive allocation rate limits as UNLIMITED.

Throws
IllegalRealtimeArgumentException—when any value less than zero is passed as the value of maxInitialArea, maxImmortal, or allocationRate.

MemoryParameters(long, long)

Signature
public
MemoryParameters(long maxInitialArea,
    long maxImmortal)

Description
Creates a MemoryParameters object with the given values and allocationRate set to UNLIMITED. It has the same effect as MemoryParameters(maxInitialArea, maxImmortal, UNLIMITED)
MemoryParameters(long)

Signature
public
MemoryParameters(long allocationRate)

Description
Creates a MemoryParameters object with the given values and allocationRate set to allocationRate. It has the same effect as MemoryParameters(UNLIMITED, UNLIMITED, allocationRate)

Available since RTSJ 2.0

11.3.2.4.3 Methods

close

Signature
public java.lang.Object
clone()

Description
Returns 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.
• close 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

getAllocationRate

Signature
public long
getAllocationRate()
MemoryParameters

Description
Determines the limit on the rate of allocation in the heap. Units are in bytes per second.

Returns
the allocation rate in bytes per second. When zero, no allocation is allowed in the heap. When the returned value is UNLIMITED then the allocation rate on the heap is uncontrolled. Enforcement of allocation rates between zero and UNLIMITED is implementation dependent.

getMaxImmortal

Signature
public long getMaxImmortal()

Description
Gets the limit on the amount of memory the schedulable may allocate in the immortal area. Units are in bytes.

Returns
the limit on immortal memory allocation. When zero, no allocation is allowed in immortal memory. When the returned value is UNLIMITED then there is no limit for allocation in immortal memory.

getMaxInitialArea

Signature
public long getMaxInitialArea()

Description
Gets the limit on the amount of memory the schedulable may allocate in its initial memory area, when initial is a scoped memory. Units are in bytes.

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 UNLIMITED then there is no limit for allocation in the initial memory area.

Available since RTSJ 2.0
11.3.2.5 PerennialMemory

public abstract class PerennialMemory

Inheritance
    java.lang.Object
    MemoryArea
        PerennialMemory

Description
A base class for all memory areas whose contents can be unexceptionally referenced. In other words, any memory area can store a reference to an object stored in one of these areas. This includes all concrete memory areas in the core package. Only memory areas of this type can be a root for a scoped memory.

11.3.2.6 SizeEstimator

public class SizeEstimator

Inheritance
    java.lang.Object
        SizeEstimator

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

SizeEstimator is a floor on the amount of memory that should be allocated. Many objects allocate other objects when they are constructed. SizeEstimator only estimates the memory requirement of the object itself, it does not include memory required for any objects allocated at construction time. 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)
11.3.2.6.1 Constructors

SizeEstimator

Signature
public SizeEstimator()

Description
Create an empty size estimator.

11.3.2.6.2 Methods

reserve(Class, int)

Signature
public void reserve(java.lang.Class<?> c, int number)

Description
Takes into account additional number instances of Class c when estimating the size of the MemoryArea.

Parameters
c—The class to take into account.
number—The number of instances of c to estimate.

Throws
IllegalRealtimeArgumentException—when c is null or number is negative.

Available since RTSJ 2.0 throws IllegalRealtimeArgumentException also when number is less than zero.
reserve(SizeEstimator, int)

Signature

public void
reserve(SizeEstimator estimator,
      int number)

Description

Takes into account additional number of the estimations from instances of SizeEstimator size when estimating the size of the MemoryArea.

Parameters

estimator—The given instance of SizeEstimator.
number—The number of times to reserve the size denoted by estimator.

Throws

IllegalRealtimeArgumentException—when estimator is null or number is less than zero.

Available since RTSJ 2.0 throws IllegalRealtimeArgumentException also when number is less than zero.

reserve(SizeEstimator)

Signature

public void
reserve(SizeEstimator size)

Description

Takes into account an additional estimation from the instance of SizeEstimator size when estimating the size of the MemoryArea.

Parameters

size—The given instance of SizeEstimator.

Throws

IllegalRealtimeArgumentException—when size is null.

reserveArray(int)

Signature

488 RTSJ 2.0 (Draft 66)
public void reserveArray(int length)

Description

Takes into account an additional instance of an array of length reference values when estimating the size of the MemoryArea.

Parameters

length—The number of entries in the array.

Throws

IllegalRealtimeArgumentException—when length is negative.

Available since RTSJ 1.0.1

reserveArray(int, Class)

Signature

public void reserveArray(int length, java.lang.Class<?> type)

Description

Takes into account an additional instance of an array of length primitive values when estimating the size of the MemoryArea.

Class values for the primitive types are available from the corresponding class types; e.g., Byte.TYPE, Integer.TYPE, and Short.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

IllegalRealtimeArgumentException—when length is negative, or type does not represent a primitive type.

Available since RTSJ 1.0.1
reserveLambda(EnclosedType, EnclosedType, EnclosedType)

**Signature**

```java
class SizeEstimator
{
    public void reserveLambda(EnclosedType first,
                               EnclosedType second,
                               EnclosedType[] others)
}
```

**Description**

Determines the size of a lambda with more than two variables in its closure and add it to this size estimator.

**Parameters**

- `first`—Type of first variable in closure.
- `second`—Type of second variable in closure.
- `others`—Types of additional variables in closure.

**Available since** RTSJ 2.0

reserveLambda(EnclosedType, EnclosedType)

**Signature**

```java
class SizeEstimator
{
    public void reserveLambda(EnclosedType first,
                               EnclosedType second)
}
```

**Description**

Determines the size of a lambda with two variables in its closure and add it to this size estimator.

**Parameters**

- `first`—Type of first variable in closure.
- `second`—Type of second variable in closure.

**Available since** RTSJ 2.0

reserveLambda(EnclosedType)

**Signature**

```java
class SizeEstimator
{
    public void reserveLambda(EnclosedType)
}
```
public void reserveLambda(EnclosedType first)

Description
Determines the size of a lambda with one variable in its closure and add it to this size estimator.

Parameters
first—Type of first variable in closure.
Available since RTSJ 2.0

reserveLambda

Signature
public void reserveLambda()

Description
Determines the size of a lambda with no closure and add it to this size estimator.
Available since RTSJ 2.0

gEstimate

Signature
public long getEstimate()

Description
Gets an estimate of the number of bytes needed to store all the objects reserved.

Returns
the estimated size in bytes.

clear

Signature
public void
clear()

Description
Restores the estimate value to zero for reuse.

Available since rtsj 2.0
11.4  javax.realtime.memory

11.4.1  Interfaces

11.4.1.1  PhysicalMemoryCharacteristic

```java
public interface PhysicalMemoryCharacteristic

Description
A tagging interface used to identify physical memory characteristics. Applications can give names to regions of memory that are described by PhysicalMemoryRegion. The names are defined by creating instances of this interface. For example, final static PhysicalMemoryCharacteristic STATIC_RAM = ...;
```

Available since RTSJ 2.0

11.4.2  Enumerations

11.4.2.1  PhysicalMemorySelector.CachingBehavior

```java
public enum PhysicalMemorySelector.CachingBehavior

Inheritance
java.lang.Object
   java.lang.Enum<PhysicalMemorySelector.CachingBehavior>
      PhysicalMemorySelector.CachingBehavior

Description
Marker for standard caching behaviors. Not all need be supported. For example, a VM running in Kernel mode might only support DISABLED.
```

11.4.2.1.1  Enumeration Constants

```java
DISABLED
```

public static final PhysicalMemorySelector.CachingBehavior DISABLED
11 Alternative Memory Areas

PhysicalMemorySelector.CachingBehavior

Description

Represents direct mapping, i.e., caching disabled or not present.

WRITE_THROUGH

public static final PhysicalMemorySelector.CachingBehavior
         WRITE_THROUGH

Description

Represents caching where writes are immediately written to cache and memory.

WRITE_BACK

public static final PhysicalMemorySelector.CachingBehavior
         WRITE_BACK

Description

Represents caching where writes are immediately written to cache, but the write
to memory is delayed, such as until the cache line is flushed.

11.4.2.1.2 Methods

values

Signature

public static javax.realtime.memory.PhysicalMemorySelector.CachingBehavior[]
         values()

Description

Gets all enumeration constants.
valueOf(String)

Signature

public static javax.realtime.memory.PhysicalMemorySelector.CachingBehavior

valueOf(String name)

Description

Gets enumeration constants corresponding to name.

11.4.2.2 PhysicalMemorySelector.PagingBehavior

public enum PhysicalMemorySelector.PagingBehavior

Inheritance

java.lang.Object

java.lang.Enum<PhysicalMemorySelector.PagingBehavior>

PhysicalMemorySelector.PagingBehavior

Description

Marker for standard paging behaviors. Not all need be supported. For example, a VM running in Kernel mode might only support DIRECT.

11.4.2.2.1 Enumeration Constants

DIRECT

public static final PhysicalMemorySelector.PagingBehavior DIRECT

Description

Represents when the page is always mapped with the same virtual address as its physical address.
FIXED

public static final PhysicalMemorySelector.PagingBehavior FIXED

Description
Represents when the page stays resident in memory, so the mapping does not change.

SWAPPABLE

public static final PhysicalMemorySelector.PagingBehavior SWAPPABLE

Description
Represents when the page may be swapped to disk; hence the mapping may change and encounter delay when accessed while swapped out.

11.4.2.2.2 Methods

class values

Signature
public static javax.realtime.memory.PhysicalMemorySelector.PagingBehavior[] values()

Description
Gets all enumeration constants.

class valueOf(String)

Signature
public static javax.realtime.memory.PhysicalMemorySelector.PagingBehavior valueOf(String name)

Description
Gets enumeration constants corresponding to name.
public class LTMemory

*Inheritance*

java.lang.Object
    javax.realtime.MemoryArea
    ScopedMemory
    LTMemory

*Description*

**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 *size*.

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 for a **javax.realtime.Schedulable** instance which may not use the **javax.realtime.HeapMemory** or to enter the memory area using the **ScopedMemory.enter** method within such an instance.

Enough memory must be committed by the completion of the constructor to satisfy the memory requirement given in the constructor. Committed means that this memory must always be available for allocation. The 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.

Creation of an **LTMemory** shall fail with a **javax.realtime.OutOfMemoryStaticError** when the current **javax.realtime.Schedulable** has been configured with a **ScopedMemoryParameters.getMaxGlobalBackingStore** that would be exceeded by said creation.

Methods from **LTMemory** should be overridden only by methods that use **super**.

See Section **javax.realtime.MemoryArea**

See Section **ScopedMemory**

See Section **javax.realtime.Schedulable**
11 Alternative Memory Areas

Available since RTSJ 2.0 moved to this package.

11.4.3.1.1 Constructors

LTMemory(long, Runnable)

Signature
   public
   LTMemory(long size,
            Runnable logic)

Description
   Create a scoped memory of the given size and with the give logic to run upon
   entry when no other logic is given.

Available since RTSJ 1.0.1

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 ini-
   tial memory area. When logic is null, this constructor is equivalent to
   LTMemory(long).

Throws
   javax.realtime.IllegalRealtimeArgumentException—when size is less than
   zero.
   javax.realtime.OutOfMemoryStaticError—when there is insufficient memory
   for the LTMemory object or for its allocation area in its backing store, or
   when the current Schedulable would exceed its configured allowance of global
   backing store.
   javax.realtime.IllegalAssignmentError—when storing logic in this would
   violate the assignment rules.

LTMemory(SizeEstimator, Runnable)
**Signature**

```java
public LTMemory(SizeEstimator size, Runnable logic)
```

**Description**

Equivalent to `LTMemory(long, Runnable)` with argument list `(size, getEstimate(), runnable)`.

**Available since** RTSJ 1.0.1

**Parameters**

- `size`—An instance of `javax.realtime.SizeEstimator` 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)`.

**Throws**

- `javax.realtime.IllegalRealtimeArgumentException`—when `size` is `null`.
- `javax.realtime.OutOfMemoryStaticError`—when there is insufficient memory for the `LTMemory` object or for its allocation area in its backing store, or when the current `Schedulable` would exceed its configured allowance of global backing store.
- `javax.realtime.IllegalAssignmentError`—when storing `logic` in `this` would violate the assignment rules.

**LTMemory(long)**

**Signature**

```java
public LTMemory(long size)
```

**Description**

Equivalent to `LTMemory(long, Runnable)` with the argument list `((size, null))`.

**Available since** RTSJ 1.0.1

**Parameters**
11 Alternative Memory Areas

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
javax.realtime.IllegalRealtimeArgumentException—when size is less than zero.
javax.realtime.OutOfMemoryStaticError—when there is insufficient memory for the LTMemory object or for its allocation area in its backing store, or when the current Schedulable would exceed its configured allowance of global backing store.

LTMemory(SizeEstimator)

Signature
public LTMemory(SizeEstimator size)

Description
Equivalent to LTMemory(long, Runnable) with argument list (size.getEstimate(), null).

Available since RTSJ 1.0.1

Parameters
size—An instance of javax.realtime.SizeEstimator used to give an estimate of the initial size. This memory must be committed before the completion of the constructor.

Throws
javax.realtime.IllegalRealtimeArgumentException—when size is null.
javax.realtime.OutOfMemoryStaticError—when there is insufficient memory for the LTMemory object, or when the current Schedulable would exceed its configured allowance of global backing store.

11.4.3.2 MemoryGroup

public class MemoryGroup

Inheritance

500 RTSJ 2.0 (Draft 66)
MemoryGroup

java.lang.Object
  java.lang.ThreadGroup
    javax.realtime.RealtimeThreadGroup
      MemoryGroup

Description
A group that limits the amount of memory available to a task. This just provides a limit for the use of backingstore, but future implementations may also limit other allocations. A nested memory group may not have a limit that exceeds that of its parent.

Available since RTSJ 2.0

11.4.3.2.1 Constructors

MemoryGroup(RealtimeThreadGroup, String)

Signature
public
MemoryGroup(RealtimeThreadGroup parent,
            String name)

Description
Creates a new group under parent named name.

MemoryGroup(String)

Signature
public
MemoryGroup(String name)
throws IllegalRealtimeStateException,
       IllegalAssignmentError

Description
Creates a new group under the current realtime thread group named name. It is the same as new MemoryGroup(null, name);
11 Alternative Memory Areas

11.4.3.2.2 Methods

**getMaxBackingStore**

*Signature*

```java
public long getMaxBackingStore()
```

*Description*

Determines how much backing store can be used by tasks in this group.

*Returns*

the total amount available.

**setMaxBackingStore(long)**

*Signature*

```java
public javax.realtime.memory.MemoryGroup setMaxBackingStore(long value)
```

*Description*

Sets the max backing store of task in this group. When the limit is exceeded, subsequent creations of LTMemory, PinnableMemory, and roots StackedMemory will fail with an Error: `javax.realtime.OutOfMemoryStaticError`.

*Parameters*

- value—The maximum backing store in bytes for tasks in this group

*Returns*

this object

11.4.3.3 PhysicalMemoryFactory

```java
public class PhysicalMemoryFactory

Inheritance

java.lang.Object
  PhysicalMemoryFactory
```

502 RTSJ 2.0 (Draft 66)
Description

A physical memory representation of memory ranges, in the form of `PhysicalMemoryRegion` instances with physical memory characteristics in the form of `PhysicalMemoryCharacteristic` instances for creating memory areas in those ranges.

Each physical memory module can have one or more physical memory characteristic. A physical memory characteristic can apply to many physical memory modules. The range of physical addresses of modules shall not overlap. A memory that spans more than one physical memory module may not be created.

The `PhysicalMemoryFactory` determines the physical addresses from the modules and keeps a relation between instances of `PhysicalMemoryRegion` and Physical Memory Addresses. The range of physical addresses of modules shall not overlap. A created memory area may not span more than one physical memory module. To find a memory range that supports PMC A and PMC B, it uses set intersection modules(A) \( \cap \) modules(B)

Available since RTSJ 2.0

11.4.3.3.1 Constructors

`PhysicalMemoryFactory`

Signature

```java
public
PhysicalMemoryFactory()
```

Description

Creates an empty factory, but when only one factory is required, uses `getDefault` instead.

11.4.3.3.2 Methods
getDefault

Signature
public static javax.realtime.memory.PhysicalMemoryFactory
default()
public void
associate(PhysicalMemoryCharacteristic[] names,
PhysicalMemoryRegion module)
throws IllegalRealtimeArgumentException,
RangeOutOfBoundsException

Description
Associates an array of programmer-defined names with a physical address range.

Parameters
names—The array of physical memory characteristics. e.g { STATIC_RAM }.
module—The object representing a range of contiguous physical addresses.

Throws
javax.realtime.IllegalRealtimeArgumentException—when either names or module is null.
RangeOutOfBoundsException—when module overlaps a previously associated
PhysicalMemoryRegion instance or cannot be mapped as Java object storage.

associate(PhysicalMemoryCharacteristic, PhysicalMemoryRegion)

Signature
public static void
associate(PhysicalMemoryCharacteristic name,
PhysicalMemoryRegion[] modules)
throws IllegalRealtimeArgumentException,
RangeOutOf BoundsException

Description
Associates a programmer-defined name with an array of physical address ranges.

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
javax.realtime.IllegalRealtimeArgumentException—when either name or modules is null.
RangeOutOfBoundsException—when module overlaps a previously associated
PhysicalMemoryRegion instance or cannot be mapped as Java object storage.
createImmortalMemory(PhysicalMemorySelector, long, Runnable)

Signature

```java
public javax.realtime.ImmortalMemory
createImmortalMemory(PhysicalMemorySelector selector,
long size,
Runnable logic)
throws RealtimeSecurityException,
SizeOutOfBoundsException,
UnsupportedPhysicalMemoryException,
MemoryTypeConflictException,
IllegalRealtimeArgumentException
```

Description

Instantiates a `javax.realtime.ImmortalMemory` object in a `PhysicalMemoryRegion`, where the memory type matches the `PhysicalMemoryCharacteristic` instances in `selector` and the virtual memory parameters of `selector` are applied.

Parameters

- `selector`—Used to choose the memory module and set the virtual mapping.
- `size`—The size of memory to be taken out of the selected module.
- `logic`—The logic to execute on entry (may be null).

Throws

- `RealtimeSecurityException`—when the application does not have permission to access physical memory or the given range of memory.
- `javax.realtime.SizeOutOfBoundsException`—when the implementation detects that `size` extends beyond a physically addressable memory module.
- `javax.realtime.UnsupportedPhysicalMemoryException`—when the underlying hardware does not support the given type, or when no matching `PhysicalMemoryCharacteristic` has been registered with this `PhysicalMemoryFactory`.
- `MemoryTypeConflictException`—when the specified base does not point to memory that matches the requested type, or when `type` specifies incompatible memory attributes.
- `javax.realtime.IllegalRealtimeArgumentException`—when `size` is less than zero.

Returns

the new memory area
createLTMemory(PhysicalMemorySelector, long, Runnable)

**Signature**

```java
public javax.realtime.memory.PinnableMemory createLTMemory(PhysicalMemorySelector selector,
long size,
Runnable logic)
throws RealtimeSecurityException,
SizeOutOfBoundsException,
UnsupportedPhysicalMemoryException,
MemoryTypeConflictException,
IllegalRealtimeArgumentException
```

**Description**

Instantiates a LTMemory object in a PhysicalMemoryRegion, matching the PhysicalMemoryCharacteristic in selector with virtual memory parameters of selector applied.

**Parameters**

- **selector**—Used to choose the memory module and set the virtual mapping.
- **size**—The size of memory to be taken out of the selected module.
- **logic**—The logic to execute on entry (may be null).

**Throws**

- **RealtimeSecurityException**—when the application does not have permission to access physical memory or the given range of memory.
- **javax.realtime.SizeOutOfBoundsException**—when the implementation detects that size extends beyond a physically addressable memory module.
- **javax.realtime.UnsupportedPhysicalMemoryException**—when the underlying hardware does not support the given type, or when no matching PhysicalMemoryCharacteristic has been registered with this PhysicalMemoryFactory.
- **MemoryTypeConflictException**—when the specified base does not point to memory that matches the requested type, or when type specifies incompatible memory attributes.
- **javax.realtime.IllegalRealtimeArgumentException**—when size is less than zero.

**Returns**

the new memory area
createPinnableMemory(PhysicalMemorySelector, long, Runnable)

Signature

```java
public javax.realtime.memory.PinnableMemory createPinnableMemory(PhysicalMemorySelector selector, long size, Runnable logic)
throws RealtimeSecurityException,
SizeOutOfBoundsException,
UnsupportedPhysicalMemoryException,
MemoryTypeConflictException,
IllegalRealtimeArgumentException
```

Description

Instantiates a PinnableMemory object in a PhysicalMemoryRegion, matching the PhysicalMemoryCharacteristic in selector with virtual memory parameters of selector applied.

Parameters

selector — Used to choose the memory module and set the virtual mapping.
size — The size of memory to be taken out of the selected module.
logic — The logic to execute on entry (may be null).

Throws

- RealtimeSecurityException — when the application does not have permissions to access physical memory or the given range of memory.
- javax.realtime.SizeOutOfBoundsException — when the implementation detects that size extends beyond a physically addressable memory module.
- javax.realtime.UnsupportedPhysicalMemoryException — when the underlying hardware does not support the given type, or when no matching PhysicalMemoryCharacteristic has been registered with this PhysicalMemoryFactory.
- MemoryTypeConflictException — when the specified base does not point to memory that matches the requested type, or when type specifies incompatible memory attributes.
- javax.realtime.IllegalRealtimeArgumentException — when size is less than zero.

Returns

the new memory area
createStackedMemory(PhysicalMemorySelector, long, long, Runnable)

Signature

```java
public javax.realtime.memory.StackedMemory
createStackedMemory(PhysicalMemorySelector selector,
    long scopeSize,
    long backingSize,
    Runnable logic)
```

throws RealtimeSecurityException,
SizeOutOfBoundsException,
UnsupportedPhysicalMemoryException,
MemoryTypeConflictException,
IllegalRealtimeArgumentException

Description

Instantiates a StackedMemory object in a PhysicalMemoryRegion, matching the PhysicalMemoryCharacteristic in selector with virtual memory parameters of selector applied.

Parameters

- `selector`—Used to choose the memory module and set the virtual mapping.
- `scopeSize`—The size of the scope to be created.
- `backingSize`—The size of the backing store to take out of the selected module.
- `logic`—The logic to execute on entry (may be null).

Throws

- RealtimeSecurityException—when the application does not have permissions to access physical memory or the given range of memory.
- javax.realtime.SizeOutOfBoundsException—when the implementation detects that size extends beyond a physically addressable memory module.
- javax.realtime.UnsupportedPhysicalMemoryException—when the underlying hardware does not support the given type, or when no matching PhysicalMemoryCharacteristic has been registered with this PhysicalMemoryFactory.
- MemoryTypeConflictException—when the specified base does not point to memory that matches the requested type, or when type specifies incompatible memory attributes.
- javax.realtime.IllegalRealtimeArgumentException—when scopeSize or backingSize is less than zero.

Returns
public class PhysicalMemoryRegion

**Inheritance**
java.lang.Object
  PhysicalMemoryRegion

**Description**
Enables an application to define a range of physical memory addresses.

**Available since** RTSJ 2.0

### 11.4.3.4.1 Constructors

**PhysicalMemoryRegion(long, long)**

**Signature**
```java
class PhysicalMemoryRegion
{
    public PhysicalMemoryRegion(long base, long length)
}
```

**Description**
Creates an instance representing a range of contiguous physical memory.

**Parameters**
- **base**—A physical address.
- **length**—The size of contiguous memory from that base.

**Throws**
- `javax.realtime.IllegalRealtimeArgumentException`—when length is less than or equal to zero, or when base is less than zero or when this module overlaps with another memory module.
- `javax.realtime.SizeOutOfBoundsException`—when base + length is greater than the physical address range of the processor.
11.4.3.4.2 Methods

**getBase**

*Signature*

```java
public long getBase()
```

*Description*

Gets the base address of the contiguous memory represented by this.

*Returns*

the base address

**getLength**

*Signature*

```java
public long getLength()
```

*Description*

Gets the length of the contiguous memory represented by this.

*Returns*

the length

11.4.3.5 PhysicalMemorySelector

```java
public class PhysicalMemorySelector
```

*Inheritance*

```
java.lang.Object
   PhysicalMemorySelector
```

*Description*

Provides characteristics both for physical memory, used to select a memory range from a memory module, and for virtual memory to be used for setting the characteristics of the mapped pages.
11 Alternative Memory Areas

**PhysicalMemorySelector**

**Available since RTSJ 2.0**

### 11.4.3.5.1 Constructors

**PhysicalMemorySelector(PhysicalMemoryCharacteristic, PhysicalMemoryCharacteristic, CachingBehavior, PagingBehavior)**

**Signature**

```java
public PhysicalMemorySelector(PhysicalMemoryCharacteristic[] request,
                             PhysicalMemoryCharacteristic[] reject,
                             PhysicalMemorySelector.CachingBehavior caching,
                             PhysicalMemorySelector.PagingBehavior paging)
```

**Description**

Creates a selector object for obtaining physical memory objects. The arguments are used to select the desired memory type.

**Parameters**

- `request`—characteristics that are required
- `reject`—characteristics that should apply to the object
- `caching`—the required caching behavior
- `paging`—the required paging behavior

### 11.4.3.5.2 Methods

**getSupportedCachingBehavior**

**Signature**

```java
public static javax.realtime.memory.PhysicalMemorySelector.CachingBehavior[]
getSupportedCachingBehavior()
```

**Description**

512 RTSJ 2.0 (Draft 66)
Gets the caching behaviors that are supported by this JVM

**Returns**

an array of the supported caching behaviors.

### getSupportedPagingBehavior

**Signature**

```java
public static javax.realtime.memory.PhysicalMemorySelector.PagingBehavior[] getSupportedPagingBehavior()
```

**Description**

Gets the paging behaviors that are supported by this JVM

**Returns**

an array of the supported paging behaviors.

### getRequestSet

**Signature**

```java
public javax.realtime.memory.PhysicalMemoryCharacteristic[] getRequestSet()
```

**Description**

A getter for the `PhysicalMemoryCharacteristic` list to be requested

**Returns**

the `PhysicalMemoryCharacteristic` list

### getRejectSet

**Signature**

```java
public javax.realtime.memory.PhysicalMemoryCharacteristic[] getRejectSet()
```

**Description**

A getter for the `PhysicalMemoryCharacteristic` list to be excluded

**Returns**

the PhysicalMemoryCharacteristic list
11 Alternative Memory Areas

getCachingBehavior

Signature

public javax.realtime.memory.PhysicalMemorySelector.CachingBehavior
getCachingBehavior()

Description

A getter for the CachingBehavior to be requested

Returns

the CachingBehavior

getPagingBehavior

Signature

public javax.realtime.memory.PhysicalMemorySelector.PagingBehavior
getPagingBehavior()

Description

A getter for the PagingBehavior to be requested

Returns

the PagingBehavior

11.4.3.6 PinnableMemory

public class PinnableMemory

Inheritance

java.lang.Object
javax.realtime.MemoryArea
    ScopedMemory
        PinnableMemory

Description

This class is for passing information between different threads as in the producer consumer pattern. One thread can enter an empty PinnableMemory, allocate some data structure, put a reference in the portal, pin the scope, exit it, and then pass it to another thread for further processing or consumption. Once the last thread is done, the memory can be unpinned, causing its contents to be freed.
Creation of a `PinnableMemory` shall fail with a `javax.realtime.OutOfMemoryStaticError` when the current `javax.realtime.Schedulable` has been configured with a `ScopedMemoryParameters.getMaxGlobalBackingStore` that would be exceeded by said creation.

Available since RTSJ 2.0

11.4.3.6.1 Constructors

### PinnableMemory(long)

**Signature**

```java
public PinnableMemory(long size)
throws IllegalRealtimeArgumentException,
        OutOfMemoryStaticError
```

**Description**

Creates a scoped memory of fixed size that can be held open when no `javax.realtime.Schedulable` has it on its scoped memory stack.

**Parameters**

- `size`—The number of bytes in the memory area.

**Throws**

- `javax.realtime.IllegalRealtimeArgumentException`—when `size` is less than zero.
- `javax.realtime.OutOfMemoryStaticError`—when there is insufficient memory for the `PinnalbeMemory` object or for its allocation area in its backing store, or when the current `Scheduled` would exceed its configured allowance of global backing store.

### PinnableMemory(SizeEstimator)

**Signature**

```java
RTSJ 2.0 (Draft 66) 515
```
public PinnableMemory(SizeEstimator size) throws IllegalRealtimeArgumentException, OutOfMemoryStaticError

Description
Equivalent to PinnableMemory(long) with size.getEstimate() as its argument.

Parameters
size—An estimator for determining the number of bytes in the memory area.

Throws
javax.realtime.IllegalRealtimeArgumentException—when size is null.
javax.realtime.OutOfMemoryStaticError—when there is insufficient memory for the PinnalbeMemory object or for its allocation area in its backing store, or when the current Schedulable would exceed its configured allowance of global backing store.

11.4.3.6.2 Methods

pin

Signature
public void pin() throws IllegalRealtimeStateException

Description
Prevents the contents from being freed.

Throws
IllegalRealtimeStateException—when the current allocation context is not this allocation context.

unpin

Signature
public void unpin()
    throws IllegalRealtimeStateException

Description

Allows the contents to be freed the next time no \texttt{javax.realtime.Schedulable} is active within the scope.

Throws

\texttt{javax.realtime.IllegalRealtimeStateException}—when schedulable does not have this memory area as its current memory area.

\textbf{isPinned}

\textit{Signature}

\begin{verbatim}
    public boolean isPinned()
\end{verbatim}

\textit{Description}

Determines whether the scope may be cleared on last exit.

\textit{Returns}

true when yes, otherwise false.

\textbf{getPinCount}

\textit{Signature}

\begin{verbatim}
    public int getPinCount()
\end{verbatim}

\textit{Description}

Finds out how many times the scope has been pinned, but not unpinned.

\textit{Returns}

the number of outstanding pins.
join

Signature

```java
public void
join()
throws InterruptedException
```

Description

Waits until the reference count of this `ScopedMemory` goes down to zero. Returns immediately when the memory is unreferenced.

Throws

- `java.lang.InterruptedException`—when this schedulable is interrupted by `javax.realtime.RealtimeThread.interrupt()` or `javax.realtime.control.AsynchronouslyInterruptedException.fire()` while waiting for the reference count to go to zero.
- `IllegalTaskStateException`—when the execution context is not an instance of `javax.realtime.Schedulable`.

`join(HighResolutionTime)`

Signature

```java
public void
join(javax.realtime.HighResolutionTime<?> time)
throws InterruptedException
```

Description

Waits at most until the time designated by the `time` parameter for the reference count of this `ScopedMemory` to drop to zero. Returns immediately when the memory area is unreferenced.

Since the time is expressed as a `javax.realtime.HighResolutionTime`, 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 `join` returns immediately.

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

java.lang.InterruptedException—when this schedulable is interrupted by javax.realtime.RealtimeThread.interrupt() or javax.realtime.control.AsynchronouslyInterruptedException.fire() while waiting for the reference count to go to zero.

IllegalTaskStateException—when the execution context is not an instance of javax.realtime.Schedulable.

javax.realtime.IllegalRealtimeArgumentException—when the execution context is a schedulable and time is null.

UnsupportedRealtimeOperationException—when the wait operation is not supported using the clock associated with time.

joinAndEnter

Signature

public void joinAndEnter() throws InterruptedException

Description

Same as ScopedMemory.joinAndEnter() except the memory must also not be pinned.

Throws

java.lang.InterruptedException—when this schedulable is interrupted by javax.realtime.RealtimeThread.interrupt() or javax.realtime.control.AsynchronouslyInterruptedException.fire() while waiting for the reference count to go to zero.

IllegalTaskStateException—when the execution context is not an instance of javax.realtime.Schedulable or when this method is invoked during finalization of objects in scoped memory and entering this scoped memory area would force deletion of the task that triggered finalization. This would include the scope containing the task, and the scope (if any) containing the scope containing the task.
ThrowBoundaryError—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 `javax.realtime.IllegalAssignmentError`, so the JVM cannot be permitted to deliver the exception. The `javax.realtime.ThrowBoundaryError` is allocated in the current allocation context and contains information about the exception it replaces.

`javax.realtime.ScopedCycleException`—when this invocation would break the single parent rule.

`javax.realtime.IllegalRealtimeArgumentException`—when the execution context 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(Runnable)**

*Signature*

```java
public void joinAndEnter(Runnable logic)
throws InterruptedException
```

*Description*

Same as `ScopedMemory.joinAndEnter(Runnable)` except the memory must also not be pinned.

*Parameters*

`logic`—The `Runnable` object which contains the code to execute.

*Throws*

`java.lang.InterruptedException`—when this schedulable is interrupted by `javax.realtime.RealtimeThread.interrupt()` or `javax.realtime.control.AsynchronouslyInterruptedException.fire()` while waiting for the reference count to go to zero.

`IllegalTaskStateException`—when the execution context is not an instance of `javax.realtime.Schedulable` or when this method is invoked during finalization of objects in scoped memory and entering this scoped memory area would force deletion of the task that triggered finalization. This would include the scope containing the task, and the scope (if any) containing the scope containing the task.

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 `javax.realtime.IllegalAssignmentError`, so the JVM cannot be permitted to deliver the exception. The `javax.realtime.ThrowBoundaryError` is allocated in the current allocation context and contains information about the exception it replaces.

`javax.realtime.ScopedCycleException`—when this invocation would break the single parent rule.

`javax.realtime.IllegalRealtimeArgumentException`—when the execution context is a schedulable and `logic` is `null`.

### joinAndEnter(Supplier)

**Signature**

```java
public T
joinAndEnter(java.util.function.Supplier<T> logic)
```

**Description**

Same as `ScopedMemory.joinAndEnter(Supplier)` except the memory must also not be pinned.

**Parameters**

- `logic`—The object whose get method will be executed.

**Returns**

- a result from the computation.

### joinAndEnter(BooleanSupplier)

**Signature**

```java
public boolean
joinAndEnter(BooleanSupplier logic)
```

**Description**

Same as `ScopedMemory.joinAndEnter(BooleanSupplier)` except the memory must also not be pinned.

**Parameters**

- `logic`—The object whose get method will be executed.

**Returns**

- a result from the computation.
joinAndEnter(IntSupplier)

Signature
   public int
   joinAndEnter(IntSupplier logic)

Description
   Same as ScopedMemory.joinAndEnter(IntSupplier) except the memory must also not be pinned.

Parameters
   logic—The object whose get method will be executed.

Returns
   a result from the computation.

joinAndEnter(LongSupplier)

Signature
   public long
   joinAndEnter(LongSupplier logic)

Description
   Same as ScopedMemory.joinAndEnter(LongSupplier) except the memory must also not be pinned.

Parameters
   logic—The object whose get method will be executed.

Returns
   a result from the computation.

joinAndEnter(DoubleSupplier)

Signature
   public double
   joinAndEnter(DoubleSupplier logic)

Description
   Same as ScopedMemory.joinAndEnter(DoubleSupplier) except the memory must also not be pinned.
Parameters

logic—The object whose get method will be executed.

Returns

a result from the computation.

joinAndEnter(HighResolutionTime)

Signature

public void
joinAndEnter(javax.realtime.HighResolutionTime<?> time)
throws InterruptedException

Description

Same as ScopedMemory.joinAndEnter(HighResolutionTime) except the memory must also not be pinned.

Parameters

time—The time that bounds the wait.

Throws

javax.realtime.ThrowBoundaryError—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 javax.realtime.IllegalAssignmentError, so the JVM cannot be permitted to deliver the exception. The javax.realtime.ThrowBoundaryError is allocated in the current allocation context and contains information about the exception it replaces.

java.lang.InterruptedException—when this schedulable is interrupted by javax.realtime.RealtimeThread.interrupt() or javax.realtime.control.AsynchronouslyInterruptedException.fire() while waiting for the reference count to go to zero.

javax.realtime.IllegalTaskStateException—when the execution context is not an instance of javax.realtime.Schedulable or when this method is invoked during finalization of objects in scoped memory and entering this scoped memory area would force deletion of the task that triggered finalization. This would include the scope containing the task, and the scope (if any) containing the scope containing the task.

javax.realtime.ScopedCycleException—when the execution context is a schedulable and this invocation would break the single parent rule.
javax.realtime.IllegalRealtimeArgumentException—when the execution context is a schedulable, and time is null or no non-null logic value was supplied to the memory area’s constructor.

javax.realtime.UnsupportedRealtimeOperationException—when the wait operation is not supported using the clock associated with time.

javax.realtime.MemoryAccessError—when calling schedulable may not use the heap and this memory area’s logic value is allocated in heap memory.

**joinAndEnter(Runnable, HighResolutionTime)**

*Signature*

```java
public void joinAndEnter(Runnable logic,
                          javax.realtime.HighResolutionTime<?> time)
  throws InterruptedException
```

*Description*

Same as ScopedMemory.joinAndEnter(Runnable, HighResolutionTime) except the memory must also not be pinned.

*Parameters*

- **logic**—The Runnable object which contains the code to execute.
- **time**—The time that bounds the wait.

*Throws*

- java.lang.InterruptedException—when this schedulable is interrupted by javax.realtime.RealtimeThread.interrupt() or javax.realtime.control.AsynchronouslyInterruptedException.fire() while waiting for the reference count to go to zero.

- IllegalTaskStateException—when the execution context is not an instance of javax.realtime.Schedulable or when this method is invoked during finalization of objects in scoped memory and entering this scoped memory area would force deletion of the task that triggered finalization. This would include the scope containing the task, and the scope (if any) containing the scope containing the task.

- ThrowBoundaryError—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 a javax.realtime.IllegalAssignmentError, so the JVM cannot be permitted to deliver the exception. The javax.realtime.ThrowBoundaryError is preallocated and saves information about the exception it replaces.
javax.realtime.ScopedCycleException—when the execution context is a sched-
ulable and this invocation would break the single parent rule.
javax.realtime.IllegalRealtimeArgumentException—when the execution con-
text is a schedulable and time or logic is null.
javax.realtime.UnsupportedRealtimeOperationException—when the wait op-
eration is not supported using the clock associated with time.

joinAndEnter(Supplier, HighResolutionTime)

Signature
public P
joinAndEnter(java.util.function.Supplier<P> logic,
javax.realtime.HighResolutionTime<?> time)

Description
Same as ScopedMemory.joinAndEnter(Supplier, HighResolutionTime) ex-
cept the memory must also not be pinned.

Parameters
logic—The object whose get method will be executed.

Returns
a result from the computation.

joinAndEnter(BooleanSupplier, HighResolutionTime)

Signature
public boolean
joinAndEnter(BooleanSupplier logic,
javax.realtime.HighResolutionTime<?> time)

Description
Same as ScopedMemory.joinAndEnter(BooleanSupplier, HighResolution-
Time) except the memory must also not be pinned.

Parameters
logic—The object whose get method will be executed.

Returns
a result from the computation.
joinAndEnter(IntSupplier, HighResolutionTime)

Signature
public int
  joinAndEnter(IntSupplier logic,
               javax.realtime.HighResolutionTime<?> time)

Description
Same as ScopedMemory.joinAndEnter(IntSupplier, HighResolutionTime)
except the memory must also not be pinned.

Parameters
logic—The object whose get method will be executed.

Returns
a result from the computation.

joinAndEnter(LongSupplier, HighResolutionTime)

Signature
public long
  joinAndEnter(LongSupplier logic,
               javax.realtime.HighResolutionTime<?> time)

Description
Same as ScopedMemory.joinAndEnter(LongSupplier, HighResolutionTime)
except the memory must also not be pinned.

Parameters
logic—The object whose get method will be executed.

Returns
a result from the computation.

joinAndEnter(DoubleSupplier, HighResolutionTime)

Signature
public double
  joinAndEnter(DoubleSupplier logic,
               javax.realtime.HighResolutionTime<?> time)

Description

526
Same as `ScopedMemory.joinAndEnter(DoubleSupplier, HighResolution-Time)` except that the area must be pinned so the memory has not been cleared.

**Parameters**
logic—The object whose get method will be executed.

**Returns**
a result from the computation.

### joinPinned

**Signature**

```java
public void joinPinned()
throws InterruptedException
```

**Description**
Same as `ScopedMemory.join()` except that the area must be pinned so the memory has not been cleared.

**Throws**

`java.lang.InterruptedIOException`—when this schedulable is interrupted by `javax.realtime.RealtimeThread.interrupt()` or `javax.realtime.control.AsynchronouslyInterruptedException.fire()` while waiting for the reference count to go to zero.

### joinPinned(HighResolutionTime)

**Signature**

```java
public void joinPinned(javax.realtime.HighResolutionTime<?> limit)
throws InterruptedException
```

**Description**
Same as `ScopedMemory.join(HighResolutionTime)` except that the area must be pinned so the memory has not been cleared.

**Parameters**

limit—The maximum time to wait.

**Throws**
java.lang.InterruptedException—when this schedulable is interrupted by javax.realtime.RealtimeThread.interrupt() or javax.realtime.control.AsynchronouslyInterruptedException.fire() while waiting for the reference count to go to zero.

joinAndEnterPinned

Signature
public void
joinAndEnterPinned()
throws InterruptedException,
IllegalTaskStateException,
ThrowBoundaryError,
ScopedCycleException,
MemoryAccessError

Description
Same as ScopedMemory.joinAndEnter() except that the area must be pinned so the memory has not been cleared.

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 javax.realtime.IllegalAssignmentError, so the JVM cannot be permitted to deliver the exception. The javax.realtime.ThrowBoundaryError 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.

java.lang.InterruptedException—when this schedulable is interrupted by javax.realtime.RealtimeThread.interrupt() or javax.realtime.control.AsynchronouslyInterruptedException.fire() while waiting for the reference count to go to zero.

IllegalTaskStateException—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.
MemoryAccessError—when calling schedulable may not use the heap and this memory area’s logic value is allocated in heap memory.

joinAndEnterPinned(Runnable)

Signature
public void
joinAndEnterPinned(Runnable logic)
throws InterruptedException,
ScopedCycleException

Description
Same as ScopedMemory.joinAndEnter(Runnable) except that the area must be pinned so the memory has not been cleared.

Parameters
logic—The logic to execute upon entry.

Throws
java.lang.InterruptedException—when this schedulable is interrupted by javax.realtime.RealtimeThread.interrupt() or javax.realtime.control.AsynchronouslyInterruptedException.fire() while waiting for the reference count to go to zero.
ScopedCycleException—when the caller is a schedulable and this invocation would break the single parent rule.

joinAndEnterPinned(Supplier)

Signature
public T
joinAndEnterPinned(java.util.function.Supplier<T> logic)

Description
Same as joinAndEnterPinned(Runnable) except that the executed method is called get and an object is returned.

Parameters
logic—The object whose get method will be executed.

Returns
a result from the computation.
joinAndEnterPinned(BooleanSupplier)

Signature

    public boolean
    joinAndEnterPinned(BooleanSupplier logic)

Description

    Same as joinAndEnterPinned(Runnable) except that the executed method is called get and a boolean is returned.

Parameters

    logic—The object whose get method will be executed.

Returns

    a result from the computation.

joinAndEnterPinned(IntSupplier)

Signature

    public int
    joinAndEnterPinned(IntSupplier logic)

Description

    Same as joinAndEnterPinned(Runnable) except that the executed method is called get and an int is returned.

Parameters

    logic—The object whose get method will be executed.

Returns

    a result from the computation.

joinAndEnterPinned(LongSupplier)

Signature

    public long
    joinAndEnterPinned(LongSupplier logic)

Description

    Same as joinAndEnterPinned(Runnable) except that the executed method is called get and a long is returned.
Parameters
logic—The object whose get method will be executed.

Returns
a result from the computation.

Available since RTSJ 2.0

joinAndEnterPinned(DoubleSupplier)

Signature
public double
joinAndEnterPinned(DoubleSupplier logic)

Description
Same as joinAndEnterPinned(Runnable) except that the executed method is called get and a double is returned.

Parameters
logic—The object whose get method will be executed.

Returns
a result from the computation.

joinAndEnterPinned(HighResolutionTime)

Signature
public void
joinAndEnterPinned(javax.realtime.HighResolutionTime<?> limit)
throws InterruptedException,
IllegalStateException,
IllegalRealtimeArgumentException,
UnsupportedRealtimeOperationException,
ThrowBoundaryError,
ScopedCycleException,
MemoryAccessError

Description
Same as ScopedMemory.joinAndEnter(HighResolutionTime) except that pinning is ignored so the memory may not have been cleared.

Parameters
11 Alternative Memory Areas

PinnableMemory

limit—The maximum time to 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 javax.realtime.IllegalAssignmentError, so the JVM cannot be permitted to deliver the exception. The javax.realtime.ThrowBoundaryError 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.

java.lang.InterruptedException—when this schedulable is interrupted by javax.realtime.RealtimeThread.interrupt() or javax.realtime.control.AsynchronouslyInterruptedException.fire() while waiting for the reference count to go to zero.

IllegalTaskStateException—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.

javax.realtime.IllegalRealtimeArgumentException—when the caller is a schedulable, and time is null or no non-null logic value was supplied to the memory area’s constructor.

MemoryAccessError—when calling schedulable may not use the heap and this memory area’s logic value is allocated in heap memory.

javax.realtime.UnsupportedRealtimeOperationException—when the wait operation is not supported using the clock associated with time.

joinAndEnterPinned(Runnable, HighResolutionTime)

Signature

public void
joinAndEnterPinned(Runnable logic,
javax.realtime.HighResolutionTime<?> limit)
throws InterruptedException,
ScopedCycleException
Same as `ScopedMemory.joinAndEnter(Runnable, HighResolutionTime)` except that pinning is ignored so the memory may not have been cleared.

**Parameters**

- `logic`—The logic to execute upon entry.
- `limit`—The maximum time to wait.

**Throws**

- `java.lang.InterruptedException`—when this schedulable is interrupted by `javax.realtime.RealtimeThread.interrupt()` or `javax.realtime.control.AsynchronouslyInterruptedException.fire()` while waiting for the reference count to go to zero.
- `ScopedCycleException`—when the caller is a schedulable and this invocation would break the single parent rule.

### joinAndEnterPinned(Supplier, HighResolutionTime)

**Signature**

```java
class PinnableMemory
{
  public P
      joinAndEnterPinned(java.util.function.Supplier<P> logic,
                      javax.realtime.HighResolutionTime<?> time)
}
```

**Description**

Same as `joinAndEnterPinned(Runnable, HighResolutionTime)` except that the executed method is called `get` and an object is returned.

**Parameters**

- `logic`—The object whose get method will be executed.

**Returns**

A result from the computation.

### joinAndEnterPinned(BooleanSupplier, HighResolutionTime)

**Signature**

```java
class PinnableMemory
{
  public boolean
      joinAndEnterPinned(BooleanSupplier logic,
                         javax.realtime.HighResolutionTime<?> time)
}
```

**Description**

RTSJ 2.0 (Draft 66)
11 Alternative Memory Areas

PinnableMemory

Same as `joinAndEnterPinned(Runnable, HighResolutionTime)` except that the executed method is called `get` and a `boolean` is returned.

Parameters
logic—The object whose `get` method will be executed.

Returns
a result from the computation.

`joinAndEnterPinned(IntSupplier, HighResolutionTime)`

Signature
public int
joinAndEnterPinned(IntSupplier logic,
javax.realtime.HighResolutionTime<??> time)

Description
Same as `joinAndEnterPinned(Runnable, HighResolutionTime)` except that the executed method is called `get` and an `int` is returned.

Parameters
logic—The object whose `get` method will be executed.

Returns
a result from the computation.

`joinAndEnterPinned(LongSupplier, HighResolutionTime)`

Signature
public long
joinAndEnterPinned(LongSupplier logic,
javax.realtime.HighResolutionTime<??> time)

Description
Same as `joinAndEnterPinned(Runnable, HighResolutionTime)` except that the executed method is called `get` and a `long` is returned.

Parameters
logic—The object whose `get` method will be executed.

Returns
a result from the computation.
joinAndEnterPinned(DoubleSupplier, HighResolutionTime)

Signature

```
public double joinAndEnterPinned(DoubleSupplier logic,
                                javax.realtime.HighResolutionTime<?> time)
```

Description
Same as `joinAndEnterPinned(Runnable, HighResolutionTime)` except that the executed method is called `get` and a `double` is returned.

Parameters

- `logic`—The object whose `get` method will be executed.

Returns
a result from the computation.

11.4.3.7 ScopedConfigurationParameters

Public class `ScopedConfigurationParameters`

Inheritance

```
java.lang.Object
    javax.realtime.ConfigurationParameters
        ScopedConfigurationParameters
```

Description
This is the same as `javax.realtime.ConfigurationParameters` except an instance of `javax.realtime.BoundSchedulable` that uses this parameters object may not access the heap and one that uses the super type may.

11.4.3.7.1 Constructors

ScopedConfigurationParameters(int, int, long)

Signature
**11 Alternative Memory Areas**

**ScopedConfigurationParameters**

```java
public ScopedConfigurationParameters(int messageLength,
                                      int stackTraceLength,
                                      long[] sizes)
    throws IllegalRealtimeStateException
```

**Description**

Similar to `javax.realtime.ConfigurationParameters. ConfigurationParameters(int, int, long[])`, except the receiver may not use the heap.

**Throws**

`IllegalRealtimeStateException`—when the current memory context is a heap memory.

### 11.4.3.7.2 Methods

**getDefaultRunner**

**Signature**

```java
public static javax.realtime.ReleaseRunner
    getDefaultRunner()
```

**Description**

Gets the system default release runner.

**Returns**

a general runner to be used when none is set.

**setDefaultRunner(ReleaseRunner)**

**Signature**

```java
public static void
    setDefaultRunner(ReleaseRunner runner)
    throws IllegalRealtimeArgumentException
```

**Description**

Sets the system default release runner.
Parameters
runner—The runner to be used when none is set. When null, the default release runner is set to the original system default.

Throws
IllegalArgumentException—when runner is allocated in a heap memory or its javax.realtime.ConfigurationParameters are not ScopedConfigurationParameters.

11.4.3.8 ScopedMemory

public abstract class ScopedMemory

Inheritance
java.lang.Object
    javax.realtime.MemoryArea
        ScopedMemory

Description
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 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 (its 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 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 an 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 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. since RTSJ 2.0, moved from javax.realtime.

11.4.3.8.1 Methods

globalBackingStoreSize

Signature
public static long
globalBackingStoreSize()

Description
Determines the total amount of memory in the global backing store.

Returns
the total amount of global backing store in bytes.

Available since RTSJ 2.0

globalBackingStoreRemaining

Signature
public static long
globalBackingStoreRemaining()

Description

538 RTSJ 2.0 (Draft 66)
Determined the amount of memory remaining for allocation to new scoped memories in the backing store of this scoped memory.

**Returns**
the amount of global backing store remaining in bytes.

**Available since** RTSJ 2.0

**globalBackingStoreConsumed**

**Signature**
public static long
globalBackingStoreConsumed()

**Description**
Determines the amount of memory consumed by existing scoped memories from the global backing store.

**Returns**
the amount of backing store available in bytes.

**Available since** RTSJ 2.0

**visitScopeRoots(Consumer)**

**Signature**
public static void
visitScopeRoots(java.util.function.Consumer<ScopedMemory> visitor)
throws IllegalRealtimeArgumentException,
ForEachTerminationException

**Description**
A means of accessing all live scoped memories whose parent is a primordial memory area, even those to which no reference exists, such a [javax.realtime.memory.PinnableMemory](#) that is pinned or another [javax.realtime.memoryScopedMemory](#) 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 guaranteed 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.

The visitor’s `accept` method is called on all live roots scopes, so long as the `visitor` does not throw `javax.realtime.ForEachTerminationException`. When that is thrown, the visit terminates. A closure could be used to capture the last element visited.

When execution of the visitor’s `accept` method is terminated abruptly by throwing an exception, then execution of `visitScopedChildren` also terminates abruptly by throwing the same exception.

**Parameters**

`visitor`—Determines the action to be performed on each of the children scopes.

**Throws**

`javax.realtime.IllegalRealtimeArgumentException`—when visitor is `null`.  
`javax.realtime.ForEachTerminationException`—when the traversal ends prematurely.  
`javax.realtime.RealtimeSecurityException`—when the application does not have permissions to access visit root scopes.

**Available since** RTSJ 2.0

### enter

**Signature**

```java
public void enter()
throws ScopedCycleException,
        ThrowBoundaryError,
        IllegalTaskStateException,
        IllegalRealtimeArgumentException,
        MemoryAccessError
```

**Description**

Associates 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`) or the `enter` method exits.

**Throws**

RTSJ 2.0 (Draft 66)
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 javax.realtime.IllegalAssignmentError, so the JVM cannot be permitted to deliver the exception. The javax.realtime.ThrowBoundaryError is allocated in the current allocation context and contains information about the exception it replaces.

IllegalTaskStateException—when the execution context is not an instance of javax.realtime.Schedulable or when this method is invoked during finalization of objects in scoped memory and entering this scoped memory area would force deletion of the execution context that triggered finalization. This would include the scope containing the execution context, and the scope (if any) containing the scope containing execution context.

javax.realtime.IllegalRealtimeArgumentException—null

MemoryAccessException—when caller is a schedulable that may not use the heap and this memory area’s logic value is allocated in heap memory.

**enter(Runnable)**

*Signature*

```java
public void enter(Runnable logic)
throws ScopedCycleException,
ThrowBoundaryError,
IllegalTaskStateException,
IllegalRealtimeArgumentException
```

*Description*

Associates 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`) or the `enter` method exits.

*Parameters*

- `logic`—The Runnable object whose `run()` method should be invoked.

*Throws*

- ScopedCycleException—when this invocation would break the single parent rule.
**ScopedMemory**

**ThrowBoundaryError**—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 `javax.realtime.IllegalAssignmentError`, so the JVM cannot be permitted to deliver the exception. The `javax.realtime.ThrowBoundaryError` is allocated in the current allocation context and contains information about the exception it replaces.

**IllegalTaskStateException**—when the execution context is not an instance of `javax.realtime.Schedulable` or when this method is invoked during finalization of objects in scoped memory and entering this scoped memory area would force deletion of the task that triggered finalization. This would include the scope containing the task, and the scope (if any) containing the scope containing task.

**javax.realtime.IllegalRealtimeArgumentException**—null

### enter(Supplier)

**Signature**

```java
public T enter(java.util.function.Supplier<T> logic)
throws ScopedCycleException,
    ThrowBoundaryError,
    IllegalTaskStateException,
    IllegalRealtimeArgumentException
```

**Description**

Same as `enter(Runnable)` except that the executed method is called `get` and an object is returned.

**Parameters**

- `logic`—The object whose `get` method will be executed.

**Returns**

A result from the computation.

### enter(BooleanSupplier)

**Signature**

```java
public boolean enter(BooleanSupplier logic)
```

542 RTSJ 2.0 (Draft 66)
ScopedMemory

javax.realtime.memory 11.4

throws ScopedCycleException,
   ThrowBoundaryError,
   IllegalTaskStateException,
   IllegalRealtimeArgumentException

Description
Same as enter(Runnable) except that the executed method is called get and a boolean is returned.

Parameters
logic—the object whose get method will be executed.

Returns
a result from the computation.

enter(IntSupplier)

Signature
public int
   enter(IntSupplier logic)
   throws ScopedCycleException,
      ThrowBoundaryError,
      IllegalTaskStateException,
      IllegalRealtimeArgumentException

Description
Same as enter(Runnable) except that the executed method is called get and an int is returned.

Parameters
logic—the object whose get method will be executed.

Returns
a result from the computation.

enter(LongSupplier)

Signature
public long
   enter(LongSupplier logic)
11 Alternative Memory Areas

ScopedMemory

throws ScopedCycleException,
ThrowBoundaryError,
IllegalTaskStateException,
IllegalRealtimeArgumentException

Description
Same as enter(Runnable) except that the executed method is called get and a long is returned.

Parameters
logic—the object whose get method will be executed.

Returns
a result from the computation.

enter(DoubleSupplier)

Signature
public double
enter(DoubleSupplier logic)
throws ScopedCycleException,
ThrowBoundaryError,
IllegalTaskStateException,
IllegalRealtimeArgumentException

Description
Same as enter(Runnable) except that the executed method is called get and a double is returned.

Parameters
logic—the object whose get method will be executed.

Returns
a result from the computation.

executeInArea(Runnable)

Signature
public void
executeInArea(Runnable logic)
ScopedMemory

java.realtime.memory 11.4

```java
throws IllegalTaskStateException,
    IllegalRealtimeArgumentException,
    InaccessibleAreaException
```

**Description**

Executes 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.

**Parameters**

logic—The runnable object whose run() method should be executed.

**Throws**

- **IllegalTaskStateException**—when the execution context is not an instance of javax.realtime.Schedulable.
- **InaccessibleAreaException**—when the memory area is not in the schedulable’s scope stack.
- **javax.realtime.IllegalRealtimeArgumentException**—when the execution context is an instance of javax.realtime.Schedulable schedulable and logic is null.

**executeInArea(Supplier)**

**Signature**

```java
public T executeInArea(java.util.function.Supplier<T> logic)
```

**Description**

Same as `executeInArea(Runnable)` except that the executed method is called get and an object is returned.

**Parameters**

logic—the object whose get method will be executed.

**Returns**

a result from the computation.
executeInArea(BooleanSupplier)

Signature
public boolean executeInArea(BooleanSupplier logic)
throws IllegalTaskStateException,
IllegalRealtimeArgumentException,
InaccessibleAreaException

Description
Same as executeInArea(Runnable) except that the executed method is called get and a boolean is returned.

Parameters
logic—the object whose get method will be executed.

Returns
a result from the computation.

executeInArea(IntSupplier)

Signature
public int executeInArea(IntSupplier logic)
throws IllegalTaskStateException,
IllegalRealtimeArgumentException,
InaccessibleAreaException

Description
Same as executeInArea(Runnable) except that the executed method is called get and an int is returned.

Parameters
logic—the object whose get method will be executed.

Returns
a result from the computation.

executeInArea(LongSupplier)

Signature

546 RTSJ 2.0 (Draft 66)
public long executeInArea(LongSupplier logic)
throws IllegalTaskStateException,
   IllegalRealtimeArgumentException,
   InaccessibleAreaException

Description
Same as `executeInArea(Runnable)` except that the executed method is called get and a long is returned.

Parameters
logic—the object whose get method will be executed.

Returns
a result from the computation.

`executeInArea(DoubleSupplier)`

Signature
public double executeInArea(DoubleSupplier logic)
throws IllegalTaskStateException,
   IllegalRealtimeArgumentException,
   InaccessibleAreaException

Description
Same as `executeInArea(Runnable)` except that the executed method is called get and a double is returned.

Parameters
logic—the object whose get method will be executed.

Returns
a result from the computation.

`getPortal`

Signature
public java.lang.Object getPortal()
11 Alternative Memory Areas

ScopedMemory

throws IllegalAssignmentError, IllegalTaskStateException

Description

Returns 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.

Throws

javax.realtime.IllegalAssignmentError—when a reference to the portal object cannot be stored in the caller’s allocation context; that is, when the object is allocated in a more deeply nested scoped memory than the current allocation context or not on the caller’s scope stack.

IllegalTaskStateException—when the execution context is not an instance of javax.realtime.Schedulable.

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.

ggetReferenceCount

Signature

public int
ggetReferenceCount()

Description

Returns the reference count of this ScopedMemory.

Note that a reference count of zero reliably means that the scope is not referenced, but other reference counts are subject to artifacts of lazy/eager maintenance by the implementation.

Returns

the reference count of this ScopedMemory.

join

Signature

548 RTSJ 2.0 (Draft 66)
public void
join()
throws InterruptedException

Description
Waits until the reference count of this ScopedMemory goes down to zero. Returns immediately when the memory is unreferenced.

Throws
java.lang.InterruptedException—when this schedulable is interrupted by javax.realtime.RealtimeThread.interrupt() or javax.realtime.control.AsynchronouslyInterruptedException.fire() while waiting for the reference count to go to zero.
IllegalStateException—when the execution context is not an instance of javax.realtime.Schedulable.

join(HighResolutionTime)

Signature
public void
join(javax.realtime.HighResolutionTime<?> time)
throws InterruptedException

Description
Waits at most until the time designated by the time parameter for the reference count of this ScopedMemory to drop to zero. Returns immediately when the memory area is unreferenced.

Since the time is expressed as a javax.realtime.HighResolutionTime, 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 join returns immediately.

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**

`java.lang.InterruptedException`—when this schedulable is interrupted by `javax.realtime.RealtimeThread.interrupt()` or `javax.realtime.control.AsynchronouslyInterruptedException.fire()` while waiting for the reference count to go to zero.

`IllegalTaskStateException`—when the execution context is not an instance of `javax.realtime.Schedulable`.

`javax.realtime.IllegalRealtimeArgumentException`—when the execution context is a schedulable and `time` is null.

`UnsupportedRealtimeOperationException`—when the wait operation is not supported using the clock associated with `time`.

### `joinAndEnter`

**Signature**

```java
public void joinAndEnter() throws InterruptedException
```

**Description**

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 enters the `ScopedMemory` and executes 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 `IllegalRealtimeArgumentException` 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.

**Throws**

`java.lang.InterruptedException`—when this schedulable is interrupted by `javax.realtime.RealtimeThread.interrupt()` or `javax.realtime.
control.AsynchronouslyInterruptedException.fire() while waiting for
the reference count to go to zero.

IllegalTaskStateException—when the execution context is not an instance of
javax.realtime.Schedulable or when this method is invoked during final-
ization of objects in scoped memory and entering this scoped memory area
would force deletion of the task that triggered finalization. This would include
the scope containing the task, and the scope (if any) containing the scope
containing the task.

ThrowBoundaryError—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 javax.realtime.IllegalAssignmentError, so
the JVM cannot be permitted to deliver the exception. The javax.realtime.
ThrowBoundaryError is allocated in the current allocation context and contains
information about the exception it replaces.

javax.realtime.ScopedCycleException—when this invocation would break the
single parent rule.

javax.realtime.IllegalRealtimeArgumentException—when the execution con-
text 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(HighResolutionTime)

Signature

public void
joinAndEnter(javax.realtime.HighResolutionTime<?> time)
throws InterruptedException

Description

In the error-free case, joinAndEnter combines join();enter(); such that no
enter() from another schedulable can intervene between the two method invoca-
tions. 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 Runnable object passed
to the constructor. When no instance of Runnable was passed to the memory
area’s constructor, the method throws IllegalRealtimeArgumentException
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 `javax.realtime.HighResolutionTime`, 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`.

Note that expiration of `time` may cause control to enter the memory area before its reference count has gone to zero.

**Parameters**

- `time`—The time that bounds the wait.

**Throws**

- `javax.realtime.ThrowBoundaryError`—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 `javax.realtime.IllegalAssignmentError`, so the JVM cannot be permitted to deliver the exception. The `javax.realtime.ThrowBoundaryError` is allocated in the current allocation context and contains information about the exception it replaces.

- `java.lang.InterruptedException`—when this schedulable is interrupted by `javax.realtime.RealtimeThread.interrupt()` or `javax.realtime.control.AsynchronouslyInterruptedException.fire()` while waiting for the reference count to go to zero.

- `javax.realtime.IllegalTaskStateException`—when the execution context is not an instance of `javax.realtime.Schedulable` or when this method is invoked during finalization of objects in scoped memory and entering this scoped memory area would force deletion of the task that triggered finalization. This would include the scope containing the task, and the scope (if any) containing the scope containing the task.

- `javax.realtime.ScopedCycleException`—when the execution context is a schedulable and this invocation would break the single parent rule.

- `javax.realtime.IllegalRealtimeArgumentException`—when the execution context is a schedulable, and `time` is `null` or no non-null `logic` value was supplied to the memory area’s constructor.

- `javax.realtime.UnsupportedRealtimeOperationException`—when the wait op-
eration is not supported using the clock associated with `time`.

`javax.realtime.MemoryAccessError`—when calling schedulable may not use the heap and this memory area’s logic value is allocated in heap memory.

**joinAndEnter(Runnable)**

**Signature**

```java
public void joinAndEnter(Runnable logic)
throws InterruptedException
```

**Description**

In the error-free case, `joinAndEnter` combines `join();` and `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`, the method throws `IllegalRealtimeArgumentException` 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.

**Parameters**

- `logic`—The `Runnable` object which contains the code to execute.

**Throws**

- `java.lang.InterruptedException`—when this schedulable is interrupted by `javax.realtime.RealtimeThread.interrupt()` or `javax.realtime.control.AsynchronouslyInterruptedException.fire()` while waiting for the reference count to go to zero.

- `IllegalTaskStateException`—when the execution context is not an instance of `javax.realtime.Schedulable` or when this method is invoked during finalization of objects in scoped memory and entering this scoped memory area would force deletion of the task that triggered finalization. This would include the scope containing the task, and the scope (if any) containing the scope containing the task.
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 `javax.realtime.IllegalAssignmentError`, so the JVM cannot be permitted to deliver the exception. The `javax.realtime.ThrowBoundaryError` is allocated in the current allocation context and contains information about the exception it replaces.

`javax.realtime.ScopedCycleException`—when this invocation would break the single parent rule.

`javax.realtime.IllegalRealtimeArgumentException`—when the execution context is a schedulable and `logic` is null.

### joinAndEnter(Supplier)

**Signature**

```
public T joinAndEnter(java.util.function.Supplier<T> logic)
```

**Description**

Same as `joinAndEnter(Runnable)` except that the executed method is called `get` and an object is returned.

**Parameters**

- `logic`—The object whose `get` method will be executed.

**Returns**

- a result from the computation.

**Available since** RTSJ 2.0

### joinAndEnter(BooleanSupplier)

**Signature**

```
public boolean joinAndEnter(BooleanSupplier logic)
```

**Description**

Same as `joinAndEnter(Runnable)` except that the executed method is called `get` and a `boolean` is returned.

**Parameters**

- `logic`—The object whose `get` method will be executed.

**Available since** RTSJ 2.0
logic—The object whose get method will be executed.

Returns
a result from the computation.

Available since RTSJ 2.0

joinAndEnter(IntSupplier)

Signature
public int
joinAndEnter(IntSupplier logic)

Description
Same as joinAndEnter(Runnable) except that the executed method is called get and an int is returned.

Parameters
logic—The object whose get method will be executed.

Returns
a result from the computation.

Available since RTSJ 2.0

joinAndEnter(LongSupplier)

Signature
public long
joinAndEnter(LongSupplier logic)

Description
Same as joinAndEnter(Runnable) except that the executed method is called get and a long is returned.

Parameters
logic—The object whose get method will be executed.

Returns
a result from the computation.

Available since RTSJ 2.0
**joinAndEnter(DoubleSupplier)**

*Signature*

```java
class ScopedMemory {
    public double joinAndEnter(DoubleSupplier logic)
}
```

*Description*

Same as `joinAndEnter(Runnable)` except that the executed method is called `get` and a `double` is returned.

*Parameters*

- `logic` — The object whose `get` method will be executed.

*Returns*

- a result from the computation.

*Available since* RTSJ 2.0

---

**joinAndEnter(Runnable, HighResolutionTime)**

*Signature*

```java
class ScopedMemory {
    public void joinAndEnter(Runnable logic,
                             javax.realtime.HighResolutionTime<?> time)
                                      throws InterruptedException
}
```

*Description*

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 `javax.realtime.HighResolutionTime`, 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)`.

---

556 RTSJ 2.0 (Draft 66)
The method throws `IllegalRealtimeArgumentException` 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.

**Parameters**
- `logic`—The `Runnable` object which contains the code to execute.
- `time`—The time that bounds the wait.

**Throws**
- `java.lang.InterruptedException`—when this schedulable is interrupted by `javax.realtime.RealtimeThread.interrupt()` or `javax.realtime.control.AsynchronouslyInterruptedException.fire()` while waiting for the reference count to go to zero.
- `IllegalTaskStateException`—when the execution context is not an instance of `javax.realtime.Schedulable` or when this method is invoked during finalization of objects in scoped memory and entering this scoped memory area would force deletion of the task that triggered finalization. This would include the scope containing the task, and the scope (if any) containing the scope containing the task.
- `ThrowBoundaryError`—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 a `javax.realtime.IllegalAssignmentError`, so the JVM cannot be permitted to deliver the exception. The `javax.realtime.ThrowBoundaryError` is preallocated and saves information about the exception it replaces.
- `javax.realtime.ScopedCycleException`—when the execution context is a schedulable and this invocation would break the single parent rule.
- `javax.realtime.IllegalRealtimeArgumentException`—when the execution context is a schedulable and `time` or `logic` is null.
- `javax.realtime.UnsupportedRealtimeOperationException`—when the wait operation is not supported using the clock associated with `time`.

**joinAndEnter(Supplier, HighResolutionTime)**

**Signature**

```java
public P
```

RTSJ 2.0 (Draft 66) 557
joinAndEnter(java.util.function.Supplier<P> logic,
            javax.realtime.HighResolutionTime<?> time)

Description
Same as `joinAndEnter(Runnable, HighResolutionTime)` except that the executed method is called `get` and an object is returned.

Parameters
- `logic`—The object whose `get` method will be executed.

Returns
- a result from the computation.

Available since RTSJ 2.0

joinAndEnter(BooleanSupplier, HighResolutionTime)

Signature
- \text{public boolean}
- `joinAndEnter(BooleanSupplier logic,
                 javax.realtime.HighResolutionTime<?> time)`

Description
Same as `joinAndEnter(Runnable, HighResolutionTime)` except that the executed method is called `get` and a boolean is returned.

Parameters
- `logic`—The object whose `get` method will be executed.

Returns
- a result from the computation.

Available since RTSJ 2.0

joinAndEnter(IntSupplier, HighResolutionTime)

Signature
- \text{public int}
- `joinAndEnter(IntSupplier logic,
                 javax.realtime.HighResolutionTime<?> time)`

Description

RTSJ 2.0 (Draft 66)
Same as `joinAndEnter(Runnable, HighResolutionTime)` except that the executed method is called `get` and an `int` is returned.

**Parameters**

- `logic`—The object whose get method will be executed.

**Returns**

- a result from the computation.

**Available since** RTSJ 2.0

\textbf{joinAndEnter(LongSupplier, HighResolutionTime)}

**Signature**

```
public long joinAndEnter(LongSupplier logic,
                          javax.realtime.HighResolutionTime<?> time)
```

**Description**

Same as `joinAndEnter(Runnable, HighResolutionTime)` except that the executed method is called `get` and a `long` is returned.

**Parameters**

- `logic`—The object whose get method will be executed.

**Returns**

- a result from the computation.

**Available since** RTSJ 2.0

\textbf{joinAndEnter(DoubleSupplier, HighResolutionTime)}

**Signature**

```
public double joinAndEnter(DoubleSupplier logic,
                          javax.realtime.HighResolutionTime<?> time)
```

**Description**

Same as `joinAndEnter(Runnable, HighResolutionTime)` except that the executed method is called `get` and a `double` is returned.

**Parameters**
logic—The object whose get method will be executed.

Returns
a result from the computation.

Available since RTSJ 2.0

visitNestedScopes(Consumer)

Signature

public void
visitNestedScopes(java.util.function.Consumer<ScopedMemory> visitor)
throws IllegalRealtimeArgumentException,
     ForEachTerminationException

Description
A means of accessing all live nested scoped memories parented in this scoped
memory, even those to which no reference exists, such as ajavax.realtime.
memory.PinnableMemory that is pinned or anotherjavax.realtime.memory.
ScopedMemory that contains a Schedulable. It has the same semantics as the
method visitScopeRoots(Consumer), except for the following:
• what scoped memories are visited,
• the memory area must be reachable from the current scope stack, and
• there is not security manager check.

Parameters

visitor—Determines the action to be performed on each of the children scopes.

Throws
javax.realtime.IllegalRealtimeArgumentException—when visitor is null.
ForEachTerminationException—when the traversal ends prematurely.

Available since RTSJ 2.0

newArray(Class, int)

Signature

public java.lang.Object
newArray(java.lang.Class<?> type,
        int number)

Description

RTSJ 2.0 (Draft 66)
Allocates an array of the given type in this memory area. This method may be concurrently used by multiple threads.

**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**

- `javax.realtime.IllegalRealtimeArgumentException`—null
- `javax.realtime.OutOfMemoryStaticError`—null
- `IllegalTaskStateException`—when the execution context is not an instance of `javax.realtime.Schedulable`.
- `InaccessibleAreaException`—when the memory area is not in the schedulable’s scope stack.

**Returns**
a new array of class type, of number elements.

**newInstance(Class)**

**Signature**

```java
public T
newInstance(java.lang.Class<T> type)
throws IllegalAccessException,
IllegalRealtimeArgumentException,
ExceptionInInitializerError,
OutOfMemoryStaticError,
InstantiationException,
IllegalTaskStateException,
InaccessibleAreaException
```

**Description**

Allocates an object in this memory area. This method may be concurrently used by multiple threads.

**Parameters**

type—The class of which to create a new instance.

**Throws**

- `IllegalAccessException`—The class or initializer is inaccessible.
11 Alternative Memory Areas

**ScopedMemory**

```java
javax.realtime.IllegalRealtimeArgumentException—null
ExceptionInInitializerError—when an unexpected exception has occurred in
  a static initializer.
javax.realtime.OutOfMemoryStaticError—null
InstantiationException—when the specified class object could not be instanti-
  ated. Possible causes are it is an interface, it is abstract, or it is an array.
IllegalTaskStateException—when the execution context is not an instance of
  javax.realtime.Schedulable.
InaccessibleAreaException—when the memory area is not in the schedulable’s
  scope stack.
```

**Returns**

a new instance of class type.

### newInstance(Constructor, Object)

**Signature**

```java
public T
newInstance(java.lang.reflect.Constructor<T> c,
  java.lang.Object[] args)
  throws IllegalAccessException,
    InstantiationException,
    InvocationTargetException
```

**Description**

Allocates an object in this memory area. This method may be concurrently used
by multiple threads.

**Parameters**

- `c`—T The constructor for the new instance.
- `args`—An array of arguments to pass to the constructor.

**Throws**

- `IllegalAccessException`—when the class or initializer is inaccessible under Java
  access control.
- `InstantiationException`—when the specified class object could not be instanti-
  ated. Possible causes are it is an interface, it is abstract, or it is an array.
- `javax.realtime.OutOfMemoryStaticError—null`
- `javax.realtime.IllegalRealtimeArgumentException—null`

562   RTSJ 2.0 (Draft 66)
IllegalTaskStateException—when the execution context is not an instance of javax.realtime.Schedulable.
InvocationTargetException—when the underlying constructor throws an exception.
InaccessibleAreaException—when the memory area is not in the schedulable’s scope stack.

Returns
a new instance of the object constructed by c.

setPortal(Object)

Signature
public void setPortal(Object object)
throws IllegalTaskStateException, IllegalAssignmentError, InaccessibleAreaException

Description
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.

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
IllegalTaskStateException—when the execution context is not an instance of javax.realtime.Schedulable.
IllegalAssignmentError—when the execution context is an instance of javax.realtime.Schedulable, and object is not allocated in this scoped memory instance and not null.
InaccessibleAreaException—when the execution context is a schedulable, this memory area is not in the caller’s scope stack and object is not null.

toString

Signature

RTSJ 2.0 (Draft 66)
public java.lang.String toString()

Description
Returns a user-friendly representation of this ScopedMemory of the form `<classname>@<num>` where `<classname>` is the name of the class, e.g. `javax.realtime.memory.ScopedMemory`, and `<num>` is a number that uniquely identifies this scoped memory area.

Returns
the string representation

11.4.3.9 ScopedMemoryParameters

public class ScopedMemoryParameters

Inheritance
java.lang.Object
javax.realtime.MemoryParameters
ScopedMemoryParameters

Description
Extends memory parameters to provide limits for scoped memory.

See Section `javax.realtime.MemoryParameters`

Available since RTSJ 2.0

11.4.3.9.1 Constructors

ScopedMemoryParameters(long, long, long, long, long, long, long)

Signature
564 RTSJ 2.0 (Draft 66)
public ScopedMemoryParameters(long maxInitialArea,
   long maxImmortal,
   long allocationRate,
   long maxContainingArea,
   long maxInitialBackingStore,
   long maxGlobalBackingStore)

throws IllegalRealtimeArgumentException

Description

Creates a ScopedMemoryParameters instance with the given values that can allow access to any ScopedMemory

Parameters

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

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

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 UNLIMITED. 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 positive allocation rate limits as UNLIMITED.

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

maxInitialBackingStore—A limit on the amount of backing store the schedulable may allocate from backing store of its initial memory area when that memory area is an instance of StackedMemory, in bytes. When zero, no allocation is allowed in that backing store. Backing store that is returned to the area backing store is subtracted from the limit. To specify no limit, use UNLIMITED.

maxGlobalBackingStore—A limit on the amount of backing store the schedulable
may allocate from the global backing store to scoped memory areas in bytes. When zero, no allocation is allowed in the memory area. To specify no limit, use UNLIMITED.

**Throws**
javax.realtime.IllegalRealtimeArgumentException—when any value other less than zero is passed as the value of maxInitialArea, maxImmortal, allocationRate, maxBackingStore, or maxContainingArea.

### ScopedMemoryParameters(long, long, long, long)

**Signature**

```java
public ScopedMemoryParameters(long maxInitialArea,
                               long maxImmortal,
                               long maxContainingArea,
                               long maxInitialBackingStore)
throws IllegalRealtimeArgumentException
```

**Description**

Same as `ScopedMemoryParameters(maxInitialArea, maxImmortal, 0, maxContainingArea, maxInitialBackingStore, 0)`. This constructor disallows root StackedMemory, LTMemory, and Heap allocation.

### ScopedMemoryParameters(long, long, long)

**Signature**

```java
public ScopedMemoryParameters(long maxInitialArea,
                               long maxImmortal,
                               long maxContainingArea)
throws IllegalRealtimeArgumentException
```

**Description**

Same as `ScopedMemoryParameters(maxInitialArea, maxImmortal, MemoryParameters.UNLIMITED, maxGlobalBackingStore, 0, 0)`. This constructor disallows host StackedMemory and LTMemory allocation.
11.4.3.9.2 Methods

getMaxGlobalBackingStore

Signature
   public long
   getMaxGlobalBackingStore()

Description
   Determines the limit on backing store for this task from the global pool.

Returns
   the limit on backing store.

getMaxInitialBackingStore

Signature
   public long
   getMaxInitialBackingStore()

Description
   Determines the limit on backing store for this task from its parent StackedMemory.

Returns
   the limit on backing store.

getMaxContainingArea

Signature
   public long
   getMaxContainingArea()

Description
   Determines the limit on allocation in the area where the task was created.

Returns
   the limit on allocation in the area where the task was created.
11 Alternative Memory Areas

11.4.3.10 StackedMemory

public class StackedMemory

Inheritance

java.lang.Object
javax.realtime.MemoryArea
ScopedMemory
StackedMemory

Description

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 this case it is returned to its enclosing scope's backing store when the object is finalized. Implementations should 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, and
- **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 object: **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.
Creation of a StackedMemory shall fail with a javax.realtime.OutOfMemoryStaticError when the current javax.realtime.Schedulable is configured with a limit on ScopedMemoryParameters.maxGlobalBackingStore and creation of the root StackedMemory would exceed that limit.

Creation of a StackedMemory is subject to additional restrictions when the current Schedulable is configured with an explicit initial memory area of type StackedMemory. In this case, the following rules apply.

- Construction of a root StackedMemory will fail and throw a OutOfMemoryStaticError regardless of the value of the Schedulable’s ScopedMemoryParameters.maxGlobalBackingStore.
- Construction of a StackedMemory from a current allocation context that is not the Schedulable’s explicit initial memory area or one of its descendants in the scope stack will fail and throw OutOfMemoryStaticError.
- A maximum of ScopedMemoryParameters.maxInitialBackingStore bytes may be allocated directly from the backing store of the Schedulable’s explicit initial memory area over the lifetime of the Schedulable. Any operation that would exceed this limit (whether by resizing the allocation area of the explicit initial memory area or a guest area in the same backing store, or by allocating a new StackedMemory with the explicit initial memory area as the current allocation context) will fail and throw a OutOfMemoryStaticError.

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, and
- a guest StackedMemory cannot be created from a StackedMemory that currently has another child area that is also a guest StackedMemory, i.e., 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 `javax.realtime.OutOfMemoryStaticError`. This overhead must be bounded by a constant.

**Available since RTSJ 2.0**

### 11.4.3.10.1 Constructors

`StackedMemory(long, long, Runnable)`

**Signature**

```java
public StackedMemory(long scopeSize,
    long backingSize,
    Runnable logic)
```

**Description**

Creates 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.

**Parameters**

- `scopeSize`—Size of the allocation area within the backing store.
- `backingSize`—Size of the total backing store.
- `logic`—`Runnable` to be entered using `this` as its current memory area when `enter()` is called.

**Throws**

- `javax.realtime.IllegalRealtimeArgumentException`—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`. 

570 RTSJ 2.0 (Draft 66)
StackedMemory

javafx.realtime.OutOfMemoryStaticError—when there is insufficient memory available to reserve the requested backing store.

javafx.realtime.IllegalTaskStateException—when the current Schedulable has a StackedMemory as its explicit initial scoped memory area and that area is not on the scope stack.

StackedMemory(SizeEstimator, SizeEstimator, Runnable)

Signature
public
StackedMemory(SizeEstimator scopeSize,
SizeEstimator backingSize,
Runnable logic)

Description
Equivalent to StackedMemory(long, long, Runnable) with argument list (scopeSize.getEstimate(), backingSize.getEstimate(), runnable).

Parameters
scopeSize—SizeEstimator indicating the size of the object allocation area within the backing store.
backingSize—SizeEstimator indicating the size of the total backing store.
logic—Runnable to be entered using this as its current memory area when enter() is called.

Throws
javafx.realtime.IllegalRealtimeArgumentException—when either scopeSize or backingSize is null, or when scopeSize.getEstimate() is too large to be allocated from a backing store of size backingSize.getEstimate().

javafx.realtime.OutOfMemoryStaticError—when there is insufficient memory available to reserve the requested backing store.

javafx.realtime.IllegalTaskStateException—when the current Schedulable has a StackedMemory as its explicit initial scoped memory area and that area is not on the scope stack.

StackedMemory(long, long)

Signature
RTSJ 2.0 (Draft 66) 571
public
StackedMemory(long scopeSize,
long backingSize)

Description
Equivalent to StackedMemory(long, long, Runnable) with argument list
(scopeSize, backingSize, null).

Parameters
scopeSize—Size of the allocation area within the backing store.
backingSize—Size of the total backing store.

Throws
javax.realtime.IllegalRealtimeArgumentException—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.
javax.realtime.OutOfMemoryStaticError—when there is insufficient memory
available to reserve the requested backing store.
javax.realtime.IllegalTaskStateException—when the current Schedulable
has a StackedMemory as its explicit initial scoped memory area and that area
is not on the scope stack.

StackedMemory(SizeEstimator, SizeEstimator)

Signature
public
StackedMemory(SizeEstimator scopeSize,
SizeEstimator backingSize)

Description
Equivalent to StackedMemory(long, long, Runnable) with argument list
(scopeSize.getEstimate(), backingSize.getEstimate(), null).

Parameters
scopeSize—SizeEstimator indicating the size of the object allocation area within
the backing store.
backingSize—SizeEstimator indicating the size of the total backing store.

Throws
javax.realtime.IllegalRealtimeArgumentException—when either scopeSize or backingSize is null, or when scopeSize.getEstimate() is too large to be allocated from a backing store of size backingSize.getEstimate().

calendar.

javax.realtime.OutOfMemoryStaticError—when there is insufficient memory available to reserve the requested backing store.

javax.realtime.IllegalTaskStateException—when the current Schedulable has a StackedMemory as its explicit initial scoped memory area and that area is not on the scope stack.

StackedMemory(long, Runnable)

Signature

public
StackedMemory(long scopeSize,
Runnable logic)

Description

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 same backing store as the parent scope’s object allocation area. The parent scope must be a StackedMemory.

Parameters

scopeSize—Size of the allocation area within the backing store.

logic—Runnable to be entered using this as its current memory area when enter() is called.

Throws

javax.realtime.IllegalRealtimeArgumentException—when the parent memory area is not a StackedMemory.

javax.realtime.MemoryInUseException—when the parent StackedMemory already has a child that is also a guest StackedMemory.

javax.realtime.IllegalRealtimeArgumentException—when scopeSize is less than zero.

javax.realtime.OutOfMemoryStaticError—when there is insufficient memory available in the backing store of the parent StackedMemory’s object allocation area to reserve the requested object allocation area.

javax.realtime.IllegalTaskStateException—when the current Schedulable has a StackedMemory as its explicit initial scoped memory area and that area

RTSJ 2.0 (Draft 66)  573
is not on the scope stack.

**StackedMemory(SizeEstimator, Runnable)**

**Signature**

```java
public
StackedMemory(SizeEstimator scopeSize,
               Runnable logic)
```

**Description**

Equivalent to `StackedMemory(long, Runnable)` with argument list `(scopeSize.getEstimate(), runnable)`. 

**Parameters**

- `scopeSize`—SizeEstimator indicating the size of the object allocation area within the backing store.
- `logic`—Runnable to be entered using this as its current memory area when `enter()` is called.

**Throws**

- `javax.realtime.IllegalRealtimeArgumentException`—when the parent memory area is not a StackedMemory.
- `javax.realtime.MemoryInUseException`—when the parent StackedMemory already has a child that is also a guest StackedMemory.
- `javax.realtime.IllegalRealtimeArgumentException`—when `scopeSize` is null.
- `javax.realtime.OutOfMemoryStaticError`—when there is insufficient memory available in the backing store of the parent StackedMemory's object allocation area to reserve the requested object allocation area.
- `javax.realtime.IllegalTaskStateException`—when the current Schedulable has a StackedMemory as its explicit initial scoped memory area and that area is not on the scope stack.

**StackedMemory(long)**

**Signature**

```java
public
StackedMemory(long scopeSize)
```
StackedMemory  \hspace{1cm} \textit{javax.realtime.memory 11.4}

\textit{Description}

Equivalent to \texttt{StackedMemory(long, Runnable)} with argument list \((\text{scopeSize}, \text{null})\).

\textit{Parameters}

\texttt{scopeSize}—Size of the allocation area within the backing store.

\textit{Throws}

\texttt{javax.realtime.IllegalRealtimeArgumentException}—when the parent memory area is not a \texttt{StackedMemory}.

\texttt{javax.realtime.MemoryInUseException}—when the parent \texttt{StackedMemory} already has a child that is also a guest \texttt{StackedMemory}.

\texttt{javax.realtime.IllegalRealtimeArgumentException}—when \texttt{scopeSize} is less than zero.

\texttt{javax.realtime.OutOfMemoryStaticError}—when there is insufficient memory available in the backing store of the parent \texttt{StackedMemory}'s object allocation area to reserve the requested object allocation area.

\texttt{javax.realtime.IllegalTaskStateException}—when the current \texttt{Schedulable} has a \texttt{StackedMemory} as its explicit initial scoped memory area and that area is not on the scope stack.

\textbf{StackedMemory(SizeEstimator)}

\textit{Signature}

\begin{verbatim}
public
StackedMemory(SizeEstimator scopeSize)
\end{verbatim}

\textit{Description}

Equivalent to \texttt{StackedMemory(long, Runnable)} with argument list \((\text{scopeSize}.\text{getEstimate}(), \text{null})\).

\textit{Parameters}

\texttt{scopeSize}—\texttt{SizeEstimator} indicating the size of the object allocation area within the backing store.

\textit{Throws}

\texttt{javax.realtime.IllegalRealtimeArgumentException}—when the parent memory area is not a \texttt{StackedMemory}.

\texttt{javax.realtime.MemoryInUseException}—when the parent \texttt{StackedMemory} already has a child that is also a guest \texttt{StackedMemory}.

RTSJ 2.0 (Draft 66) 575
11 Alternative Memory Areas

javax.realtime.IllegalRealtimeArgumentException—when scopeSize is null.
javax.realtime.OutOfMemoryStaticError—when there is insufficient memory available in the backing store of the parent StackedMemory's object allocation area to reserve the requested object allocation area.
javax.realtime.IllegalTaskStateException—when the current Schedulable has a StackedMemory as its explicit initial scoped memory area and that area is not on the scope stack.

11.4.3.10.2 Methods

resize(long)

Signature

public void
resize(long scopeSize)

Description

Changes 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 grow the allocation area, or to shrink the allocation area no smaller than 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.

Parameters

scopeSize—The new allocation area size for this scope.

Throws

javax.realtime.RealtimeSecurityException—when the caller is not permitted to perform the requested adjustment.
javax.realtime.IllegalRealtimeArgumentException—there are additional guest StackedMemory allocation areas after this one in the backing store.
javax.realtime.OutOfMemoryStaticError—when the remaining backing store is insufficient for the requested adjustment, or when the current Schedulable has a StackedMemory as its explicit initial scoped memory area and that area is not on the scope stack.
getMaximumSize

Signature
    public long
    getMaximumSize()

Description
    Gets the maximum size this memory area can attain. The value returned by
    this function is the maximum size that can currently be passed to \texttt{resize(long)}
    without triggering an \texttt{OutOfMemoryStaticError}.

Returns
    the maximum size attainable.

hostBackingStoreSize

Signature
    public long
    hostBackingStoreSize()

Description
    Determines the total amount of memory in the backing store of this stacked
    memory. For a guest stacked memory, this is always zero.

Returns
    the total amount of backing store in bytes.

hostBackingStoreRemaining

Signature
    public long
    hostBackingStoreRemaining()

Description
    Determines the amount of memory remaining for allocation to new stacked
    memories in the backing store of this stacked memory. For a guest stacked
    memory, this is always zero.

Returns
    the amount of backing store remaining in bytes.
**hostBackingStoreConsumed**

**Signature**

```java
class StackedMemory
{
    public long hostBackingStoreConsumed()
}
```

**Description**

Determines the amount of memory consumed by existing stacked memories from the backing store of this stacked memory. For a guest stacked memory, this is always zero.

**Returns**

the amount of backing store consumed in bytes.

**enter**

**Signature**

```java
class StackedMemory
{
    public void enter()
}
```

**Description**

Associates 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.

**Throws**

- `javax.realtime.IllegalRealtimeArgumentException`—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`—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 `javax.realtime.IllegalAssignmentError`, so the JVM cannot be permitted to deliver the exception. The `javax.realtime.ThrowBoundaryError` is allocated in the current allocation context and contains information about the exception it replaces.
IllegalStateException—when the execution context is not an instance of javax.realtime.Schedulable or when this method is invoked during finalization of objects in scoped memory and entering this scoped memory area would force deletion of the execution context that triggered finalization. This would include the scope containing the execution context, and the scope (if any) containing the scope containing execution context.

MemoryAccessError—when caller is a schedulable that may not use the heap and this memory area’s logic value is allocated in heap memory.

See Section ScopedMemory.enter()

**enter(Runnable)**

**Signature**

```java
public void enter(Runnable logic)
```

**Description**

Associates 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.

**Throws**

java.realtime.IllegalRealtimeArgumentException—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.

java.realtime.ThrowBoundaryError—null

java.realtime.IllegalTaskStateException—null

java.realtime.MemoryAccessError—null

See Section ScopedMemory.enter(Runnable)

**joinAndEnter**

**Signature**
public void
joinAndEnter()

Description

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 enters the `ScopedMemory` and executes 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 `IllegalRealtimeArgumentException` 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.

Throws

- `java.lang.InterruptedException`—when this schedulable is interrupted by `javax.realtime.RealtimeThread.interrupt()` or `javax.realtime.control.AsynchronouslyInterruptedException.fire()` while waiting for the reference count to go to zero.
- `IllegalTaskStateException`—when the execution context is not an instance of `javax.realtime.Schedulable` or when this method is invoked during finalization of objects in scoped memory and entering this scoped memory area would force deletion of the task that triggered finalization. This would include the scope containing the task, and the scope (if any) containing the scope containing the task.
- `ThrowBoundaryError`—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 `javax.realtime.IllegalAssignmentError`, so the JVM cannot be permitted to deliver the exception. The `javax.realtime.ThrowBoundaryError` is allocated in the current allocation context and contains information about the exception it replaces.
- `javax.realtime.ScopedCycleException`—when this invocation would break the single parent rule.
- `javax.realtime.IllegalRealtimeArgumentException`—when the execution context 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(HighResolutionTime)

**Signature**

```java
class ScopedMemory {
    public void joinAndEnter(javax.realtime.HighResolutionTime<?> time) throws InterruptedException;
}
```

**Description**

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 `Runnable` object passed to the constructor. When no instance of `Runnable` was passed to the memory area’s constructor, the method throws `IllegalRealtimeArgumentException` 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 `javax.realtime.HighResolutionTime`, 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`.

Note that expiration of `time` may cause control to enter the memory area before its reference count has gone to zero.

**Parameters**

- `time`—The time that bounds the wait.

**Throws**

- `javax.realtime.ThrowBoundaryError`—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 `javax.realtime.IllegalAssignmentError`, so the JVM cannot be permitted to deliver the exception. The `javax.realtime.ThrowBoundaryError` is allocated in the current allocation context and contains information about the exception it replaces.

`java.lang.InterruptedException`—when this schedulable is interrupted by `javax.realtime.RealtimeThread.interrupt()` or `javax.realtime.control.AsynchronouslyInterruptedException.fire()` while waiting for the reference count to go to zero.

`javax.realtime.IllegalTaskStateException`—when the execution context is not an instance of `javax.realtime.Schedulable` or when this method is invoked during finalization of objects in scoped memory and entering this scoped memory area would force deletion of the task that triggered finalization. This would include the scope containing the task, and the scope (if any) containing the scope containing the task.

`javax.realtime.ScopedCycleException`—when the execution context is a schedulable and this invocation would break the single parent rule.

`javax.realtime.IllegalRealtimeArgumentException`—when the execution context is a schedulable, and `time` is `null` or no non-null `logic` value was supplied to the memory area’s constructor.

`javax.realtime.UnsupportedRealtimeOperationException`—when the wait operation is not supported using the clock associated with `time`.

`javax.realtime.MemoryAccessError`—when calling schedulable may not use the heap and this memory area’s logic value is allocated in heap memory.

### `joinAndEnter(Runnable)`

**Signature**

```java
public void joinAndEnter(Runnable logic)
throws InterruptedException
```

**Description**

In the error-free case, `joinAndEnter` combines `join();` and `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, the method throws IllegalRealtimeArgumentException 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.

Parameters
logic—The Runnable object which contains the code to execute.

Throws
java.lang.InterruptedException—when this schedulable is interrupted by javax.realtime.RealtimeThread.interrupt() or javax.realtime.control.AsynchronouslyInterruptedException.fire() while waiting for the reference count to go to zero.

IllegalTaskStateException—when the execution context is not an instance of javax.realtime.Schedulable or when this method is invoked during finalization of objects in scoped memory and entering this scoped memory area would force deletion of the task that triggered finalization. This would include the scope containing the task, and the scope (if any) containing the scope containing the task.

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 javax.realtime.IllegalAssignmentError, so the JVM cannot be permitted to deliver the exception. The javax.realtime.ThrowBoundaryError is allocated in the current allocation context and contains information about the exception it replaces.

javax.realtime.ScopedCycleException—when this invocation would break the single parent rule.

javax.realtime.IllegalRealtimeArgumentException—when the execution context is a schedulable and logic is null.

joinAndEnter(Runnable, HighResolutionTime)

Signature
public void joinAndEnter(Runnable logic,
javax.realtime.HighResolutionTime<?> time)
    throws InterruptedException

Description
In the error-free case, joinAndEnter combines join();enter(); such that no
enter() from another schedulable can intervene between the two method invoca-
tions. 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 javax.realtime.HighResolutionTime, 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).

The method throws IllegalRealtimeArgumentException 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.

Parameters
logic—The Runnable object which contains the code to execute.
time—The time that bounds the wait.

Throws
java.lang.InterruptedIOException—when this schedulable is interrupted
by javax.realtime.RealtimeThread.interrupt() or javax.realtime.
control.AsynchronouslyInterruptedIOException.fire() while waiting for
the reference count to go to zero.

IllegalTaskStateException—when the execution context is not an instance of
javax.realtime.Schedulable or when this method is invoked during final-
ization of objects in scoped memory and entering this scoped memory area
would force deletion of the task that triggered finalization. This would include
the scope containing the task, and the scope (if any) containing the scope
containing the task.

ThrowBoundaryError—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 a `javax.realtime.IllegalAssignmentError`, so the JVM cannot be permitted to deliver the exception. The `javax.realtime.ThrowBoundaryError` is preallocated and saves information about the exception it replaces.

`javax.realtime.ScopedCycleException`—when the execution context is a schedulable and this invocation would break the single parent rule.

`javax.realtime.IllegalRealtimeArgumentException`—when the execution context is a schedulable and `time` or `logic` is `null`.

`javax.realtime.UnsupportedRealtimeOperationException`—when the wait operation is not supported using the clock associated with `time`.

11.5 The Rationale

The memory area support has three main areas of change: package separation, extensions for safer off-heap memory management, and factory based, type compatible physical memory areas. Each of these changes was driven by its own set of requirements. Still they where designed to produce a coherent set of features for the new specification.

11.5.1 Package Separation

RTSJ 2.0 has been reorganized to support a core group of classes and three subgroups. To do this, some facilities that were in the main `javax.realtime` package have had to be moved to other packages. Though not strictly necessary to support Java 8, these changes make it possible to define Java 9 modules for each of these groups of classes. For alternate memory management, the corresponding classes have been moved to `javax.realtime.memory` and could be in a module with the same name. This separation has been used as a means of separating the attributes used to raw memory from physical memory as well, since raw memory is in the `javax.realtime.device` package.

11.5.2 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 indeterminacy 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.

When a scoped memory area is used by both heap and extraheap tasks, there could be cases where a finalizer executed in extraheap context could attempt to use a heap reference left by a heap-using task. 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 explicit initial scoped memory areas (EISMAs) for schedulables has repercussions.

1. The EISMA’s parent is set when its realtime thread is constructed or its ASEH becomes firable, but its reference count is not incremented until the thread is started or the asynchronous event handler is released. 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.
2. Any action that makes an event handler not firable must block until all the resulting finalization completes.
3. Any action that makes an event handler firable must block until any ongoing finalization of its EISMA completes.

Since an EISMA is only entered upon release and exited at the completion of release, the handler of the release can generally run finalization. A thread collecting the event that triggers the handler will not have any effect on EISMA finalization. Only another execution context can prevent finalization by the handler at release end.
11.5.3 Reservation Management

The reservation management semantics in RTSJ 2.0 were created to maintain the maximum semantic compatibility with RTSJ 1.0, while providing the default EISMA behavior more commonly desired (that is, that every release is begun in an empty memory area). Since a memory area that is the EISMA of a task but is not otherwise in use is not on any scope stack, without reservation management it or one of the scopes in the scope stack below it could be erroneously placed at a location in the tree of memory areas different from that where it will be entered on the task’s release. This would cause an exception to be thrown at release, due to a violation of the single parent rule. By reserving its location in the tree at the time that it is configured, this exception (which cannot be conveniently caught by any application context) is avoided.

11.5.4 Backing Store Management

Management of backing store memory for memory areas was overhauled substantially for RTSJ 2.0. In this edition, access to the global and physical backing stores is limited on a per-Schedulable basis, and the ability to use and reuse allocations from these backing stores in a fragmentation-free manner is provided by StackedMemory. The rules for access to the global and physical backing stores and the creation of StackedMemory objects for Schedulables with an explicit initial scoped memory area of type StackedMemory are strict, but necessary to achieve full control of backing store resources. In particular, It seems impractical to enforce that a Schedulable is incapable of examining the MemoryAreas that lie on its own scope stack, whether directly (e.g., via a visitor) or indirectly (e.g., by requesting the MemoryArea from which objects it can reach are allocated). Therefore, an implementation must enforce specific rules on a) the global and physical backing stores, and b) the StackedMemory hierarchy in which a Schedulable operates. These rules are designed to constrain the backing store allocations of a Schedulable regardless of the MemoryAreas with which it may come in contact.

11.5.5 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

1. volatile – whether it maintains its state when the power is turned off,
2. writable – whether it can be written at all, written once or written many times and whether writing is under program control,
3. synchronous or asynchronous – whether the memory is synchronized with the system bus,
4. 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,
5. fast to access – both for reading and writing.

Examples include the following [2].

1. 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 cost per byte. SRAM has faster access times and is more expensive. Both DRAM and SRAM are examples 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.

2. 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.

3. 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.
   (a) 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.
   (b) FLASH memory – this is nonvolatile memory 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
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.

1. Ninety percent of performance-critical memory access is to a set of objects that could fit in a half the total memory.

2. The system enables the locking of a small amount of data in the cache, and a small number of pages in the translation look-aside 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.

3. 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 one ensures that all data that might interact with disk, network, or video is not stored on the daughter board.

4. Improved video performance can be obtained by using an array as a video buffer. This will only be effective if a physically contiguous, unpagable, 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.

5. The system has banks of SRAM and saves power by automatically putting them to “sleep” whenever they stay unused for 100ms or so. To exploit this, the objects used by each phase of this 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 13.2.1). 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\textsuperscript{4}. 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.

11.5.5.1 The Original Physical Memory Framework

The RTSJ 1.0.x supported three ways to allocate objects that can be placed in particular types of memory.

1. **ImmortalPhysicalMemory** allocates immortal objects in memory with specified characteristics.
2. **LTPhysicalMemory** allocates scoped memory objects in a memory with specified characteristics using a linear time memory allocation algorithm.
3. **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 standard memory 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.

Originally, the RTSJ supported access to physical memory via a memory manager and one or more memory filters. The goal of the memory manager was to provide a single interface with which the programmer could interact to access memory with a particular characteristic. A memory filter provided access to a particular type of physical memory. Memory filters could be dynamically added and removed from the system, and there could only be a single filter for each memory type. The memory manager was unaware of the physical addresses of each type of memory. This was encapsulated by the filters. The filters also know the virtual memory characteristics that had been allocated to their memory type. For example, whether the memory is readable or writable.

In theory, any developer could create a new physical memory filter and register it with the Physical Memory Model (PMM). However, the programming of filters is difficult for the following reasons.

1. 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, when it is not, the physical address of the next free physical memory of the requested type. This is complex because requests for

\textsuperscript{4}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.
compound types of physical memory must find a free segment that satisfies all attributes of the compound type.

2. The Java runtime must continue to behave correctly under the Java memory model when 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. For instance, write-through cache is more restricted than copy-back cache. When 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 interpreter, 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.

3. The physical memory filters where passed as type Object to physical memory type constructors, so no type checking supported proper usage.

Hence, the utility of the physical memory filter framework at Version 1.0.2 is questionable, and hence is replaced in 2.0 with a simpler, factory-based framework.

11.5.5.2 The RTSJ 2.0 Physical Memory Framework

The main problem with the 1.0.x framework is that it placed too great 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. Therefore, the JVM 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.

1. 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 in 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, nonvolatile memory could be used, but any objects stored 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 can be accessed via the RTSJ Raw Memory facilities (see Section 13.2.1).

2. Any API must delegate 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\(^5\) 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.

The program must 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 options for mapping physical memory into the virtual memory of the application. Examples include whether the range is to be permanently resident in memory and whether data is written to the cache and the main memory simultaneously, i.e., a write through caching. By default, memory is subject to paging or swapping.

Given the required physical memory characteristics, the programmer creates a `PhysicalMemoryRegion` for accessing this memory and registers it with a `PhysicalMemoryFactory`. This factory can then be used with new constructors on the physical memory classes. For example,

```java
PhysicalMemoryCharacteristic sram = new
    PhysicalMemoryCharacteristic();
PhysicalMemoryCharacteristic[] characteristics =
    new PhysicalMemoryCharacteristic[]{ sram };
PhysicalMemorySelector selector =
    new PhysicalMemorySelector(null, null, WRITE_THROUGH, FIXED);
MemoryArea memory = factory.createImmortalMemory(selectors,
    size, logic);
```

Use of this factory enables the programmer to specify the allocation of the backing store in a particular type of memory with particular memory characteristics. The selector 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.

Hence, once physical memory regions have been created and registered, physical memory areas can be created and objects can be allocated within those memory regions using the usual RTSJ mechanisms for changing the allocation context of the `new` operator.

---

\(^5\)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/](http://www.acpi.info/)
11.5.5.3 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.

11.5.5.3.1 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.

```java
public class SRAMType implements PhysicalMemoryCharacteristic {}  
```

The initialization method must now create instances of the `PhysicalMemoryRegion` class to represent the physical memory modules needed.

```java
PhysicalMemoryRegion staticRam = new PhysicalMemoryRegion(0x10000000L, 0x100000000L);
PhysicalMemoryRegion staticSleepableRam = new PhysicalMemoryRegion(0xA0000000L, 0x100000000L);
```

It then creates names for the characteristics that the program wants to associate with each memory module.

```java
PhysicalMemoryCharacteristic STATIC_RAM = new MyMemoryType();
PhysicalMemoryCharacteristic AUTO_SLEEPABLE = new MyMemoryType();
```

It then informs the PMM of the appropriate associations:

```java
PhysicalMemoryFactory factory = PhysicalMemoryFactory.getDefault();
factory.associate(STATIC_RAM, staticRam);
factory.associate(STATIC_RAM, staticSleepableRam);
factory.associate(AUTO_SLEEPABLE, staticSleepableRam);
```

Once this is done, the program can create a selector with the required properties. In
this case, some SRAM must be auto sleepable.

```java
PhysicalMemoryCharacteristic[] PMC =
new PhysicalMemoryCharacteristic[2];
PMC[0] = STATIC_RAM;
PMC[1] = AUTO_SLEEPABLE;

PhysicalMemorySelector selector =
new PhysicalMemorySelector<>(PMC, null, 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.

### 11.5.5.3.2 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 straightforward. Here is an example.

```java
ImmortalMemory IM = factory.createImmortalMemory(selector, 0
x1000);
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.
    }
});
```

The physical memory factory keeps track of previously allocated memory and is able to determine whether memory is available with the appropriate characteristics. Of course, the physical memory factory has no knowledge of what these names mean; it is merely providing a look-up service.
Chapter 12

Asynchronous Control Flow

An important aspect of this specification is its support for asynchronous control flow. The Control Module provides that support through a mechanism that

- enable asynchronous transfer of control and
- facilitate the asynchronous termination of realtime threads.

Two related mechanisms are provided: a general transfer mechanism called Asynchronous Transfer of Control (ATC), which provides a means of stopping some calculation prematurely and a task abort mechanism to safely terminate any task called Asynchronous Task Termination (ATT).

The `interrupt()` method in `java.lang.Thread` provides rudimentary asynchronous communication by setting a pollable and resettable flag in the target thread, and by throwing a synchronous exception when the target thread is blocked at an invocation of `wait()`, `sleep()`, `join()`, or an operation that throws `InterruptedException`. This specification generalizes the notion of interrupt to all tasks, offering a more comprehensive asynchronous execution control facility without requiring polling. For RealtimeThreads, the effect of `Thread.interrupt()` must be extended by adding an overridden version in `RealtimeThread`.

This mechanism, called Asynchronous Transfer of Control (ATC), is based on throwing and propagating an exception that, though asynchronous, is 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 `Schedulable`.

A method indicates its eligibility for asynchronous interruption 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 enables specifying the interruptible
code, the handler, and the result retrieval as separate methods.

Whereas ATC provides a general transfer mechanism for code declared to be
interruptible, ATT is designed to safely terminate code that is not explicitly pro-
grammed for being interrupted asynchronously. Thus ATC is recoverable and ATT
is not. Otherwise, the mechanisms are quite similar.

12.1 Definitions

Termination Pending — The attribute of a task which has been aborted before
its begins the termination process.

Control Transfer Pending — The attribute of a task which has been interrupted
and either has not reached an asynchronous transition point or is in an ATC
deferred section.

Asynchronous Interrupt — A request for a transfer of control.

Abort — A request for a task to be terminated.

Terminating — The state of a task that is in the process of unwinding its execution
stack following its being aborted.

Transferring Control — The state of a task that is in the process of unwinding
its execution stack to the next point where the AsynchronouslyInterruptEx-
ception is caught.

Safe Point — A point in the code, where the task is guaranteed not to mutate the
heap.

Self-Suspending Operation — Any operation that causes the task to potentially
self suspend. Self-suspend operations include

• an attempt to acquire a monitor lock,
• a call to Object.wait(),
• a call to Thread.sleep(),
• a call to RealtimeThread.waitForNextRelease(), and
• any IO operation

Asynchronous Transition Point — Any point in the execution of a task when
that task must cease executing user code and begin any pending asynchronous
control transfer.

Transition Deferred Section — A section of code during the execution of which
a task may not transfer from a pending state to a Transferring or Terminating
state, i.e., contains no Asynchronous Transition Points. Transition deferred
section are the following:

• synchronized statements,
• static initializers, and
• the execution of finally clauses.
Definitions 12.1

Asynchronously Interrupted Exception (AIE) — An instance of the `javax.realtime.AsynchronouslyInterruptedException` class (a subclass of `java.lang.InterruptedException`).

Asynchronously Interruptible Method (AI-Method) — A method or constructor that includes `AsynchronouslyInterruptedException` explicitly (that is, not a subclass of `AsynchronouslyInterruptedException`) in its throws clause.

Asynchronous Transfer of Control (ATC) — A nonlocal transfer of program control in a task initiated from outside that task.

ATC Deferred Section — A superset of transition deferred sections with the inclusion of all methods that do not declare `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 `AsynchronouslyInterruptedException` (AIE). The interruptible blocking methods comprise

- `HighResolutionTime.waitForObject()`,
- `Object.wait()`,
- `Thread.sleep()`,
- `RealtimeThread.sleep()`,
- `Thread.join()`,
- `ScopedMemory.join()`,
- `ScopedMemory.joinAndEnter()`,
- `RealtimeThread.waitForNextReleaseInterruptible()`,
- `WaitFreeWriteQueue.read()`,
- `WaitFreeReadQueue.waitForData()`,
- `WaitFreeReadQueue.write()`,
- `WaitFreeDequeue.blockingRead()`,
- `WaitFreeDequeue.blockingWrite()`

and their overloaded forms. Furthermore, the method `waitForNextRelease` in `RealtimeThread` is interruptible when the thread’s release parameters `isRousable` method returns `true`. Similarly instances of `AsyncBaseEventHandlers` are released early when their release parameters `isRousable` method returns `true`.

Generation of an asynchronous control change request — this is the event in the underlying system that makes the AIE available to the program.

Delivery of the asynchronous control change request — this is the action in...
the target thread that begins unwinding the stack to reach point where, in the case of ATC, control continues or where, in the case of ATT, the stack is empty and control in that task ceases.

12.2 Semantics

Both asynchronous transfer of control (ATC) and asynchronous task termination (ATT) provide a means of ending computation asynchronously. Since the programmer must declare in which code ATC can be taken, the programmer can ensure that it is safe to asynchronously pass control to another section of code. ATT enables ending computation more generally, but always results in task termination.

12.2.1 Asynchronous Transition Point

Whether it be an ATT or an ATC request, there are two steps to engage asynchronous control flow: generating and delivering the change of control flow request. The first is done by the task requesting the change, either ATC or ATT, and the second by the task in which the change takes place. Activation should take place as soon as possible, but the program must be in a well-defined state when this change of control happens. For this, the runtime must define points, called asynchronous transition points, where this control change may take place.

There are three requirements for asynchronous transition points.

1. An asynchronous transition point (ATP) should occur at each safe point and must be at regular intervals in the code.
2. There must be an ATP at least at the following points:
   - any self-suspending operation,
   - the release of a monitor lock,
   - any transition between a deferred and a nondeferred section (in either direction),
   - any call to Thread.interrupt(),
   - any call to Thread.start(), and
   - any call to AsyncBaseEvent.attach()
3. In general, all calls to methods (inlining excepted) should be asynchronous transition points.

Whether or not reaching an ATP results in delivery of a pending control change depends on whether or not the point is in a Deferred Section for the type of the pending control change.

As with throwing any instance of Throwable, delivering a pending control change request begins the unwinding of the stack. During this unwinding, all finally
12.2.2 Asynchronous Transfer of Control

An ATC is generated by calling either the `fire()` method on an AIE where a task is in its `doInterruptible` method or the `Schedulable.interrupt()` method on a task. Between the generation of an AIE and its delivery, the exception is held `pending`. The AIE returns to pending after delivery when caught. It must be `cleared` by program logic, explicitly by using the `AsynchronouslyInterruptedException.clear()` method or implicitly when `AsynchronouslyInterruptedException.doInterruptible` completes, to proceed to the normal execution state at the new control flow point. Figure 12.1 illustrates the state transitions for ATC.

The following points define the semantics of ATC. Semantics that apply to particular classes, constructors, methods, and fields will be found in their detail sections, respectively.

1. An AIE is generated for a given task when
• the `fire()` method is called on an AIE for which the task is executing within the `doInterrruptible` method
• or the `Schedulerable.interrupt()` method is called.

The latter is also effectively called when an AIE is generated by internal virtual machine mechanisms (such as an interrupted I/O operation) that are asynchronous to the execution of the program logic which is the target of the AIE.

2. An AIE becomes pending upon generation and remains pending until explicitly cleared or replaced by another AIE.

3. An AIE is delivered to a schedulable when it is executing in a method declared to throw AIE, except in an ATC-deferred section as defined below.

   (a) The generation of an AIE through the `fire()` mechanism behaves as if it sets an asynchronously-interrupted status in the schedulable.

   i. When the schedulable is blocked within an interruptible blocking method or invokes an interruptible blocking method when this asynchronously-interrupted status is set, the invocation immediately completes by throwing the pending AIE and clearing the asynchronously-interrupted status.

   ii. When a pending AIE is explicitly cleared, the asynchronously-interrupted status is also cleared.

   (b) Blocking methods which are declared to throw `java.lang.IOException` but are not declared to throw `java.io.InterruptedException`, e.g., blocking methods in `java.io.*`, must be prevented from blocking indefinitely when invoked from a method with `AsyncronouslyInterruptedException` in its `throws` clause. When an AIE is generated and the target schedulable’s control is blocked inside one of these methods with an AIE-method on the call stack, the implementation may either unblock the blocked call, raise `java.lang.InterruptedIOException` on behalf of the call, or allow the call to complete normally if the implementation determines that the call would unblock within a bounded period of time defined by the implementation.

   (c) When an AIE-method is attempting to acquire an object lock when an associated AIE is generated, the attempt to acquire the lock is abandoned.

   (d) When 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 `try` statement that (i) has a handler for the generated AIE (that is a handler naming the AIE’s class or any of its superclasses, or a `finally` clause)
and ii) 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 that if synchronized code is abandoned as a result of this control transfer, the associated locks are released.

4. A pending ATC becomes active at the next asynchronous transition point not in an ATC deferred section.

5. 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.

6. Native methods that include `AsynchronouslyInterruptedException` in their `throws` clause have implementation-specific behavior.

7. 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.

8. Instances of the `Timed` class have a logically 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.

9. An AIE only has the semantics defined here when it originates with the `AsynchronouslyInterruptedException.fire()` method, the `Schedulable.interrupt()` method or from within the realtime VM. If an AIE is thrown from program logic using the Java throw statement, it uses the same semantics as throwing any other instance of a subclass of `Exception`, it is processed as a normal exception, and has no effect on the pending state of any AIE, and no effect on the firing of the AIE concerned.

10. The `Schedulable.interrupt()` method is a special case of ATC.

   (a) it causes the target task to throw the generic AIE and has the behaviors defined for `Thread.interrupt()`. This is the only interaction between the ATC mechanism and the conventional `interrupt()` mechanism.

   (b) An AEH that is waiting for a release and is rousable will release immediately as per Section 6.2.1.3.4 above with the generic AIE pending when it is interrupted.

   (c) A RealtimeThread blocked in `waitForNextRelease` that is rousable will
immediately return as per Section 6.2.1.3.2 with the generic AIE pending when it is interrupted.

11. The virtual machine must throw AsynchronouslyInterruptedException for interrupted java.nio methods as if RealtimeThread.interrupt() were called on the interrupted thread.

12.2.2.1 Extending Conventional Java Interrupts

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.

When so is an instance of a schedulable and the interrupt() method is called on the schedulable associated with that object, then the following holds.

1. When control is in an ATC-deferred section, then the AIE remains in a pending state. Execution 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 2 as appropriate.

2. When control is not in an ATC-deferred section, then control is transferred to the catch or finally clause of the nearest dynamically-enclosing try statement that is in an ATC-deferred section and has a handler for the generated AIE, i.e., is a handler naming the AIE’s class or any of its superclasses, or a finally clause. 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 [5] for an explanation of the terms dynamically enclosing and handlers. The RTSJ uses those definitions unaltered.

3. When 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). ATC then follows option 1 or 2 as appropriate.

4. When control is transferred from an ATC-deferred section to an AI method through the action of propagating an exception while an AIE is pending, when the transition to the AI-method occurs, the thrown exception is discarded and replaced by the pending AIE.

12.2.2.2 Nesting AsynchronouslyInterruptedExceptions

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 Schedulable.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 stack frame further from stack base. The phrase “a frame shallower on the stack than this frame” refers to a stack frame nearer to the stack base.

1. When the current AIE is an $AIE_0$ and the new AIE is an $AIE_x$ associated with any frame on the stack, the new AIE ($AIE_x$) is discarded.
2. When the current AIE is an $AIE_x$ and the new AIE is an $AIE_0$, the current AIE ($AIE_x$) is replaced by the new AIE ($AIE_0$).
3. When the current AIE is an $AIE_x$ and the new AIE is an $AIE_y$ from a frame deeper on the stack, the new AIE ($AIE_y$) is discarded.
4. When 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 ($AIE_y$).
5. When the current AIE is an $AIE_0$ and the new AIE is an $AIE_0$, or when the current AIE is an $AIE_x$ and the new AIE is an $AIE_x$, the new AIE is discarded.

When $clear()$ is called on a pending AIE or that AIE is superseded by another, the first AIE’s pending state is cleared. Clearing a nonpending AIE (with the $clear()$ method) has no effect.

12.2.3 Asynchronous Task Termination

Asynchronous task termination provides a best effort termination facility for tasks. It ensures that each task unwinds its stack in a well-defined way, so that it has a chance to release all resources it has taken. As illustrated in Figure 12.1, ATT has simpler state transitions than ATC. As with ATC, a generated ATT is first pending, but once it is delivered, it cannot return to the pending state.

The following rules govern ATT.

1. Any task, with the required security permissions, can abort any instance of $AsynchronousControlGroup$ to which it has a reference.
2. A call to method $AsynchronousControlGroup.abort()$ shall immediately mark the group and all tasks it contains as termination pending.
3. A task marked with termination pending shall enter its abort phase at its next asynchronous transition point not in an ATT deferred section.
4. An asynchronous transition point for ATT is the same as for ATC except for what sections of code are deferred.
5. A task, on entering into its termination phase, shall behave as if an uncatchable $Error$ has been thrown at the point in the code where the task moves from termination pending to terminating. When the task enters the Terminating state, any uncaught exception handler set for the task shall be executed.
6. A task in the terminating phase shall be prohibited from execution any self-suspending operations except $close$ and acquiring a lock.
In order for the abort mechanism to work, the system must prevent the application from doing any of the following.

1. All `finally` and catch `Throwable` clauses must terminate.
2. No `finally` or catch `Throwable` clause may call may self suspend except for calls to close and taking a monitor.

These rules cannot be enforced completely at runtime. Static analysis and signing the resultant code before loading is needed to ensure proper functioning of ATT.
12.3  javax.realtime.control

12.3.1 Interfaces

12.3.1.1 Interruptible

```java
public interface Interruptible {
  // Description
  Interruptible is an interface implemented by classes that will be used as arguments on the methods doInterruptible() of AsynchronouslyInterruptedException and its subclasses. doInterruptible() invokes the implementations of the methods in this interface.

12.3.1.1.1 Methods

run(AsynchronouslyInterruptedException)

**Signature**

```java
public void run(AsynchronouslyInterruptedException exception)
throws AsynchronouslyInterruptedException
```

**Description**

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.

**Parameters**

- `exception`—The AIE object whose doInterruptible method is calling the `run` method. Used to invoke methods on AsynchronouslyInterruptedException from within the `run()` method.

interruptAction(AsynchronouslyInterruptedException)

**Signature**
public void
interruptAction(AsynchronouslyInterruptedException exception)

Description
This method is called by the system when the run() method is interrupted. By using this, the program logic can determine when the run() method completed normally or had its control asynchronously transferred to its caller.

Parameters
exception—The currently pending AIE. Used to invoke methods on AsynchronouslyInterruptedException from within the interruptAction() method.

12.3.2 Classes
12.3.2.1 AsynchronousControlGroup

public class AsynchronousControlGroup

Inheritance
java.lang.Object
  java.lang.ThreadGroup
    javax.realtime.RealtimeThreadGroup
      AsynchronousControlGroup

Description
An enhanced RealtimeThreadGroup in which asynchronous task termination can be performed. It defines a set of tasks, both instances of javax.realtime.Schedulable and java.lang.Thread, that can be terminated together. By combining this with a class loader, one can build a virtual process within a Java runtime environment.

See Section javax.realtime.ProcessingGroup

See Section javax.realtime.memory.MemoryGroup

12.3.2.1.1 Constructors
AsynchronousControlGroup(RealtimeThreadGroup, String)

Signature
public
AsynchronousControlGroup(RealtimeThreadGroup parent,
String name)

Description
Creates a new asynchronous control group with its scheduler type inherited from
parent.

Parameters
parent—The parent group of the new group
name—The name of the new group

Throws
IllegalArgumentException—when the parent ThreadGroup instance is not
an instance of RealtimeThreadGroup.

IllegalArgumentException—when the parent ThreadGroup instance is not
assignable to this.

AsynchronousControlGroup(String)

Signature
public
AsynchronousControlGroup(String name)
throws IllegalArgumentException,
IllegalArgumentException

Description
Creates a new group with the current ThreadGroup instance as its parent and
that parent’s scheduler type for its scheduler type. That parent must be an
instance of RealtimeThreadGroup. The primordial realtime thread group has
Scheduler.class as its scheduler type.

Parameters
name—The name of the new group

Throws

RTSJ 2.0 (Draft 66) 607
AsynchronouslyInterruptedException—when the parent ThreadGroup instance is not an instance of RealtimeThreadGroup.

IllegalAssignmentError—when the parent ThreadGroup instance is not assignable to this.

12.3.2.1.2 Methods

abort

Signature
public void abort()

Description
Terminate all tasks running in this thread group.

12.3.2.2 AsynchronouslyInterruptedException

public class AsynchronouslyInterruptedException

Inheritance
java.lang.Object
java.lang.Throwable
java.lang.Exception
javaint.InterruptedException
AsynchronouslyInterruptedException

Description
A special exception that is thrown in response to an attempt to asynchronously transfer the locus of control of a schedulable.

An instance of javax.realtime.Schedulable executing a method or constructor, which lists AsynchronouslyInterruptedException 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 Schedulable, s, when the s.interrupt() method is called or the fire method is called of an AIE for which s has a doInterruptible method call in progress.

When an asynchronous interrupt is generated when the target Schedulable is executing within an ATC-deferred section, the asynchronous interrupt becomes pending. A pending asynchronous interrupt is delivered when the target 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 java.lang.Thread.interrupted().

When javax.realtime.Schedulable.interrupt, 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 AsynchronouslyInterruptedException 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.

Schedulable.interrupt() generates the 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-instance of Schedulable.

Other sources (e.g., AsynchronouslyInterruptedException.fire() and Timed) will generate specific instances of AsynchronouslyInterruptedException which applications can identify and thus limit propagation.

12.3.2.2.1 Constructors
AsynchronouslyInterruptedException

**Signature**

```java
public AsynchronouslyInterruptedException()
```

**Description**

Creates an instance of `AsynchronouslyInterruptedException`.

**AsynchronouslyInterruptedException(String)**

**Signature**

```java
public AsynchronouslyInterruptedException(String message)
```

**Description**

Creates an instance of `AsynchronouslyInterruptedException`.

**Parameters**

- `message`—A message to identify this instance.

12.3.2.2.2 Methods

**getGeneric**

**Signature**

```java
public static javax.realtime.control.AsynchronouslyInterruptedException getGeneric()
```

**Description**

Gets the singleton system generic `AsynchronouslyInterruptedException` that is generated when `javax.realtime.Schedulable.interrupt()` is invoked.

**Throws**

610 RTSJ 2.0 (Draft 66)
IllegalTaskStateException—when the current thread context is not an instance of \texttt{javax.realtime.Schedulable}.

\textit{Returns}

the generic \texttt{AsynchronouslyInterruptedException}.

\textbf{enable}

\textit{Signature}

\begin{verbatim}
    public boolean enable()
\end{verbatim}

\textit{Description}

Enables the throwing of this exception. This method is valid only when the caller has a call to \texttt{doInterruptible} in progress. When invoked when no call to \texttt{doInterruptible} is in progress, \texttt{enable} returns \texttt{false} and does nothing.

\textit{Returns}

\texttt{true}, when this was disabled before the method was called and the call was invoked whilst the associated \texttt{doInterruptible} was in progress, and \texttt{false} otherwise.

\textbf{disable}

\textit{Signature}

\begin{verbatim}
    public synchronized boolean disable()
\end{verbatim}

\textit{Description}

Disables the throwing of this exception. When the \texttt{fire} 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 \texttt{doInterruptible} in progress. If invoked when no call to \texttt{doInterruptible} is in progress, \texttt{disable} returns \texttt{false} and does nothing.

\textit{Returns}

\texttt{true}, when this was enabled before the method was called and the call was invoked with the associated \texttt{doInterruptible} in progress, and \texttt{false} otherwise.
isEnabled

Signature
public boolean
isEnabled()

Description
Queries 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.

Returns
ture, when this is enabled and the method call was invoked in the context of the associated doInterruptible, and false otherwise.

fire

Signature
public boolean
fire()

Description
Generates 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) 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.
Behaves as if Thread.interrupt() were called on the task currently operating within this exception’s doInterruptible.

Returns
ture, when this is not disabled and it has an invocation of a doInterruptible in progress and there is no outstanding fire request, and false otherwise.

doInterruptible(Interruptible)

Signature
public boolean
doInterruptible(Interruptible logic)

**Description**

Executes the run() method of the given Interruptible. This method may be on the stack in exactly one javax.realtime.Schedulable 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 the given Interruptible is always entered with the exception in the enabled state, but that state can be modified with enable() and disable(), and the state can be observed with isEnabled().

This AIE is cleared on return from doInterruptible.

**Parameters**

logic—An instance of an Interruptible whose run() method will be called.

**Throws**

- IllegalTaskStateException—when called on the generic AsynchronouslyInterruptedException.
- IllegalRealtimeArgumentException—when logic is null.

**Returns**

true, when the method call completed normally, and false, when another call to doInterruptible has not completed.

**Available since** RTSJ 2.0 no longer throws an exception when called from a Java thread.

clear

**Signature**

    public boolean
clear()

**Description**

Atomically checks whether or not this is pending on the currently executing schedulable, and when so, makes it non-pending.

This method may be called at any time, and in particular need not be called in a try or catch block.

**Returns**

true, when this was pending, and false, when this was not pending.
Available since RTSJ 1.0.1

Available since RTSJ 2.0 no longer throws an exception when called from a task that is not an instance of `javax.realtime.Schedulable`.

**throwPending**

*Signature*

```java
public static void throwPending()
    throws AsynchronouslyInterruptedException
```

*Description*

Causes a pending `AsynchronouslyInterruptedException` to be thrown as a synchronous exception in an ATC-deferred region if one exists.

*Throws*

`AsynchronouslyInterruptedException`—if an AIE is pending.

Available since RTSJ 2.0

**fillInStackTrace**

*Signature*

```java
public java.lang.Throwable fillInStackTrace()
```

*Description*

Does nothing, since no stacktrace is kept.

*Returns*

this instance.

**setStackTrace(StackTraceElement)**

*Signature*

```java
public void setStackTrace(java.lang.StackTraceElement[] new_stackTrace)
```

614 RTSJ 2.0 (Draft 66)
AsynchronouslyInterruptedException

throws NullPointerException

Description
Does nothing, since no stacktrace is kept.

getStackTrace

Signature
public java.lang.StackTraceElement[]
getStackTrace()

Description
No stacktrace is kept, so none can be returned.

Returns
an empty array.

printStackTrace

Signature
public void
printStackTrace()

Description
No stacktrace is kept, so a message to that effect is printed.

printStackTrace(PrintStream)

Signature
public void
printStackTrace(PrintStream stream)

Description
No stacktrace is kept, so a message to that effect is printed.

Parameters
stream—A PrintStream for printing
printStackTrace(PrintWriter)

Signature
   public void
   printStackTrace(PrintWriter writer)

Description
   No stacktrace is kept, so a message to that effect is printed.

Parameters
   writer—A PrintWriter for printing

12.3.2.3 Timed

public class Timed

Inheritance
   java.lang.Object
      java.lang.Throwable
         java.lang.Exception
            java.lang.InterruptedException
               AsynchronouslyInterruptedException
                  Timed

Description
   Creates a scope in a javax.realtime.Schedulable object which will be asyn-
   chronously 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 when 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.

   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);
12.3.2.3.1 Constructors

**Timed(HighResolutionTime)**

*Signature*

```java
public Timed(javax.realtime.HighResolutionTime<?> time)
throws IllegalRealtimeArgumentException,
Un.supportedRealtimeOperationException
```

*Description*

Creates an instance of Timed with a timer set to time. When the time is in the past the AsynchronouslyInterruptedException mechanism is activated immediately after or when the doInterruptible method is called.

*Parameters*

time—When time is a javax.realtime.RelativeTime value, it is the interval of time between the invocation of doInterruptible and the time when the schedulable is asynchronously interrupted. When time is an javax.realtime.AbsoluteTime value, the timer asynchronously interrupts at this time (assuming the timer has not been cancelled).

*Throws*

IllegalRealtimeArgumentException—when time is null.

UnsupportedRealtimeOperationException—when time is not based on a javax.realtime.Clock.

12.3.2.3.2 Methods

**doInterruptible(Interruptible)**

*Signature*

```java
public boolean doInterruptible(Interruptible logic)
```

*Description*
Executes a timeout method by starting the timer and executing the \texttt{run()} method of the given \texttt{Interruptible} object.

\textbf{Parameters}
logic—An instance of an \texttt{Interruptible} whose \texttt{run()} method will be called.

\textbf{Throws}
\texttt{IllegalRealtimeArgumentException}—when \texttt{logic} is \texttt{null}.
\texttt{IllegalThreadStateException}—\texttt{null}

\textbf{Returns}
\texttt{true}, when the method call completed normally, and \texttt{false}, when another call to \texttt{doInterruptible} has not completed.

\textbf{resetTime(HighResolutionTime)}

\textbf{Signature}
\begin{verbatim}
public void resetTime(javax.realtime.HighResolutionTime<?> time)
\end{verbatim}

\textbf{Description}
Sets the timeout for the next invocation of \texttt{doInterruptible}.

\textbf{Parameters}
time—This can be an absolute time or a relative time. When \texttt{null} or not based on a \texttt{javax.realtime.Clock}, the timeout is not changed.

\textbf{restart(HighResolutionTime)}

\textbf{Signature}
\begin{verbatim}
public void restart(javax.realtime.HighResolutionTime<?> time)
\end{verbatim}

\textbf{Description}
Resets the timeout. When this \texttt{Timed} instance is executing, it adjusts the timeout to \texttt{time} and restarts the timer. When the instance is not executing, it adjusts the timeout for the next invocation.

\textbf{Parameters}
time—The new timeout.

\textbf{Throws}
IllegalRealtimeArgumentException—when time is null or a relative time less than zero.

UnsupportedRealtimeOperationException—when time is not based on a javax.realtime.Clock

Available since RTSJ 2.0

12.4 Rationale

The ability to interrupt a task asynchronously is necessary in many kinds of applications, but such a facility 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.

12.4.1 Asynchronous Transfer of Control

One basic decision was provide ATC, but only allow it in methods that explicitly permits it. The default of no ATC is reasonable, since most code is not written expecting 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 schedule s as an effect of calling s.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

```java
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().

### 12.4.2 Asynchronous Task Termination

The trend towards microservices are just as applicable to realtime and embedded
systems as to more conventional systems. It is often necessary to be able to terminate
a single service in the system without affecting the rest of the system.

Though Java provides class loaders and thread groups to help insulate one service
from another, there is no way of forcing a service to stop. ATT provides this, but
it can only provide limited safety, therefore it has been designed to take an entire
group of threads down at once. Using it together with a class loader can minimize
some of the problems associated with Thread.stop().

In order to help preserve the state of the rest of the system, two decisions were
made:

- all finally clauses are executed and
- only an entire realtime thread group can be aborted.

This means that malicious code could still prevent termination with faulty finalization
code. For instance, the code sequence `while(true);` can prevent termination.
Therefore, some static analysis and signing should be used for code that one might
want to terminate.
Chapter 13

Devices and Triggering

Interacting with the external environment in a timely manner is an important requirement for realtime, embedded systems. From an embedded system’s 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, version 2.0 has extended the goals of the device interfaces to be type safe and user extensible, so that the user can define 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.

This specification should be implementable for all of these.

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 the other execution environments, 2, 3, and 4, a 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

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 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 implementors. Interrupt service routines were not considered at all.

Version 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 Signal, as well as the new RealtimeSignal, are now unified under ActiveEvent. This means that Happening, Signal, and RealtimeSignal, like Timer are now subclasses of AsyncBaseEvent. 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, TimeDispatcher, SignalDispatcher, and RealtimeSignalDispatcher. Finally, provisions for interrupt service routines have been made as well.

13.1 Definitions

Direct Memory Access (DMA) — A data transfer directly to memory without CPU intervention, as in DMA controller.

DMA Controller — A device that can move data in memory without using the
Semantics 13.2

CPU.

**Happening** — 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.

**Interrupt Service Routine (ISR)** — A special task that is executed when an interrupt happens. This code runs with execution eligibility higher than normal execution eligibilities and can only be interrupted by another interrupt.

**Raw Memory** — A means of mapping memory locations, such as device registers, into Java objects for direct access from Java code without using JNI. The memory to map can be in an arbitrary address space.

**Raw Memory Region** — An address space for Raw Memory.

**Stride** — The distance between two memory locations. Adjacent memory locations have a stride of one. Stride is measured as units of the memory location size. For example, the stride between two bytes that are adjacent and two integers that are adjacent is both one, but the actual address offsets are one and four bytes respectively.

Open issue 13.1.1 (elb)

Check consistency with the JMM.

End of issue 13.1.1

13.2 Semantics

The classes in this Chapter are part of the Device Module introduced in Section 3.2.2.2 and are only required in implementations that include that module. 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 internal active events.

13.2.1 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.

1. Each area of memory supporting raw memory access is identified by a subclass of **RawMemoryRegion**.

RTSJ 2.0 (Draft 66)
Figure 13.1: Raw Memory Interface

```
javax.realtime::RawLong
  <<interface>>
    +getLong():long
    +getLong(int offset):long
    +get(int offset, long[] v):int
    +get(int offset, long[] v, int start, int count):int
    +address():long

javax.realtime::RawLongReader
  <<interface>>
    +getLong():long
    +getLong(int offset):long
    +get(int offset, long[] v):int
    +get(int offset, long[] v, int start, int count):int
    +address():long

javax.realtime::RawLongWriter
  <<interface>>
    +setLong(long v)
    +setLong(int offset, long v)
    +set(int offset, long[] v, int start, int count):int
    +address():long

javax.realtime::RawFloat
  <<interface>>
    +get():float
    +getDouble(int offset):double
    +get(int offset, double[] v):int
    +get(int offset, double[] v, int start, int count):int
    +address():long

javax.realtime::RawFloatReader
  <<interface>>
    +get():float
    +getDouble(int offset):double
    +get(int offset, double[] v):int
    +get(int offset, double[] v, int start, int count):int
    +address():long

javax.realtime::RawFloatWriter
  <<interface>>
    +setDouble(double v)
    +setDouble(int offset, double data)
    +set(int offset, double[] v, int start, int count):int
    +address():long

javax.realtime::RawDouble
  <<interface>>
    +setDouble(double v)
    +setDouble(int offset, double data)
    +set(int offset, double[] v, int start, int count):int
    +address():long

javax.realtime::RawDoubleReader
  <<interface>>
    +get():double
    +getDouble(int offset):double
    +get(int offset, double[] v):int
    +get(int offset, double[] v, int start, int count):int
    +address():long

javax.realtime::RawDoubleWriter
  <<interface>>
    +setDouble(double v)
    +setDouble(int offset, double data)
    +set(int offset, double[] v, int start, int count):int
    +address():long

javax.realtime::RawShort
  <<interface>>
    +getShort():short
    +getShort(int offset):short
    +get(int offset, short[] v):int
    +get(int offset, short[] v, int start, int count):int
    +address():long

javax.realtime::RawShortReader
  <<interface>>
    +getShort():short
    +getShort(int offset):short
    +get(int offset, short[] v):int
    +get(int offset, short[] v, int start, int count):int
    +address():long

javax.realtime::RawShortWriter
  <<interface>>
    +setShort(short v)
    +setShort(int offset, short v)
    +set(int offset, short[] v, int start, int count):int
    +address():long

javax.realtime::RawByte
  <<interface>>
    +getByte():byte
    +getByte(int offset):byte
    +get(int offset, byte[] v):int
    +get(int offset, byte[] v, int start, int count):int
    +address():long

javax.realtime::RawByteReader
  <<interface>>
    +getByte():byte
    +getByte(int offset):byte
    +get(int offset, byte[] v):int
    +get(int offset, byte[] v, int start, int count):int
    +address():long

javax.realtime::RawByteWriter
  <<interface>>
    +setByte(byte v)
    +setByte(int offset, byte v)
    +set(int offset, byte[] v, int start, int count):int
    +address():long

javax.realtime::RawInt
  <<interface>>
    +getInt():int
    +getInt(int offset):int
    +get(int offset, int[] v):int
    +get(int offset, int[] v, int start, int count):int
    +address():long

javax.realtime::RawIntReader
  <<interface>>
    +getInt():int
    +getInt(int offset):int
    +get(int offset, int[] v):int
    +get(int offset, int[] v, int start, int count):int
    +address():long

javax.realtime::RawIntWriter
  <<interface>>
    +setInt(int v)
    +setInt(int offset, int v)
    +set(int offset, int[] v, int start, int count):int
    +address():long

Visibility
+ = public
# = protected
~ = package
```
(a) The raw memory region \texttt{RawMemoryFactory.MEMORY\_MAPPED\_REGION} facilitates access to memory locations that are outside the main memory used by the JVM. It is used to access input and output device registers when such registers are memory mapped.

(b) The raw memory region \texttt{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.

(c) 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.

2. Access to raw memory is controlled by implementation-defined objects, called \textit{accessor objects}. These implement specification-defined interfaces (e.g.,\texttt{RawByte, RawShort, RawInt}, etc.) and are created by implementation-defined factory objects. Each factory implements the \texttt{RawMemoryRegionFactory} interface, and is identified by its \texttt{RawMemoryRegion}.

3. The \texttt{RawMemoryFactory} class defines the application programmer’s interface to the raw memory facilities.

4. The \texttt{RawMemoryRegionFactory} interface defines the interface that all factories must support for creating accessor objects.

### 13.2.1.1 Raw Memory Region

Raw memory is designed to support arbitrary I/O address spaces. The simplest of which is through the processor address space and is accessible via standard memory access instructions, such as \texttt{load} and \texttt{store}. This provides access to memory mapped I/O devices, but there are other address spaces as well. Each of these address spaces is referred to as a \textit{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 \texttt{in} and \texttt{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 have the necessary I/O instructions.

All other raw memory regions are optional and may be provided by a system integrator or an application developer. The API provides an interface, \texttt{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 emulate 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 from and writing to an I/O bus directly. The instances are used to get accessors of a region instead of using a `RawMemoryRegionFactory` directly.

### 13.2.1.2 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.

### 13.2.1.3 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.

### 13.2.2 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 for 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 are different than for other `java.nio` mechanisms.

These differences are covered with a special byte buffer factory: `DirectMemoryBufferFactory`. 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 `DirectMemoryBufferFactory` class also provides static methods for ensuring that Java-generated changes to DMA-mapped memory buffers are visible to native code, and vice versa.

### 13.2.3 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 handling 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 and 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.

#### 13.2.3.1 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 depicted in
As with other active events, `Happening` has its own dispatcher class: `HappeningDispatcher`. There is a 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 having 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 13.3 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 its 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).

There are three main differences between this mechanism and the string-based API.
1. The Happening class is now a first-class entity, rather than being buried in the implementation and identified only by a String object.
2. They include the Happening.trigger(int) method that enables a happening to be explicitly triggered by Java code, and at the implementation’s option, also include a native code function that permits native application code to trigger the happening.
3. Finally, Happening is a subclass of AsyncEvent, just as Timer, instead of being attached to an AsyncEvent.

13.2.4 Interrupt Service Routines

In Java-based systems, JNI is typically used to transfer control between an interrupt service routine (ISR) written in assembler or C and the program. RTSJ 2.0 supports
the possibility of the ISR being written in 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.

1. An occurrence of an interrupt consists of its generation and delivery.
2. Generation of the interrupt is the mechanism in the underlying hardware or system that makes the interrupt available to the Java program.
3. 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 the application native code linked with the JVM, or directly by the hardware interrupt mechanism.
4. Between generation and delivery, the interrupt is *pending*.

5. 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.

6. 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.

7. An application-defined ISR can be registered with one or more nonreserved 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.

8. 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`, the implementation must use the available interrupt masking facilities of the processor.

9. In order to safely share data between an interrupt service routine and other tasks, usually an Happening, a special monitor control policy is provided: `InterruptMasking`. When in use, it raises the eligibility of the entering task above all other task in the system and locks out the requisite interrupts. This synchronization is similar to priority ceiling emulation, as it always raises the eligibility of the entering task, but it also inhibits the interrupt for which it is configured, and potentially all less eligible interrupts.

10. The default allocation context of the `handle` method is the memory area passed during construction.

11. Any exception propagated from the `handle` method shall be caught by the JVM and ignored.

12. Code running in the context of an ISR may only attempt to acquire a lock that has a corresponding `InterruptMasking` as its monitor control policy. A `CeilingViolationException` is thrown, when an ISR attempts to acquire a lock with a different monitor control policy type.

13. An ISR object may be allocated in any memory area that does not move the handler. This includes immortal and scoped memory, but might not include heap memory on all systems.

14. As long as the ISR is registered, the memory area containing it is an execution context and thus may not be released.

The model assumes that
1. the processor has a (logical) interrupt controller that monitors a number of interrupt lines;
2. the interrupt controller may associate each interrupt line with a particular, but not necessarily unique, interrupt mask;
3. associated with the interrupt lines is a (logical) interrupt vector that contains the addresses of the ISRs;
4. the processor has instructions that enable interrupts from a particular line to be disabled or masked irrespective of the device attached or its type;
5. disabling interrupts from a specific line should disable the interrupts from lines having lower eligibility;
6. a device can be connected to an arbitrary interrupt line;
7. when an interrupt is signaled 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); and
8. on return from the ISR, interrupts are automatically re-enabled.

For interrupts, the RTSJ has an associated hardware priority that is more eligible that any task running on any scheduler. That priority can be used to set the eligibility of a task entering an object with a InterruptMasking monitor control policy. The RTSJ virtual machine may use this policy to disable the interrupts from the associated interrupt line and less eligible interrupts, as if the task was servicing the corresponding interrupt. On a multicore system, the situation is more complex, since there may be other cores available to handle other interrupts, even at lower eligibility, 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 realtime JVM may place restrictions on the code that can be written in this method. The process is illustrated in Figure 13.4, and is described below.

1. The ISR is instantiated by some application realtime thread for a given interrupt id.
2. The created ISR is registered with the JVM.
3. As part of the registration process, system dependent code sets up the underlying interrupt vectors linkage for calling 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 second level handler. The `InterruptServiceRoutine` mechanism is designed to work with `Happening`. In the simplest case, the `handle` method simply calls the `trigger` method on an instance of `Happening`. Though `handle` method could use `Object.notify` to wake a thread as an alternative, happenings are more flexible. In any case, using `Object.notifyAll` is not recommended and `Object.wait` should not be used at all.

### 13.2.4.1 Synchronization

Sometimes it is necessary to share data between happenings or between happenings and their associated interrupt service routines. Synchronization is the paradigm used in Java, but sharing data with handlers required not just eligibility inversion avoidance, but also interrupt masking. Since normal monitor control policies do not suffice for this, the device module provides additional monitor control policies for this: the abstract base type `InterruptMasking` along with its subtypes `InterruptInheritance` and `InterruptCeilingEmulation`.

### 13.2.4.2 Required Documentation

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 noninhibitable interrupts (if this is permitted).
2. Which runtime stack the `handle` method uses when it executes.
3. Any implementation-specific or hardware-specific activity that happens before the `handle` method is invoked, e.g., reading device registers or 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.
13.3  javax.realtime.device

13.3.1  Interfaces

13.3.1.1  DirectMemoryByteBuffer

public interface DirectMemoryByteBuffer

Description
An interface that can be implemented by a subclass of ByteBuffer for supporting DMA.

Available since RTSJ 2.0

13.3.1.1.1  Methods

isReadOnly

Signature
public boolean
isReadOnly()

Description
Determines whether or not one can write to the buffer.

Returns
true when and only when the buffer is read only.

duplicate

Signature
public javax.realtime.device.DirectMemoryByteBuffer
duplicate()
Creates a new memory buffer pointing to the same underlying memory. The content of the new buffer will remain the same. Changes to this buffer’s content will be visible in the new buffer and vice versa. Initially the two buffers’ position, limit, and mark values will be the same, but independent of one another. Changes to one will not be reflected in the other.

_Returns_
the new memory buffer

**get**

_Signature_

```
    public byte get()
```

_Description_
Obtains the byte at the current position and then increments the position.

_Returns_
the byte at the old position.

**get(int)**

_Signature_

```
    public byte get(int index)
```

Throws
_IndexOutOfBoundsException_—when index is negative or not smaller than the buffer’s limit.

_Returns_
the byte at index.
**getChar**

*Signature*

```java
public char
getChar()
```

*Description*

Obtains the char at the current position and then increments the position.

*Returns*

the char at the old position.

**getChar(int)**

*Signature*

```java
public char
getChar(int arg0)
```

*Description*

Obtains the char at index.

*Throws*

`IndexOutOfBoundsException`—when index is negative or not smaller than the buffer’s limit.

*Returns*

the char at index.

**getDouble**

*Signature*

```java
public double
getDouble()
```

*Description*

Obtains the double at the current position and then increments the position.

*Returns*

the double at the old position.
**getDouble(int)**

*Signature*
```
public double
g.getDouble(int arg0)
```

*Description*
Obtains the double at \textit{index}.

*Throws*
\texttt{IndexOutOfBoundsException}—when index is negative or not smaller than the buffer’s limit.

*Returns*
the double at \textit{index}.

**getFloat**

*Signature*
```
public float
getFloat()
```

*Description*
Obtains the float at the current position and then increments the position.

*Returns*
the float at the old position.

**getFloat(int)**

*Signature*
```
public float
getFloat(int arg0)
```

*Description*
Obtains the float at \textit{index}.

*Throws*
\texttt{IndexOutOfBoundsException}—when index is negative or not smaller than the buffer’s limit.

*Returns*
the float at \textit{index}.
**getInt**

*Signature*

```java
class DirectMemoryByteBuffer {
    public int getInt()
}
```

*Description*

Obtains the int at the current position and then increments the position.

*Returns*

the int at the old position.

**getInt(int)**

*Signature*

```java
class DirectMemoryByteBuffer {
    public int getInt(int arg0)
}
```

*Description*

Obtains the int at index.

*Throws*

`IllegalArgumentException`—when index is negative or not smaller than the buffer’s limit.

*Returns*

the int at index.

**getLong**

*Signature*

```java
class DirectMemoryByteBuffer {
    public long getLong()
}
```

*Description*

Obtains the long at the current position and then increments the position.

*Returns*

the long at the old position.
13 Devices and Triggering

DirectMemoryByteBuffer

**getLong(int)**

*Signature*

```java
class DirectMemoryByteBuffer {
    public long getLong(int arg0)
}
```

*Description*

Obtains the long at index.

*Throws*

- `IndexOutOfBoundsException`—when index is negative or not smaller than the buffer’s limit.

*Returns*

the long at index.

**getShort**

*Signature*

```java
class DirectMemoryByteBuffer {
    public short getShort()
}
```

*Description*

Obtains the short at the current position and then increments the position.

*Returns*

the short at the old position.

**getShort(int)**

*Signature*

```java
class DirectMemoryByteBuffer {
    public short getShort(int arg0)
}
```

*Description*

Obtains the short at index.

*Throws*

- `IndexOutOfBoundsException`—when index is negative or not smaller than the buffer’s limit.

*Returns*

the short at index.
put(byte)

Signature

public javax.realtime.device.DirectMemoryByteBuffer
      put(byte value)
      throws BufferOverflowException,
               ReadOnlyBufferException

Description

Sets the byte at the current position to value and then increments the position
by one.

Throws

BufferOverflowException—when the current position is not smaller than its limit.
ReadOnlyBufferException—when this buffer is read only.

Returns

this

put(int, byte)

Signature

public javax.realtime.device.DirectMemoryByteBuffer
      put(int index,
           byte value)
      throws BufferOverflowException,
               ReadOnlyBufferException

Description

Sets the byte at the index to value.

Throws

BufferOverflowException—when index is negative or not smaller than the
buffer’s limit.
ReadOnlyBufferException—when this buffer is read only.

putChar(char)

Signature
public javax.realtime.device.DirectMemoryByteBuffer
putChar(char arg0)

Description
Sets the char at the current position to value and then increments the position by one.

Throws
BufferOverflowException—when the current position is not smaller than its limit.
ReadOnlyBufferException—when this buffer is read only.

Returns
this

putChar(int, char)

Signature
public javax.realtime.device.DirectMemoryByteBuffer
putChar(int arg0,
        char arg1)

Description
Sets the char at the index to value.

Throws
BufferOverflowException—when index is negative or not smaller than the buffer’s limit.
ReadOnlyBufferException—when this buffer is read only.

putDouble(double)

Signature
public javax.realtime.device.DirectMemoryByteBuffer
putDouble(double arg0)

Description
Sets the double at the current position to value and then increments the position by one.

Throws
BufferOverflowException—when the current position is not smaller than its limit.
ReadOnlyBufferException—when this buffer is read only.

Returns
this

putDouble(int, double)

Signature
   public javax.realtime.device.DirectMemoryByteBuffer
      putDouble(int arg0,
                double arg1)

Description
   Sets the double at the index to value.

Throws
   BufferOverflowException—when index is negative or not smaller than the
       buffer’s limit.
   ReadOnlyBufferException—when this buffer is read only.

putFloat(float)

Signature
   public javax.realtime.device.DirectMemoryByteBuffer
      putFloat(float arg0)

Description
   Sets the float at the current position to value and then increments the position
       by one.

Throws
   BufferOverflowException—when the current position is not smaller than its limit.
   ReadOnlyBufferException—when this buffer is read only.

Returns
this
putFloat(int, float)

Signature
    public javax.realtime.device.DirectMemoryByteBuffer
    putFloat(int arg0,
              float arg1)

Description
    Sets the float at the index to value.

Throws
    BufferOverflowException—when index is negative or not smaller than the buffer’s limit.
    ReadOnlyBufferException—when this buffer is read only.

putInt(int)

Signature
    public javax.realtime.device.DirectMemoryByteBuffer
    putInt(int arg0)

Description
    Sets the int at the current position to value and then increments the position by one.

Throws
    BufferOverflowException—when the current position is not smaller than its limit.
    ReadOnlyBufferException—when this buffer is read only.

Returns
    this

putInt(int, int)

Signature
    public javax.realtime.device.DirectMemoryByteBuffer
    putInt(int arg0,
           int arg1)

Description
Sets the int at the index to value.

Throws
BufferOverflowException—when index is negative or not smaller than the buffer’s limit.
ReadOnlyBufferException—when this buffer is read only.

putLong(long)

Signature
public javax.realtime.device.DirectMemoryByteBuffer putLong(long arg0)

Description
Sets the long at the current position to value and then increments the position by one.

Throws
BufferOverflowException—when the current position is not smaller than its limit.
ReadOnlyBufferException—when this buffer is read only.

Returns
this

putLong(int, long)

Signature
public javax.realtime.device.DirectMemoryByteBuffer putLong(int arg0, long arg1)

Description
Sets the long at the index to value.

Throws
BufferOverflowException—when index is negative or not smaller than the buffer’s limit.
ReadOnlyBufferException—when this buffer is read only.
putShort(short)

Signature
public javax.realtime.device.DirectMemoryByteBuffer
putShort(short arg0)

Description
Sets the short at the current position to value and then increments the position by one.

Throws
BufferOverflowException—when the current position is not smaller than its limit.
ReadOnlyBufferException—when this buffer is read only.

Returns
this

putShort(int, short)

Signature
public javax.realtime.device.DirectMemoryByteBuffer
putShort(int arg0,
        short arg1)

Description
Sets the short at the index to value.

Throws
BufferOverflowException—when index is negative or not smaller than the buffer’s limit.
ReadOnlyBufferException—when this buffer is read only.

slice

Signature
public javax.realtime.device.DirectMemoryByteBuffer
slice()

Description
DirectMemoryByteBuffer

Creates a direct new byte buffer whose content is shared with a subsequence of this buffer’s content. The content of the new buffer will start at the current value of this buffer’s position. Changes to the content of the new buffer will be visible in this new buffer, and vice versa. The two buffers’ position, limit, and mark values are independent or one another. The new buffer’s position at start is zero, its capacity and its limit start at the number of bytes remaining in this buffer, and its mark is initially undefined.

**position**

*Signature*

```java
public int position()
```

*Description*

Determines the reference position of this buffer.

*Returns*

the current position

**position(int)**

*Signature*

```java
public javax.realtime.device.DirectMemoryByteBuffer position(int position)
throws IllegalRealtimeArgumentException
```

*Description*

Sets the reference position for this buffer. When the mark is defined and is larger than the new position, the mark becomes undefined.

*Parameters*

*position*—The new current position, which must be a natural number no larger than the current limit.

*Throws*

`javax.realtime.IllegalRealtimeArgumentException`—when the preconditions on *position* do not hold.

*Returns*

the buffer itself.
limit

Signature
   public int
   limit()

Description
   Determines the buffer's limit.

Returns
   the limit.

limit(int)

Signature
   public javax.realtime.device.DirectMemoryByteBuffer
   limit(int limit)
   throws IllegalRealtimeArgumentException

Description
   Sets this buffer's limit. When the position is larger than the new limit, position
   is set to the new limit. When the mark is defined and is larger than the new
   limit, the mark becomes undefined.

Parameters
   limit—The new limit value which must be a natural number no larger than this
           buffer's capacity

Throws
   javax.realtime.IllegalRealtimeArgumentException—when the preconditions
   on limit do not hold.

Returns
   the buffer itself

mark

Signature
   public javax.realtime.device.DirectMemoryByteBuffer
   mark()

Description
Sets this buffer’s mark to the current position.

Returns
the buffer itself.

reset

Signature
public javax.realtime.device.DirectMemoryByteBuffer
reset()
throws InvalidMarkException

Description
Resets this buffer’s position to the previously marked position leaving the mark value unchanged.

Throws
InvalidMarkException—when the mark is undefined.

Returns
the buffer itself

flip

Signature
public javax.realtime.device.DirectMemoryByteBuffer
flip()
throws IllegalRealtimeArgumentException

Description
Flips this buffer, i.e., the limit is set to the current position, then the position is set to zero, and the mark becomes undefined. After a sequence of channel-read or put operations, invoke this method to prepare for a sequence of channel-write or relative get operations. Here is an example.

buf.put(magic);    // Prepend header
in.read(buf);      // Read data into rest of buffer
buf.flip();        // Flip buffer
out.write(buf);    // Write header + data to channel

This method is often used in conjunction with the compact method when transferring data from one place to another.
13 Devices and Triggering

Returns
the buffer itself.

remaining

Signature
public int
remaining()

Description
Determines number of elements remaining in the buffer.

Returns
the number of elements between the current position and the limit.

hasRemaining

Signature
public boolean
hasRemaining()

Description
Determines whether or not there are any elements between the current position and the limit.

Returns
true when and only when there is at least one element between the current position and the limit.

13.3.1.2 RawByte

public interface RawByte

Interfaces
javax.realtime.device.RawByteReader
javax.realtime.device.RawByteWriter

Description

650 RTSJ 2.0 (Draft 66)
A marker for an object that can be used to access 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

13.3.1.3 RawByteReader

public interface RawByteReader

Interfaces
javax.realtime.device.RawMemory

Description
A marker for a byte accessor object encapsulating the protocol for reading bytes from raw memory. A byte accessor can always access at least one byte. Each byte is transferred in a single atomic operation. Groups of bytes may be transferred together; however, this is not required.

Objects of this type are created with the method RawMemoryFactory.createRawByteReader and RawMemoryFactory.createRawByte. Each object references a range of elements in the RawMemoryRegion starting at the base address provided to the factory method. The size provided to the factory method determines the number of accessible elements.

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

13.3.1.3.1 Methods

getByte

Signature

RTSJ 2.0 (Draft 66)
public byte
getByte()

Description
Gets the value at the first position referenced by this instance, i.e., the value at its start address. This operation must be atomic with respect to all other raw memory accesses to the address.

Returns
the value at the base address.

getByte(int)

Signature
public byte
getByte(int offset)
throws OffsetOutOfBoundsException

Description
Gets the value at the address: base address + offset * stride * element size in bytes. When an exception is thrown, no data is transferred.

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)

Signature
public int
get(int offset,
byte[] values)
throws OffsetOutOfBoundsException,
NullPointerException

652
RTSJ 2.0 (Draft 66)
Description
Fills values with elements from this instance, where the n\textsuperscript{th} element is at the address base address + (offset + n) \times stride \times element size in bytes. Only the bytes in the intersection of the start and end of values and the base address and the end of the memory region are transferred. When an exception is thrown, no data is transferred.

Parameters
offset—of the first byte in the memory region to transfer.
values—The array to receive 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 actually transferred to values.

get(int, byte, int, int)

Signature
public int
get(int offset,
byte[] values,
int start,
int count)
throws OffsetOutOfBoundsException,
ArrayIndexOutOfBoundsException,
NullPointerException

Description
Fills values from index start with elements from this instance, where the n\textsuperscript{th} element is at the address: base address + (offset+n) \times stride \times element size in bytes. The number of bytes transferred 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 transferred.

Parameters
offset—of the first byte in the memory region to transfer.
values—The array to receive 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 transferred.

13.3.1.4 RawByteWriter

public interface RawByteWriter

Interfaces

javax.realtime.device.RawMemory

Description

A marker for a byte accessor object encapsulating the protocol for writing bytes to raw memory. A byte accessor can always access at least one byte. Each byte is transferred in a single atomic operation. Groups of bytes may be transferred together; however, this is not required.

Objects of this type are created with the method RawMemoryFactory. createRawByteWriter and RawMemoryFactory.createRawByte. Each object references a range of elements in the RawMemoryRegion starting at the base address provided to the factory method. The size provided to the factory method determines the number of elements accessible.

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

13.3.1.4.1 Methods
setByte(byte)

Signature

    public void
    setByte(byte value)

Description

Sets 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.

Parameters

    value—The new value for the element.

setByte(int, byte)

Signature

    public void
    setByte(int offset,
            byte value)
        throws OffsetOutOfBoundsException

Description

Sets the value of the \( n^{th} \) element referenced by this instance, where \( n \) is \( offset \) and the address is \( base\ address + offset \times size\ of\ Byte \). This operation must be atomic with respect to all other raw memory accesses to the address. When an exception is thrown, no data is transferred.

Parameters

    offset—of byte in the memory region.
    value—The new value for the element.

Throws

    OffsetOutOfBoundsException—when \( offset \) is negative or greater than or equal to the number of elements in the raw memory region.

set(int, byte)

Signature
public int set(int offset, byte[] values) throws OffsetOutOfBoundsException, NullPointerException

Description
Copies from values to the memory region, from index start to elements where the nth element is at the address: base address + (offset + n) * stride * element size in bytes. Only the bytes in the intersection of values and the end of the memory region are transferred. When an exception is thrown, no data is transferred.

Parameters
offset—of first byte in the memory region to be set.
values—The source of the data to write.

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 actually transferred to values.

set(int, byte, int, int)

Signature
public int set(int offset, byte[] values, int start, int count) throws OffsetOutOfBoundsException, ArrayIndexOutOfBoundsException, NullPointerException

Description
Copies 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 transferred 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 transferred.

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 copy.
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 transferred.

13.3.1.5 RawDouble

public interface RawDouble

Interfaces
javax.realtime.device.RawDoubleReader
javax.realtime.device.RawDoubleWriter

Description
A marker for an object that can be used to access 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

13.3.1.6 RawDoubleReader

public interface RawDoubleReader

Interfaces
javax.realtime.device.RawMemory

Description
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 transferred in a single atomic operation. Groups of doubles may be transferred together; however, this is not required.

Objects of this type are created with the method RawMemoryFactory.createRawDoubleReader and RawMemoryFactory.createRawDouble. Each object references a range of elements in the RawMemoryRegion starting at the base address provided to the factory method. The size provided to the factory method determines the number of accessible elements.

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

13.3.1.6.1 Methods

getDouble

Signature
public double
getDouble()

Description
Gets the value at the first position referenced by this instance, i.e., the value at its start address. This operation must be atomic with respect to all other raw memory accesses to the address.

Returns
the value at the base address.

getDouble(int)

Signature
658

RTSJ 2.0 (Draft 66)
public double
    getDouble(int offset)
    throws OffsetOutOfBoundsException

Description
    Gets the value at the address: base address + offset * stride * element size in bytes. When an exception is thrown, no data is transferred.

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)

Signature
    public int
    get(int offset,
        double[] values)
    throws OffsetOutOfBoundsExcpetion,
            NullPointerException

Description
    Fills values with elements from this instance, where the n<sup>th</sup> element is at the address base address + (offset + n) * stride * element size in bytes. Only the doubles in the intersection of the start and end of values and the base address and the end of the memory region are transferred. When an exception is thrown, no data is transferred.

Parameters
    offset—of the first double in the memory region to transfer.
    values—The array to receive 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 actually transferred to `values`.

`get(int, double, int, int)`

Signature
```
public int
get(int offset,
    double[] values,
    int start,
    int count)
throws OffsetOutOfBoundsException,
       ArrayIndexOutOfBoundsException,
       NullPointerException
```

Description
Fills `values` from index `start` with elements from this instance, where the nth element is at the address: base address + (offset+n) * stride * element size in bytes. The number of bytes transferred 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 transferred.

Parameters
- `offset`—of the first double in the memory region to transfer.
- `values`—The array to receive 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 transferred.
13.3.1.7 RawDoubleWriter

public interface RawDoubleWriter

Interfaces
javax.realtime.device.RawMemory

Description
A marker for a double accessor object encapsulating the protocol for writing doubles to raw memory. A double accessor can always access at least one double. Each double is transferred in a single atomic operation. Groups of doubles may be transferred together; however, this is not required.

Objects of this type are created with the method RawMemoryFactory.createRawDoubleWriter and RawMemoryFactory.createRawDouble. Each object references a range of elements in the RawMemoryRegion starting at the base address provided to the factory method. The size provided to the factory method determines the number of elements accessible.

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

13.3.1.7.1 Methods

setDouble(double)

Signature
public void
setDouble(double value)

Description
Sets 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.

Parameters
value—The new value for the element.

**setDouble(int, double)**

*Signature*

```java
public void setDouble(int offset,
  double value)
  throws OffsetOutOfBoundsException
```

*Description*

Sets 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 transferred.

*Parameters*

- `offset`—of double in the memory region.
- `value`—The new value for the element.

*Throws*

- `OffsetOutOfBoundsException`—when `offset` is negative or greater than or equal to the number of elements in the raw memory region.

**set(int, double)**

*Signature*

```java
public int set(int offset,
  double[] values)
  throws OffsetOutOfBoundsException,
    NullPointerException
```

*Description*

Copies from `values` to the memory region, from index `start` to elements where the $n^{th}$ element is at the address: `base address + (offset + n) * stride * element size` in bytes. Only the doubles in the intersection of `values` and the end of the memory region are transferred. When an exception is thrown, no data is transferred.

*Parameters*
offset—of first double in the memory region to be set.
values—The source of the data to write.

Throws

OutOfBoundsException—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 actually transferred to values.

set(int, double, int, int)

Signature
public int
set(int offset,
    double[] values,
    int start,
    int count)
throws OffsetOutOfBoundsException,
        ArrayIndexOutOfBoundsException,
        NullPointerException

Description
Copies 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 transferred 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 transferred.

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 copy.
count—The maximum number of doubles to copy.

Throws

OutOfBoundsException—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 transferred.

13.3.1.8 RawFloat

public interface RawFloat

    Interfaces
    javax.realtime.device.RawFloatReader
    javax.realtime.device.RawFloatWriter

    Description
    A marker for an object that can be used to access 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

13.3.1.9 RawFloatReader

public interface RawFloatReader

    Interfaces
    javax.realtime.device.RawMemory

    Description
    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 transferred in a single atomic operation. Groups of floats may be transferred together; however, this is not required.

    Objects of this type are created with the method RawMemoryFactory.createRawFloatReader and RawMemoryFactory.createRawFloat. Each object references a range of elements in the RawMemoryRegion starting at the base address provided to the factory method. The size provided to the factory method determines the number of accessible elements.

    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

13.3.1.9.1 Methods

getFloat

Signature

class RawFloatReader

public float getFloat()

Description

Gets the value at the first position referenced by this instance, i.e., the value at its start address. This operation must be atomic with respect to all other raw memory accesses to the address.

Returns

the value at the base address.

getFloat(int)

Signature

class RawFloatReader

public float getFloat(int offset)

throws OffsetOutOfBoundsException

Description

Gets the value at the address: base address + offset * stride * element size in bytes. When an exception is thrown, no data is transferred.

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)**

*Signature*

```java
class RawFloatReader {
    public int get(int offset, float[] values) throws OffsetOutOfBoundsException, NullPointerException {
        // Implementation
    }
}
```

*Description*

Fills `values` with elements from this instance, where the \( n^{th} \) element is at the address `base address + (offset + n) \times \text{stride} \times \text{element size in bytes}`. Only the floats in the intersection of the start and end of `values` and the `base address` and the end of the memory region are transferred. When an exception is thrown, no data is transferred.

*Parameters*

- `offset`—of the first float in the memory region to transfer.
- `values`—The array to receive 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 actually transferred to `values`.

---

**get(int, float, int, int)**

*Signature*

```java
class RawFloatReader {
    public int get(int offset, float[] values, int start, int count) throws OffsetOutOfBoundsException, ArrayIndexOutOfBoundsException, NullPointerException {
        // Implementation
    }
}
```

*Description*

---

666 RTSJ 2.0 (Draft 66)
Fills values from index start with elements from this instance, where the n\textsuperscript{th} element is at the address: base address + (offset+n) \times \text{stride} \times \text{element size in bytes}. The number of bytes transferred 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 transferred.

**Parameters**
- offset—of the first float in the memory region to transfer.
- values—The array to receive 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 transferred.

### 13.3.1.10 RawFloatWriter

public interface RawFloatWriter

**Interfaces**
javax.realtime.device.RawMemory

**Description**
A marker for a float accessor object encapsulating the protocol for writing floats to raw memory. A float accessor can always access at least one float. Each float is transferred in a single atomic operation. Groups of floats may be transferred together; however, this is not required.

Objects of this type are created with the method RawMemoryFactory.createRawFloatWriter and RawMemoryFactory.createRawFloat. Each object references a range of elements in the RawMemoryRegion starting at the base address provided to the factory method. The size provided to the factory method determines the number of elements accessible.

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

### 13.3.1.10.1 Methods

#### setFloat(float)

**Signature**

```java
class RawFloatWriter {
    public void setFloat(float value)
}
```

**Description**

Sets 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.

**Parameters**

- `value`—The new value for the element.

#### setFloat(int, float)

**Signature**

```java
class RawFloatWriter {
    public void setFloat(int offset, float value)
        throws OffsetOutOfBoundsException
}
```

**Description**

Sets 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 transferred.

**Parameters**

- `offset`—of float in the memory region.
- `value`—The new value for the element.
Throws

OffsetOutOfBoundsException—when offset is negative or greater than or equal to the number of elements in the raw memory region.

set(int, float)

Signature

public int set(int offset, float[] values)
    throws OffsetOutOfBoundsException, NullPointerException

Description

Copies from values to the memory region, from index start to elements where the n\textsuperscript{th} element is at the address: base address + (offset + n) * stride * element size in bytes. Only the floats in the intersection of values and the end of the memory region are transferred. When an exception is thrown, no data is transferred.

Parameters

offset—of first float in the memory region to be set.
values—The source of the data to write.

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 actually transferred to values.

set(int, float, int, int)

Signature

public int set(int offset, float[] values, int start, int count)
13 Devices and Triggering

```java
throws OffsetOutOfBoundsException,
       ArrayIndexOutOfBoundsException,
       NullPointerException
```

**Description**

Copies `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 transferred 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 transferred.

**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 copy.
- `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 transferred.

### 13.3.1.11 RawInt

```java
public interface RawInt
```

**Interfaces**

- `javax.realtime.device.RawIntReader`
- `javax.realtime.device.RawIntWriter`

**Description**

A marker for an object that can be used to access 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
13.3.1.12 RawIntReader

public interface RawIntReader

Interfaces
javax.realtime.device.RawMemory

Description
A marker for a int accessor object encapsulating the protocol for reading ints from raw memory. A int accessor can always access at least one int. Each int is transferred in a single atomic operation. Groups of ints may be transferred together; however, this is not required.

Objects of this type are created with the method RawMemoryFactory.createRawIntReader and RawMemoryFactory.createRawInt. Each object references a range of elements in the RawMemoryRegion starting at the base address provided to the factory method. The size provided to the factory method determines the number of accessible elements.

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

13.3.1.12.1 Methods

getInt

Signature
   public int
   getInt()

Description
Gets the value at the first position referenced by this instance, i.e., the value at its start address. This operation must be atomic with respect to all other raw memory accesses to the address.

Returns
the value at the base address.

getInt(int)

Signature
public int
getInt(int offset)
throws OffsetOutOfBoundsException

Description
Gets the value at the address: base address + offset * stride * element size in bytes. When an exception is thrown, no data is transferred.

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)

Signature
public int
get(int offset,
    int[] values)
throws OffsetOutOfBoundsException,
    NullPointerException

Description
Fills values with elements from this instance, where the n\textsuperscript{th} element is at the address base address + (offset + n) * stride * element size in bytes. Only the ints in the intersection of the start and end of values and the base address and the end of the memory region are transferred. When an exception is thrown, no data is transferred.

Parameters
offset—of the first int in the memory region to transfer.
values—The array to receive 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 actually transferred to values.

get(int, int, int, int)

Signature

public int
get(int offset,
    int[] values,
    int start,
    int count)
throws OffsetOutOfBoundsException,
    ArrayIndexOutOfBoundsException,
    NullPointerException

Description

Fills values from index start with elements from this instance, where the nth element is at the address: base address + (offset+n) * stride * element size in bytes. The number of bytes transferred 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 transferred.

Parameters

offset—of the first int in the memory region to transfer.
values—The array to receive 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 transferred.

13.3.1.13 RawIntWriter

public interface RawIntWriter

 Interfaces
javax.realtime.device.RawMemory

 Description
A marker for a int accessor object encapsulating the protocol for writing ints
to raw memory. A int accessor can always access at least one int. Each int
is transferred in a single atomic operation. Groups of ints may be transferred
together; however, this is not required.

Objects of this type are created with the method RawMemoryFactory.
createRawIntWriter and RawMemoryFactory.createRawInt. Each object ref-
ences a range of elements in the RawMemoryRegion starting at the base address
provided to the factory method. The size provided to the factory method deter-
mines the number of elements accessible.

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

13.3.1.13.1 Methods

setInt(int)

 Signature
   public void
   setInt(int value)

 Description

674 RTSJ 2.0 (Draft 66)
Sets 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.

**Parameters**

*value*—The new value for the element.

### setInt(int, int)

**Signature**

```java
public void setInt(int offset, int value)
throws OffsetOutOfBoundsException
```

**Description**

Sets the value of the n\textsuperscript{th} 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 transferred.

**Parameters**

*offset*—of int in the memory region.

*value*—The new value for the element.

**Throws**

`OffsetOutOfBoundsException`—when `offset` is negative or greater than or equal to the number of elements in the raw memory region.

### set(int, int)

**Signature**

```java
public int set(int offset, int[] values)
throws OffsetOutOfBoundsException, NullPointerException
```

**Description**

Copies from `values` to the memory region, from index `start` to elements where the n\textsuperscript{th} element is at the address: base address + (offset + n) * stride * element size.
in bytes. Only the ints in the intersection of values and the end of the memory region are transferred. When an exception is thrown, no data is transferred.

Parameters
offset—of first int in the memory region to be set.
values—The source of the data to write.

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 actually transferred to values.

set(int, int, int, int)

Signature
public int
set(int offset,
    int[] values,
    int start,
    int count)
throws OffsetOutOfBoundsException,
        ArrayIndexOutOfBoundsException,
        NullPointerException

Description
Copies 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 transferred 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 transferred.

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 copy.
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 \texttt{start} is negative or either \texttt{start} or \texttt{start + count} is greater than or equal to the size of values.

NullPointerException—when \texttt{values} is null.

\textit{Returns}

the number of ints actually transferred.

### 13.3.1.14 RawLong

**public interface RawLong**

\textit{Interfaces}

javax.realtime.device.RawLongReader

javax.realtime.device.RawLongWriter

\textit{Description}

A marker for an object that can be used to access 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.

\textit{Available since} RTSJ 2.0

### 13.3.1.15 RawLongReader

**public interface RawLongReader**

\textit{Interfaces}

javax.realtime.device.RawMemory

\textit{Description}

A marker for a long accessor object encapsulating the protocol for reading longs from raw memory. A long accessor can always access at least one long. Each long is transferred in a single atomic operation. Groups of longs may be transferred together; however, this is not required.

Objects of this type are created with the method \texttt{RawMemoryFactory.createRawLongReader} and \texttt{RawMemoryFactory.createRawLong}. Each object references a range of elements in the \texttt{RawMemoryRegion} starting at the \textit{base address} provided to the factory method. The size provided to the factory method determines the number of accessible elements.
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

13.3.1.15.1 Methods

getLong

Signature

public long
getLong()

Description

Gets the value at the first position referenced by this instance, i.e., the value at its start address. This operation must be atomic with respect to all other raw memory accesses to the address.

Returns

the value at the base address.

getLong(int)

Signature

public long
getLong(int offset)
throws OffsetOutOfBoundsException

Description

Gets the value at the address: base address + offset * stride * element size in bytes. When an exception is thrown, no data is transferred.

Parameters

offset—of long in the memory region starting from the address specified in the associated factory method.

Throws

678

RTSJ 2.0 (Draft 66)
**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)**

**Signature**
```java
public int get(int offset, long[] values) throws OffsetOutOfBoundsException, NullPointerException
```

**Description**
Fills values with elements from this instance, where the n\(^{th}\) element is at the address base address + (offset + n) * stride * element size in bytes. Only the longs in the intersection of the start and end of values and the base address and the end of the memory region are transferred. When an exception is thrown, no data is transferred.

**Parameters**
offset—of the first long in the memory region to transfer.
values—The array to receive 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 actually transferred to values.

**get(int, long, int, int)**

**Signature**
```java
public int get(int offset, long[] values, int start, int count)
```
throws OffsetOutOfBoundsException,
        ArrayIndexOutOfBoundsException,
        NullPointerException

Description
Fills values from index start with elements from this instance, where the n\textsuperscript{th}
        element is at the address: base address + (offset+n) \times \text{stride} \times \text{element size in}
bytes. The number of bytes transferred 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 transferred.

Parameters
offset — of the first long in the memory region to transfer.
values — The array to receive 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 transferred.

13.3.1.16 RawLongWriter

public interface RawLongWriter

        Interfaces
javax.realtime.device.RawMemory

Description
A marker for a long accessor object encapsulating the protocol for writing longs
to raw memory. A long accessor can always access at least one long. Each long
is transferred in a single atomic operation. Groups of longs may be transferred
together; however, this is not required.

Objects of this type are created with the method RawMemoryFactory.
createRawLongWriter and RawMemoryFactory.createRawLong. Each object
references a range of elements in the `RawMemoryRegion` starting at the base address provided to the factory method. The size provided to the factory method determines the number of elements accessible.

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

### 13.3.1.16.1 Methods

#### setLong(long)

**Signature**

```java
public void setLong(long value)
```

**Description**

Sets 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.

**Parameters**

- `value`—The new value for the element.

#### setLong(int, long)

**Signature**

```java
public void setLong(int offset,
                    long value)
    throws OffsetOutOfBoundsException
```

**Description**

Sets the value of the \(n\)th 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 transferred.

Parameters
offset—of long in the memory region.
value—The new value for the element.

Throws
OffsetOutOfBoundsException—when offset is negative or greater than or equal to the number of elements in the raw memory region.

set(int, long)

Signature
public int set(int offset,
    long[] values)
throws OffsetOutOfBoundsException,
        NullPointerException

Description
Copies from values to the memory region, from index start to elements where the n\textsuperscript{th} element is at the address: base address + (offset + n) \times \text{stride} \times \text{element size in bytes}. Only the longs in the intersection of values and the end of the memory region are transferred. When an exception is thrown, no data is transferred.

Parameters
offset—of first long in the memory region to be set.
values—The source of the data to write.

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 actually transferred to values.

set(int, long, int, int)

Signature

682   RTSJ 2.0 (Draft 66)
public int
set(int offset,
    long[] values,
    int start,
    int count)
throws OffsetOutOfBoundsException,
    ArrayIndexOutOfBoundsException,
    NullPointerException

Description
Copies 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 transferred 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 transferred.

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 copy.
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 transferred.

13.3.1.17 RawMemory

public interface RawMemory

Description
A marker for all raw memory accessor objects.

Available since RTSJ 2.0
13.3.1.17.1 Methods

**getAddress**

*Signature*

```java
public long getAddress()
```

*Description*

Gets the base physical address of this object.

*Returns*

the first physical address this raw memory object can access.

**getSize**

*Signature*

```java
public int getSize()
```

*Description*

Gets the number of bytes that this object spans.

*Returns*

the size of this raw memory.

**getStride**

*Signature*

```java
public int getStride()
```

*Description*

Gets the distance between elements in multiples of element size.

*Returns*

the span between elements of this raw memory.
13.3.1.18 RawMemoryRegionFactory

public interface RawMemoryRegionFactory

Description
A means of giving an application the ability to provide support for a RawMemoryRegion that is not already provided by the standard. An instance of this interface can be registered with a RawMemoryFactory and provide the object that that factory should return for with a given RawMemoryRegion. It is responsible for checking all requests and throwing the proper exception when a request is invalid or the requester is not authorized to make the request.

Available since RTSJ 2.0

13.3.1.18.1 Methods

getRegion

Signature
public javax.realtime.device.RawMemoryRegion getRegion()

Description
Determines for what region this factory creates raw memory objects.

Returns
the region of this factory.

getName

Signature
public java.lang.String getName()

Description
Determines the name of the region for which this factory creates raw memory objects.

Returns
the name of the region of this factory.

createRawByte(long, int, int)

Signature
public javax.realtime.device.RawByte createRawByte(long base,
int count,
int stride)
throws SecurityException,
OutOfBoundsException,
SizeOutOfBoundsException,
UnsupportedRawMemoryRegionException,
MemoryTypeConflictException

Description
Creates an instance of a class that implements RawByte and accesses memory of getRegion in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride \times \text{size of RawByte} \times \text{count}. The object is allocated in the current memory area of the calling thread.

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 multiple 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.
OutOfBoundsException—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` and supports access to the specified range in
the memory region.

Available since RTSJ 2.0

createRawByteReader(long, int, int)

Signature
```
public javax.realtime.device.RawByteReader
createRawByteReader(long base,
    int count,
    int stride)
```

throws SecurityException,
OffsetOutOfBoundsException,
SizeOutOfBoundsException,
UnsupportedRawMemoryRegionException,
MemoryTypeConflictException

Description
Creates an instance of a class that implements `RawByteReader` and accesses
memory of `getRegion` 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.

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 multiple 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 and supports access to the specified range in the memory region.

Available since RTSJ 2.0

createRawByteWriter(long, int, int)

Signature
public javax.realtime.device.RawByteWriter
createRawByteWriter(long base,
    int count,
    int stride)
throws SecurityException,
    OffsetOutOfBoundsException,
    SizeOutOfBoundsException,
    UnsupportedRawMemoryRegionException,
    MemoryTypeConflictException

Description
Creates an instance of a class that implements RawByteWriter and accesses memory of getRegion 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.

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 multiple 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 and supports access to the specified range in the memory region.

Available since RTSJ 2.0

createRawShort(long, int, int)

Signature
public javax.realtime.device.RawShort createRawShort(long base,
int count,
int stride)
throws SecurityException,
OffsetOutOfBoundsException,
SizeOutOfBoundsException,
UnsupportedRawMemoryRegionException,
MemoryTypeConflictException

Description
Creates an instance of a class that implements RawShort and accesses memory of getRegion 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.

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 multiple 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 and supports access to the specified range in the memory region.

Available since RTSJ 2.0

createRawShortReader(long, int, int)

Signature
dpublic javax.realtime.device.RawShortReader
createRawShortReader(long base,
        int count,
        int stride)

throws SecurityException,
    OffsetOutOfBoundsException,
    SizeOutOfBoundsException,
    UnsupportedRawMemoryRegionException,
    MemoryTypeConflictException

Description
Creates an instance of a class that implements RawShortReader and accesses memory of getRegion 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.

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 and supports access to the specified range in the memory region.

Available since RTSJ 2.0

createRawShortWriter(long, int, int)

Signature
public javax.realtime.device.RawShortWriter createRawShortWriter(long base,
int count,
int stride)
throws SecurityException,
OffsetOutOfBoundsException,
SizeOutOfBoundsException,
UnsupportedRawMemoryRegionException,
MemoryTypeConflictException

Description
Creates an instance of a class that implements RawShortWriter and accesses memory of getRegion 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.

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 multiple 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.

OutOfBoundsException—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 and supports access to the specified range in the memory region.

Available since RTSJ 2.0

createRawInt(long, int, int)

Signature

public javax.realtime.device.RawInt
createRawInt(long base,
     int count,
     int stride)
throws SecurityException,
       OffsetOutOfBoundsException,
       SizeOutOfBoundsException,
       UnsupportedRawMemoryRegionException,
       MemoryTypeConflictException

Description

Creates an instance of a class that implements RawInt and accesses memory of getRegion 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.

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 multiple 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.
IllegalArgumentException—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 and supports access to the specified range in the memory region.

Available since RTSJ 2.0

createRawIntReader(long, int, int)

Signature
public javax.realtime.device.RawIntReader createRawIntReader(long base,
int count,
int stride)
throws SecurityException,
OffsetOutOfBoundsException,
SizeOutOfBoundsException,
UnsupportedRawMemoryRegionException,
MemoryTypeConflictException

Description
Creates an instance of a class that implements RawIntReader and accesses memory of getRegion 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.

Parameters
base—The starting physical address accessible through the returned instance.
count—The number of memory elements accessible through the returned instance.
**13 Devices and Triggering**

**RawMemoryRegionFactory**

**stride**—The distance to the next element in multiple 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.
- `OutOfBoundsException`—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` and supports access to the specified range in the memory region.

*Available since RTSJ 2.0*

**createRawIntWriter(long, int, int)**

*Signature*

```java
public javax.realtime.device.RawIntWriter createRawIntWriter(long base,
        int count,
        int stride)
throws SecurityException,
        OffsetOutOfBoundsException,
        SizeOutOfBoundsException,
        UnsupportedRawMemoryRegionException,
        MemoryTypeConflictException
```

*Description*

Creates an instance of a class that implements `RawIntWriter` and accesses memory of `getRegion` 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.

*Parameters*
**createRawLong(long, int, int)**

**Signature**

```java
define javax.realtime.device.RawLong createRawLong(long base,
    int count,
    int stride)
throws SecurityException,
    OffsetOutOfBoundsException,
    SizeOutOfBoundsException,
    UnsupportedRawMemoryRegionException,
    MemoryTypeConflictException
```

**Description**

Creates an instance of a class that implements `RawLong` and accesses memory of `getRegion` 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.
### 13 Devices and Triggering

**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 multiple 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` and supports access to the specified range in the memory region.

**Available since** RTSJ 2.0

### `createRawLongReader(long, int, int)`

**Signature**
```
public javax.realtime.device.RawLongReader createRawLongReader(long base,
                   int count,
                   int stride)
```

**Throws**
- `SecurityException`,
- `OffsetOutOfBoundsException`,
- `SizeOutOfBoundsException`,
- `UnsupportedRawMemoryRegionException`,
- `MemoryTypeConflictException`

**Description**
Creates an instance of a class that implements `RawLongReader` and accesses memory of `getRegion` 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.

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 multiple 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 and supports access to the specified range in the memory region.

Available since RTSJ 2.0

createRawLongWriter(long, int, int)

Signature
public javax.realtime.device.RawLongWriter
createRawLongWriter(long base,
        int count,
        int stride)

throws SecurityException,
OffsetOutOfBoundsException,
SizeOutOfBoundsException,
UnsupportedRawMemoryRegionException,
MemoryTypeConflictException

Description
RTSJ 2.0 (Draft 66) 697
13 Devices and Triggering

RawMemoryRegionFactory

Creates an instance of a class that implements RawLongWriter and accesses memory of getRegion 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.

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 multiple 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.
OutOfBoundsException—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 and supports access to the specified range in the memory region.

Available since RTSJ 2.0

createRawFloat(long, int, int)

Signature

public javax.realtime.device.RawFloat createRawFloat(long base,
        int count,
        int stride)
throws SecurityException,
        OffsetOutOfBoundsException,
        SizeOutOfBoundsException,
        UnsupportedRawMemoryRegionException,
        MemoryTypeConflictException

698 RTSJ 2.0 (Draft 66)
Description

Creates an instance of a class that implements RawFloat and accesses memory of getRegion in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is \( \text{stride} \times \text{size of RawFloat} \times \text{count} \). The object is allocated in the current memory area of the calling thread.

Parameters

- \text{base}—The starting physical address accessible through the returned instance.
- \text{count}—The number of memory elements accessible through the returned instance.
- \text{stride}—The distance to the next element in multiple 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 and supports access to the specified range in the memory region.

Available since RTSJ 2.0

createRawFloatReader(long, int, int)

Signature

public javax.realtime.device.RawFloatReader
createRawFloatReader(long base,
int count,
int stride)
throws SecurityException,
OffsetOutOfBoundsException,
SizeOutOfBoundsException,
UnsupportedRawMemoryRegionException,
MemoryTypeConflictException

Description
Creates an instance of a class that implements `RawFloatReader` and accesses memory of `getRegion` 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.

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 multiple 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.
- `OutOfBoundsException`—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` and supports access to the specified range in the memory region.

Available since RTSJ 2.0

createRawFloatWriter(long, int, int)

Signature
```java
public javax.realtime.device.RawFloatWriter createRawFloatWriter(long base,
                      int count,
                      int stride)
```

700 RTSJ 2.0 (Draft 66)
Description

Creates an instance of a class that implements `RawFloatWriter` and accesses memory of `getRegion` 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.

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 multiple 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.
- `OffsetOutOf BoundsException`—when `base` is invalid.
- `SizeOutOf BoundsException`—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` and supports access to the specified range in the memory region.

Available since RTSJ 2.0

createRawDouble(long, int, int)

Signature

```java
public javax.realtime.device.RawDouble createRawDouble(long base,
```
int count,
int stride)
throws SecurityException,
    OffsetOutOfBoundsException,
    SizeOutOfBoundsException,
    UnsupportedRawMemoryRegionException,
    MemoryTypeConflictException

Description

Creates an instance of a class that implements RawDouble and accesses memory of getRegion in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is \( \text{stride} \times \text{size of RawDouble} \times \text{count} \). The object is allocated in the current memory area of the calling thread.

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 multiple 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 and supports access to the specified range in the memory region.

Available since RTSJ 2.0

createRawDoubleReader(long, int, int)

Signature

702 RTSJ 2.0 (Draft 66)
public javax.realtime.device.RawDoubleReader
createRawDoubleReader(long base,
   int count,
   int stride)
throws SecurityException,
   OffsetOutOfBoundsException,
   SizeOutOfBoundsException,
   UnsupportedRawMemoryRegionException,
   MemoryTypeConflictException

Description

Creates an instance of a class that implements RawDoubleReader and accesses memory of getRegion 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.

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 multiple 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 and supports access to the specified range in the memory region.

Available since RTSJ 2.0
createRawDoubleWriter(long, int, int)

Signature

public javax.realtime.device.RawDoubleWriter
createRawDoubleWriter(long base,
int count,
int stride)

throws SecurityException,
OffsetOutOfBoundsException,
SizeOutOfBoundsException,
UnsupportedRawMemoryRegionException,
MemoryTypeConflictException

Description

Creates an instance of a class that implements RawDoubleWriter and accesses memory of getRegion in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is \( \text{stride} \times \text{size of RawDoubleWriter} \times \text{count} \). The object is allocated in the current memory area of the calling thread.

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 multiple 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 and supports access to the specified range in the memory region.
Available since RTSJ 2.0

13.3.1.19 RawShort

```java
public interface RawShort

    Interfaces
    javax.realtime.device.RawShortReader
    javax.realtime.device.RawShortWriter

Description
A marker for an object that can be used to access 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

13.3.1.20 RawShortReader

```java
public interface RawShortReader

    Interfaces
    javax.realtime.device.RawMemory

Description
A marker for a short accessor object encapsulating the protocol for reading
shorts from raw memory. A short accessor can always access at least one short.
Each short is transferred in a single atomic operation. Groups of shorts may be
transferred together; however, this is not required.

Objects of this type are created with the method `RawMemoryFactory.createRawShortReader` and `RawMemoryFactory.createRawShort`. Each object
references a range of elements in the `RawMemoryRegion` starting at the `base address` provided to the factory method. The size provided to the factory method
determines the number of accessible elements.

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

### 13.3.1.20.1 Methods

#### getShort

**Signature**

```java
public short getShort()
```

**Description**

Gets the value at the first position referenced by this instance, i.e., the value at its start address. This operation must be atomic with respect to all other raw memory accesses to the address.

**Returns**

the value at the *base address*.

#### getShort(int)

**Signature**

```java
public short getShort(int offset)
```

**Description**

Gets the value at the address: base address + offset * stride * element size in bytes. When an exception is thrown, no data is transferred.

**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)

Signature

```java
public int get(int offset,
        short[] values)
        throws OffsetOutOfBoundsException,
        NullPointerException
```

Description

Fills `values` with elements from this instance, where the \(n^{th}\) element is at the address `base address + (offset + n) \times \text{stride} \times \text{element size in bytes}`. Only the shorts in the intersection of the start and end of `values` and the `base address` and the end of the memory region are transferred. When an exception is thrown, no data is transferred.

Parameters

- `offset`—of the first short in the memory region to transfer.
- `values`—The array to receive 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 actually transferred to `values`.

get(int, short, int, int)

Signature

```java
public int get(int offset,
        short[] values,
        int start,
        int count)
        throws OffsetOutOfBoundsException,
        ArrayIndexOutOfBoundsException,
        NullPointerException
```

Description

RTSJ 2.0 (Draft 66) 707
Fills values from index start with elements from this instance, where the n\text{th} element is at the address: base address + (offset+n) \ast stride \ast element size in bytes. The number of bytes transferred 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 transferred.

Parameters

offset—of the first short in the memory region to transfer.
values—The array to receive 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 transferred.

13.3.1.21 RawShortWriter

public interface RawShortWriter

Interfaces
javax.realtime.device.RawMemory

Description
A marker for a short accessor object encapsulating the protocol for writing shorts to raw memory. A short accessor can always access at least one short. Each short is transferred in a single atomic operation. Groups of shorts may be transferred together; however, this is not required.

Objects of this type are created with the method RawMemoryFactory.createRawShortWriter and RawMemoryFactory.createRawShort. Each object references a range of elements in the RawMemoryRegion starting at the base address provided to the factory method. The size provided to the factory method determines the number of elements accessible.

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

### 13.3.1.21.1 Methods

#### setShort(short)

**Signature**

```java
public void setShort(short value)
```

**Description**

Sets 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.

**Parameters**

- `value`—The new value for the element.

#### setShort(int, short)

**Signature**

```java
public void setShort(int offset,
                     short value)
    throws OffsetOutOfBoundsException
```

**Description**

Sets the value of the \( n^{th} \) 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 transferred.

**Parameters**

- `offset`—of short in the memory region.
- `value`—The new value for the element.
13 Devices and Triggering

Throws
OffsetOutOfBoundsException—when offset is negative or greater than or equal to the number of elements in the raw memory region.

set(int, short)

Signature
public int
set(int offset,
    short[] values)
throws OffsetOutOfBoundsException,
    NullPointerException

Description
Copies from values to the memory region, from index start to elements where the n\textsuperscript{th} element is at the address: base address + (offset + n) \times stride \times element size in bytes. Only the shorts in the intersection of values and the end of the memory region are transferred. When an exception is thrown, no data is transferred.

Parameters
offset—of first short in the memory region to be set.
values—The source of the data to write.

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 actually transferred to values.

set(int, short, int, int)

Signature
public int
set(int offset,
    short[] values,
    int start,
    int count)
throws OffsetOutOfBoundsException,
     ArrayIndexOutOfBoundsException,
     NullPointerException

Description
Copies 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 transferred 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 transferred.

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 copy.
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 transferred.

13.3.2 Classes
13.3.2.1 DirectMemoryBufferFactory

public class DirectMemoryBufferFactory

Inheritance
java.lang.Object
     DirectMemoryBufferFactory

Description
A factory class for generating raw byte buffers. This enables the infrastructure to limit the address ranges from which a buffer may be taken. The address range managed by a DirectMemoryBufferFactory instance may overlap that of another DirectMemoryBufferFactory instance.
Available since RTSJ 2.0

13.3.2.1.1 Constructors

DirectMemoryBufferFactory(DirectMemoryRegion, long, long)

Signature
public
   DirectMemoryBufferFactory(DirectMemoryRegion region,
   long base,
   long size)
   throws MemoryInUseException,
            RangeOutOfBoundsException

Description
Creates a factory for allocating buffers in a particular address range. Whether
the address is physical or virtual is system dependent.

Parameters
region—The area of memory a DMA controller can reference, from which this
   factory takes its memory.
base—The base address of a memory range in region for buffer allocation.
size—The number of bytes in the memory range.

Throws
MemoryInUseException—when the memory area provided is already in use by or
   reserved for a javax.realtime.MemoryArea, program code, or other sytem or
   VM structure.
RangeOutOfBoundsException—when the memory region overlaps with another or
   cannot be used for DMA.

13.3.2.1.2 Methods
allocateByteBuffer(int)

Signature
    public javax.realtime.device.DirectMemoryByteBuffer
    allocateByteBuffer(int capacity)

Description
    Creates a direct byte buffer with the given capacity within the range of this
    factory.

Parameters
    capacity—The number of bytes in the buffer.

Throws
    javax.realtime.OutOfMemoryStaticError—when no memory is available.

Returns
    the new buffer.

free(DirectMemoryByteBuffer)

Signature
    public void
    free(DirectMemoryByteBuffer buffer)

Description
    Frees the memory associated with the given DirectMemoryByteBuffer instance.
    The capacity and limit of the buffer are both set to zero, so data can no longer
    be transferred with the buffer. The buffer range can then be safely reallocated.

Parameters
    buffer—The DirectMemoryByteBuffer to free.

Throws
    javax.realtime.IllegalRealtimeArgumentException—when buffer was not
    allocated from this factory.
    javax.realtime.IllegalRealtimeStateException—when buffer has already
    been freed.

Open issue 13.3.1
    Unless we get a better answer from the JDK team, we must have our own buffer
    class!
End of issue 13.3.1
inRange(DirectMemoryByteBuffer)

**Signature**

public boolean
inRange(DirectMemoryByteBuffer buffer)

**Description**

Checks to see whether or not the buffer’s data area is within the range of this factory.

**Parameters**

- buffer—The DirectMemoryByteBuffer to check.

**Returns**

true when and only when the buffer’s data area is within the range of this factory; otherwise false.

addressOf(DirectMemoryByteBuffer)

**Signature**

public long
addressOf(DirectMemoryByteBuffer buffer)

**Description**

Gives the location of this buffer’s data in memory. The address shall be in the address space of the DMA controller.

**Parameters**

- buffer—The DirectMemoryByteBuffer of which to get the address.

**Returns**

the start address of the data range of this buffer.

writeFence(DirectMemoryByteBuffer)

**Signature**

public static void
writeFence(DirectMemoryByteBuffer buffer)

**Description**

714 RTSJ 2.0 (Draft 66)
Ensures that all changes to the `DirectMemoryByteBuffer` 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 shall 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.

**Parameters**

- `buffer`—The byte buffer which will be flushed.

### readFence(DirectMemoryByteBuffer)

**Signature**

```
public static void readFence(DirectMemoryByteBuffer buffer)
```

**Description**

Ensures that any previous changes to the memory represented by the given `DirectMemoryByteBuffer` 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 shall 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(DirectMemoryByteBuffer)` or its native equivalent on the buffer’s memory.

**Parameters**

- `buffer`—The byte buffer which will be updated.

### 13.3.2.2 DirectMemoryRegion

```
public class DirectMemoryRegion extends java.lang.Object {

   // Class implementation

   // Constructor
   public DirectMemoryRegion() {
      // Constructor implementation
   }

   // Method implementations
   // ...

   // Inherited methods from Object
   // ...

   // Additional methods specific to DirectMemoryRegion
   // ...

   // Class description
   RTSJ 2.0 (Draft 66) 715

   // Class description
}
```

Inheritance

- `java.lang.Object`
  - `DirectMemoryRegion`
13 Devices and Triggering

DirectMemoryRegion

Defines the reachable memory for a given DMA controller in terms of the physical
address space of the system.

Available since RTSJ 2.0

13.3.2.2.1 Constructors

DirectMemoryRegion(long, long)

Signature
public
DirectMemoryRegion(long start,
    long size)
throws IllegalRealtimeArgumentException

Description
Creates a DMA memory definition.

Parameters
start—The DMA address space in the physical address space of the main processor.
size—The number of bytes in the DMA address space.

Throws
javax.realtime.IllegalRealtimeArgumentException—when start is less than
zero or start + size is larger than the physical memory of the system.

13.3.2.2.2 Methods

regionAddressOf(long)

Signature
public long
regionAddressOf(long address)
throws IllegalRealtimeArgumentException

716 RTSJ 2.0 (Draft 66)
Happening

javax.realtime.device 13.3

Description

Translates a physical address into a DMA region address.

Parameters

t—The address to translate.

Throws

d—when the result is outside the DMA space.

Returns

the equivalent address in DMA space.

physicalAddressOf(long)

Signature

public long
physicalAddressOf(long address)
throws IllegalRealtimeArgumentException

Description

Translates a DMA space address into a physical address.

Parameters

t—The address to translate.

Throws

d—when the input is outside the DMA space.

Returns

the corresponding physical address.

13.3.2.3 Happening

public class Happening

Inheritance

java.lang.Object
javax.realtime.AsyncBaseEvent
javax.realtime.AsyncEvent
Happening
Interfaces
javax.realtime.ActiveEvent

Description
This class provides second level handling for external events such as interrupts. A happening can be triggered by an InterruptServiceRoutine 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 provided by the system which is a string beginning with @.

Available since RTSJ 2.0

13.3.2.3.1 Constructors

Happening(String, HappeningDispatcher)

Signature
public
Happening(String name,
            HappeningDispatcher dispatcher)
throws IllegalRealtimeArgumentException

Description
Creates a happening with the given name.

Parameters
name—A string to name the happening.
dispatcher—To use when being triggered.

Throws
javax.realtime.IllegalRealtimeArgumentException—when name is null or 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.
Happening(String)

Signature

```
public Happening(String name)
```  

Throws `IllegalRealtimeArgumentException`—when `name` is null or 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`.

Description

Creates a happening with the given `name` and the default dispatcher.

Parameters

`name`—A string to name the happening.

Throws

`javax.realtime.IllegalRealtimeArgumentException`—when `name` is null or 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`.

13.3.2.3.2 Methods

gHappening(String)

Signature

```
public static javax.realtime.device.Happening getHappening(String name)
```  

Returns

a reference to the happening with the given `name`, or `null`, if no happening with the given name is found.
isHappening(String)

**Signature**

```java
public static boolean isHappening(String name)
```

**Description**

Determines whether or not there is an active happening with `name` given as parameter.

**Parameters**

- `name`—A string that might name an active happening.

**Throws**

- `javax.realtime.IllegalRealtimeArgumentException`—when `name` is null.

**Returns**

- `true` only when there is a registered happening with the given `name`.

createId(String)

**Signature**

```java
public static int createId(String name)
throws IllegalRealtimeStateException
```

**Description**

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` 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 $O(\log_2(n))$.

**Parameters**

- `name`—A string to name a happening.

**Throws**

- `javax.realtime.IllegalRealtimeStateException`—when `name` is already registered.

- `javax.realtime.IllegalRealtimeArgumentException`—when `name` is null.

**Returns**

- an ID assigned by the system.
getIdentifier(String)

**Signature**

```java
class Happening {  
    public static int getId(String name)  
}
```

**Description**

Obtains 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(\(\log_2(n)\)).

**Parameters**

- `name`—A happening name string.

**Throws**

- `javax.realtime.IllegalArgumentException`—when `name` is null.

**Returns**

The ID, or -1 when no happening is found with that name.

get(int)

**Signature**

```java
class Happening {  
    public static Happening get(int id)  
}
```

**Description**

Gets the external event corresponding to a given `id`.

**Parameters**

- `id`—The identifier of a registered signal.

**Returns**

the signal corresponding to `id`.

get(String)

**Signature**

```java
class Happening {  
    public static Happening get(String name)  
}
```

**Description**

Obtains 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(\(\log_2(n)\)).

**Parameters**

- `name`—A happening name string.

**Throws**

- `javax.realtime.IllegalArgumentException`—when `name` is null.

**Returns**

The ID, or -1 when no happening is found with that name.
Gets the external event corresponding to a given name.

Parameters
name—The name of a registered signal.

Throws
javax.realtime.IllegalRealtimeArgumentException—when name is null.

Returns
the signal corresponding to name.

trigger(int)

Signature
public static boolean
trigger(int id)

Description
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 InterruptServiceRoutine.handle method.

trigger() and any native code analog to it interact with other javax.realtime.ActiveEvent 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.

Parameters
id—Identifies which happening to trigger.

Returns
true when a happening with ID happeningId was found, false otherwise.

ggetId

Signature
public final int
ggetId()

Description
Gets the number of this happening.
Returns
the happening number or -1, when not registered.

getName

Signature
public java.lang.String
getName()

Description
Gets the name of this happening.

Returns
the name of this happening.

start

Signature
public void
start()
throws IllegalRealtimeStateException

Description
Starts this happening, i.e., changes its state to the active and enabled. Once a happening is started for 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.

Throws
javax.realtime.IllegalRealtimeStateException—when this happening has already been started or its name is already in use by another happening that has been started.

See Section stop()

start(boolean)

Signature
public void
start(boolean disabled)
throws IllegalRealtimeStateException

Description
Starts this happening, but leaves it in the disabled state. When fired before being enabled, it does not dispatch its handlers.

Parameters
disabled—true for starting in a disabled state.

Throws
javax.realtime.IllegalRealtimeStateException—when this happening has already been started.

See Section stop()

stop

Signature
public boolean
stop()
throws IllegalRealtimeStateException

Description
Stops this happening from responding to the fire and trigger methods.

Throws
javax.realtime.IllegalRealtimeStateException—when this happening is not active.

Returns
ture when this is in the enabled state; false otherwise.

isActive

Signature
public boolean
isActive()

Description

Determines the activation state of this happening, i.e., if it has been started.

Returns
true when active; false otherwise.

**isRunning**

*Signature*
```java
class Happening
{
    public boolean isRunning();
}
```

*Description*
Determines whether or not this happening is both active and enabled.

Returns
true when this happening is both active and enabled; false otherwise.

**trigger**

*Signature*
```java
class Happening
{
    public void trigger();
}
```

*Description*
Causes the event dispatcher associated with this to be scheduled for execution.
The implementation should be simple enough so that it can be done in the context of an InterruptServiceRoutine.handle method.
This method must execute in constant time.

**getDispatcher**

*Signature*
```java
class Happening
{
    public javax.realtime.device.HappeningDispatcher getDispatcher();
}
```

*Description*
Obtains the dispatcher for this.

Returns
that dispatcher.
addHandler(AsyncBaseEventHandler)

*Signature*

```java
public void addHandler(AsyncBaseEventHandler handler)
```

*Description*

Adds a handler to the set of handlers associated with this event. An instance of `AsyncBaseEvent` 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.

Note that there is an implicit reference to the handler stored in `this`. The assignment must be valid under any applicable memory assignment rules.

setHandler(AsyncBaseEventHandler)

*Signature*

```java
public void setHandler(AsyncBaseEventHandler handler)
```

*Description*

Associates a new handler with this event and removes all existing handlers. The execution of this method is atomic with respect to the execution of the `fire()` method.

removeHandler(AsyncBaseEventHandler)

*Signature*

```java
public void removeHandler(AsyncBaseEventHandler handler)
```

*Description*

Removes 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 unireal, this method shall not return until all related finalization has completed.

### 13.3.2.4 HappeningDispatcher

**Inheritance**

```
java.lang.Object
  javax.realtime.ActiveEventDispatcher<HappeningDispatcher, Happening>
  HappeningDispatcher
```

**Description**

This class provides a means of dispatching a set of `Happening`.

**Available since** RTSJ 2.0

### 13.3.2.4.1 Constructors

**HappeningDispatcher(SchedulingParameters, RealtimeThreadGroup)**

**Signature**

```
public
HappeningDispatcher(SchedulingParameters schedule,
RealtimeThreadGroup group)
throws IllegalRealtimeStateException
```

**Description**

Creates a new dispatcher, whose dispatching thread runs with the given scheduling parameters.

**Parameters**

- `schedule`—The parameters to use for scheduling this dispatcher.
- `group`—The realtime thread group to use for the dispatcher.
13 Devices and Triggering

HappeningDispatcher

Throws

IllegalRealtimeStateException—when the intersection of affinity in schedule and the affinity of group does not correspond to a valid affinity.

HappeningDispatcher(SchedulingParameters)

Signature

public HappeningDispatcher(SchedulingParameters schedule)
throws IllegalRealtimeStateException

Description

Creates a new dispatcher, whose dispatching thread runs with the given scheduling parameters.

Parameters

schedule—The parameters to use for scheduling this dispatcher.

Throws

IllegalRealtimeStateException—when the intersection of affinity in schedule and the affinity of the current thread group does not correspond to a valid affinity.

13.3.2.4.2 Methods

setDefaultDispatcher(HappeningDispatcher)

Signature

public static void setDefaultDispatcher(HappeningDispatcher dispatcher)

Description

Sets the system default happening dispatcher.

Parameters

dispatcher—The default to use when no dispatcher is provided. When null, the happening dispatcher is set to the original system default.
register(Happening)

Signature
public synchronized void
register(Happening happening)
throws RegistrationException,
    IllegalRealtimeStateException,
    IllegalRealtimeArgumentException

Description
Registers a Happening with this dispatcher.

Parameters
happening—The instance to be registered.

Throws
RegistrationException—when happening is already registered.
javax.realtime.IllegalRealtimeStateException—when this object has been destroyed.
javax.realtime.IllegalRealtimeArgumentException—when happening is not stopped.

deregister(Happening)

Signature
public synchronized void
deregister(Happening happening)
throws DeregistrationException,
    IllegalRealtimeStateException,
    IllegalRealtimeArgumentException

Description
Unregisters a Happening from this dispatcher.

Parameters
happening—The instance to be unregistered.

Throws
DeregistrationException—when happening is not already registered.
javax.realtime.IllegalRealtimeStateException—when this object has been destroyed.
13 Devices and Triggering

javax.realtime.IllegalRealtimeArgumentException—when happening is not stopped.

destroy

Signature
public void destroy()
throws IllegalRealtimeStateException

Description
Releases all resources thereby making the dispatcher unusable.

Throws
javax.realtime.IllegalRealtimeStateException—when called on a dispatcher that has one or more registered Happening objects.

13.3.2.5 InterruptCeilingEmulation

public class InterruptCeilingEmulation

Inheritance
java.lang.Object
javax.realtime.MonitorControl
InterruptMasking
InterruptCeilingEmulation

Description
The default monitor control policy for all instances of InterruptServiceRoutine. It is for use synchronizing between second level interrupt handler while the corresponding interrupt line is blocked. The ceiling used depends on the state of the interrupt service routine used as synchronization object: when registered, it is the hardware priority associated with its interrupt line; otherwise, it is the highest software priority available for priority ceiling emulation. When set on anything but an interrupt handler associated with a maskable interrupt, a javax.realtime.UnsupportedRealtimeOperationException is thrown.

See Section javax.realtime.PriorityCeilingEmulation
13.3.2.5.1 Methods

**instance**

**Signature**

```java
public static javax.realtime.device.InterruptCeilingEmulation
instance()
```

**Description**

Obtain the instance of this class for a given interrupt. The instance returned is
not affected by garbage collection and is universally reachable. It is also effectively
immutable.

**Returns**

The instance of this class for the use with an instance of `InterruptServiceRoutine`
associate with a maskable interrupt.

13.3.2.6 InterruptDescriptor

```java
public class InterruptDescriptor
```

**Inheritance**

- `java.lang.Object`
  - `InterruptDescriptor`

**Description**

A class to manage valid interrupts and provide names for them. It provides
both lookup and enumeration via a visitor. Both the names and the values are
implementation dependent.

13.3.2.6.1 Methods
**create(long, String)**

*Signature*

```java
class InterruptDescriptor {
    public static javax.realtime.device.InterruptDescriptor create(long id,
                        String name)
        throws IllegalRealtimeArgumentException,
                IllegalRealtimeStateException
}
```

*Description*

Create a new interrupt descriptor.

*Parameters*

- **id** — the numerical ID of the new descriptor.
- **name** — of the descriptor

*Throws*

- `IllegalRealtimeArgumentException` — when `name` is null.
- `IllegalRealtimeStateException` — when a descriptor with `name` already exists.

*Returns*

the corresponding interrupt descriptor object.

**get(String)**

*Signature*

```java
class InterruptDescriptor {
    public static javax.realtime.device.InterruptDescriptor get(String name)
        throws IllegalRealtimeArgumentException
}
```

*Description*

Get a memory interrupt descriptor by name.

*Parameters*

- **name** — of the descriptor

*Throws*

- `IllegalRealtimeArgumentException` — when `name` is null.

*Returns*

the interrupt descriptor object or `null`, when none with `name` exists.
isInterrupt(String)

Signature

public static boolean
isInterrupt(String name)

Description

Ask whether or not there is a memory interrupt descriptor of a given name.

Parameters

name—for which to search

Returns

true when there is one and false otherwise.

visitInterrupts(Consumer)

Signature

public static void
visitInterrupts(java.util.function.Consumer<InterruptDescriptor> visitor)

Description

Visit all valid interrupt identifiers. Each valid interrupt identifier corresponds to hardware interrupt line. Their values are system dependent.

Parameters

visitor—the code to execute on each descriptor.

getName

Signature

public final java.lang.String
getName()

Description

Obtains the name of this interrupt descriptor.

Returns

the interrupt descriptors name
**13 Devices and Triggering**

**getID**

*Signature*

```java
class InterruptInheritance {
    public final long getID()
}
```

*Description*

Obtains the id of this interrupt descriptor.

*Returns*

the interrupt descriptors name

**toString**

*Signature*

```java
class InterruptInheritance {
    public final java.lang.String toString()
}
```

*Description*

Gets a printable representation for a descriptor.

*Returns*

the printable form of this memory interrupt descriptor.

**13.3.2.7 InterruptInheritance**

public class InterruptInheritance

*Inheritance*

java.lang.Object

    javax.realtime.MonitorControl
        InterruptMasking
            InterruptInheritance

*Description*

An alternative monitor control policy for instances of `InterruptServiceRoutine`. It is for use synchronizing between and interrupt service routine and its second level interrupt handler while the corresponding interrupt line is blocked. Though masking only takes place when the interrupt service routine used as synchronization object is registered, eligibility inheritance is always applied. When set
on anything but an interrupt handler associated with a maskable interrupt, an
`javax.realtime.UnsupportedRealtimeOperationException` is thrown.

13.3.2.7.1 Methods

instance

*Signature*

```java
global static javax.realtime.device.InterruptInheritance
instance()
```

*Description*

Obtain the instance of this class for a given interrupt. The instance returned is
not affected by garbage collection and is universally reachable. It is also effectively
immutable.

*Returns*

The instance of this class for the use with an instance of `InterruptServiceRoutine`
associate with a maskable interrupt.

13.3.2.8 InterruptMasking

```java
public abstract class InterruptMasking
```

*Inheritance*

java.lang.Object
```
javax.realtime.MonitorControl
InterruptMasking
```

*Description*

A monitor control policy for use with Interrupt Service Routines. Data shared
between an interrupt service routine and a happening should use a subclass of
this monitor control policy. Synchronizing on an object causes the corresponding
interrupt of its `InterruptServiceRoutine` instance to be masked out when its
routine is registered. When its routine is not registered, masking does not occur.

Monitor control policies of this type can only be set on interrupt service
routines and interrupt service routines may only using this monitor control policy
type.
13 Devices and Triggering

13.3.2.9 InterruptServiceRoutine

public abstract class InterruptServiceRoutine

Inheritance
java.lang.Object
    InterruptServiceRoutine

Description
A class for defining a first level interrupt handler. The implementation must override the handle 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. Furthermore, it may only have its monitor set to one of type InterruptMasking. The default is InterruptInheritance for instance associated with a maskable interrupt and InterruptUnmaskable for one associated with an unmaskable interrupt.

Available since RTSJ 2.0

13.3.2.9.1 Constructors

InterruptServiceRoutine(long, MemoryArea)

Signature
public
    InterruptServiceRoutine(long Interrupt,
                                MemoryArea area)

throws NullPointerException,
            IllegalRealtimeArgumentException

Description
Creates an interrupt service routine for a particular interrupt line with a particular memory area. An interrupt line may only have one handler registered at any given time and an instance of this class may only be used for one interrupt. The default monitor control policy for an instance of this class is InterruptInheritance.

736 RTSJ 2.0 (Draft 66)
InterruptServiceRoutine

Parameters
Interrupt—A reference to the Interrupt associate with this handler.
area—The allocation context in which the handle method runs.

Throws
NullPointerException—when area is null.
javax.realtime.IllegalRealtimeArgumentException—when area is a memory area that cannot be accessed from an ISR.

13.3.2.9.2 Methods

getHandler(long)

Signature
public static javax.realtime.device.InterruptServiceRoutine
getHandler(long interrupt)

Description
Gets the InterruptServiceRoutine that is handling a given interrupt.

Parameters
interrupt—A system-dependent identifier for the interrupt.

Returns
the InterruptServiceRoutine registered to the given interrupt. When nothing is registered, null is returned.

getAffinity(long)

Signature
public static javax.realtime.Affinity
getAffinity(long interrupt)
throws IllegalRealtimeArgumentException

Description
Generally, the program cannot determine the affinity of interrupt handling; however, it is necessary for determining schedulability. This method provides that information, where possible.

Parameters
interrupt—The identifier for the interrupt to test

Throws
javax.realtime.IllegalRealtimeArgumentException—when interrupt is not in the set of descriptors managed by the InterruptDescriptor class.

Returns
an affinity when the system can determine the affinity of interrupt or null when it cannot.

isRegistered

Signature
public final boolean
isRegistered()

Description
Obtains the registration state.

Returns
true when registered, otherwise false.

register

Signature
public void
register()
throws RegistrationException,
ScopedCycleException

Description
Registers this interrupt service routine with the system so that it can be triggered. Its initial memory area, if any, will be placed in the scope stack and have its reservation count increased. Registering may change the affinity, when the currently set affinity is not compatible with the affinities available for the specific interrupt for which this routine is registered.

Throws
RegistrationException—when this is already registered, some other InterruptServiceRoutine is registered for interrupt, or the handle method is synchronized.
ScopedCycleException—when the initial memory area for this InterruptServiceRoutine is already present in the scope tree at a different location than the rules for reservation would imply.

**unregister**

*Signature*

```java
public void unregister()
throws DeregistrationException
```

*Description*

Deregisters this interrupt service routine with the system so that it can no longer be triggered. Its initial memory area, if any, will have its reservation count decreased.

*Throws*

DeregistrationException—when this interrupt service routine is not registered.

**handle**

*Signature*

```java
protected abstract void handle()
```

*Description*

The code to execute for first level interrupt handling. A subclass defines this to give the required behavior. RawMemory classes may be used to access the associated device registers and a Happening may be triggered for second level interrupt handling.

The code used to implement this method should not block itself or induce a context switch, e.g., sleep or perform I/O. Only spin waits may be used. The effects of unbounded blocking and inducing a context switch here are undefined and could result in a deadlock. Object.notify() and Object.notifyAll() may be called, but Happening.trigger is preferred for ease of linking with the event handling system and javax.realtime.RealtimeThread.release() for releasing a realtime thread with javax.realtime.AperiodicParameters. In any case, Object.wait() should not be called.

Synchronizing with the code in a handle() method can only be done by synchronizing with the instance of InterruptServiceRoutine containing that
handle() method. The handle() method itself may not be synchronized, since this generates unnecessary extra code which could cause blocking. Instead, the platform interrupt masking is used to ensure mutual exclusion. The containing instance can only have a monitor control policy of type InterruptMasking.

When a memory area is provided, that memory is entered before this method is invoked and exited after it returns. When no memory area is provided, the method may not allocate. Any attempt to allocate will result in an OutOfMemoryError.

Any exceptions thrown by this method cause the method to about and are silently caught and discarded by the infrastructure.

13.3.2.10 InterruptUnmaskable

public class InterruptUnmaskable

Inheritance
java.lang.Object
javax.realtime.MonitorControl
InterruptMasking
InterruptUnmaskable

Description
A control policy for instances of InterruptServiceRoutine associated with an unmaskable interrupt. Since such an interrupt cannot be masked, synchronization is not possible. Synchronization on an object with this monitor policy always fails with an javax.realtime.CeilingViolationException. When set on anything but an interrupt handler associated with an unmaskable interrupt, an javax.realtime.UnsupportedRealtimeOperationException is thrown.

13.3.2.10.1 Methods

instance

Signature
public static javax.realtime.deviceInterruptUnmaskable
instance()
Obtain the instance of this class for a given interrupt. The instance returned is not affected by garbage collection and is universally reachable. It is also effectively immutable.

**Returns**
The instance of this class for the use with an instance of `InterruptServiceRoutine` associate with an unmaskable interrupt.

### 13.3.2.11 RawMemoryFactory

**Description**
This class is the hub of a system that constructs special purpose objects to access particular types and ranges of raw memory. This facility is supported by the `register(RawMemoryRegionFactory)` methods. An application developer can use this method to add support for additional memory regions.

Each `create method` returns an object of the corresponding type, e.g., the `createRawByte(RawMemoryRegion, long, int, int)` method returns a reference to an object that implements the `RawByte` interface and supports access to the requested type of memory and address range. Each `create method` is permitted to optimize error checking and access based on the requested memory type and address range.

The usage pattern for raw memory, assuming the necessary factory has been registered, is illustrated by this example.

```java
// Get an accessor object that can access memory starting at
// baseAddress, for size bytes.
RawInt memory =
    RawMemoryFactory.
    createRawInt(RawMemoryFactory.MEMORY_MAPPED_REGION,
                 address, count, stride, false);
// Use the accessor to load from and store to raw memory.
int loadedData = memory.getInt(someOffset);
memory.setInt(otherOffset, intVal);
```
When an application needs to access a class of memory that is not already supported by a registered factory, the developer must implement and register a factory that implements the `RawMemoryRegionFactory` which can create objects to access memory in that region.

A raw memory region factory is identified by a `RawMemoryRegion` that is used by each `create` method, e.g., `createRawByte(RawMemoryRegion, long, int, int)`, to locate the appropriate factory. The name is provided to `register(RawMemoryRegionFactory)` through the factory’s `RawMemoryRegionFactory.getName` method.

The `register(RawMemoryRegionFactory)` method is only used by the application code when it needs to add support for a new type of raw memory.

Whether a given offset addresses a high-order or low-order byte of an aligned `short` in memory is determined by the value of the `javax.realtime.RealtimeSystem.BYTE_ORDER` static byte variable in class `javax.realtime.RealtimeSystem`, by the start address of the object, and by the offset given the `stride` of the object. Regardless of the byte ordering, accessor methods continue to select bytes starting at offset from the base address and continuing toward greater addresses.

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

Atomic loads and stores on raw memory are defined in terms of physical memory. This memory may be accessible to threads outside the JVM and to non-programmed access (e.g., DMA). Consequently, atomic access must be supported by hardware. This specification is written with the assumption that all suitable hardware platforms support atomic loads from raw memory for aligned bytes, shorts, and ints. Atomic access beyond the specified minimum may be supported by the implementation.

Storing values into raw memory is more hardware dependent than loading values. Many processor architectures do not support atomic stores of variables except for aligned stores of the processor’s word size. For instance, storing a byte into memory might require reading a 32-bit quantity into a processor register, updating the register to reflect the new byte value, then restoring the whole 32-bit quantity. Changes to other bytes in the 32-bit quantity that take place between the load and the store are lost.

Some processors have mechanisms that can be used to implement an atomic store of a byte, but those mechanisms are often slow and not universally supported.
This class need not support unaligned access to data; but if it does, it is not required from 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 tasks. A raw memory region could be updated by another task, 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 can be represented by 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 array is described in the following table.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Values</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access type</td>
<td>read, write</td>
<td></td>
</tr>
<tr>
<td>Data type</td>
<td>byte, short, int, long, float, double</td>
<td></td>
</tr>
<tr>
<td>Alignment</td>
<td>0 aligned</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 to one less than data type size</td>
<td>the first byte of the data is the value of alignment bytes away from natural alignment.</td>
</tr>
<tr>
<td>Atomicity</td>
<td>processor</td>
<td>means access is atomic with respect to other tasks on processor.</td>
</tr>
<tr>
<td></td>
<td>smp</td>
<td>means access is processor atomic, and atomic with respect to all processors in an SMP.</td>
</tr>
<tr>
<td></td>
<td>memory</td>
<td>means that access is smp atomic, and atomic with respect to all access to the memory including DMA.</td>
</tr>
</tbody>
</table>

The true values in the table are represented by properties of the following form. `javax.realtime.atomicaccess_<access>_<type>_<alignment>_atomicity=true` for example,
Table entries with a value of false may be explicitly represented, but since false is the default value, such properties are redundant.

All raw memory access is treated as volatile, and serialized. The infrastructure must be forced to read memory or write to memory on each call to a raw memory objects’s getter or setter method, and to complete the reads and writes in the order they appear in the program order.

Available since RTSJ 2.0

13.3.2.11.1 Fields

MEMORY_MAPPED_REGION

public static final RawMemoryRegion MEMORY_MAPPED_REGION

Description

This raw memory region is predefined for request access to memory mapped I/O devices.

IO_PORT_MAPPED_REGION

public static final RawMemoryRegion IO_PORT_MAPPED_REGION

Description

This raw memory region is predefined for access to I/O device space implemented by processor instructions, such as the x86 in and out instructions.

13.3.2.11.2 Constructors
RawMemoryFactory

Signature
public
RawMemoryFactory()

Description
Creates an empty factory. For a factory with support for the platform defined regions, use getDefaultFactory instead.

13.3.2.11.3 Methods

getDefaultFactory

Signature
public static javax.realtime.device.RawMemoryFactory
getDefaultFactory()

Description
Gets the factory with support for the platform defined regions.

Returns
the platform-defined factory.

register(RawMemoryRegionFactory)

Signature
public void
register(RawMemoryRegionFactory factory)
throws RegistrationException

Description
Adds support for a new memory region.

Parameters
factory—The RawMemoryRegionFactory instance to use for creating RawMemory objects for the RawMemoryRegion instances it makes available.
Throws

RegistrationException—when the factory already is already registered.

deregister(RawMemoryRegionFactory)

Signature

public void
deregister(RawMemoryRegionFactory factory)
throws DeregistrationException

Description

Removes support for a new memory region.

Parameters

factory—The RawMemoryRegionFactory to be made unavailable.

Throws

RegistrationException—when the factory is not registered.

createRawByte(RawMemoryRegion, long, int, int)

Signature

public javax.realtime.device.RawByte
createRawByte(RawMemoryRegion region,
   long base, int count, int stride)
throws SecurityException,
   OffsetOutOfBoundsException,
   SizeOutOfBoundsException,
   MemoryTypeConflictException,
   UnsupportedRawMemoryRegionException

Description

Creates an instance of a class that implements RawByte and accesses memory of region in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride * size of RawByte * count. The object is allocated in the current memory area of the calling thread.

Parameters

region—The address space from which the new instance should be taken.
base—The starting physical address accessible through the returned instance.
count—The number of memory elements accessible through the returned instance.
stride—The distance to the next element as a multiple 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 a memory that matches the type served by this factory.

**Returns**

an object that implements RawByte and supports access to the specified range in the memory region.

createRawByteReader(RawMemoryRegion, long, int, int)

**Signature**

```java
public javax.realtime.device.RawByteReader
createRawByteReader(RawMemoryRegion region,
long base,
int count,
int stride)
```

**Throws**

SecurityException,
OffsetOutOfBoundsException,
SizeOutOfBoundsException,
MemoryTypeConflictException,
UnsupportedRawMemoryRegionException

**Description**

Creates an instance of a class that implements RawByteReader and accesses memory of region in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride * size of RawByteReader * count. The object is allocated in the current memory area of the calling thread.
Parameters
region—The address space from which the new instance should be taken.
base—The starting physical address accessible through the returned instance.
count—The number of memory elements accessible through the returned instance.
stride—The distance to the next element as a multiple 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.
OutOfBoundsException—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 a memory that matches the type served by this factory.

Returns
an object that implements RawByteReader and supports access to the specified range in the memory region.

createRawByteWriter(RawMemoryRegion, long, int, int)

Signature
public javax.realtime.device.RawByteWriter
createRawByteWriter(RawMemoryRegion region,
   long base,
   int count,
   int stride)
throws SecurityException,
   OutOfBoundsException,
   SizeOutOfBoundsException,
   MemoryTypeConflictException,
   UnsupportedRawMemoryRegionException

Description
Creates an instance of a class that implements RawByteWriter and accesses memory of region in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride * size of
RawMemoryFactory

**RawByteWriter** * count. The object is allocated in the current memory area of the calling thread.

**Parameters**
- region—The address space from which the new instance should be taken.
- base—The starting physical address accessible through the returned instance.
- count—The number of memory elements accessible through the returned instance.
- stride—The distance to the next element as a multiple 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 a memory that matches the type served by this factory.

**Returns**
- an object that implements RawByteWriter and supports access to the specified range in the memory region.

**createRawShort(RawMemoryRegion, long, int, int)**

**Signature**
```java
public javax.realtime.device.RawShort createRawShort(RawMemoryRegion region,
    long base,
    int count,
    int stride)
```

**Throws**
- SecurityException,
- OffsetOutOfBoundsException,
- SizeOutOfBoundsException,
- MemoryTypeConflictException,
- UnsupportedRawMemoryRegionException

**Description**

RTSJ 2.0 (Draft 66) 749
Creates an instance of a class that implements `RawShort` and accesses memory of the `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.

**Parameters**
- `region`—The address space from which the new instance should be taken.
- `base`—The starting physical address accessible through the returned instance.
- `count`—The number of memory elements accessible through the returned instance.
- `stride`—The distance to the next element as a multiple 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 a memory that matches the type served by this factory.

**Returns**
an object that implements `RawShort` and supports access to the specified range in the memory region.

createRawShortReader(RawMemoryRegion, long, int, int)

**Signature**

```java
class javax.realtime.device.RawShortReader

createRawShortReader(RawMemoryRegion region,
                        long base,
                        int count,
                        int stride)
```
Description

Creates an instance of a class that implements RawShortReader and accesses memory of region in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is \( \text{stride} \times \text{size of RawShortReader} \times \text{count} \). The object is allocated in the current memory area of the calling thread.

Parameters

region—The address space from which the new instance should be taken.
base—The starting physical address accessible through the returned instance.
count—The number of memory elements accessible through the returned instance.
stride—The distance to the next element as a multiple 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.

OutOfBoundsException—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 a memory that matches the type served by this factory.

Returns

an object that implements RawShortReader and supports access to the specified range in the memory region.

createRawShortWriter(RawMemoryRegion, long, int, int)

Signature

public javax.realtime.device.RawShortWriter
createRawShortWriter(RawMemoryRegion region,
    long base,
    int count,
    int stride)

throws SecurityException,
OutOfBoundsException,
SizeOutOfBoundsException,
MemoryTypeConflictException,
UnsupportedRawMemoryRegionException

Description

Creates an instance of a class that implements `RawShortWriter` and accesses memory of `region` in the address range described by `base`, `stride`, and `count`. The actual extent of the memory addressed by the object is `stride * size of RawShortWriter * count`. The object is allocated in the current memory area of the calling thread.

Parameters

- `region`—The address space from which the new instance should be taken.
- `base`—The starting physical address accessible through the returned instance.
- `count`—The number of memory elements accessible through the returned instance.
- `stride`—The distance to the next element as a multiple 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 a memory that matches the type served by this factory.

Returns

an object that implements `RawShortWriter` and supports access to the specified range in the memory region.

createRawInt(RawMemoryRegion, long, int, int)

Signature

```java
public javax.realtime.device.RawInt createRawInt(RawMemoryRegion region,
        long base,
        int count,
        int stride)
```

752 RTSJ 2.0 (Draft 66)
throws SecurityException,
    OffsetOutOfBoundsException,
    SizeOutOfBoundsException,
    MemoryTypeConflictException,
    UnsupportedRawMemoryRegionException

Description

Creates an instance of a class that implements RawInt and accesses memory of region in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride * size of RawInt * count. The object is allocated in the current memory area of the calling thread.

Parameters

region—The address space from which the new instance should be taken.
base—The starting physical address accessible through the returned instance.
count—The number of memory elements accessible through the returned instance.
stride—The distance to the next element as a multiple 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 a memory that matches the type served by this factory.

Returns

an object that implements RawInt and supports access to the specified range in the memory region.

createRawIntReader(RawMemoryRegion, long, int, int)

Signature

public javax.realtime.device.RawIntReader
createRawIntReader(RawMemoryRegion region,
            long base,
            int count,
int stride)
throws SecurityException,
    OffsetOutOfBoundsException,
    SizeOutOfBoundsException,
    MemoryTypeConflictException,
    UnsupportedRawMemoryRegionException

Description
Creates an instance of a class that implements RawIntReader and accesses memory of region in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is \( \text{stride} \times \text{size of RawIntReader} \times \text{count} \). The object is allocated in the current memory area of the calling thread.

Parameters
region—The address space from which the new instance should be taken.
base—The starting physical address accessible through the returned instance.
count—The number of memory elements accessible through the returned instance.
stride—The distance to the next element as a multiple 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 a memory that matches the type served by this factory.

Returns
an object that implements RawIntReader and supports access to the specified range in the memory region.

cREATEWIntWriter(RawMemoryRegion, long, int, int)

Signature
public javax.realtime.device.RawIntWriter
cREATEWIntWriter(RawMemoryRegion region,
long base,
   int count,
   int stride)

throws SecurityException,
   OffsetOutOfBoundsException,
   SizeOutOfBoundsException,
   MemoryTypeConflictException,
   UnsupportedRawMemoryRegionException

Description

Creates an instance of a class that implements RawIntWriter and accesses memory of region in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride * size of RawIntWriter * count. The object is allocated in the current memory area of the calling thread.

Parameters

region—The address space from which the new instance should be taken.
base—The starting physical address accessible through the returned instance.
count—The number of memory elements accessible through the returned instance.
stride—The distance to the next element as a multiple 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 a memory that matches the type served by this factory.

Returns

an object that implements RawIntWriter and supports access to the specified range in the memory region.

createRawLong(RawMemoryRegion, long, int, int)

Signature

RTSJ 2.0 (Draft 66)
public javax.realtime.device.RawLong
createRawLong(RawMemoryRegion region,
    long base,
    int count,
    int stride)
throws SecurityException,
    OffsetOutOfBoundsException,
    SizeOutOfBoundsException,
    MemoryTypeConflictException,
    UnsupportedRawMemoryRegionException

Description

Creates an instance of a class that implements RawLong and accesses memory of
region in the address range described by base, stride, and count. The actual
extent of the memory addressed by the object is stride * size of RawLong *
count. The object is allocated in the current memory area of the calling thread.

Parameters

region—The address space from which the new instance should be taken.
base—The starting physical address accessible through the returned instance.
count—The number of memory elements accessible through the returned instance.
stride—The distance to the next element as a multiple 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 a memory that
matches the type served by this factory.

Returns

an object that implements RawLong and supports access to the specified range in
the memory region.

createRawLongReader(RawMemoryRegion, long, int, int)
Signature

```java
public javax.realtime.device.RawLongReader
createRawLongReader(RawMemoryRegion region,
    long base,
    int count,
    int stride)
```

throws SecurityException,
OffsetOutOfBoundsException,
SizeOutOfBoundsException,
MemoryTypeConflictException,
UnsupportedRawMemoryRegionException

Description

Creates an instance of a class that implements RawLongReader and accesses memory of region in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride * size of RawLongReader * count. The object is allocated in the current memory area of the calling thread.

Parameters

- **region**—The address space from which the new instance should be taken.
- **base**—The starting physical address accessible through the returned instance.
- **count**—The number of memory elements accessible through the returned instance.
- **stride**—The distance to the next element as a multiple 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 a memory that matches the type served by this factory.

Returns

an object that implements RawLongReader and supports access to the specified range in the memory region.
createRawLongWriter(RawMemoryRegion, long, int, int)

Signature

public javax.realtime.device.RawLongWriter
createRawLongWriter(RawMemoryRegion region, long base, int count, int stride)
throws SecurityException,
    OffsetOutOfBoundsException,
    SizeOutOfBoundsException,
    MemoryTypeConflictException,
    UnsupportedRawMemoryRegionException

Description

Creates an instance of a class that implements RawLongWriter and accesses memory of region in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride * size of RawLongWriter * count. The object is allocated in the current memory area of the calling thread.

Parameters

region—The address space from which the new instance should be taken.
base—The starting physical address accessible through the returned instance.
count—The number of memory elements accessible through the returned instance.
stride—The distance to the next element as a multiple 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 a memory that matches the type served by this factory.

Returns

an object that implements RawLongWriter and supports access to the specified range in the memory region.
createRawFloat(RawMemoryRegion, long, int, int)

Signature

public javax.realtime.device.RawFloat
createRawFloat(RawMemoryRegion region,
    long base,
    int count,
    int stride)
throws SecurityException,
    OffsetOutOfBoundsException,
    SizeOutOfBoundsException,
    MemoryTypeConflictException,
    UnsupportedRawMemoryRegionException

Description

Creates an instance of a class that implements RawFloat and accesses memory of region in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is \( \text{stride} \times \text{size of RawFloat} \times \text{count} \). The object is allocated in the current memory area of the calling thread.

Parameters

region—The address space from which the new instance should be taken.
base—The starting physical address accessible through the returned instance.
count—The number of memory elements accessible through the returned instance.
stride—The distance to the next element as a multiple 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 a memory that matches the type served by this factory.

Returns

an object that implements RawFloat and supports access to the specified range in the memory region.
createRawFloatReader(RawMemoryRegion, long, int, int)

Signature

public javax.realtime.device.RawFloatReader
createRawFloatReader(RawMemoryRegion region,
    long base,
    int count,
    int stride)
throws SecurityException,
    OffsetOutOfRangeException,
    SizeOutOfRangeException,
    MemoryTypeConflictException,
    UnsupportedRawMemoryRegionException

Description

Creates an instance of a class that implements RawFloatReader and accesses memory of region in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride * size of RawFloatReader * count. The object is allocated in the current memory area of the calling thread.

Parameters

region—The address space from which the new instance should be taken.
base—The starting physical address accessible through the returned instance.
count—The number of memory elements accessible through the returned instance.
stride—The distance to the next element as a multiple 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.
OffsetOutOfRangeException—when base is invalid.
SizeOutOfRangeException—when the memory addressed by the object would extend into an invalid range of memory.
MemoryTypeConflictException—when base does not point to a memory that matches the type served by this factory.

Returns

an object that implements RawFloatReader and supports access to the specified range in the memory region.
createRawFloatWriter(RawMemoryRegion, long, int, int)

**Signature**

```java
public javax.realtime.device.RawFloatWriter
createRawFloatWriter(RawMemoryRegion region,
                     long base,
                     int count,
                     int stride)
```

**Description**

Creates an instance of a class that implements `RawFloatWriter` and accesses memory of `region` in the address range described by `base`, `stride`, and `count`. The actual extent of the memory addressed by the object is `stride * size of RawFloatWriter * count`. The object is allocated in the current memory area of the calling thread.

**Parameters**

- `region`—The address space from which the new instance should be taken.
- `base`—The starting physical address accessible through the returned instance.
- `count`—The number of memory elements accessible through the returned instance.
- `stride`—The distance to the next element as a multiple 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.
- `OutOfBoundsException`—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 a memory that matches the type served by this factory.

**Returns**

an object that implements `RawFloatWriter` and supports access to the specified range in the memory region.
createRawDouble(RawMemoryRegion, long, int, int)

Signature

public javax.realtime.device.RawDouble
createRawDouble(RawMemoryRegion region,
    long base,
    int count,
    int stride)

throws SecurityException,
    OffsetOutOfBoundsException,
SizeOutOfBoundsException,
    MemoryTypeConflictException,
UnSupportedRawMemoryRegionException

Description

Creates an instance of a class that implements RawDouble and accesses memory of region in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride * size of RawDouble * count. The object is allocated in the current memory area of the calling thread.

Parameters

region—The address space from which the new instance should be taken.
base—The starting physical address accessible through the returned instance.
count—The number of memory elements accessible through the returned instance.
stride—The distance to the next element as a multiple 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 a memory that matches the type served by this factory.

Returns

an object that implements RawDouble and supports access to the specified range in the memory region.
createRawDoubleReader(RawMemoryRegion, long, int, int)

Signature

public javax.realtime.device.RawDoubleReader
createRawDoubleReader(RawMemoryRegion region,
    long base,
    int count,
    int stride)

throws SecurityException,
    OffsetOutOfBoundsException,
    SizeOutOfBoundsException,
    MemoryTypeConflictException,
    UnsupportedRawMemoryRegionException

Description

Creates an instance of a class that implements RawDoubleReader and accesses memory of region in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride * size of RawDoubleReader * count. The object is allocated in the current memory area of the calling thread.

Parameters

region—The address space from which the new instance should be taken.
base—The starting physical address accessible through the returned instance.
count—The number of memory elements accessible through the returned instance.
stride—The distance to the next element as a multiple 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 a memory that matches the type served by this factory.

Returns
an object that implements `RawDoubleReader` and supports access to the specified range in the memory region.

**createRawDoubleWriter(RawMemoryRegion, long, int, int)**

---

**Signature**

```java
public javax.realtime.device.RawDoubleWriter createRawDoubleWriter(RawMemoryRegion region,
                                                                            long base,
                                                                            int count,
                                                                            int stride)

throws SecurityException,
          OffsetOutOfBoundsException,
          SizeOutOfBoundsException,
          MemoryTypeConflictException,
          UnsupportedRawMemoryRegionException
```

**Description**

Creates an instance of a class that implements `RawDoubleWriter` and accesses memory of `region` in the address range described by `base`, `stride`, and `count`. The actual extent of the memory addressed by the object is `stride * size of RawDoubleWriter * count`. The object is allocated in the current memory area of the calling thread.

**Parameters**

- `region`—The address space from which the new instance should be taken.
- `base`—The starting physical address accessible through the returned instance.
- `count`—The number of memory elements accessible through the returned instance.
- `stride`—The distance to the next element as a multiple 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 a memory that matches the type served by this factory.

Returns
an object that implements RawDoubleWriter and supports access to the specified range in the memory region.

13.3.2.12 RawMemoryRegion

public class RawMemoryRegion

Inheritance
java.lang.Object
  RawMemoryRegion

Description
RawMemoryRegion is a class for typing raw memory regions. It is returned by the RawMemoryRegionFactory.getRegion methods of the raw memory region factory classes, and it is used with methods such as RawMemoryFactory.createRawByte(RawMemoryRegion, long, int, int) and RawMemoryFactory.createRawDouble(RawMemoryRegion, long, int, int) methods to identify the region from which the application wants to get an accessor instance.

Available since RTSJ 2.0

13.3.2.12.1 Methods

get(String)

Signature
public static javax.realtime.device.RawMemoryRegion get(String name)
  throws IllegalRealtimeArgumentException

Description
Get a memory region type by name.

Parameters

RTSJ 2.0 (Draft 66)
name—of the region

Throws

IllegalRealtimeArgumentException—when name is null.

Returns

the region type object or null, when none with name exists.

isRawMemoryRegion(String)

Signature

public static boolean
isRawMemoryRegion(String name)

Description

Ask whether or not there is a memory region type of a given name.

Parameters

name—for which to search

Returns

true when there is one and false otherwise.

getName

Signature

public final java.lang.String
getName()

Description

Obtains the name of this region type.

Returns

the region types name

toString

Signature

public final java.lang.String
toString()

Description

766
13.4 Rationale

As with alternative memory management, support for device access has been improved in several ways: typed access to device memory, and extensions for DMA, and a new first-level interrupt handling API, as well as package separation. Package separation was discussed in the rationale for alternative memory management. Each of the other changes was driven by its own set of requirements. Still they were designed to produce a coherent set of features for the new specification.

13.4.1 Typed 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*. RTSJ Version 2.0 provides two categories:

1. memory that is used to access memory-mapped device registers, and
2. 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 device 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 encapsulate 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 13.5. There are two groups of methods:

1. those that enable a factory to be registered, and
2. those that 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 objects, 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 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_MEMORY_MAPPED_REGION` raw memory name, then the following code will create the objects that facilitate the memory access.

```java
RawMemoryFactory factor = RawMemoryFactory.getDefault();
```
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.

```java
// where l is of type long and is data to be sent to the device
dataReg.setLong(l);

// where i is of type int and is the command to the device
controlReg.setInt(i);
```

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.

The facilities provided by the RTSJ allow an application to support 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. Hence, the RTSJ facilities allow it only to be accessed as a raw memory device.

### 13.4.2 Direct memory access

DMA requires access to memory outside 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.

1. 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.
2. The RTSJ requires that RTSJ programs can be compiled by regular Java compilers. Different bytecode compilers, and their supporting JVMs, 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.

3. 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; however, with raw memory and raw byte buffers, one could construct a higher level API that would cover most DMA controllers. Even so, there will always be odd cases that would still not fit the general pattern, especially for embedded systems. For this reason, only this low level API is provided.

The DMA interface is designed to minimize the points where actual physical addresses are provided. If nothing else, this reduces the number of places where security checks are needed. Actual physical addresses are only needed when a DMARegion is created. When a DMA buffer is needed, the application developer can draw it from one of the previously defined regions. When exact addresses are needed for each buffer, a DMARegion can be defined for each buffer. Otherwise, a large region can be defined for each controller and the system can manage allocation out of these regions.

Figure 13.6: Flash memory device

13.4.3 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:

1. **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.

2. **Interrupt vector**—a dedicated (or configurable) location that specifies the location of an interrupt handler.

3. **Interrupt handler**—code that is reachable from the interrupt vector.

4. **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:

1. **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.

2. **Interrupt overload**—the problem of ensuring that noninterrupt-driven processing is not swamped by unexpected or misbehaving interrupts.

3. **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 Controller1. 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.

1. **Stack overflow**—static analysis techniques can normally be used to determine the worst-case stack usage of all interrupt handlers. When stack is borrowed from the executing thread, this amount must be added to the worst-case stack usage of all threads.

2. **Interrupt overload**—this is typically managed by aperiodic server technology in combination with interrupt masking (see Section 13.6 of [3]).

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

1. 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 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.

2. 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.

3. 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.

An new monitor control policy was added to separate synchronization with an interrupt service routine from other types of synchronization. This is to ensure that the interrupt masking policy is independent of all task schedulers. Furthermore, other types of monitor control policies should not have any inherent knowledge of interrupt masking.
Chapter 14

Interprocess Signaling

On many operating systems, it is possible for one process to signal another. POSIX provides a well defined means of signaling other processes and receiving signals from them, therefore one would like to be able to use this facility when it or a similar mechanism is available. The POSIX module provides the means to do this. It provides a common idiom for binding signals to instances of AsyncEventHandler.

14.1 Definitions

**Signal** — A notification between two system processes, which may or may not contain a data packet.

**Realtime Signal** — A special type of signal that carries a bit of data with it.

14.2 Semantics

The POSIX interface provides two main facilities: sending signals and receiving signals. These are supported by a means of determining which signals are supported on an implementation. In addition, not only stateless signals, but also signals with data are also supported. All classes are in the javax.realtime.posix package.

Some signals have predefined uses. For example, **SIG_SEGV** is used to signal an application that the program has attempted to reference the content of an address that is not available in the current address space. There is even a signal that can never be caught because it terminates the process before anything else can happen.

Therefore, an implementation must document what signals an RTSJ runtime or virtual machine uses, whether or not the signal handler can be overridden, and what would happen when a program attaches a handler to that signal or sends that signal to another instance of the virtual machine. Whenever possible, an RTSJ program
should be able to handle any signal it receives. The only exception is SIG_KILL, which never reaches the program.

14.2.1 POSIX Signals
The Signal class represents POSIX signals and is required on platforms that provide POSIX signals. As with 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 Signal.get(int) and Signal.get(String) methods.

14.2.2 POSIX Realtime Signals
The RealtimeSignal class represents POSIX realtime events. It also implements ActiveEvent, but as a subclass of AsyncLongEvent, so that it can pass the data sent with its signal. As with Signal, it cannot be instantiated by the user, rather an instance exists for each POSIX signal defined on the system, which can be retrieved either by name or number using the RealtimeSignal.get(int) and RealtimeSignal.get(String) methods.
14.3 javax.realtime.posix

14.3.1 Classes

14.3.1.1 RealtimeSignal

public class RealtimeSignal

Inheritance
java.lang.Object
javax.realtime.AsyncBaseEvent
javax.realtime.AsyncLongEvent
RealtimeSignal

Interfaces
javax.realtime.ActiveEvent

Description
A javax.realtime.ActiveEvent subclass for defining a POSIX realtime signal.

Available since RTSJ 2.0

14.3.1.1.1 Methods

isPOSIXRealtimeSignal(String)

Signature
public static boolean
isPOSIXRealtimeSignal(String name)

Description
Determines whether or not the given name represents a POSIX realtime signal.

Parameters
name—The name of the signal check.

Returns
true when a signal with the given name is registered.
getId(String)

Signature
   public static int
   getId(String name)

Description
   Gets the ID of a supported POSIX realtime signal by its name.

Parameters
   name—The name of the signal whose ID should be determined.

Returns
   the ID of the signal named by name.

get(String)

Signature
   public static javax.realtime.posix.RealtimeSignal
   get(String name)

Description
   Gets a POSIX realtime signal by its name.

Parameters
   name—The name of the signal to get.

Returns
   the realtime signal with name or null when none exists.

get(int)

Signature
   public static javax.realtime.posix.RealtimeSignal
   get(int id)

Description
   Gets a POSIX realtime signal by its ID.

Parameters
   id—The identifier of a POSIX realtime signal.

Returns
   the realtime signal corresponding to ID or -1 when none exists.
get\textit{Id}

\textit{Signature}

\begin{verbatim}
    public int getId()
\end{verbatim}

\textit{Description}

Gets the name of this realtime signal.

\textit{Returns}

the ID of this signal.

get\textit{Name}

\textit{Signature}

\begin{verbatim}
    public final java.lang.String getName()
\end{verbatim}

\textit{Description}

Gets the name of this signal.

\textit{Returns}

the name of this signal.

get\textit{Dispatcher}

\textit{Signature}

\begin{verbatim}
    public javax.realtime.posix.RealtimeSignalDispatcher getDispatcher()
\end{verbatim}

\textit{Description}

Obtains the dispatcher for \texttt{this}.

\textit{Returns}

that dispatcher.
isActive

Signature
public boolean
isActive()

Description
Determines the activation state of this signal, i.e., whether or not it has been
started.

Returns
ture when active; false otherwise.

isRunning

Signature
public boolean
isRunning()

Description
Determines the firing state (releasing or skipping) of this signal, i.e., whether or
not is active and enabled.

Returns
ture when releasing, false when skipping.

start

Signature
public final synchronized void
start()
throws IllegalRealtimeStateException

Description
Starts this RealtimeSignal, i.e., changes 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.

Throws

778          RTSJ 2.0 (Draft 66)
javax.realtime.IllegalRealtimeStateException—when this RealtimeSignal has already been started.

See Section stop()

start(boolean)

Signature
  public final synchronized void
  start(boolean disabled)
  throws IllegalRealtimeStateException

Description
  Starts this RealtimeSignal, i.e., changes 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.

Parameters
  disabled—true for starting in a disabled state.

Throws
  javax.realtime.IllegalRealtimeStateException—when this RealtimeSignal has already been started.

See Section stop()

stop

Signature
  public final boolean
  stop()
  throws IllegalRealtimeStateException

Description
  Stops this RealtimeSignal. A stopped realtime signal ceases to be a source of activation and no longer causes any AE attached to it to be a source of activation.

Throws
  javax.realtime.IllegalRealtimeStateException—when this RealtimeSignal is not running.

Returns
true when this was enabled; false otherwise.

**send(long, long)**

*Signature*

```java
public native boolean send(long pid,
long payload)
```

*Description*

Sends this signal to another process.

*Parameters*

- `pid`—The identifier of the process to which to send the signal.
- `payload`—The long value associated with a fire.

*Returns*

true when signal can be sent; otherwise false.

**addHandler(AsyncBaseEventHandler)**

*Signature*

```java
public void addHandler(AsyncBaseEventHandler handler)
```

*Description*

Adds a handler to the set of handlers associated with this event. An instance of `AsyncBaseEvent` 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.

Note that there is an implicit reference to the handler stored in this. The assignment must be valid under any applicable memory assignment rules.

**setHandler(AsyncBaseEventHandler)**

*Signature*

```java
780
```
public void
setHandler(AsyncBaseEventHandler handler)

Description
Associates a new handler with this event and removes all existing handlers. The execution of this method is atomic with respect to the execution of the fire() method.

removeHandler(AsyncBaseEventHandler)

Signature
public void
removeHandler(AsyncBaseEventHandler handler)

Description
Removes 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 unfirable, this method shall not return until all related finalization has completed.

14.3.1.2 RealtimeSignalDispatcher

public class RealtimeSignalDispatcher

Inheritance
java.lang.Object
javax.realtime.ActiveEventDispatcher<RealtimeSignalDispatcher, RealtimeSignal>
RealtimeSignalDispatcher

Description
Provides a means of dispatching a set of RealtimeSignals. 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
14 Interprocess Signaling

14.3.1.2.1 Constructors

RealtimeSignalDispatcher(SchedulingParameters, RealtimeThreadGroup)

Signature
public
RealtimeSignalDispatcher(SchedulingParameters schedule, 
RealtimeThreadGroup group)
throws IllegalRealtimeStateException

Description
Creates a new dispatcher, whose dispatching thread runs with the given SchedulingParameters.

Parameters
schedule—Parameters for scheduling this dispatcher.
group—Container for this dispatcher.

Throws
IllegalRealtimeStateException—when the intersection of affinity in schedule and the affinity of group does not correspond to a valid affinity.

RealtimeSignalDispatcher(SchedulingParameters)

Signature
public
RealtimeSignalDispatcher(SchedulingParameters schedule)
throws IllegalRealtimeStateException

Description
Creates a new dispatcher, whose dispatching thread runs with the given SchedulingParameters.

Parameters
schedule—Parameters for scheduling this dispatcher.

Throws
IllegalRealtimeStateException
IllegalRealtimeStateException—when the intersection of affinity in schedule and the affinity of the current thread group does not correspond to a valid affinity.

14.3.1.2.2 Methods

setDefaultDispatcher(RealtimeSignalDispatcher)

Signature
   public static void
   setDefaultDispatcher(RealtimeSignalDispatcher dispatcher)

Description
   Sets the system default realtime signal dispatcher.

Parameters
   dispatcher—The instance to be used when no dispatcher is provided. When null, the realtime signal dispatcher is set to the original system default.

register(RealtimeSignal)

Signature
   public void
   register(RealtimeSignal signal)
   throws RegistrationException,
           IllegalRealtimeStateException,
           IllegalRealtimeArgumentException

Description
   Registers signal with this dispatcher.

Parameters
   signal—The signal object to register.

Throws
   RegistrationException—when signal is already registered.
   javax.realtime.IllegalRealtimeStateException—when this object has been destroyed.
javax.realtime.IllegalRealtimeArgumentException—when signal is not stopped.

deregister(RealtimeSignal)

Signature

public void
deregister(RealtimeSignal signal)
throws DeregistrationException,
   IllegalRealtimeStateException,
   IllegalRealtimeArgumentException

Description

Deregisters the signal from this dispatcher.

Parameters

signal—The signal object to deregister.

Throws

DeregistrationException—when signal is not already registered.
javax.realtime.IllegalRealtimeStateException—when this object has been destroyed.
javax.realtime.IllegalRealtimeArgumentException—when signal is not stopped.

destroy

Signature

public void
destroy()
throws IllegalRealtimeStateException

Description

Releases all resources thereby making the dispatcher unusable.

Throws

javax.realtime.IllegalRealtimeStateException—when called on a dispatcher that has one or more registered RealtimeSignal objects.

14.3.1.3 Signal

public class Signal

Inheritance
java.lang.Object
    javax.realtime.AsyncBaseEvent
    javax.realtime.AsyncEvent
    Signal

Interfaces
javax.realtime.ActiveEvent

Description
A javax.realtime.ActiveEvent subclass for defining a POSIX signal.

Available since RTSJ 2.0

14.3.1.3.1 Fields

MAX_NUM_SIGNALS

public static final int MAX_NUM_SIGNALS

Description
Obtains the maximum number a signals may have.

14.3.1.3.2 Methods

isPOSIXSignal(String)

Signature
public static boolean isPOSIXSignal(String name)

Description
Determines if the given name represents a POSIX signal.

**Parameters**
- name — The string passed as the name of the signal.

**Returns**
- true when a signal with the given name is registered.

### getId(String)

**Signature**
```
public static int
getId(String name)
```

**Description**
- Gets the ID of a supported signal by its name.

**Parameters**
- name — The name of the signal for which to search.

**Returns**
- the ID of the signal named by name or -1 when no signal with the given name exists.

### get(String)

**Signature**
```
public static javax.realtime.posix.Signal
get(String name)
```

**Description**
- Gets a supported signal by its name.

**Parameters**
- name — The name identifying the signal to get.

**Returns**
- the signal associated with name or null when none exists.

### get(int)

**Signature**
```
786
```

RTSJ 2.0 (Draft 66)
public static javax.realtime.posix.Signal get(int id)

Description

Gets a supported signal by its ID.

Parameters

id—The identifier of a registered signal.

Returns

the signal corresponding to ID or null when none exists.

getProcessId

Signature

public static long getProcessId()

Description

Obtains the OS Id of the JVM process. When running in kernel space, the result is VM dependent and must be documented. This number returned is only usable with Signal.send(long).

Returns

the OS process ID.

getId

Signature

public int getId()

Description

Gets the number of this signal.

Returns

the signal number.
14 Interprocess Signaling

**getName**

*Signature*

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

*Description*

Gets the name of this signal.

*Returns*

the name of this signal.

**getDispatcher**

*Signature*

```java
public javax.realtime.posix.SignalDispatcher
getDispatcher()
```

*Description*

Obtains the dispatcher for this.

*Returns*

that dispatcher.

**isActive**

*Signature*

```java
public boolean
isActive()
```

*Description*

Determines the activation state of this signal, i.e., whether or not it has been started.

*Returns*

true when active; false otherwise.
isRunning

Signature
public boolean
isRunning()

Description
Determines the firing state, releasing or skipping, of this signal, i.e., whether or not it is active and enabled.

Returns
true when releasing, false when skipping.

start

Signature
public void
start()
throws IllegalRealtimeStateException

Description
Starts this Signal, i.e., changes 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.

Throws
javax.realtime.IllegalRealtimeStateException—when this Signal has already been started.

See Section stop()

start(boolean)

Signature
public void
start(boolean disabled)
throws IllegalRealtimeStateException

Description
Starts this Signal, i.e., changes 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.

**Parameters**
- `disabled`—true for starting in a disabled state.

**Throws**
- `javax.realtime.IllegalRealtimeStateException`—when this Signal has already been started.

See Section stop()

### stop

**Signature**
```java
public boolean stop()
throws IllegalRealtimeStateException
```

**Description**
Stops this Signal. A stopped signal ceases to be a source of activation and no longer causes any AE attached to it to be a source of activation.

**Throws**
- `javax.realtime.IllegalRealtimeStateException`—when this Signal is not running.

**Returns**
true when this was enabled and false otherwise.

### send(long)

**Signature**
```java
public void send(long pid)
throws POSIXInvalidSignalException,
POSIXSignalPermissionException,
POSIXInvalidTargetException
```

**Description**

790 RTSJ 2.0 (Draft 66)
Sends this signal to another process or process group.

On POSIX systems running in user space, the following holds:

- when pid is positive, the signal is sent to pid;
- when pid equals 0, the signal is sent to every process in the process group of the current process;
- when pid equals -1, the signal is sent to every process for which the calling process has permission to send signals, except for possibly OS-defined system processes; otherwise
- when pid is less than -1, the signal is sent to every process in the process group -pid.

POSIX.1-2001 requires the underlying mechanism of `signal.send(-1)` to send signal to all processes for which the current process may signal, except possibly for some OS-defined system processes.

For an RTVM running in kernel space, the meaning of the `pid` is implementation dependent, though it should be as closed to the standard definition as possible.

Parameters

`pid`—ID of the process to which to send the signal.

Throws

- `POSIXInvalidSignalException`—when the signal number is not valid.
- `POSIXSignalPermissionException`—when the process does not have permission to send the target.
- `POSIXInvalidTargetException`—when the target does not exist.

### addHandler(AsyncBaseEventHandler)

**Signature**

```
public void addHandler(AsyncBaseEventHandler handler)
```

**Description**

Adds a handler to the set of handlers associated with this event. An instance of `AsyncBaseEvent` 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.

Note that there is an implicit reference to the handler stored in `this`. The assignment must be valid under any applicable memory assignment rules.
14 Interprocess Signaling

**setHandler(AsyncBaseEventHandler)**

*Signature*

```java
class SignalDispatcher {
    public void setHandler(AsyncBaseEventHandler handler)
}
```

*Description*

Associates a new handler with this event and removes all existing handlers. The execution of this method is atomic with respect to the execution of the `fire()` method.

**removeHandler(AsyncBaseEventHandler)**

*Signature*

```java
class SignalDispatcher {
    public void removeHandler(AsyncBaseEventHandler handler)
}
```

*Description*

Removes 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 unirable, this method shall not return until all related finalization has completed.

### 14.3.1.4 SignalDispatcher

```java
public class SignalDispatcher {
    Inheritance
        java.lang.Object
            javax.realtime.ActiveEventDispatcher<SignalDispatcher, Signal>
                SignalDispatcher

    Description
        Provides a means of dispatching a set of `Signals`. 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.
```
SignalDispatcher

Available since RTSJ 2.0

14.3.1.4.1 Constructors

SignalDispatcher(SchedulingParameters, RealtimeThreadGroup)

Signature

public SignalDispatcher(SchedulingParameters scheduling, RealtimeThreadGroup group)
throws IllegalRealtimeStateException

Description

Creates a new dispatcher, whose dispatching thread runs with the given SchedulingParameters.

Parameters

scheduling—Parameters for scheduling this dispatcher.

group—Container for this dispatcher.

Throws

IllegalRealtimeStateException—when the intersection of affinity in schedule
and the affinity of group does not correspond to a valid affinity.

SignalDispatcher(SchedulingParameters)

Signature

public SignalDispatcher(SchedulingParameters scheduling)
throws IllegalRealtimeStateException

Description

Creates a new dispatcher, whose dispatching thread runs with the given SchedulingParameters.
14 Interprocess Signaling

SignalDispatcher

Parameters
scheduling—For scheduling this dispatcher.

Throws
IllegalRealtimeStateException—when the intersection of affinity in schedule and the affinity of the current thread group does not correspond to a valid affinity.

14.3.1.4.2 Methods

setDefaultDispatcher(SignalDispatcher)

Signature
public static void
setDefaultDispatcher(SignalDispatcher dispatcher)

Description
Sets the system default signal dispatcher.

Parameters
dispatcher—An instance to be used when no dispatcher is provided. When null, the signal dispatcher is set to the original system default.

register(Signal)

Signature
public synchronized void
register(Signal signal)
throws RegistrationException,
         IllegalRealtimeStateException,
         IllegalRealtimeArgumentException

Description
Registers a POSIX signal with this dispatcher.

Parameters
signal—The signal instance to register.

Throws
RegistrationException—when Signal is already registered.
javax.realtime.IllegalRealtimeStateException—when this object has been destroyed.
javax.realtime.IllegalRealtimeArgumentException—when Signal is not stopped.

deregister(Signal)

Signature
public synchronized void
deregister(Signal signal)
throws DeregistrationException,
    IllegalRealtimeStateException,
    IllegalRealtimeArgumentException

Description
Deregisters the POSIX Signal from this dispatcher.

Parameters
signal—The signal instance to deregister.

Throws
DeregistrationException—when Signal is not already registered.
javax.realtime.IllegalRealtimeStateException—when this object has been destroyed.
javax.realtime.IllegalRealtimeArgumentException—when Signal is not stopped.

destroy

Signature
public void
destroy()
throws IllegalRealtimeStateException

Description
Releases all resources thereby making the dispatcher unusable.

Throws
javax.realtime.IllegalRealtimeStateException—when called on a dispatcher that has one or more registered Signal objects.
14 Interprocess Signaling

14.4 Rationale

POSIX is the most widely supported standard for operating systems, both conventional and realtime. Providing support for sending and receiving signals as encapsulated in the Signal and RealtimeSignal enables realtime java programs to interact, not just with the environment, but also other processes in a system. Even for systems that are not strictly POSIX compatible, one can implement this interface for encapsulating similar functionality in a common API.

The old interface was not consistent with the event and handler model used by Timer and other AsyncEventHandler types. Providing a new module for posix signal handling was used as a means of fixing this inconsistency. Thus the javax.realtime.posix contains new classes for interacting with signals using the event and handler model, as well as adding support for realtime signals.

The Signal and RealtimeSignal classes are singletons for each underlying signal. This provides minimum delay, but makes isolation more difficult. For OSGi and other modular platforms, this can be circumvented at a small additional cost. The application just needs to provide a handler for each isolation group that just dispatches to a secondary event for handlers in that group. This is the same mechanism that can be used to emulate the deprecated AsyncEvent.bindTo(String).
Chapter 15
System and Options

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. There are even ones that vary in their degree of compliance. Also, the type of garbage collection provided may vary from one implementation to another. This specification offers the means and facilities to manage these differences by 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 class that provides some basic information about the garbage collector.

15.1 Semantics

There are three classes with semantics that do not fall into other categories: `RealtimeSystem`, `RealtimeSecurity`, and `GarbageCollection`. Their overall semantics is detailed below. Thereafter, semantics applying to methods, constructors, and fields of these classes are provided.

15.1.1 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 2.0, a program can also ask what modules are supported. The enumeration `RTSJModule` supports this capability.
15.1.2 Realtime Security

Security for the classes in `javax.realtime` and its subpackages is provided by a set of permission classes, all of which are subclasses of `RealtimePermission`. These classes control access to key realtime features. Particularly critical is access to memory outside the heap. Core RTSJ features also have security checks. These should enable an application to restrict the use of the RTSJ, particularly for dynamically loaded code. Of particular concern are classes that can create or control resources, such as creating threads, both explicitly and implicitly, controlling scheduling and affinity, creating persistent objects, and accessing resources outside RTSJ memory areas.

Detailed information is provided in the class documentation below, where the arguments types are described in the subsequent table.

- **AffinityPermission**

<table>
<thead>
<tr>
<th>Method</th>
<th>Target</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Affinity.setProcessorAddedEvent</code></td>
<td>Group</td>
<td>monitor</td>
</tr>
<tr>
<td><code>Affinity.setProcessorRemovedEvent</code></td>
<td>Group</td>
<td>monitor</td>
</tr>
</tbody>
</table>

- **CoreMemoryPermission**

<table>
<thead>
<tr>
<th>Method</th>
<th>Target</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>MemoryArea.executeInArea</code></td>
<td>Area</td>
<td>enter</td>
</tr>
<tr>
<td><code>MemoryArea.enter</code></td>
<td>Area</td>
<td>enter</td>
</tr>
<tr>
<td><code>MemoryArea.newInstance</code></td>
<td>---</td>
<td>allocate</td>
</tr>
<tr>
<td><code>MemoryArea.newInstance</code></td>
<td>---</td>
<td>allocate</td>
</tr>
</tbody>
</table>

- **SchedulingPermission**

<table>
<thead>
<tr>
<th>Method</th>
<th>Target</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Scheduler.reschedule</code></td>
<td>Group</td>
<td>control</td>
</tr>
<tr>
<td><code>ProcessingGroup.enforceCost</code></td>
<td>Group</td>
<td>control</td>
</tr>
<tr>
<td><code>ProcessingGroup.setPeriod</code></td>
<td>Group</td>
<td>control</td>
</tr>
<tr>
<td><code>RealtimeThreadGroup.setMaxEligibility</code></td>
<td>Group</td>
<td>tune</td>
</tr>
<tr>
<td><code>ProcessingGroup.setMaximumCost</code></td>
<td>Group</td>
<td>tune</td>
</tr>
<tr>
<td><code>ProcessingGroup.setMinimumCost</code></td>
<td>Group</td>
<td>tune</td>
</tr>
<tr>
<td><code>ProcessingGroup.setCostOverrunHandler</code></td>
<td>Group</td>
<td>monitor</td>
</tr>
<tr>
<td><code>ProcessingGroup.setCostUnderrunHandler</code></td>
<td>Group</td>
<td>monitor</td>
</tr>
<tr>
<td><code>Scheduler.setDefaultScheduler</code></td>
<td>---</td>
<td>system</td>
</tr>
<tr>
<td><code>ProcessingGroup.setGranularity</code></td>
<td>---</td>
<td>system</td>
</tr>
<tr>
<td><code>MonitorControl.setMonitorControl</code></td>
<td>---</td>
<td>system</td>
</tr>
</tbody>
</table>
### TaskPermission

<table>
<thead>
<tr>
<th>Method</th>
<th>Target</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>ActiveEvent.start</td>
<td>Group</td>
<td>control</td>
</tr>
<tr>
<td>ActiveEvent.stop</td>
<td>Group</td>
<td>control</td>
</tr>
<tr>
<td>AsyncBaseEvent.enable</td>
<td>Group</td>
<td>control</td>
</tr>
<tr>
<td>AsyncBaseEvent.disable</td>
<td>Group</td>
<td>control</td>
</tr>
<tr>
<td>Schedulable.setDaemon</td>
<td>Group</td>
<td>control</td>
</tr>
<tr>
<td>Clock.Clock</td>
<td>—</td>
<td>create</td>
</tr>
<tr>
<td>RealtimeThread.RealtimeThread</td>
<td>—</td>
<td>create</td>
</tr>
<tr>
<td>Timer.Timer</td>
<td>—</td>
<td>create</td>
</tr>
<tr>
<td>AsyncBaseEvent.addHandler</td>
<td>Group</td>
<td>handle</td>
</tr>
<tr>
<td>AsyncBaseEvent.setHandler</td>
<td>Group</td>
<td>handle, override</td>
</tr>
<tr>
<td>AsyncBaseEvent.removeHandler</td>
<td>Group</td>
<td>override</td>
</tr>
<tr>
<td>ActiveEventDispatcher.register</td>
<td>—</td>
<td>system</td>
</tr>
<tr>
<td>ActiveEventDispatcher.deregister</td>
<td>—</td>
<td>system</td>
</tr>
<tr>
<td>ActiveEventDispatcher.destroy</td>
<td>—</td>
<td>system</td>
</tr>
<tr>
<td>ConfigurationParameters.setDefaultRunner</td>
<td>—</td>
<td>system</td>
</tr>
<tr>
<td>Clock.setRealtimeClock</td>
<td>—</td>
<td>system</td>
</tr>
<tr>
<td>Clock.setUniversalClock</td>
<td>—</td>
<td>system</td>
</tr>
<tr>
<td>TimeDispatcher.setDefaultDispatcher</td>
<td>—</td>
<td>system</td>
</tr>
</tbody>
</table>

### POSIXPermission

<table>
<thead>
<tr>
<th>Method</th>
<th>Target</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>RealtimeSignal.addHandler</td>
<td>Signals</td>
<td>handle</td>
</tr>
<tr>
<td>RealtimeSignal.setHandler</td>
<td>Signals</td>
<td>handle, override</td>
</tr>
<tr>
<td>RealtimeSignal.removeHandler</td>
<td>Signals</td>
<td>override</td>
</tr>
<tr>
<td>RealtimeSignal.send</td>
<td>Signals</td>
<td>send</td>
</tr>
<tr>
<td>RealtimeSignal.start</td>
<td>Signals</td>
<td>control</td>
</tr>
<tr>
<td>RealtimeSignal.stop</td>
<td>Signals</td>
<td>control</td>
</tr>
<tr>
<td>RealtimeSignalDispatcher.</td>
<td>—</td>
<td>system</td>
</tr>
<tr>
<td>setDefaultDispatcher</td>
<td>—</td>
<td>system</td>
</tr>
<tr>
<td>Signal.addHandler</td>
<td>Signals</td>
<td>handle</td>
</tr>
<tr>
<td>Signal.setHandler</td>
<td>Signals</td>
<td>handle, override</td>
</tr>
<tr>
<td>Signal.removeHandler</td>
<td>Signals</td>
<td>override</td>
</tr>
<tr>
<td>Signal.send</td>
<td>Signals</td>
<td>send</td>
</tr>
<tr>
<td>Signal.start</td>
<td>Signals</td>
<td>control</td>
</tr>
<tr>
<td>Signal.stop</td>
<td>Signals</td>
<td>control</td>
</tr>
<tr>
<td>SignalDispatcher.setDefaultDispatcher</td>
<td>—</td>
<td>system</td>
</tr>
</tbody>
</table>
### System and Options

- **DirectMemoryPermission**

<table>
<thead>
<tr>
<th>Method</th>
<th>Target</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>DirectMemoryBufferFactory.</td>
<td>Addresses</td>
<td>define</td>
</tr>
<tr>
<td>DirectMemoryBufferFactory</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DirectMemoryBufferFactory.allocateByteBuffer</td>
<td>Store map</td>
<td></td>
</tr>
<tr>
<td>DirectMemoryBufferFactory.free</td>
<td>Group</td>
<td>override</td>
</tr>
</tbody>
</table>

- **HappeningPermission**

<table>
<thead>
<tr>
<th>Method</th>
<th>Target</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Happening.createId</td>
<td></td>
<td>create</td>
</tr>
<tr>
<td>Happening.Happening</td>
<td></td>
<td>create</td>
</tr>
<tr>
<td>Happening.addHandler</td>
<td>Group</td>
<td>handle</td>
</tr>
<tr>
<td>Happening.setHandler</td>
<td>Group</td>
<td>handle, override</td>
</tr>
<tr>
<td>Happening.removeHandler</td>
<td>Group</td>
<td>override</td>
</tr>
<tr>
<td>HappeningDispatcher.setDefaultDispatcher</td>
<td></td>
<td>system</td>
</tr>
</tbody>
</table>

- **RawMemoryPermission**

<table>
<thead>
<tr>
<th>Method</th>
<th>Target</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>RawMemoryFactory.register</td>
<td>Addresses</td>
<td>define</td>
</tr>
<tr>
<td>RawMemoryFactory.deregister</td>
<td>Addresses</td>
<td>define</td>
</tr>
<tr>
<td>RawMemoryFactory.deregister</td>
<td>Addresses</td>
<td>define</td>
</tr>
<tr>
<td>RawMemoryFactory.createRawByte</td>
<td>Store map</td>
<td></td>
</tr>
<tr>
<td>RawMemoryFactory.createRawByteReader</td>
<td>Store map</td>
<td></td>
</tr>
<tr>
<td>RawMemoryFactory.createRawByteWriter</td>
<td>Store map</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RawMemoryFactory.createRawDouble</td>
<td>Store map</td>
<td></td>
</tr>
<tr>
<td>RawMemoryFactory.createRawDoubleReader</td>
<td>Store map</td>
<td></td>
</tr>
<tr>
<td>RawMemoryFactory.createRawDoubleWriter</td>
<td>Store map</td>
<td></td>
</tr>
</tbody>
</table>

- **PhysicalMemoryPermission**

<table>
<thead>
<tr>
<th>Method</th>
<th>Target</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>PhysicalMemoryFactory.associate</td>
<td>Addresses</td>
<td>define</td>
</tr>
<tr>
<td>PhysicalMemoryFactory.createImmortalMemory</td>
<td>Store</td>
<td>map</td>
</tr>
<tr>
<td>PhysicalMemoryFactory.createLTMemory</td>
<td>Store</td>
<td>map</td>
</tr>
<tr>
<td>PhysicalMemoryFactory.createPinnableMemory</td>
<td>Store</td>
<td>map</td>
</tr>
<tr>
<td>PhysicalMemoryFactory.createStackedMemory</td>
<td>Store</td>
<td>map</td>
</tr>
</tbody>
</table>

- **ScopedMemoryPermission**

<table>
<thead>
<tr>
<th>Method</th>
<th>Target</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>LTMemory.LTMemory</td>
<td>Store</td>
<td>map</td>
</tr>
<tr>
<td>PinnableMemory.PinnableMemory</td>
<td>Store</td>
<td>map</td>
</tr>
<tr>
<td>StackedMemory.StackedMemory</td>
<td>Store</td>
<td>map</td>
</tr>
<tr>
<td>ScopedMemory.joinAndEnter</td>
<td>Area</td>
<td>enter</td>
</tr>
</tbody>
</table>
In order to make the system easier to understand, targets are kept as consistent as possible in accordance with the following table.

<table>
<thead>
<tr>
<th>Target</th>
<th>Values</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addresses</td>
<td>a set of address ranges or *</td>
<td>0x10000100-0x10000200,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x10000400-0x10000500</td>
</tr>
<tr>
<td>Area</td>
<td>a memory area type or *</td>
<td>StackedMemory</td>
</tr>
<tr>
<td>Group</td>
<td>either current or *</td>
<td>current</td>
</tr>
<tr>
<td>Signals</td>
<td>a list of signal names or *</td>
<td>INT, STOP</td>
</tr>
<tr>
<td>Store</td>
<td>an amount of backing store or *</td>
<td>1024k</td>
</tr>
<tr>
<td>—</td>
<td>no specific target</td>
<td></td>
</tr>
</tbody>
</table>

Similarly, each action is also used for similar functions in different permission classes in accordance with the following table.

<table>
<thead>
<tr>
<th>Action</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>allocate</td>
<td>allows critical memory allocation</td>
</tr>
<tr>
<td>control</td>
<td>allows changing the mode of realtime tasks</td>
</tr>
<tr>
<td>create</td>
<td>allows creating critical realtime objects</td>
</tr>
<tr>
<td>define</td>
<td>allows defining a range of memory for later use</td>
</tr>
<tr>
<td>enter</td>
<td>allows entering a memory area</td>
</tr>
<tr>
<td>handle</td>
<td>allows event handling</td>
</tr>
<tr>
<td>map</td>
<td>allows mapping defined memory into the system</td>
</tr>
<tr>
<td>monitor</td>
<td>allows resource monitoring</td>
</tr>
<tr>
<td>override</td>
<td>allows overriding code from another part of the system</td>
</tr>
<tr>
<td>send</td>
<td>allows sending signals</td>
</tr>
<tr>
<td>system</td>
<td>allows a critical operation</td>
</tr>
<tr>
<td>tune</td>
<td>allows system tuning</td>
</tr>
</tbody>
</table>

15.1.3 GarbageCollection

It is extremely difficult to characterize garbage collectors in a uniform manner. The only information that can be provided by all collectors is the preemption latency. Each implementation may provide its own subclass of GarbageCollector to provide additional information, which may be queried via reflection.

15.1.4 Compliance Version

Determining the current version is supported by a system property. When an application calls 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 the version defined by this specification should return the string "2.0.0".
15.2 javax.realtime

15.2.1 Enumerations

15.2.1.1 RTSJModule

public enum RTSJModule

Inheritance
java.lang.Object
    java.lang.Enum<RTSJModule>
        RTSJModule

Description
Modules an RTSJ implementation may provide.

Available since RTSJ 2.0

15.2.1.1.1 Enumeration Constants

CORE

public static final RTSJModule CORE

Description
Indicates the presence of the core module.

CONTROL

public static final RTSJModule CONTROL

Description
Indicates the presence of the CONTROL module.
15 System and Options

15.2.1.1.2 Methods

values

**Signature**

```java
public static javax.realtime.RTSJModule[] values()
```

**Description**

Gets all enumeration constants.
valueOf(String)

Signature

public static javax.realtime.RTSJModule
valueOf(String name)

Description

Gets enumeration constants corresponding to name.

value

Signature

public int
value()

Description

Determines the numeric value of an element of this enumeration. This value can be used in bit sets to determine the presence of the given element.

Returns

a number with a single bit set representing this element.

in(int)

Signature

public boolean
in(int value)

Description

Given an int representing a set of enumeration elements via bit value, sees whether or not this element is contained within that set.

Parameters

value—The set to test against.

Returns

true when and only when value has the bit set that represents this.
15.2.2 Classes

15.2.2.1 AffinityPermission

```java
class AffinityPermission {
    // Inheritance
    inherits java.lang.Object
    inherits java.security.Permission
    RealtimePermission
    AffinityPermission

    Description
    The core module provides a permission for the security manager to administer
    CPU affinities. The following table describes the actions to check. Each permission
    can be limited to the current `ThreadGroup` by specifying the target `group` or it
    can apply to all, either with no target or through the target `*`.

<table>
<thead>
<tr>
<th>Action Name</th>
<th>Description</th>
<th>Risks of grant</th>
</tr>
</thead>
<tbody>
<tr>
<td>control</td>
<td>Changes the affinity of a task</td>
<td>CPU Assignment Risk</td>
</tr>
<tr>
<td>monitor</td>
<td>One could change the event used to monitor adding or removing a processor.</td>
<td>Lost Events Risk</td>
</tr>
</tbody>
</table>

The risk classes are defined in `RealtimePermission`.

Available since RTSJ 2.0

15.2.2.1.1 Constructors

AffinityPermission(String, String)

Signature

806
RTSJ 2.0 (Draft 66)
public
AffinityPermission(String target,
String actions)

Description
Creates a new AffinityPermission object for a given action, i.e., the symbolic
name of an action. The target string specifies additional limitations on the
action.

Parameters
target—Specifies the domain for the action, or * for no limit on the permission.
actions—The names of the actions to allow, or * for all actions.

Throws
NullPointerException—when action is null.
IllegalRealtimeArgumentException—when target or action is empty.

15.2.2.1.2 Methods

equals(Object)

Signature
public boolean
equals(Object other)

Description
Compare two Permission objects for equality.

Parameters
other—is the object with which to compare.

Returns
true when yes and false otherwise.

getActions

Signature
public java.lang.String
getActions()
Description
Obtain the actions as a String in canonical form.

Returns
the actions represented as a string.

hashCode

Signature
public int
hashCode()

Description
Obtain the hash code value for this object.

Returns
the hash code value.

implies(Permission)

Signature
public boolean
implies(Permission permission)

Description
Checks if the given permission’s actions are 'implied by' this object’s actions. This method is used by the AccessController to determine whether or not a requested permission is implied by another permission that is known to be valid in the current execution context.

Parameters
permission—is the permission to check.

Returns
true when yes and false otherwise.

15.2.2.2 CoreMemoryPermission

public class CoreMemoryPermission

Inheritance

808	RTSJ 2.0 (Draft 66)
Memory permission are divided into those for the core module and those for the memory module. The following table describes the actions to check for the core module. The name of a primordial memory area type can be given.

<table>
<thead>
<tr>
<th>Action Name</th>
<th>Description</th>
<th>Risks of grant</th>
</tr>
</thead>
<tbody>
<tr>
<td>enter</td>
<td>Allows execution in a given area.</td>
<td>Memory Leak Risk</td>
</tr>
<tr>
<td>allocate</td>
<td>Allows the creation of an object in Immortal without entering it.</td>
<td>Can cause a Memory Leak Risk</td>
</tr>
</tbody>
</table>

The wildcard *, or no target, allows access to primordial memory area. The risk classes are defined in RealtimePermission.

Available since RTSJ 2.0

15.2.2.2.1 Constructors

CoreMemoryPermission(String)

Signature

```java
public CoreMemoryPermission(String actions)
```

Description

Creates a new CoreMemoryPermission object for a given action, i.e., the symbolic name of an action.

Parameters

actions—The names of the actions to allow, or * for all actions.

Throws

RTSJ 2.0 (Draft 66) 809
NullPointerException—when action is null.

IllegalRealtimeArgumentException—when action is empty.

CoreMemoryPermission(String, String)

Signature

```java
public CoreMemoryPermission(String target, String actions)
```

Description

Creates a new CoreMemoryPermission object for a given action, i.e., the symbolic name of an action.

Parameters

- **target**—The names of the memory area class for the action, or * for all memory areas.
- **actions**—The names of the actions to allow, or * for all actions.

Throws

- NullPointerException—when action is null.
- IllegalRealtimeArgumentException—when action is empty.

15.2.2.2.2 Methods

equals(Object)

Signature

```java
public boolean equals(Object other)
```

Description

Compare two Permission objects for equality.

Parameters

- **other**—is the object with which to compare.

Returns

- true when yes and false otherwise.
getActions

Signature

    public java.lang.String
getActions()

Description
Obtain the actions as a String in canonical form.

Returns
the actions represented as a string.

hashCode

Signature

    public int
hashCode()

Description
Obtain the hash code value for this object.

Returns
the has code value.

implies(Permission)

Signature

    public boolean
implies(Permission permission)

Description
Checks if the given permission’s actions are "implied by" this object’s actions. This method is used by the AccessController to determine whether or not a requested permission is implied by another permission that is known to be valid in the current execution context.

Parameters
permission—is the permission to check.

Returns
true when yes and false otherwise.
15.2.2.3 GarbageCollector

public abstract class GarbageCollector

Inheritance
java.lang.Object
   GarbageCollector

Description
The system shall provide dynamic and static information characterizing the
temporal behavior and imposed overhead of any garbage collection algorithm
provided by the implementation. 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().

15.2.2.3.1 Methods

getPreemptionLatency

Signature
public abstract javax.realtime.RelativeTime
getPreemptionLatency()

Description
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.

Schedulables which may not use the heap 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 the release of a schedulable during garbage
collection, and the time the schedulable starts execution or gains full access to heap memory, whichever comes later.

**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.

### RealtimePermission

```java
public abstract class RealtimePermission
```

**Inheritance**

```java
java.lang.Object
  java.security.Permission
    RealtimePermission
```

**Description**

All permission classes in the RTSJ inherit from this class. The following table lists common risk classes that correspond to granting specific permissions.
<table>
<thead>
<tr>
<th>Risk Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Assignment Risk</td>
<td>Interferes with critical tasks by assigning too many other tasks to the same CPU.</td>
</tr>
<tr>
<td>Encapsulation Risk</td>
<td>Could break out of encapsulation.</td>
</tr>
<tr>
<td>External Risk</td>
<td>Could adversely effect other processes on the system.</td>
</tr>
<tr>
<td>Interference Risk</td>
<td>Could interfere with the function of other parts of the system.</td>
</tr>
<tr>
<td>Load Risk</td>
<td>Could increase the load on the system.</td>
</tr>
<tr>
<td>Lost Events Risk</td>
<td>Another task could no longer receive the expected events.</td>
</tr>
<tr>
<td>Memory Leak Risk</td>
<td>Could cause memory to be lost to the system.</td>
</tr>
<tr>
<td>Scheduling Risk</td>
<td>Interferes with the timeliness of other parts of the system.</td>
</tr>
<tr>
<td>Device Range Risk</td>
<td>Could specify memory outside the desired Device range.</td>
</tr>
<tr>
<td>Device Map Risk</td>
<td>Could map too much or too little Device memory.</td>
</tr>
<tr>
<td>DMA Range Risk</td>
<td>Could specify memory outside the desired DMA range.</td>
</tr>
<tr>
<td>DMA Map Risk</td>
<td>Could map too much or too little DMA memory for DMA.</td>
</tr>
<tr>
<td>Physical Range Risk</td>
<td>Could specify memory outside the desired Physical range.</td>
</tr>
<tr>
<td>Physical Map Risk</td>
<td>Could take too much memory.</td>
</tr>
</tbody>
</table>

**Available since** RTSJ 2.0

### 15.2.2.4.1 Constructors

**RealtimePermission(String)**
RealtimePermission

**Signature**

```java
protected
RealtimePermission(String action)
```

**Description**

Creates a new `RealtimePermission` object for a given action, i.e., the symbolic name of an action. The `target` string specifies additional limitations on the action.

**Parameters**

- `action`—The names of the actions to allow, or `*` for all actions.

**Throws**

- `NullPointerException`—when `action` is `null`
- `IllegalRealtimeArgumentException`—when `action` is empty.

**RealtimePermission(String, String)**

**Signature**

```java
protected
RealtimePermission(String target,
                    String actions)
```

**Description**

Creates a new `RealtimePermission` object for a given action, i.e., the symbolic name of an action. The `target` string specifies additional limitations on the action.

**Parameters**

- `target`—Specifies the domain for the action, or `*` for no limit on the permission.
- `actions`—The names of the actions to allow, or `*` for all actions.

**Throws**

- `NullPointerException`—when `action` is `null`
- `IllegalRealtimeArgumentException`—when `target` or `action` is empty.

### 15.2.2.4.2 Methods
equals(Permission)

**Signature**

```java
public abstract boolean equals(Permission other)
```

**Description**

Compare two Permission objects for equality.

**Parameters**

- `other` — is the object with which to compare.

**Returns**

- `true` when yes and `false` otherwise.

getActions

**Signature**

```java
public abstract String getActions()
```

**Description**

Obtain the actions as a String in canonical form.

**Returns**

- the actions represented as a string.

hashCode

**Signature**

```java
public abstract int hashCode()
```

**Description**

Obtain the hash code value for this object.

**Returns**

- the hash code value.
implies(Permission)

Signature
   public abstract boolean
   implies(Permission permission)

Description
   Checks if the given permission’s actions are "implied by" this object’s actions. This method is used by the AccessController to determine whether or not a requested permission is implied by another permission that is known to be valid in the current execution context.

Parameters
   permission—is the permission to check.

Returns
   true when yes and false otherwise.

15.2.2.5 RealtimeSystem

public class RealtimeSystem

Inheritance
   java.lang.Object
   RealtimeSystem

Description
   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.

15.2.2.5.1 Fields

BIG_ENDIAN

public static final byte 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**

```java
class RealtimeSystem {
    public static final byte LITTLE_ENDIAN = ...;
```

Description

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**

```java
class RealtimeSystem {
    public static final byte PDP_ENDIAN = ...;
```

Description

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.

Available since RTSJ 2.0

**CROSS_ENDIAN**

```java
class RealtimeSystem {
    public static final byte CROSS_ENDIAN = ...;
```

Description

Value indicating a mixed endian mode: the int 0x0A0B0C0D is stored in the byte sequence 0x0D, 0x0C, 0x0B, 0x0A, and the long 0x0102030405060708 is stored in the sequence 0x05, 0x06, 0x07, 0x08, 0x01, 0x02, 0x03, 0x04.

Available since RTSJ 2.0

818 RTSJ 2.0 (Draft 66)
BYTE_ORDER

public static final byte BYTE_ORDER

Description
The byte ordering of the underlying hardware.

Deprecated RTSJ 2.0

15.2.2.5.2 Methods

getByteOrder

Signature
public static byte getByteOrder()

Description
Obtains the byte order of the byte order of the system.

Returns
one of the defined byte order constants.

currentGC

Signature
public static javax.realtime.GarbageCollector currentGC()

Description
Returns a reference to the currently active garbage collector for the heap.

Returns
a GarbageCollector object which is the current collector collecting objects on the conventional Java heap.
getConcurrentLocksUsed

Signature
public static int
getConcurrentLocksUsed()

Description
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.

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, returns -1. Note that when the number of concurrent locks is not tracked, the number of available concurrent locks is effectively unlimited.

getMaximumConcurrentLocks

Signature
public static int
getMaximumConcurrentLocks()

Description
Gets the maximum number of locks that can be used concurrently without incurring an execution time increase as set by the setMaximumConcurrentLocks() methods.

Note that 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 gives the implementation a hint as to the maximum number of concurrent locks it should expect.

Returns
an integer, whose value is the maximum number of locks that can be in simultaneous use.
setMaximumConcurrentLocks(int)

Signature

public static void
setMaximumConcurrentLocks(int numLocks)

Description

Sets the anticipated maximum number of locks that may be held or waited on concurrently. Provides a hint to systems that use a monitor cache as to how much space to dedicate to the cache.

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().

setMaximumConcurrentLocks(int, boolean)

Signature

public static void
setMaximumConcurrentLocks(int number,
    boolean hard)

Description

Sets the anticipated maximum number of locks that may be held or waited on concurrently. Provides a limit for the size of the monitor cache on systems that provide one when hard is true.

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().

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 is thrown. When the system does not limit use of concurrent locks, this parameter is silently ignored.
setSecurityManager(RealtimeSecurity)

**Signature**
```
public static void
setSecurityManager(RealtimeSecurity manager)
```

**Description**
Sets a new realtime security manager.

**Parameters**
- `manager` — A `RealtimeSecurity` object which will become the new security manager.

**Throws**
- `RealtimeSecurityException` — when security manager has already been set.

getInitialMonitorControl

**Signature**
```
public static javax.realtime.MonitorControl
getInitialMonitorControl()
```

**Description**
Returns the monitor control object that represents the initial monitor control policy.

**Returns**
the initial monitor control policy.

**Available since** RTSJ 1.0.1

supports(RTSJModule)

**Signature**
```
public static boolean
supports(RTSJModule module)
```

**Description**
Determines whether or not a particular module is supported.

**Parameters**

822 RTSJ 2.0 (Draft 66)
module—The identifier of the module to be checked for support.

Returns
true when module is supported; otherwise false.

Available since RTSJ 2.0

modules

Signature
public static int
modules()

Description
The set of modules supported.

Returns
an integer representing all the modules supported.

Available since RTSJ 2.0

hasUniversalClock

Signature
public static boolean
hasUniversalClock()

Description
Determines whether or not this system supports a universal time clock.

Returns
ture when the system can provide a universal time clock.

canEnforceCost

Signature
public static boolean
canEnforceCost()

Description
Determines whether or not hard cost enforcement is supported.

**Returns**

true when cost enforcement is supported; otherwise false.

**Available since** RTSJ 2.0

---

**canEnforceAllocationRate**

**Signature**

```java
public static boolean canEnforceAllocationRate()
```

**Description**

Determines whether or not allocation rate enforcement is supported.

**Returns**

true when allocation rate enforcement is supported, otherwise false.

**Available since** RTSJ 2.0

---

**setDefaultConfiguration(ConfigurationParameters)**

**Signature**

```java
public static void setDefaultConfiguration(ConfigurationParameters parameters)
```

**Description**

Sets the default configuration used to by tasks that are not explicitly provided with one.

**Parameters**

- **parameters**—contains the new default configuration.

**Available since** RTSJ 2.0

---

**getDefaultValue**

**Signature**

```
824 RTSJ 2.0 (Draft 66)
```
public static javax.realtime.ConfigurationParameters getDefaultConfiguration()

Description

Determines the current configurations used by tasks that are not explicitly provided with one.

Returns

the current configurations.

Available since RTSJ 2.0

15.2.2.6 SchedulingPermission

public class SchedulingPermission

Inheritance

java.lang.Object
java.security.Permission
    RealtimePermission
        SchedulingPermission

Description

Scheduling has its own security permission that covers APIs in Scheduler, RealtimeThreadGroup, and ProcessingGroup. The following table describes the actions to check. Either the permission is limited to the current ThreadGroup by specifying the target group or it can apply to all, either with no target or the target *.

RTSJ 2.0 (Draft 66) 825
<table>
<thead>
<tr>
<th>Action Name</th>
<th>Description</th>
<th>Risks of grant</th>
</tr>
</thead>
<tbody>
<tr>
<td>system</td>
<td>Changes system wide behavior, such as how scheduling is done.</td>
<td>Scheduling Risk</td>
</tr>
<tr>
<td>control</td>
<td>Changes someone else’s scheduling limits or raises your own limits.</td>
<td>Scheduling Risk</td>
</tr>
<tr>
<td>monitor</td>
<td>Adds overrun and underrun handlers to someone else’s group.</td>
<td>Load Risk</td>
</tr>
<tr>
<td>tune</td>
<td>Changes task scheduling.</td>
<td>Scheduling Risk</td>
</tr>
</tbody>
</table>

The wildcard * is allowed for both signal and action. The risk classes are defined in `RealtimePermission`.

**Available since** RTSJ 2.0

### 15.2.2.6.1 Constructors

**SchedulingPermission(String, String)**

**Signature**

```java
public SchedulingPermission(String target,
                           String actions)
```

**Description**

Creates a new `SchedulingPermission` object for a given action, i.e., the symbolic name of an action. The `target` string specifies additional limitations on the action.

**Parameters**

- **target**—Specifies the domain for the action, or * for no limit on the permission.
- **actions**—The names of the actions to allow, or * for all actions.
**SchedulingPermission**

Throws

NullPointerException—when action is null.

IllegalRealtimeArgumentException—when target or action is empty.

**SchedulingPermission(String)**

Signature

```public SchedulingPermission(String actions)`

Description

Creates a new SchedulingPermission object for a given action, i.e., the symbolic name of an action.

Parameters

actions—The names of the actions to allow, or * for all actions.

Throws

NullPointerException—when action is null.

IllegalRealtimeArgumentException—when action is empty.

15.2.2.6.2 Methods

**equals(Object)**

Signature

```public boolean equals(Object other)`

Description

Compare two Permission objects for equality.

Parameters

other—is the object with which to compare.

Returns

ture when yes and false otherwise.
getActions

Signature
public java.lang.String
getActions()

Description
Obtain the actions as a String in canonical form.

Returns
the actions represented as a string.

hashCode

Signature
public int
hashCode()

Description
Obtain the hash code value for this object.

Returns
the hash code value.

implies(Permission)

Signature
public boolean
implies(Permission permission)

Description
Checks if the given permission’s actions are 'implied by' this object’s actions.
This method is used by the AccessController to determine whether or not a requested permission is implied by another permission that is known to be valid in the current execution context.

Parameters
permission—is the permission to check.

Returns
true when yes and false otherwise.
15.2.2.7 TaskPermission

public class TaskPermission

Inheritance
java.lang.Object
java.security.Permission
RealtimePermission
TaskPermission

Description
Task permissions are for controlling threads and handlers, as well as creating clocks. The following table describes the actions to check. For all but `create`, which takes no target, either the permission is limited to the current `ThreadGroup` by specifying the target `group`, or it can apply to all with no target or the target `*`.

<table>
<thead>
<tr>
<th>Action Name</th>
<th>Description</th>
<th>Risks of grant</th>
</tr>
</thead>
<tbody>
<tr>
<td>control</td>
<td>Enables controlling the activity of a task.</td>
<td>Scheduling Risk</td>
</tr>
<tr>
<td>create</td>
<td>Enables new thread, timers, and tasks to be created.</td>
<td>Scheduling Risk</td>
</tr>
<tr>
<td>handle</td>
<td>Adds handler to an asynchronous event.</td>
<td>Load Risk</td>
</tr>
<tr>
<td>override</td>
<td>Interference Risk.</td>
<td></td>
</tr>
<tr>
<td>system</td>
<td>Changes system wide tasking behavior.</td>
<td>Load and Scheduling Risk</td>
</tr>
</tbody>
</table>

The risk classes are defined in `RealtimePermission`.

Available since RTSJ 2.0

15.2.2.7.1 Constructors
TaskPermission(String, String)

**Signature**

```java
class TaskPermission {
    public TaskPermission(String target, String actions) {
    }
}
```

**Description**

Creates a new `TaskPermission` object for a given action, i.e., the symbolic name of an action. The `target` string specifies additional limitations on the action.

**Parameters**

- `target`—Specifies the domain for the action, or `*` for no limit on the permission.
- `actions`—The names of the actions to allow, or `*` for all actions.

**Throws**

- `NullPointerException`—when `action` is `null`.
- `IllegalRealtimeArgumentException`—when `target` or `action` is empty.

TaskPermission(String)

**Signature**

```java
class TaskPermission {
    public TaskPermission(String actions) {
    }
}
```

**Description**

Creates a new `TaskPermission` object for a given action, i.e., the symbolic name of an action.

**Parameters**

- `actions`—The names of the actions to allow, or `*` for all actions.

**Throws**

- `NullPointerException`—when `action` is `null`.
- `IllegalRealtimeArgumentException`—when `action` is empty.

15.2.2.7.2 Methods
equals(Object)

Signature
   public boolean
equals(Object other)

Description
   Compare two Permission objects for equality.

Parameters
   other—is the object with which to compare.

Returns
   true when yes and false otherwise.

getActions

Signature
   public java.lang.String
   getActions()

Description
   Obtain the actions as a String in canonical form.

Returns
   the actions represented as a string.

hashCode

Signature
   public int
   hashCode()

Description
   Obtain the hash code value for this object.

Returns
   the hash code value.
implies(Permission)

Signature
public boolean
implies(Permission permission)

Description
Checks if the given permission’s actions are 'implied by' this object’s actions. This method is used by the AccessController to determine whether or not a requested permission is implied by another permission that is known to be valid in the current execution context.

Parameters
permission—is the permission to check.

Returns
true when yes and false otherwise.
15.3  javax.realtime.device

15.3.1  Classes

15.3.1.1  DirectMemoryPermission

public class DirectMemoryPermission

Inheritance
java.lang.Object
  java.security.Permission
    javax.realtime.RealtimePermission
      DirectMemoryPermission

Description
The device management module provides a permission for the security manager to control access to DMA memory. The following table describes the actions to check. An address range can be given as the target or * for any legal address range.

<table>
<thead>
<tr>
<th>Action Name</th>
<th>Description</th>
<th>Risks of grant</th>
</tr>
</thead>
<tbody>
<tr>
<td>define</td>
<td>Defines a DMA address range for use by raw memory.</td>
<td>DMA Range Risk</td>
</tr>
<tr>
<td>map</td>
<td>Maps a DMA address range for use by a DMA object.</td>
<td>DMA Mapping risk</td>
</tr>
</tbody>
</table>

The risk classes are defined in javax.realtime.RealtimePermission.

Available since RTSJ 2.0

15.3.1.1.1  Constructors

DirectMemoryPermission(String, String)

Signature

RTSJ 2.0 (Draft 66)
public DirectMemoryPermission(String target, 
String actions)

Description

Creates a new DirectMemoryPermission object for a given action, i.e., the 
symbolic name of an action. The target string specifies additional limitations 
on the action.

Parameters

target—Specifies the domain for the action, or * for no limit on the permission.
actions—The names of the actions to allow, or * for all actions.

Throws

NullPointerException—when action is null.
javax.realtime.IllegalRealtimeArgumentException—when target or action 
is empty.

15.3.1.1.2 Methods

equals(Object)

Signature

public boolean 
equals(Object other)

Description

Compare two Permission objects for equality.

Parameters

other—is the object with which to compare.

Returns

true when yes and false otherwise.

getActions

Signature

834 RTSJ 2.0 (Draft 66)
public java.lang.String
getActions()

Description
Obtain the actions as a String in canonical form.

Returns
the actions represented as a string.

hashCode

Signature
public int
hashCode()

Description
Obtain the hash code value for this object.

Returns
the has code value.

implies(Permission)

Signature
public boolean
implies(Permission permission)

Description
Checks if the given permission’s actions are "implied by" this object’s actions. This method is used by the AccessController to determine whether or not a requested permission is implied by another permission that is known to be valid in the current execution context.

Parameters
permission—is the permission to check.

Returns
true when yes and false otherwise.
15.3.1.2 HappeningPermission

public class HappeningPermission

Inheritance
java.lang.Object
  java.security.Permission
    javax.realtime.RealtimePermission
      HappeningPermission

Description
The device management module provides a permission for the security manager to control happenings. The following table describes the actions to check. For all but create, which takes no arguments, either the permission is limited to the current ThreadGroup by specifying the target group or it can apply to all, either with no target or with the target *.

<table>
<thead>
<tr>
<th>Action Name</th>
<th>Description</th>
<th>Risks of grant</th>
</tr>
</thead>
<tbody>
<tr>
<td>create</td>
<td>Enables new thread, timers, and tasks to be created.</td>
<td>Scheduling Risk</td>
</tr>
<tr>
<td>handle</td>
<td>Allows adding a handler to a Happening.</td>
<td>Load Risk</td>
</tr>
<tr>
<td>override</td>
<td>Enables handlers to be removed.</td>
<td>Interference Risk</td>
</tr>
<tr>
<td>system</td>
<td>Changes system’s wide happening behavior.</td>
<td>Scheduling and Load Risk</td>
</tr>
</tbody>
</table>

The risk classes are defined in javax.realtime.RealtimePermission.

Available since RTSJ 2.0

15.3.1.2.1 Constructors

HappeningPermission(String, String)

Signature

836 RTSJ 2.0 (Draft 66)
public
HappeningPermission(String target,
String actions)

Description

Creates a new HappeningPermission object for a given action, i.e., the symbolic
name of an action. The target string specifies additional limitations on the
action.

Parameters

target—Specifies the domain for the action, or * for no limit on the permission.
actions—The names of the actions to allow, or * for all actions.

Throws

NullPointerException—when action is null.
javax.realtime.IllegalRealtimeArgumentException—when target or action
is empty.

HappeningPermission(String)

Signature

public
HappeningPermission(String actions)

Description

Creates a new HappeningPermission object for a given action, i.e., the symbolic
name of an action.

Parameters

actions—The names of the actions to allow, or * for all actions.

Throws

NullPointerException—when action is null.
javax.realtime.IllegalRealtimeArgumentException—when action is empty.

15.3.1.2.2 Methods
equals(Object)

Signature
   public boolean
equals(Object other)

Description
   Compare two Permission objects for equality.

Parameters
   other—is the object with which to compare.

Returns
   true when yes and false otherwise.

getActions

Signature
   public java.lang.String
getActions()

Description
   Obtain the actions as a String in canonical form.

Returns
   the actions represented as a string.

hashCode

Signature
   public int
hashCode()

Description
   Obtain the hash code value for this object.

Returns
   the has code value.
RawMemoryPermission

15.3.1.3 RawMemoryPermission

public class RawMemoryPermission

Inheritance

java.lang.Object
java.security.Permission
javax.realtime.RealtimePermission
RawMemoryPermission

Description

The device management module provides a permission for the security manager to manage raw memory. The following table describes the actions to check. An address range can be given as the target or * for any.

<table>
<thead>
<tr>
<th>Action Name</th>
<th>Description</th>
<th>Risks of grant</th>
</tr>
</thead>
<tbody>
<tr>
<td>define</td>
<td>Defines a device address range for use by raw memory.</td>
<td>Device Range Risk</td>
</tr>
<tr>
<td>map</td>
<td>Maps a given amount of raw memory into a raw memory object.</td>
<td>Device Map Risk</td>
</tr>
</tbody>
</table>

RTSJ 2.0 (Draft 66)
The risk classes are defined in `javax.realtime.RealtimePermission`.

**Available since** RTSJ 2.0

### 15.3.1.3.1 Constructors

#### RawMemoryPermission(String, String)

**Signature**

```java
public RawMemoryPermission(String target,
    String actions)
  throws NullPointerException,
    IllegalRealtimeArgumentException
```

**Description**

Creates a new `RawMemoryPermission` object for a given action, i.e., the symbolic name of an action. The `target` string specifies additional limitations on the action.

**Parameters**

- `target`—Specifies the domain for the action, or * for no limit on the permission.
- `actions`—The names of the actions to allow, or * for all actions.

**Throws**

- `NullPointerException`—when `action` is `null`.
- `javax.realtime.IllegalRealtimeArgumentException`—when `target` or `action` is empty.

#### RawMemoryPermission(String)

**Signature**

```java
public RawMemoryPermission(String actions)
```

840 RTSJ 2.0 (Draft 66)
throws NullPointerException,
IllegalRealtimeArgumentException

Description

Creates a new RawMemoryPermission object for a given action, i.e., the symbolic name of an action.

Parameters

actions—The names of the actions to allow, or * for all actions.

Throws

NullPointerException—when action is null.
javax.realtime.IllegalRealtimeArgumentException—when action is empty.

15.3.1.3.2 Methods

equals(Object)

Signature

public boolean
equals(Object other)

Description

Compare two Permission objects for equality.

Parameters

other—is the object with which to compare.

Returns

ture when yes and false otherwise.

getActions

Signature

public java.lang.String
getActions()

Description

Obtain the actions as a String in canonical form.
Returns
the actions represented as a string.

hashCode

Signature
public int
hashCode()

Description
Obtain the hash code value for this object.

Returns
the has code value.

implies(Permission)

Signature
public boolean
implies(Permission permission)

Description
Checks if the given permission’s actions are ’implied by’ this object’s actions. This method is used by the AccessController to determine whether or not a requested permission is implied by another permission that is known to be valid in the current execution context.

Parameters
permission—is the permission to check.

Returns
true when yes and false otherwise.
15.4  javax.realtime.memory

15.4.1  Classes

15.4.1.1  PhysicalMemoryPermission

```java
public class PhysicalMemoryPermission
```

**Inheritance**

java.lang.Object
  java.security.Permission
    javax.realtime.RealtimePermission
      PhysicalMemoryPermission

**Description**

The alternate memory management module provides two permissions for the security manager to use. The following table describes the actions for checking the use of physical memory. An address range can be given as the target or * for any.

<table>
<thead>
<tr>
<th>Action Name</th>
<th>Description</th>
<th>Risks of grant</th>
</tr>
</thead>
<tbody>
<tr>
<td>define</td>
<td>Defines a physical address range for use by physical memory.</td>
<td>Physical Range Risk</td>
</tr>
<tr>
<td>map</td>
<td>Maps physical memory backing store for creating a memory area.</td>
<td>Physical Map Risk</td>
</tr>
</tbody>
</table>

The risk classes are defined in `javax.realtime.RealtimePermission`.

**Available since** RTSJ 2.0

15.4.1.1.1  Constructors

**PhysicalMemoryPermission**(String, String)

**Signature**

RTSJ 2.0 (Draft 66)
public
PhysicalMemoryPermission(String target,
    String actions)

Description
Creates a new DirectMemoryPermission object for a given action, i.e., the
symbolic name of an action. The target string specifies additional limitations
on the action.

Parameters
target—Specifies the domain for the action, or * for no limit on the permission.
actions—The names of the actions to allow, or * for all actions.

Throws
NullPointerException—when action is null.
javax.realtime.IllegalRealtimeArgumentException—when target or action
    is empty.

15.4.1.1.2 Methods

equals(Object)

Signature
    public boolean
equals(Object other)

Description
Compare two Permission objects for equality.

Parameters
other—is the object with which to compare.

Returns
true when yes and false otherwise.

getActions

Signature

844
RTSJ 2.0 (Draft 66)
public java.lang.String getActions()

Description
Obtain the actions as a String in canonical form.

Returns
the actions represented as a string.

hashCode

Signature
public int hashCode()

Description
Obtain the hash code value for this object.

Returns
the hash code value.

implies(Permission)

Signature
public boolean implies(Permission permission)

Description
Checks if the given permission’s actions are "implied by" this object’s actions. This method is used by the AccessController to determine whether or not a requested permission is implied by another permission that is known to be valid in the current execution context.

Parameters
permission—is the permission to check.

Returns
true when yes and false otherwise.
15 System and Options

15.4.1.2 ScopedMemoryPermission

public class ScopedMemoryPermission

Inheritance
java.lang.Object
   java.security.Permission
      javax.realtime.RealtimePermission
         ScopedMemoryPermission

Description
The alternate memory management module provides two permissions for the
security manager to use. The following table describes the actions for checking
the use of scoped memory. A signal name can be given as the target. The name
of a scoped memory area type can be given for enter and a maximum amount of
backing store can be used for global backing store.

<table>
<thead>
<tr>
<th>Action Name</th>
<th>Description</th>
<th>Risks of grant</th>
</tr>
</thead>
<tbody>
<tr>
<td>map</td>
<td>Uses a given amount of the global backing store.</td>
<td>Physical Map Risk</td>
</tr>
<tr>
<td>enter</td>
<td>Enters a scoped memory</td>
<td>Memory Risk</td>
</tr>
<tr>
<td>monitor</td>
<td>Visit root scoped memories</td>
<td>Encapsulation Risk</td>
</tr>
</tbody>
</table>

The wildcard * or no target allows access to any scoped memory area or
any amount of backing store. The risk classes are defined in javax.realtime.
RealtimePermission.

Available since RTSJ 2.0

15.4.1.2.1 Constructors

ScopedMemoryPermission(String, String)

Signature

846 RTSJ 2.0 (Draft 66)
public
ScopedMemoryPermission(String target,
    String actions)

Description
Creates a new DirectMemoryPermission object for a given action, i.e., the symbolic name of an action. The target string specifies additional limitations on the action.

Parameters
target—Specifies the domain for the action, or * for no limit on the permission.
actions—The names of the actions to allow, or * for all actions.

Throws
NullPointerException—when action is null.
javax.realtime.IllegalRealtimeArgumentException—when target or action is empty.

15.4.1.2.2 Methods

equals(Object)

Signature
    public boolean
    equals(Object other)

Description
Compare two Permission objects for equality.

Parameters
other—is the object with which to compare.

Returns
true when yes and false otherwise.

getActions

Signature
public java.lang.String
getActions()

Description
Obtain the actions as a String in canonical form.

Returns
the actions represented as a string.

hashCode

Signature
public int
hashCode()

Description
Obtain the hash code value for this object.

Returns
the has code value.

implies(Permission)

Signature
public boolean
implies(Permission permission)

Description
Checks if the given permission’s actions are 'implied by' this object’s actions. This method is used by the AccessController to determine whether or not a requested permission is implied by another permission that is known to be valid in the current execution context.

Parameters
permission—is the permission to check.

Returns
true when yes and false otherwise.
15.5  javax.realtime.posix

15.5.1  Classes

15.5.1.1  POSIXPermission

```java
public class POSIXPermission
```

*Inheritance*

```java
java.lang.Object
   java.security.Permission
      javax.realtime.RealtimePermission
         POSIXPermission
```

*Description*

The POSIX module provides one permission class for the security manager to use. This permission applies to both `Signal` and `RealtimeSignal`. The following table describes the actions for checking the use of signals. A signal name or * can be given as the target.

<table>
<thead>
<tr>
<th>Action Name</th>
<th>Description</th>
<th>Risks of grant</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>Adds a handle to the given signal.</td>
<td>Load Risk</td>
</tr>
<tr>
<td>override</td>
<td>Removes a handler that belongs to another realtime thread group.</td>
<td>Interference Risk</td>
</tr>
<tr>
<td>send</td>
<td>Sends a given signal.</td>
<td>External Risk</td>
</tr>
<tr>
<td>control</td>
<td>Starts or stops this signal.</td>
<td>Scheduling Risk</td>
</tr>
<tr>
<td>system</td>
<td>Changes system’s wide signalling behavior.</td>
<td>Scheduling and Load Risk</td>
</tr>
</tbody>
</table>

The wildcard * is allowed for both signal and action. The risk classes are defined in `javax.realtime.RealtimePermission`.

*Available since* RTSJ 2.0

15.5.1.1.1  Constructors
POSIXPermission(String, String)

Signature
   public
   POSIXPermission(String target,
                    String actions)

Description
   Creates a new POSIXPermission object for a given action, i.e., the symbolic
   name of an action. The target string specifies for which POSIX signal the action
   applies.

Parameters
   target—Specifies the domain for the action, or * for no limit on the permission.
   actions—The names of the actions to allow, or * for all actions.

Throws
   NullPointerException—when action is null.
   javax.realtime.IllegalRealtimeArgumentException—when target or action
   is empty.

POSIXPermission(String)

Signature
   public
   POSIXPermission(String actions)

Description
   Creates a new POSIXPermission object for a given action, i.e., the symbolic
   name of an action.

Parameters
   actions—The names of the actions to allow, or * for all actions.

Throws
   NullPointerException—when action is null.
   javax.realtime.IllegalRealtimeArgumentException—when action is empty.
15.5.1.1.2 Methods

equals(Object)

Signature
   public boolean
   equals(Object other)

Description
   Compare two Permission objects for equality.

Parameters
   other—is the object with which to compare.

Returns
   true when yes and false otherwise.

getActions

Signature
   public java.lang.String
   getActions()

Description
   Obtain the actions as a String in canonical form.

Returns
   the actions represented as a string.

hashCode

Signature
   public int
   hashCode()

Description
   Obtain the hash code value for this object.

Returns
   the hash code value.
implies(Permission)

Signature
public boolean
implies(Permission permission)

Description
Checks if the given permission’s actions are 'implied by' this object’s actions. This method is used by the AccessController to determine whether or not a requested permission is implied by another permission that is known to be valid in the current execution context.

Parameters
permission—is the permission to check.

Returns
true when yes and false otherwise.

15.6 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 corresponds 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 support a given function, such as Signal and RealtimeSignal encapsulate common functionality for POSIX compliant systems.

Finally, the GarbageCollector class provides some basic information about the garbage collector, but this information is necessarily very limited. The specification does not require a deterministic garbage collector, and even with such a collector, the variation between collectors is quite large. For example, work-based collectors do not have garbage collector threads, so many of the parameters for thread-based collectors would not make sense for a work-based collector. Data that is easy to collect with one type of collector can be quite costly to collect with another. For this reason, collector information is provided via a factory method so that the return class can be extended to provide additional, implementation-defined information.
Chapter 16

Exceptions

As with other Java specifications, the RTSJ uses throwables, exceptions and errors, to signal conditions that are abnormal, incorrect, or disallowed. In cases where these exceptional and error conditions are thrown as a result of executing or emulating a Java bytecode or performing operations compatible with standard I/O or class instantiation, the conventional Java exceptions and errors are used. They are taken primarily from the java.lang package, but also a few from the java.lang.reflect and java.io packages as well. In cases where a conventional Java exception is defined that would fit the exceptional condition, a static subclass is used. In other cases, new static exceptions are defined in the javax.realtime package. These static exception classes provide

- additional exception classes required for other sections of this specification,
- the ability to throw exceptions without allocating memory, and
- the ability to asynchronously transfer the control of program logic (see AsynchronouslyInterruptedException).

The ability to throw exceptions without memory allocation is important for using scoped and immortal memory; otherwise, throwing an exception would require much larger scopes to ensure against memory exhaustion.

16.1 Semantics

In order to balance compatibility with conventional Java against effective memory management for realtime system, this specification uses four categories of throwables. The first category consists of those that can be thrown by an RTSJ method, but are the result of executing a byte code instruction:

- java.lang.ArithmeticException,
- java.lang.ArrayIndexOutOfBoundsException,
- java.lang.ClassCastException,
- java.lang.IllegalArgumentException,
- java.lang.IndexOutOfBoundsException,
- java.lang.NullPointerException,
- java.lang.StringIndexOutOfBoundsException, and
- java.lang.OutOfMemoryError.

The second category is similar, but result from class loading or I/O:

- java.lang.InstantiationException,
- java.lang.ExceptionInInitializerError,
- java.nio.BufferOverflowException,
- java.nio.InvalidMarkException,
- java.nio.ReadOnlyBufferException, and
- java.lang.reflect.InvocationTargetException.

The third category is exceptions that have the same meaning as conventional Java exception but are thrown directly by an RTSJ method:

<table>
<thead>
<tr>
<th>RTSJ Throwable</th>
<th>extends</th>
</tr>
</thead>
<tbody>
<tr>
<td>AsynchronouslyInterruptedException</td>
<td>InterruptedException</td>
</tr>
<tr>
<td>ConstructorCheckedException</td>
<td>InstantiationException</td>
</tr>
<tr>
<td>IllegalRealtimeArgumentException</td>
<td>IllegalArgumentException</td>
</tr>
<tr>
<td>IllegalRealtimeStateException</td>
<td>IllegalStateException</td>
</tr>
<tr>
<td>IllegalTaskStateException</td>
<td>IllegalThreadStateException</td>
</tr>
<tr>
<td>RealtimeSecurityException</td>
<td>IllegalArgumentException</td>
</tr>
<tr>
<td>UnsupportedRealtimeOperationException</td>
<td>UnsupportedOperationException</td>
</tr>
<tr>
<td>OutOfMemoryStaticError</td>
<td>OutOfMemoryError</td>
</tr>
</tbody>
</table>

These classes all implement the interface StaticThrowable. The last category consists of exceptions that are defined only in the RTSJ. Each of these extend one of

- StaticCheckedException,
- StaticRuntimeException, or
- StaticError, depending on what kind of exception it is. All three of these classes also implement StaticThrowable. Except for how information associated with a Throwable is stored and managed, the semantics of the classes that implement StaticThrowable are the same as for all other Java throwables. All classes in this section are required. 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 exceptions defined in this section implement StaticThrowable and thus are statically allocated. There is at most one instance of each of these exceptions and errors, managed by the runtime. The message and stack information they would normally carry is held in a thread-local data structure. This means this information is only valid within the context of the thread that threw the StaticThrowable, and there only until a new StaticThrowable is thrown.
The thread-local storage used by instances of `StaticThrowable` is controlled by the `ConfigurationParameters` associated with the active task when the exception is thrown. This may be the system default `ConfigurationParameters` (set on `RealtimeSystem`) in the case of Java threads or a `Schedulable` for which no `ConfigurationParameters` was provided, or it may be the `ConfigurationParameters` explicitly set for a `Schedulable`.

Though the `AsynchronouslyInterruptedException` and `Timed` classes are exceptions, they provide additional functionality for supporting Asynchronous Transfer of Control (ATC). This functionality is more closely related to asynchronous operation than to exception handling. For this reason and because they are not in the core module, they are not described in this chapter, but rather in Chapter 8 on asynchrony.
16.2 javax.realtime

16.2.1 Interfaces

16.2.1.1 StaticThrowable

public interface StaticThrowable

Description

A marker interface to indicate that a Throwable is intended to be created once and reused. Throwables that implement this interface keep their state in a ThreadLocal data structure instead of the object itself. This means that data is only valid until the next StaticThrowable is thrown in the context of the current thread. Instances of AsyncBaseEventHandler always have some instance of Thread when executing. Having a marker interface makes it easier to provide checking tools to ensure the proper throw sequence for all Throwables thrown from application code.

Throwables which implement this interface should define a get() method that returns the singleton Throwable of that class. It should also fill the stack backtrace and clear the message and cause.

An application which throws a static exception should use the following paradigm:

```java
throw LateStartException.get().initMessage("..").initCause(..);
```

The message must be initialized before the cause, because initMessage is defined on StaticThrowable but not Throwable. Setting the message and the cause are both optional.

Applications which define a static throwable by implementing StaticThrowable should extend one of predefined classes: StaticError, StaticCheckedException, or StaticRuntimeException. When this is not possible because one needs to extend an existing conventional Java exception, the new throwable must override its methods and redirect them to the local instance of StaticThrowableStorage. For example, the following code snippet reimplements initMessage using StaticThrowableStorage:

```java
public Throwable initMessage(String message)
{
    StaticThrowableStorage.getCurrent().initMessage(message);
    return this;
}
```
Stack trace, message, and cause must be stored in and retrieved from this thread’s local storage structure.

See Section ConfigurationParameters

Available since RTSJ 2.0

16.2.1.1.1 Methods

initMessage(String)

Signature
public java.lang.Throwable
initMessage(String message)

Description
Store the message of the calling exception in the instance of StaticThrowableStorage associated with this task. This is the only method not defined in java.lang.Throwable.

Parameters
message—Text to be saved describing the exception’s cause.

Returns
this object.

Available since RTSJ 2.0 implemented by all static throwables

getMessage

Signature
public java.lang.String
getMessage()

Description
Gets the message describing the exception’s cause from thread local memory.

Returns
the message when this was the last method thrown and message was set for it or null.

**getLocalizedMessage**

**Signature**
```
public java.lang.String getLocalizedMessage()
```

**Description**
Subclasses may override this message to get an error message that is localized to the default locale.
By default it returns `getMessage()`.

**Returns**
the value of `getMessage()`.

**initCause(Throwables)**

**Signature**
```
public java.lang.Throwable initCause(Throwable causingThrowable)
```

**Description**
Store the cause of calling exception in the instance of `StaticThrowableStorage` associated with this task.

**Parameters**
causingThrowable—The reason why this `Throwable` gets thrown.

**Throws**
`IllegalRealtimeArgumentException`—when the cause is this `Throwable` itself.

**Returns**
the reference to this `Throwable`.

**getCause**

**Signature**
```
public java.lang.Throwable getCause()
```

858 RTSJ 2.0 (Draft 66)
**Description**

Gets the cause from thread local memory of the calling exception or null when no cause was set. The cause is another exception that was caught before the current exception is to be thrown.

**Returns**

the cause when this was the last thrown exception and cause was set or null.

---

**fillInStackTrace**

**Signature**

```java
public java.lang.Throwable fillInStackTrace()
```

**Description**

Calls into the virtual machine to capture the current stack trace in the instance of `StaticThrowableStorage` associated with this task.

**Returns**

a reference to this `Throwable`.

---

**setStackTrace(StackTraceElement)**

**Signature**

```java
public void setStackTrace(java.lang.StackTraceElement[] new_stackTrace)
```

**Description**

This method enables 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.

**Parameters**

- `new_stackTrace` — the stack trace to be used as replace.

**Throws**

- `NullPointerException` — when `new_stackTrace` or any element of `new_stackTrace` is null.
getStackTrace

Signature

public java.lang.StackTraceElement[] getStackTrace()

Description

Gets the stack trace created by fillInStackTrace for this Throwable from the current thread local storage 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.

When memory areas of the RTSJ are used (see MemoryArea), and this Throwable was allocated in a different memory area than the current allocation context, the resulting stack trace will be allocated in either the same memory area this was allocated in or the current memory area, depending on which is the least deeply nested, thereby creating objects that are assignment compatible with both areas.

Returns

An array representing the stack trace when this was the last message thrown or StaticThrowableStorage.EMPTY_TRACE, but never null.

printStackTrace

Signature

public void printStackTrace()

Description

Prints 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.
printStackTrace(PrintStream)

Signature
public void
printStackTrace(PrintStream stream)

Description
Prints 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.

Parameters
stream—The stream to print to.

printStackTrace(PrintWriter)

Signature
public void
printStackTrace(PrintWriter s)

Description
Prints 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.

Parameters
s—The PrintWriter to write to.

16.2.2 Classes
16.2.2.1 AlignmentError

public class AlignmentError

Inheritance
java.lang.Object
16 Exceptions

ArrivalTimeQueueOverflowException

java.lang.Throwable
java.lang.Error
  StaticError
    AlignmentError

Description

The exception thrown 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.

Available since RTSJ 2.0

16.2.2.1.1 Methods

get

Signature

public static javax.realtime.AlignmentError
  get()

Description

Obtains the singleton of this static throwable. It is prepared for immediate throwing.

Returns

the single instance of this throwable.

Available since RTSJ 2.0

16.2.2.2 ArrivalTimeQueueOverflowException

public class ArrivalTimeQueueOverflowException

Inheritance

java.lang.Object
  java.lang.Throwable
ArrivalTimeQueueOverflowException

java.lang.Exception
java.lang.RuntimeException
StaticRuntimeException
EventQueueOverflowException
ArrivalTimeQueueOverflowException

Description
When an arrival time occurs and should be queued, but the queue already holds a number of times equal to the initial queue length, an instance of this class is thrown.

Available since RTSJ 1.0.1 this is unchecked

Available since RTSJ 2.0 extends EventQueueOverflowException

16.2.2.2.1 Constructors

ArrivalTimeQueueOverflowException

Signature
public
ArrivalTimeQueueOverflowException()

Description
The default constructor for ArrivalTimeQueueOverflowException, but user code should use get() instead.

16.2.2.2.2 Methods

get

Signature
public static javax.realtime.ArrivalTimeQueueOverflowException
get()
16 Exceptions

CeilingViolationException

Description

Obtains the singleton of this static throwable. It is prepared for immediate throwing.

Returns

the single instance of this Throwable.

Available since RTSJ 2.0

16.2.2.3 CeilingViolationException

public class CeilingViolationException

Inheritance

java.lang.Object
    java.lang.Throwable
        java.lang.Exception
            java.lang.RuntimeException
                java.lang.IllegalArgumentException
                    java.lang.IllegalThreadStateException
                        IllegalTaskStateException
                            CeilingViolationException

Interfaces

javax.realtime.StaticThrowable

Description

This exception is thrown when a schedulable or java.lang.Thread attempts to lock an object governed by an instance of PriorityCeilingEmulation and the thread or SO’s base priority exceeds the policy’s ceiling.

Available since RTSJ 2.0 implements StaticThrowable

16.2.2.3.1 Methods

get

Signature

864 RTSJ 2.0 (Draft 66)
public static javax.realtime.CeilingViolationException get()

Description
Obtains the singleton of this static throwable. It is prepared for immediate throwing.

Returns
the single instance of this throwable.

Available since RTSJ 2.0

getCeiling

Signature
public int
getCeiling()

Description
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.

Returns
the ceiling of the PriorityCeilingEmulation policy which caused this exception to be thrown.

getCallerPriority

Signature
public int
getCallerPriority()

Description
Gets the base priority of the SO or thread whose attempt to synchronize resulted in the throwing of this.

Returns
the synchronizing thread’s base priority.
16 Exceptions

16.2.2.4 ConstructorCheckedException

public class ConstructorCheckedException

Inheritance
java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      java.lang.ReflectiveOperationException
      java.lang.InstantiationException
      ConstructorCheckedException

Description
To throw when MemoryArea.newInstance causes the constructor of the new instance to throw a checked exception.

Available since RTSJ 2.0

16.2.2.4.1 Constructors

ConstructorCheckedException(Throwable)

Signature
public
ConstructorCheckedException(Throwable cause)

Description
A constructor that can carry the original checked exception.

Parameters
cause—The original checked exception.

16.2.2.5 DeregistrationException

public class DeregistrationException

Inheritance
EventQueueOverflowException

java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      java.lang.RuntimeException
        StaticRuntimeException
          DeregistrationException

Description
An exception to throw when trying to deregister an ActiveEvent from an ActiveEventDispatcher to which it is not registered.

Available since RTSJ 2.0

16.2.2.5.1 Methods

get

Signature
  public static javax.realtime.DeregistrationException
    get()

Description
  Obtains the singleton of this static throwable. It is prepared for immediate throwing.

Returns
  the single instance of this throwable.

16.2.2.6 EventQueueOverflowException

public class EventQueueOverflowException

Inheritance
  java.lang.Object
    java.lang.Throwable
      java.lang.Exception
        java.lang.RuntimeException
          StaticRuntimeException
            EventQueueOverflowException
16 Exceptions

**forEachTerminationException**

*Description*
When an arrival time occurs and should be queued, but the queue already holds a number of times equal to the initial queue length, an instance of this class is thrown.

*Available since* RTSJ 2.0 Generalizes `ArrivalTimeQueueOverflowException`

### 16.2.2.6.1 Methods

**get**

*Signature*
```java
class java.realtime.EventQueueOverflowException

generic get()
```

*Description*
Obtains the singleton of this static throwable. It is prepared for immediate throwing.

*Returns*
the single instance of this throwable.

### 16.2.2.7 forEachTerminationException

*public class* `forEachTerminationException`

*Inheritance*
```java
java.lang.Object
    java.lang.Throwable
        java.lang.Exception
            java.lang.RuntimeException
                StaticRuntimeException
                    `forEachTerminationException`
```

*Description*
An exception to throw when a visitor should terminate early. It is for use with the `forEach` method in collection classes. Since it is a StaticThrowable, it can be used without creating garbage.
16.2.2.7.1 Methods

get

Signature
public static javax.realtime.ForEachTerminationException get()

Description
Gets the static instance of this class and initializes its stack trace.

Returns
the static singleton of this class.

16.2.2.8 IllegalAssignmentError

public class IllegalAssignmentError

Inheritance
java.lang.Object
java.lang.Throwable
java.lang.Error
StaticError
IllegalAssignmentError

Description
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 to be an instance of javax.realtime.memory.ScopedMemory, to a field of an object in immortal memory.

Available since RTSJ 2.0 extends StaticError

16.2.2.8.1 Constructors
IllegalAssignmentError

Signature

\[
\begin{align*}
\text{public} \\
\text{IllegalAssignmentError}()
\end{align*}
\]

Description

A constructor for IllegalAssignmentError, but the application should use \text{get()} instead.

16.2.2.8.2 Methods

get

Signature

\[
\begin{align*}
\text{public static java.realtime.IllegalAssignmentError} \\
\text{get()}
\end{align*}
\]

Description

Obtains the singleton of this static throwable. It is prepared for immediate throwing.

Returns

the single instance of this throwable.

Available since RTSJ 2.0

16.2.2.9 IllegalRealtimeArgumentException

public class IllegalRealtimeArgumentException

Inheritance

java.lang.Object
java.lang.Throwable
java.lang.Exception
java.lang.RuntimeException

870 RTSJ 2.0 (Draft 66)
IllegalRealtimeArgumentException

java.lang.IllegalArgumentException

IllegalRealtimeArgumentException

Interfaces
javax.realtime.StaticThrowable

Description
A version of IllegalArgumentException to be thrown by RTSJ methods that does not require allocation.

Available since RTSJ 2.0

16.2.2.9.1 Methods

get

Signature
public static javax.realtime.IllegalRealtimeArgumentException
get()

Description
Gets the preallocated version of this Throwable. Allocation is done in memory that acts like ImmortalMemory. The message and cause are cleared and the stack trace is filled out.

Returns
the preallocated exception.

initMessage(String)

Signature
public java.lang.Throwable
initMessage(String message)

Description
Store the message of the calling exception in the instance of StaticThrowableStorage associated with this task. This is the only method not defined in java.lang.Throwable.
16 Exceptions

IllegalRealtimeArgumentException

Parameters
message—Text to be saved describing the exception’s cause.

Returns
this object.

Available since RTSJ 2.0 implemented by all static throwables

getsMessage

Signature
public java.lang.String
getMessage()

Description
Gets the message describing the exception’s cause from thread local memory.

Returns
the message when this was the last method thrown and message was set for it or null.

getsLocalizedMessage

Signature
public java.lang.String
getLocalizedMessage()

Description
Subclasses may override this message to get an error message that is localized to the default locale.
By default it returns getMessage().

Returns
the value of getMessage().

initCause(Throwable)

Signature
public java.lang.Throwable
initCause(Throwable causingThrowable)
IllegalRealtimeArgumentException
javax.realtime 16.2

Description
Store the cause of calling exception in the instance of StaticThrowableStorage associated with this task.

Parameters
causinThrowable—The reason why this Throwable gets thrown.

Throws
IllegalRealtimeArgumentException—when the cause is this Throwable itself.

Returns
the reference to this Throwable.

getcasue

Signature
public java.lang.Throwable getcas() 

Description
Gets the cause from thread local memory of the calling exception or null when no cause was set. The cause is another exception that was caught before the current exception is to be thrown.

Returns
the cause when this was the last thrown exception and cause was set or null.

fillInStackTrace

Signature
public java.lang.Throwable fillInStackTrace() 

Description
Calls into the virtual machine to capture the current stack trace in the instance of StaticThrowableStorage associated with this task.

Returns
a reference to this Throwable.
setStackTrace(StackTraceElement)

Signature
public void
setStackTrace(java.lang.StackTraceElement[] trace)
throws NullPointerException

Description
This method enables 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.

Parameters
new_stackTrace—the stack trace to be used as replace.

Throws
NullPointerException—when new_stackTrace or any element of new_stackTrace
is null.

getStackTrace

Signature
public java.lang.StackTraceElement[]
getStackTrace()

Description
Gets the stack trace created by fillInStackTrace for this Throwable from the
current thread local storage 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.

When memory areas of the RTSJ are used (see MemoryArea), and this Throwable
was allocated in a different memory area than the current allocation context,
the resulting stack trace will be allocated in either the same memory area this
was allocated in or the current memory area, depending on which is the least
deeply nested, thereby creating objects that are assignment compatible with both
areas.

Returns
An array representing the stack trace when this was the last message thrown or StaticThrowableStorage.EMPTY_TRACE, but never null.

**printStackTrace**

*Signature*

```java
public void printStackTrace()
```

*Description*

Prints 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.

**printStackTrace(PrintStream)**

*Signature*

```java
public void printStackTrace(PrintStream stream)
```

*Description*

Prints 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.

*Parameters*

- `stream`—The stream to print to.

**printStackTrace(PrintWriter)**

*Signature*

```java
public void printStackTrace(PrintWriter writer)
```

*Description*
Prints 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.

Parameters

s—The PrintWriter to write to.

16.2.2.10 IllegalRealtimeStateException

public class IllegalRealtimeStateException

Inheritance

java.lang.Object
java.lang.Throwable
java.lang.Exception
java.lang.RuntimeException
java.lang.IllegalStateException
IllegalRealtimeStateException

Interfaces

javax.realtime.StaticThrowable

Description

The exception thrown when a Schedulable instance attempts to access a memory which is illegal in the memories current state. For instance, some memory areas have limits from what other area they may be entered into.

Available since RTSJ 2.0

16.2.2.10.1 Methods

get

Signature

public static javax.realtime.IllegalRealtimeStateException get()
IllegalRealtimeStateException

Description

Gets the preallocated version of this Throwable. Allocation is done in memory that acts like ImmortalMemory. The message and cause are cleared and the stack trace is filled out.

Returns

the preallocated exception.

initMessage(String)

Signature

public java.lang.Throwable

initMessage(String message)

Description

Store the message of the calling exception in the instance of StaticThrow-ableStorage associated with this task. This is the only method not defined in java.lang.Throwable.

Parameters

message—Text to be saved describing the exception’s cause.

Returns

this object.

Available since RTSJ 2.0 implemented by all static throwables

getMessage

Signature

public java.lang.String

getMessage()

Description

Gets the message describing the exception’s cause from thread local memory.

Returns

the message when this was the last method thrown and message was set for it or null.
16 Exceptions

IllegalRealtimeStateException

getLocalizedMessage

Signature
public java.lang.String
getLocalizedMessage()

Description
Subclasses may override this message to get an error message that is localized to
the default locale.
By default it returns getMessage().

Returns
the value of getMessage().

initCause(Throwable)

Signature
public java.lang.Throwable
initCause(Throwable causingThrowable)

Description
Store the cause of calling exception in the instance of StaticThrowableStorage
associated with this task.

Parameters
causingThrowable—The reason why this Throwable gets thrown.

Throws
IllegalRealtimeArgumentException—when the cause is this Throwable itself.

Returns
the reference to this Throwable.

getcause

Signature
public java.lang.Throwable
getcause()

Description
878 RTSJ 2.0 (Draft 66)
Gets the cause from thread local memory of the calling exception or null when no cause was set. The cause is another exception that was caught before the current exception is to be thrown.

Returns
the cause when this was the last thrown exception and cause was set or null.

fillInStackTrace

Signature
public java.lang.Throwable fillInStackTrace()

Description
Calls into the virtual machine to capture the current stack trace in the instance of StaticThrowableStorage associated with this task.

Returns
a reference to this Throwable.

setStackTrace(StackTraceElement)

Signature
public void setStackTrace(java.lang.StackTraceElement[] new_stackTrace)
throws NullPointerException

Description
This method enables 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.

Parameters
new_stackTrace—the stack trace to be used as replace.

Throws
NullPointerException—when new_stackTrace or any element of new_stackTrace is null.
getStackTrace

**Signature**

```java
public java.lang.StackTraceElement[]
getStackTrace()
```

**Description**

Gets the stack trace created by fillInStackTrace for this Throwable from the current thread local storage 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.

When memory areas of the RTSJ are used (see MemoryArea), and this Throwable was allocated in a different memory area than the current allocation context, the resulting stack trace will be allocated in either the same memory area this was allocated in or the current memory area, depending on which is the least deeply nested, thereby creating objects that are assignment compatible with both areas.

**Returns**

An array representing the stack trace when this was the last message thrown or StaticThrowableStorage.EMPTY_TRACE, but never null.

printStackTrace

**Signature**

```java
public void
printStackTrace()
```

**Description**

Prints 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.
IllegalTaskStateException

**printStackTrace(PrintStream)**

*Signature*

```java
public void printStackTrace(PrintStream stream)
```

*Description*

Prints 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.

*Parameters*

- `stream`—The stream to print to.

**printStackTrace(PrintWriter)**

*Signature*

```java
public void printStackTrace(PrintWriter writer)
```

*Description*

Prints 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.

*Parameters*

- `s`—The PrintWriter to write to.

### 16.2.2.11 IllegalTaskStateException

public class IllegalTaskStateException

*Inheritance*

- java.lang.Object
- java.lang.Throwable
- java.lang.Exception
16 Exceptions

IllegalTaskStateException

java.lang.RuntimeException
java.lang.IllegalArgumentException
java.lang.IllegalThreadStateException
IllegalTaskStateException

Interfaces
javax.realtime.StaticThrowable

Description
The exception thrown when a Schedulable instance attempts an operation which is illegal in its current state. For instance, changing parameters on such instances are only allowed when the scheduler is not active or the new parameters are consistent with the current scheduler.

Available since RTSJ 2.0

16.2.2.11.1 Methods

get

Signature
public static javax.realtime.IllegalTaskStateException get()

Description
Gets the preallocated version of this Throwable. Allocation is done in memory that acts like ImmortalMemory. The message and cause are cleared and the stack trace is filled out.

Returns
the preallocated exception.

initMessage(String)

Signature
public java.lang.Throwable initMessage(String message)

Description

882 RTSJ 2.0 (Draft 66)
Store the message of the calling exception in the instance of StaticThrowableStorage associated with this task. This is the only method not defined in java.lang.Throwable.

Parameters
message—Text to be saved describing the exception’s cause.

Returns
this object.

Available since RTSJ 2.0 implemented by all static throwables

ggetMessage

Signature
public java.lang.String getMessage()

Description
Gets the message describing the exception’s cause from thread local memory.

Returns
the message when this was the last method thrown and message was set for it or null.

getLocalizedMessage

Signature
public java.lang.String getLocalizedMessage()

Description
Subclasses may override this message to get an error message that is localized to the default locale.
By default it returns getMessage().

Returns
the value of getMessage().
16 Exceptions

initCause(Throwables)

Signature

```java
public java.lang.Throwable
    initCause(Throwables causingThrowable)
```

Description

Store the cause of calling exception in the instance of StaticThrowableStorage associated with this task.

Parameters

- `causingThrowable`—The reason why this Throwable gets thrown.

Throws

- IllegalRealtimeArgumentException—when the cause is this Throwable itself.

Returns

the reference to this Throwable.

getcause

Signature

```java
public java.lang.Throwable
    getcause()
```

Description

Gets the cause from thread local memory of the calling exception or null when no cause was set. The cause is another exception that was caught before the current exception is to be thrown.

Returns

the cause when this was the last thrown exception and cause was set or null.

fillInStackTrace

Signature

```java
public java.lang.Throwable
    fillInStackTrace()
```

Description

Calls into the virtual machine to capture the current stack trace in the instance of StaticThrowableStorage associated with this task.
IllegalTaskStateException

Returns

a reference to this Throwable.

setStackTrace(StackTraceElement)

Signature

class java.lang.StackTraceElement[] setStackTrace(java.lang.StackTraceElement[] new_stackTrace)

throws NullPointerException

Description

This method enables 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.

Parameters

new_stackTrace—the stack trace to be used as replace.

Throws

NullPointerException—when new_stackTrace or any element of new_stackTrace is null.

getStackTrace

Signature

public java.lang.StackTraceElement[] getStackTrace()

Description

Gets the stack trace created by fillInStackTrace for this Throwable from the current thread local storage 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.

When memory areas of the RTSJ are used (see MemoryArea), and this Throwable was allocated in a different memory area than the current allocation context, the resulting stack trace will be allocated in either the same memory area this was allocated in or the current memory area, depending on which is the least
deeply nested, thereby creating objects that are assignment compatible with both areas.

**Returns**
An array representing the stack trace when this was the last message thrown or `StaticThrowableStorage.EMPTY_TRACE`, but never `null`.

**printStackTrace**

**Signature**
```java
public void printStackTrace()
```

**Description**
Prints 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.

**printStackTrace(PrintStream)**

**Signature**
```java
public void printStackTrace(PrintStream stream)
```

**Description**
Prints 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.

**Parameters**
- `stream`—The stream to print to.

**printStackTrace(PrintWriter)**

**Signature**
```java
886
```

RTSJ 2.0 (Draft 66)
InaccessibleAreaException

```java
public void printStackTrace(PrintWriter writer)
```

**Description**

Prints the stack trace of this Throwable to the given PrintWriter.

The printed stack trace contains the result of \texttt{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.

**Parameters**

- \texttt{s}—The PrintWriter to write to.

### 16.2.2.12 InaccessibleAreaException

```java
public class InaccessibleAreaException
```

**Inheritance**

```
java.lang.Object
default java.lang.Throwable
java.lang.Exception
java.lang.RuntimeException
StaticRuntimeException
InaccessibleAreaException
```

**Description**

The specified memory area is not on the current thread’s scope stack.

**Available since** RTSJ 1.0.1 Becomes unchecked

**Available since** RTSJ 2.0 extends StaticRuntimeException

### 16.2.2.12.1 Constructors

**InaccessibleAreaException**

**Signature**

RTSJ 2.0 (Draft 66) 887
public
InaccessibleAreaException()

Description
A constructor for InaccessibleAreaException, but application code should use get() instead.

InaccessibleAreaException(String)

Signature
public
InaccessibleAreaException(String description)

Description
A descriptive constructor for InaccessibleAreaException.

Deprecated since RTSJ 2.0; application code should use get() instead.

Parameters
description—Description of the error.

16.2.2.12.2 Methods

get

Signature
public static javax.realtime.InaccessibleAreaException
get()

Description
Obtains the singleton of this static throwable. It is prepared for immediate throwing.

Returns
the single instance of this throwable.

Available since RTSJ 2.0

888
RTSJ 2.0 (Draft 66)
16.2.2.13 LateStartException

public class LateStartException

Inheritance
java.lang.Object
   java.lang.Throwable
      java.lang.Exception
         StaticCheckedException
            LateStartException

Description
Exception thrown when a periodic realtime thread or timer is started after its assigned, absolute, start time.

Available since RTSJ 2.0

16.2.2.13.1 Methods

get

Signature
   public static javax.realtime.LateStartException
      get()

Description
Obtains the singleton of this static throwable. It is prepared for immediate throwing.

Returns
the single instance of this throwable.

16.2.2.14 MITViolationException

public class MITViolationException

Inheritance
16 Exceptions

MITViolationException

java.lang.Object
java.lang.Throwable
java.lang.Exception
java.lang.RuntimeException
StaticRuntimeException
MITViolationException

Description
Thrown by the AsyncEvent.fire() 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 became unchecked

Available since RTSJ 2.0 extends StaticRuntime Exception

16.2.2.14.1 Constructors

MITViolationException

Signature
public
MITViolationException()

Description
A constructor for MITViolationException.

MITViolationException(String)

Signature
public
MITViolationException(String description)

Description
A descriptive constructor for MITViolationException.

890 RTSJ 2.0 (Draft 66)
MemoryAccessError

Parameters

description—Description of the error.

16.2.2.14.2 Methods

get

Signature

public static javax.realtime.MITViolationException
get()

Description

Obtains the singleton of this static throwable. It is prepared for immediate throwing.

Returns

the single instance of this throwable.

Available since RTSJ 2.0

16.2.2.15 MemoryAccessError

public class MemoryAccessError

Inheritance

java.lang.Object
java.lang.Throwable
java.lang.Error
StaticError
MemoryAccessError

Description

This error is thrown on an attempt to refer to an object in an inaccessible MemoryArea. For example, when logic in a RealtimeThread or AsyncBaseEventHandler configured with a javax.realtime.memory.ScopedConfigurationParameters object, attempts to refer to an object in a HeapMemory area.

Available since RTSJ 2.0 extends StaticError
16 Exceptions  

16.2.2.15.1 Constructors  

MemoryAccessError  

*Signature*  

```java  
public MemoryAccessError()  
```

*Description*  

A constructor for `MemoryAccessError`, but application code should use `get()` instead.

16.2.2.15.2 Methods  

get  

*Signature*  

```java  
public static javax.realtime.MemoryAccessError get()  
```

*Description*  

Obtains the singleton of this static `Throwable`. It is prepared for immediate throwing.

*Returns*  

the single instance of this `Throwable`.  

**Available since** RTSJ 2.0

16.2.2.16 MemoryInUseException  

```java  
public class MemoryInUseException  
```

*Inheritance*  

892  

RTSJ 2.0 (Draft 66)
MemoryInUseException

java.lang.Object
    java.lang.Throwable
        java.lang.Exception
            java.lang.RuntimeException
                StaticRuntimeException
                    MemoryInUseException

Description
There has been attempt to allocate a range of physical or virtual memory that is already in use.

Available since RTSJ 2.0 extends StaticRuntimeException

16.2.2.16.1 Constructors

MemoryInUseException

Signature
    public
        MemoryInUseException()

Description
A constructor for MemoryInUseException, but application code should use get() instead.

MemoryInUseException(String)

Signature
    public
        MemoryInUseException(String description)

Description
A descriptive constructor for MemoryInUseException.

Deprecated since RTSJ 2.0; application code should use get() instead.

Parameters
16 Exceptions

MemoryScopeException

description—Description of the error.

16.2.2.16.2 Methods

get

Signature
public static javax.realtime.MemoryInUseException
get()

Description
Obtains the singleton of this static throwable. It is prepared for immediate throwing.

Returns
the single instance of this throwable.

Available since RTSJ 2.0

16.2.2.17 MemoryScopeException

public class MemoryScopeException

Inheritance
java.lang.Object
java.lang.Throwable
java.lang.Exception
java.lang.RuntimeException
StaticRuntimeException
MemoryScopeException

Description
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.

Available since RTSJ 2.0 extends StaticRuntimeException

894 RTSJ 2.0 (Draft 66)
16.2.2.17.1 Constructors

MemoryScopeException

Signature
   public
   MemoryScopeException()

Description
   A constructor for MemoryScopeException, but application code should use get() instead.

16.2.2.17.2 Methods

get

Signature
   public static javax.realtime.MemoryScopeException
   get()

Description
   Obtains the singleton of this static throwable. It is prepared for immediate throwing.

Returns
   the single instance of this throwable.

Available since RTSJ 2.0

16.2.2.18 MemoryTypeConflictException

public class MemoryTypeConflictException

Inheritance

RTSJ 2.0 (Draft 66)
MemoryTypeConflictException

java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      java.lang.RuntimeException
        StaticRuntimeException
          MemoryTypeConflictException

Description

This exception is thrown when the PhysicalMemoryManager is given conflicting specifications for memory. The conflict can be between types in an array of memory type specifiers, or between the specifiers and a specified base address.

Available since RTSJ 1.0.1 Changed to an unchecked exception.

Available since RTSJ 2.0 extends StaticRuntimeException

16.2.2.18.1 Constructors

MemoryTypeConflictException

Signature
public
MemoryTypeConflictException()

Description
A constructor for MemoryTypeConflictException, but application code should use get() instead.

MemoryTypeConflictException(String)

Signature
public
MemoryTypeConflictException(String description)

Description
A descriptive constructor for MemoryTypeConflictException.
Deprecated since RTSJ 2.0; application code should use `get()` instead.

Parameters
- `description`—A description of the exception.

16.2.2.18.2 Methods

get

Signature
```java
public static javax.realtime.MemoryTypeConflictException get()
```

Description
Obtains the singleton of this static throwable. It is prepared for immediate throwing.

Returns
the single instance of this throwable.

Available since RTSJ 2.0

16.2.2.19 OffsetOutOfBoundsException

public class OffsetOutOfBoundsException

Inheritance
- java.lang.Object
- java.lang.Throwable
- java.lang.Exception
- java.lang.RuntimeException
  - StaticRuntimeException
  - OffsetOutOfBoundsException

Description
When the constructor of an `ImmortalPhysicalMemory`, `LTPhysicalMemory`, `VT-PhysicalMemory`, `RawMemoryAccess`, or `RawMemoryFloatAccess` is given an invalid address.
Available since RTSJ 1.0.1 became unchecked

Available since RTSJ 2.0 extends StaticRuntimeException

16.2.2.19.1 Constructors

**OffsetOutOfBoundsException**

*Signature*

```java
public
OffsetOutOfBoundsException()
```

*Description*

A constructor for `OffsetOutOfBoundsException`, application code should use `get()` instead.

16.2.2.19.2 Methods

**get**

*Signature*

```java
public static javax.realtime.OffsetOutOfBoundsException
get()
```

*Description*

Obtains the singleton of this static throwable. It is prepared for immediate throwing.

*Returns*

the single instance of this throwable.

Available since RTSJ 2.0
public class OutOfMemoryStaticError

Inheritance
java.lang.Object
  java.lang.Throwable
    java.lang.Error
      java.lang.VirtualMachineError
        java.lang.OutOfMemoryError
          OutOfMemoryStaticError

Interfaces
javax.realtime.StaticThrowable

Description
A version of OutOfMemoryError that does not require allocation. It should be thrown from all RTSJ memory subclasses except HeapMemory. It is up to the implementation as to whether HeapMemory throws this exception or its parent.

Available since RTSJ 2.0

16.2.2.20.1 Methods

get

Signature
public static javax.realtime.OutOfMemoryStaticError get()

Description
Gets the preallocated version of this Throwable. Allocation is done in memory that acts like ImmortalMemory. The message and cause are cleared and the stack trace is filled out.

Returns
the preallocated exception.
initMessage(String)

Signature
public java.lang.Throwable
initMessage(String message)

Description
Store the message of the calling exception in the instance of StaticThrowableStorage associated with this task. This is the only method not defined in java.lang.Throwable.

Parameters
message—Text to be saved describing the exception’s cause.

Returns
this object.

Available since RTSJ 2.0 implemented by all static throwables

getMessage

Signature
public java.lang.String
getMessage()

Description
Gets the message describing the exception’s cause from thread local memory.

Returns
the message when this was the last method thrown and message was set for it or null.

getLocalizedMessage

Signature
public java.lang.String
getLocalizedMessage()

Description

900 RTSJ 2.0 (Draft 66)
OutOfMemoryStaticError

Subclasses may override this message to get an error message that is localized to the default locale.

By default it returns `getMessage()`.

**Returns**

the value of `getMessage()`.

**initCause(Throwables)**

**Signature**

```java
public java.lang.Throwable initCause(Throwables cause)
```

**Description**

Store the cause of calling exception in the instance of `StaticThrowableStorage` associated with this task.

**Parameters**

`causingThrowable`—The reason why this `Throwables` gets thrown.

**Throws**

`IllegalRealtimeArgumentException`—when the cause is this `Throwables` itself.

**Returns**

the reference to this `Throwables`.

**getCause**

**Signature**

```java
public java.lang.Throwable getCause()
```

**Description**

Gets the cause from thread local memory of the calling exception or `null` when no cause was set. The cause is another exception that was caught before the current exception is to be thrown.

**Returns**

the cause when this was the last thrown exception and cause was set or `null`. 
fillInStackTrace

**Signature**

```java
public java.lang.Throwable
fillInStackTrace()
```

**Description**

Calls into the virtual machine to capture the current stack trace in the instance of `StaticThrowableStorage` associated with this task.

**Returns**

a reference to this Throwable.

**setStackTrace(StackTraceElement)**

**Signature**

```java
public void
setStackTrace(java.lang.StackTraceElement[] trace)
throws NullPointerException
```

**Description**

This method enables 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.

**Parameters**

- `new_stackTrace`—the stack trace to be used as replace.

**Throws**

- `NullPointerException`—when `new_stackTrace` or any element of `new_stackTrace` is null.

**getStackTrace**

**Signature**

```java
public java.lang.StackTraceElement[]
getStackTrace()
```

**Description**

902  RTSJ 2.0 (Draft 66)
 Gets the stack trace created by fillInStackTrace for this Throwable from the current thread local storage 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.  

When memory areas of the RTSJ are used (see MemoryArea), and this Throwable was allocated in a different memory area than the current allocation context, the resulting stack trace will be allocated in either the same memory area this was allocated in or the current memory area, depending on which is the least deeply nested, thereby creating objects that are assignment compatible with both areas.

**Returns**
An array representing the stack trace when this was the last message thrown or StaticThrowableStorage.EMPTY_TRACE, but never null.

### printStackTrace

**Signature**

```java
public void printStackTrace()
```

**Description**

Prints 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.

### printStackTrace(PrintStream)

**Signature**

```java
public void printStackTrace(PrintStream stream)
```

**Description**

RTSJ 2.0 (Draft 66)
Prints 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.

Parameters

stream—The stream to print to.

printStackTrace(PrintWriter)

Signature

public void
printStackTrace(PrintWriter s)

Description

Prints 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.

Parameters

s—The PrintWriter to write to.

16.2.2.21 POSIXException

public abstract class POSIXException

Inheritance

java.lang.Object
java.lang.Throwable
java.lang.Exception
StaticCheckedException
POSIXException

Description

A base class for all POSIX exceptions.

Available since RTSJ 2.0
16.2.2.22  POSIXInvalidSignalException

public class POSIXInvalidSignalException

Inheritance
java.lang.Object
    java.lang.Throwable
        java.lang.Exception
            StaticCheckedException
                POSIXException
                    POSIXInvalidSignalException

Description
An invalid POSIX signal number has been specified.

Available since RTSJ 2.0

16.2.2.22.1  Methods

get

Signature
    public static javax.realtime.POSIXInvalidSignalException
        get()

Description
Obtains the singleton of this static throwable. It is prepared for immediate throwing.

Returns
the single instance of this throwable.

16.2.2.23  POSIXInvalidTargetException

public class POSIXInvalidTargetException

Inheritance

RTSJ 2.0 (Draft 66) 905
16 Exceptions

POSIXSignalPermissionException

java.lang.Object
java.lang.Throwable
java.lang.Exception
    StaticCheckedException
    POSIXException
    POSIXInvalidTargetException

Description
The target of the signal does not exist.

Available since RTSJ 2.0

16.2.2.23.1 Methods

get

Signature
public static javax.realtime.POSIXInvalidTargetException
    get()

Description
Obtains the singleton of this static throwable. It is prepared for immediate throwing.

Returns
the single instance of this throwable.

16.2.2.24 POSIXSignalPermissionException

public class POSIXSignalPermissionException

Inheritance
java.lang.Object
java.lang.Throwable
java.lang.Exception
    StaticCheckedException
    POSIXException
    POSIXSignalPermissionException
ProcessorAffinityException

Description

The process does not have permission to send the given signal to the given target.

Available since RTSJ 2.0

16.2.2.24.1 Methods

get

Signature

public static javax.realtime.POSIXSignalPermissionException get()

Description

Obtains the singleton of this static throwable. It is prepared for immediate throwing.

Returns

the single instance of this throwable.

16.2.2.25 ProcessorAffinityException

public class ProcessorAffinityException

Inheritance

java.lang.Object
	java.lang.Throwable
	java.lang.Exception
		StaticCheckedException
		ProcessorAffinityException

Description

Exception used to report processor affinity-related errors.

Available since RTSJ 2.0
16 Exceptions

16.2.2.25.1 Methods

get

Signature

public static javax.realtime.ProcessorAffinityException get()

Description

Obtains the singleton of this static throwable. It is prepared for immediate throwing.

Returns

the single instance of this throwable.

16.2.2.26 RangeOutOfBoundsException

public class RangeOutOfBoundsException

Inheritance

java.lang.Object
java.lang.Throwable
java.lang.Exception
StaticCheckedException
RangeOutOfBoundsException

Description

Thrown when the memory region overlaps with another region in use or memory that may not be used.

Available since RTSJ 2.0

16.2.2.26.1 Methods

908 RTSJ 2.0 (Draft 66)
RealtimeSecurityException

get

Signature

```java
public static javax.realtime.RangeOutOfBoundsException get()
```

Description

Obtains the singleton of this static throwable. It is prepared for immediate
throwing.

Returns

the single instance of this throwable.

16.2.2.27 RealtimeSecurityException

public class RealtimeSecurityException

Inheritance

java.lang.Object
  java.lang.Throwable
  java.lang.Exception
    java.lang.RuntimeException
      java.lang.IllegalArgumentException
        RealtimeSecurityException

Interfaces

javax.realtime StaticThrowable

Description

A version of SecurityException to be thrown by RTSJ methods that does not
require allocation.

Available since RTSJ 2.0

16.2.2.27.1 Methods

RTSJ 2.0 (Draft 66)
get

Signature

    public static javax.realtime.RealtimeSecurityException
    get()

Description

    Gets the preallocated version of this Throwable. Allocation is done in memory that acts like ImmortalMemory. The message and cause are cleared and the stack trace is filled out.

Returns

    the preallocated exception.

initMessage(String)

Signature

    public java.lang.Throwable
    initMessage(String message)

Description

    Store the message of the calling exception in the instance of StaticThrowableStorage associated with this task. This is the only method not defined in java.lang.Throwable.

Parameters

    message—Text to be saved describing the exception’s cause.

Returns

    this object.

Available since RTSJ 2.0 implemented by all static throwables

getMessage

Signature

    public java.lang.String
    getMessage()

Description

    Gets the message describing the exception’s cause from thread local memory.
Returns
the message when this was the last method thrown and message was set for it or null.

getLocalizedMessage

Signature
public java.lang.String getLocalizedMessage()

Description
Subclasses may override this message to get an error message that is localized to the default locale.
By default it returns getMessage().

Returns
the value of getMessage().

initCause( Throwable)

Signature
public java.lang.Throwable initCause( Throwable causingThrowable)

Description
Store the cause of calling exception in the instance of StaticThrowableStorage associated with this task.

Parameters
causingThrowable—The reason why this Throwable gets thrown.

Throws
IllegalRealtimeArgumentException—when the cause is this Throwable itself.

Returns
the reference to this Throwable.

getcause

Signature
16 Exceptions

RealtimeSecurityException

public java.lang Throwable
getCause()

Description

Gets the cause from thread local memory of the calling exception or null when no cause was set. The cause is another exception that was caught before the current exception is to be thrown.

Returns

the cause when this was the last thrown exception and cause was set or null.

fillInStackTrace

Signature

public java.lang Throwable
fillInStackTrace()

Description

Calls into the virtual machine to capture the current stack trace in the instance of StaticThrowableStorage associated with this task.

Returns

a reference to this Throwable.

setStackTrace(StackTraceElement)

Signature

public void
setStackTrace(java.lang.StackTraceElement[] trace)
throws NullPointerException

Description

This method enables 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.

Parameters

new_stackTrace—the stack trace to be used as replace.

Throws

NullPointerException—when new_stackTrace or any element of new_stackTrace is null.

912

RTSJ 2.0 (Draft 66)
RealtimeSecurityException

getStackTrace

Signature

public java.lang.StackTraceElement[]
getStackTrace()

Description

Gets the stack trace created by fillInStackTrace for this Throwable from the current thread local storage 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.

When memory areas of the RTSJ are used (see MemoryArea), and this Throwable was allocated in a different memory area than the current allocation context, the resulting stack trace will be allocated in either the same memory area this was allocated in or the current memory area, depending on which is the least deeply nested, thereby creating objects that are assignment compatible with both areas.

Returns

An array representing the stack trace when this was the last message thrown or StaticThrowableStorage.EMPTY_TRACE, but never null.

printStackTrace

Signature

public void
printStackTrace()

Description

Prints 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.
### printStackTrace(PrintStream)

**Signature**

```java
class RegistrationException {
    public void printStackTrace(PrintStream stream)
}
```

**Description**

Prints 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.

**Parameters**

- `stream`—The stream to print to.

### printStackTrace(PrintWriter)

**Signature**

```java
class RegistrationException {
    public void printStackTrace(PrintWriter writer)
}
```

**Description**

Prints 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.

**Parameters**

- `writer`—The `PrintWriter` to write to.

### 16.2.2.28 RegistrationException

```java
class RegistrationException {
    // Constructors, methods, etc.
}
```

**Inheritance**

```java
RegistrationException extends Exception extends Throwable extends Object
```
Description
An exception to throw when trying to register an ActiveEvent with an ActiveEventDispatcher to which it is already registered.

Available since RTSJ 2.0

16.2.2.28.1 Methods

get

Signature
public static javax.realtime.RegistrationException get()

Description
Obtains the singleton of this static throwable. It is prepared for immediate throwing.

Returns
the single instance of this throwable.

16.2.2.29 ResourceLimitError

public class ResourceLimitError

Inheritance
g-java.lang.Object
  java.lang.Throwable
    java.lang.Error
      StaticError
        ResourceLimitError

Description
This error is thrown when an attempt is made to exceed a system resource limit, such as the maximum number of locks.
Available since RTSJ 2.0 extends StaticError

16.2.2.29.1 Constructors

ResourceLimitError

Signature
   public
   ResourceLimitError()

Description
   A constructor for ResourceLimitError, but application code should use \texttt{get()} instead.

ResourceLimitError(String)

Signature
   public
   ResourceLimitError(String description)

Description
   A descriptive constructor for ResourceLimitError.

Deprecated since RTSJ 2.0; application code should use \texttt{get()} instead.

Parameters
   description—The reason for throwing this error.
get

Signature

```java
public static javax.realtime.ResourceLimitError get()
```

Description

Obtains the singleton of this static `Throwable`. It is prepared for immediate throwing.

Returns

the single instance of this `Throwable`.

Available since RTSJ 2.0

16.2.2.30   ScopedCycleException

```java
public class ScopedCycleException
```

Inheritance

`java.lang.Object`
`java.lang.Throwable`
`java.lang.Exception`
`java.lang.RuntimeException`
`StaticRuntimeException`
`ScopedCycleException`

Description

Thrown when a schedulable attempts to enter an instance of `javax.realtime.memory.ScopedMemory` where that operation would cause a violation of the single parent rule.

Available since RTSJ 2.0 extends StaticRuntimeException

16.2.2.30.1   Constructors
16 Exceptions

ScopedCycleException

Signature

public
ScopedCycleException()

Description

A constructor for ScopedCycleException, but application code should use get() instead.

ScopedCycleException(String)

Signature

public
ScopedCycleException(String description)

Description

A descriptive constructor for ScopedCycleException.

Deprecated since RTSJ 2.0; application code should use get() instead.

Parameters

description—Description of the error.

16.2.2.30.2 Methods

get

Signature

public static javax.realtime.ScopedCycleException
get()

Description

Obtains the singleton of this static throwable. It is prepared for immediate throwing.
Returns
the single instance of this throwable.

Available since RTSJ 2.0

16.2.2.31 SizeOutOfBoundsException

public class SizeOutOfBoundsException

Inheritance
java.lang.Object
    java.lang.Throwable
        java.lang.Exception
            java.lang.RuntimeException
                StaticRuntimeException
                    SizeOutOfBoundsException

Description
To throw when the constructor of an ImmortalPhysicalMemory, LTPhysicalMemory, or VTPhysicalMemory is given an invalid size or when a memory access generated by a raw memory accessor instance (See javax.realtime.device.RawMemory.) would cause access to an invalid address.

Available since RTSJ 1.0.1 became unchecked

Available since RTSJ 2.0 extends StaticRuntimeException

16.2.2.31.1 Constructors

SizeOutOfBoundsException

Signature
    public
    SizeOutOfBoundsException()

Description
    RTSJ 2.0 (Draft 66)
A constructor for `SizeOutOfBoundsException`, but application code should use `get()` instead.

**SizeOutOfBoundsException(String)**

**Signature**

```java
public SizeOutOfBoundsException(String description)
```

**Description**

A descriptive constructor for `SizeOutOfBoundsException`.

**Deprecated** since RTSJ 2.0; application code should use `get()` instead.

**Parameters**

- `description`—The description of the exception.

### 16.2.2.31.2 Methods

**get**

**Signature**

```java
public static javax.realtime.SizeOutOfBoundsException get()
```

**Description**

Obtains the singleton of this static throwable. It is prepared for immediate throwing.

**Returns**

the single instance of this throwable.

**Available since** RTSJ 2.0
16.2.2.32 StaticCheckedException

public abstract class StaticCheckedException

Inheritance
java.lang.Object
   java.lang.Throwable
      java.lang.Exception
         StaticCheckedException

Interfaces
javax.realtime.StaticThrowable

Description
A base class for all checked exceptions defined in the specification which do not extend a conventional Java exception.

Available since RTSJ 2.0

16.2.2.32.1 Constructors

StaticCheckedException

Signature
protected
StaticCheckedException()  

Description
Enable this class to be extended.

16.2.2.32.2 Methods
**initMessage(String)**

*Signature*

```java
public java.lang.Throwable
initMessage(String message)
```

*Description*

Store the message of the calling exception in the instance of `StaticThrowableStorage` associated with this task. This is the only method not defined in `java.lang.Throwable`.

*Parameters*

*message*—Text to be saved describing the exception’s cause.

*Returns*

this object.

*Available since* RTSJ 2.0 implemented by all static throwables

**getMessage**

*Signature*

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

*Description*

Gets the message describing the exception’s cause from thread local memory.

*Returns*

the message when this was the last method thrown and message was set for it or null.

**getLocalizedMessage**

*Signature*

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

*Description*

922

RTSJ 2.0 (Draft 66)
Subclasses may override this message to get an error message that is localized to the default locale.
By default it returns `getMessage()`.

**Returns**
the value of `getMessage()`.

**initCause(Throwabe)**

**Signature**
```java
public java.lang.Throwable
initCause(Throwabe causingThrowabe)
```

**Description**
Store the cause of calling exception in the instance of `StaticThrowableStorage` associated with this task.

**Parameters**
`causingThrowabe`—The reason why this `Throwabe` gets thrown.

**Throws**
`IllegalRealtimeArgumentException`—when the cause is this `Throwabe` itself.

**Returns**
the reference to this `Throwabe`.

**getCause**

**Signature**
```java
public java.lang.Throwabe
getCause()```

**Description**
Gets the cause from thread local memory of the calling exception or `null` when no cause was set. The cause is another exception that was caught before the current exception is to be thrown.

**Returns**
the cause when this was the last thrown exception and cause was set or `null`. 
**fillInStackTrace**

*Signature*

```java
public java.lang.Throwable
fillInStackTrace()
```

*Description*

Calls into the virtual machine to capture the current stack trace in the instance of `StaticThrowableStorage` associated with this task.

*Returns*

a reference to this `Throwable`.

**setStackTrace(StackTraceElement)**

*Signature*

```java
public void
setStackTrace(java.lang.StackTraceElement[] new_stackTrace)
throws NullPointerException
```

*Description*

This method enables 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.

*Parameters*

`new_stackTrace`—the stack trace to be used as replace.

*Throws*

`NullPointerException`—when `new_stackTrace` or any element of `new_stackTrace` is null.

**getStackTrace**

*Signature*

```java
public java.lang.StackTraceElement[]
getStackTrace()
```

*Description*

924 RTSJ 2.0 (Draft 66)
Gets the stack trace created by fillInStackTrace for this Throwable from the current thread local storage 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.

When memory areas of the RTSJ are used (see MemoryArea), and this Throwable was allocated in a different memory area than the current allocation context, the resulting stack trace will be allocated in either the same memory area this was allocated in or the current memory area, depending on which is the least deeply nested, thereby creating objects that are assignment compatible with both areas.

**Returns**
An array representing the stack trace when this was the last message thrown or StaticThrowableStorage.EMPTY_TRACE, but never null.

**printStackTrace**

**Signature**
```
public void
printStackTrace()
```

**Description**

Prints 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.

**printStackTrace(PrintStream)**

**Signature**
```
public void
printStackTrace(PrintStream stream)
```

**Description**
Prints 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.

Parameters
stream—The stream to print to.

printStackTrace(PrintWriter)

Signature
public void
printStackTrace(PrintWriter writer)

Description
Prints 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.

Parameters
s—The PrintWriter to write to.

16.2.2.33 StaticError

public abstract class StaticError

Inheritance
java.lang.Object
java.lang.Throwable
java.lang.Error
StaticError

Interfaces
javax.realtime.StaticThrowable

Description
A base class for all errors defined in the specification which do not extend a conventional Java error.
StaticError

Available since RTSJ 2.0

16.2.2.33.1 Constructors

StaticError

Signature
    protected
    StaticError()

Description
    Enable this class to be extended.

16.2.2.33.2 Methods

initMessage(String)

Signature
    public java.lang.Throwable
    initMessage(String message)

Description
    Store the message of the calling exception in the instance of StaticThrowableStorage associated with this task. This is the only method not defined in java.lang.Throwable.

Parameters
    message—Text to be saved describing the exception’s cause.

Returns
    this object.

Available since RTSJ 2.0 implemented by all static throwables
**getException**

*Signature*

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

*Description*

Gets the message describing the exception’s cause from thread local memory.

*Returns*

the message when this was the last method thrown and message was set for it or null.

**getLocalizedMessage**

*Signature*

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

*Description*

Subclasses may override this message to get an error message that is localized to the default locale.

By default it returns `getMessage()`.

*Returns*

the value of `getMessage()`.

**initCause(Throwable)**

*Signature*

```
public java.lang.Throwable initCause(Throwable causingThrowable)
```

*Description*

Store the cause of calling exception in the instance of StaticThrowableStorage associated with this task.

*Parameters*

- `causingThrowable`—The reason why this Throwable gets thrown.

*Throws*

928 RTSJ 2.0 (Draft 66)
IllegalRealtimeArgumentException—when the cause is this Throwable itself.

Returns
the reference to this Throwable.

getcause

Signature
public java.lang.Throwable
getcause()

Description
Gets the cause from thread local memory of the calling exception or null when no cause was set. The cause is another exception that was caught before the current exception is to be thrown.

Returns
the cause when this was the last thrown exception and cause was set or null.

fillinStackTrace

Signature
public java.lang.Throwable
fillinStackTrace()

Description
Calls into the virtual machine to capture the current stack trace in the instance of StaticThrowableStorage associated with this task.

Returns
a reference to this Throwable.

setStackTrace(StackTraceElement)

Signature
public void
setStackTrace(java.lang.StackTraceElement[] new_stackTrace)
throws NullPointerException

Description
This method enables 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.

**Parameters**
- `new_stackTrace`—the stack trace to be used as replace.

**Throws**
- `NullPointerException`—when `new_stackTrace` or any element of `new_stackTrace` is null.

### getStackTrace

**Signature**

```java
public java.lang.StackTraceElement[] getStackTrace()
```

**Description**

Gets the stack trace created by `fillInStackTrace` for this `Throwable` from the current thread local storage as an array of `StackTraceElement`

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.

When memory areas of the RTSJ are used (see `MemoryArea`), and this `Throwable` was allocated in a different memory area than the current allocation context, the resulting stack trace will be allocated in either the same memory area `this` was allocated in or the current memory area, depending on which is the least deeply nested, thereby creating objects that are assignment compatible with both areas.

**Returns**

An array representing the stack trace when this was the last message thrown or `StaticThrowableStorage.EMPTY_TRACE`, but never `null`.

### printStackTrace

**Signature**

```java
public void printStackTrace()
```
Description

Prints 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.

printStackTrace(PrintStream)

Signature

public void
printStackTrace(PrintStream stream)

Description

Prints 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.

Parameters

stream—The stream to print to.

printStackTrace(PrintWriter)

Signature

public void
printStackTrace(PrintWriter writer)

Description

Prints 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.

Parameters

s—The PrintWriter to write to.
public abstract class StaticRuntimeException

Inheritance
java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      java.lang.RuntimeException

Interfaces
javax.realtime.StaticThrowable

Description
A base class for all unchecked exceptions defined in the specification which do not extend a conventional Java exception.

Available since RTSJ 2.0

16.2.2.34.1 Constructors

StaticRuntimeException

Signature
protected
StaticRuntimeException()

Description
Enable this class to be extended.

16.2.2.34.2 Methods

932 RTSJ 2.0 (Draft 66)
initMessage(String)

Signature

public java.lang.Throwable
initMessage(String message)

Description

Store the message of the calling exception in the instance of StaticThrowableStorage associated with this task. This is the only method not defined in java.lang.Throwable.

Parameters

message—Text to be saved describing the exception’s cause.

Returns

this object.

Available since RTSJ 2.0 implemented by all static throwables

getMessage

Signature

public java.lang.String
getMessage()

Description

Gets the message describing the exception’s cause from thread local memory.

Returns

the message when this was the last method thrown and message was set for it or null.

getLocalizedMessage

Signature

public java.lang.String
getLocalizedMessage()

Description

RTSJ 2.0 (Draft 66) 933
Subclasses may override this message to get an error message that is localized to the default locale.
  By default it returns `getMessage()`.

**Returns**
the value of `getMessage()`.

### initCause( Throwable )

**Signature**
```
public java.lang.Throwable
initCause( Throwable causingThrowable )
```

**Description**
Store the cause of calling exception in the instance of `StaticThrowableStorage` associated with this task.

**Parameters**
- **causingThrowable**—The reason why this `Throwable` gets thrown.

**Throws**
- `IllegalRealtimeArgumentException`—when the cause is this `Throwable` itself.

**Returns**
the reference to this `Throwable`.

### getCause

**Signature**
```
public java.lang.Throwable
getCause()
```

**Description**
Gets the cause from thread local memory of the calling exception or `null` when no cause was set. The cause is another exception that was caught before the current exception is to be thrown.

**Returns**
the cause when this was the last thrown exception and cause was set or `null`. 
fillInStackTrace

Signature
    public java.lang.Throwable
    fillInStackTrace()

Description
    Calls into the virtual machine to capture the current stack trace in the instance
    of StaticThrowableStorage associated with this task.

Returns
    a reference to this Throwable.

setStackTrace(StackTraceElement)

Signature
    public void
    setStackTrace(java.lang.StackTraceElement[] new_stackTrace)
    throws NullPointerException

Description
    This method enables 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.

Parameters
    new_stackTrace—the stack trace to be used as replace.

Throws
    NullPointerException—when new_stackTrace or any element of new_stackTrace
    is null.

getStackTrace

Signature
    public java.lang.StackTraceElement[]
    getStackTrace()

Description
    RTSJ 2.0 (Draft 66) 935
16 Exceptions

StaticRuntimeException

Gets the stack trace created by fillInStackTrace for this Throwable from the current thread local storage 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.

When memory areas of the RTSJ are used (see MemoryArea), and this Throwable was allocated in a different memory area than the current allocation context, the resulting stack trace will be allocated in either the same memory area this was allocated in or the current memory area, depending on which is the least deeply nested, thereby creating objects that are assignment compatible with both areas.

Returns

An array representing the stack trace when this was the last message thrown or StaticThrowableStorage.EMPTY_TRACE, but never null.

printStackTrace

Signature

public void
printStackTrace()

Description

Prints 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.

printStackTrace(PrintStream)

Signature

public void
printStackTrace(PrintStream stream)

Description

936 RTSJ 2.0 (Draft 66)
Prints 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.

Parameters

stream—The stream to print to.

printStackTrace(PrintWriter)

Signature

cpublic void
printStackTrace(PrintWriter writer)

Description

Prints 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.

Parameters

s—The PrintWriter to write to.

16.2.2.35 StaticThrowableStorage

public class StaticThrowableStorage

Inheritance

java.lang.Object
java.lang.Throwable
StaticThrowableStorage

Interfaces
javax.realtime.StaticThrowable

Description

Provides the methods for managing the thread local memory used for storing the data needed by preallocated throwables, i.e., exceptions and errors which implement StaticThrowable. This call is visible so that an application can
extend an existing conventional Java throwable and still implement Static-Throwables; its methods can be implemented using the methods defined in this class. An application defined throwable that does not need to extend an existing conventional Java throwable should extend one of StaticCheckedException, StaticRuntimeException, or StaticError instead.

Available since RTSJ 2.0

16.2.2.35.1 Fields

EMPTY_TRACE

public static final java.lang.StackTraceElement[] EMPTY_TRACE

Description
A single copy of an empty stack trace.

16.2.2.35.2 Methods

getCurrent

Signature
public static javax.realtime.StaticThrowableStorage getCurrent()

Description
A means of obtaining the storage object for the current task.

Returns
the storage object for the current task.
setLastThrowable(StaticThrowable)

Signature

    public javax.realtime.StaticThrowableStorage
    setLastThrowable(StaticThrowable value)

Description

    Capture the exception to be thrown, so as to provide a context for the data stored
    in this object.

g getLasthrowable

Signature

    public javax.realtime.StaticThrowable
    getLastThrowable()

Description

    Determine for what throwable the data is valid;

Returns

    the current context for the throwable data stored in this object.

fillInStackTrace

Signature

    public java.lang.Throwable
    fillInStackTrace()

Description

    Captures the current thread’s stack trace and saves it in thread local storage.
    Only the part of the stack trace that fits in the preallocated buffer is stored. This
    method should be called by a preallocated exception to implement its method of
    the same name.

Returns

    this
getMessage

Signature

public java.lang.String
getMessage()

Description

Gets the message from thread local storage that was saved by the last preallocated exception thrown. This method should be called by a preallocated exception to implement its method of the same name.

Returns

the message.

initMessage(String)

Signature

public java.lang.Throwable
initMessage(String message)

Description

Saves 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.

Parameters

message—The description to save.

getCause

Signature

public java.lang.Throwable
getCause()

Description

Gets the cause from thread local storage that was saved by the last preallocated exception thrown. The actual exception of cause is not saved, but just a reference to its type. This returns a newly allocated exception without any valid content, i.e., no valid stack trace. This method should be called by a preallocated exception to implement its method of the same name.
Returns
the message

**initCause(Throwables)**

*Signature*

\[
\text{public java.lang.Throwable}
\]
\[
\text{initCause(Throwables causingThrowable)}
\]

*Description*
Saves 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.

*Parameters*

causingThrowable—In the case of cascading throwables, the exception or error
that was the original cause.

*Returns*
this

**getStackTrace**

*Signature*

\[
\text{public java.lang.StackTraceElement[]} 
\]
\[
\text{getStackTrace()}
\]

*Description*
Gets the stack trace from thread local storage that was saved by the last pre-
allocated exception thrown. This method should be called by a preallocated
exception to implement its method of the same name.

*Returns*
an array of the elements of the stack trace.

**getLocalizedMessage**

*Signature*
public java.lang.String getLocalizedMessage()

**Description**
Subclasses may override this message to get an error message that is localized to the default locale.
By default it returns `getMessage()`.

**Returns**
the value of `getMessage()`.

**setStackTrace(StackTraceElement)**

**Signature**
public void
`setStackTrace(java.lang.StackTraceElement[] new_stackTrace)`

**Description**
This method enables 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.

**Parameters**
new_stackTrace—the stack trace to be used as replace.

**Throws**
NullPointerException—when new_stackTrace or any element of new_stackTrace is null.

**printStackTrace**

**Signature**
public void
`printStackTrace()`

**Description**
Prints 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.
printStackTrace(PrintStream)

Signature

public void
printStackTrace(PrintStream stream)

Description
Prints 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.

Parameters
stream—The stream to print to.

printStackTrace(PrintWriter)

Signature

public void
printStackTrace(PrintWriter writer)

Description
Prints 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.

Parameters
s—The PrintWriter to write to.

16.2.2.36 ThrowBoundaryError

public class ThrowBoundaryError

Inheritance
java.lang.Object
java.lang.Throwable
java.lang.Error
16 Exceptions

**StaticError**

**ThrowBoundaryError**

**Description**

The error thrown by `MemoryArea.enter(Runnable logic)` when a `Throwable` allocated from memory that is not usable in the surrounding scope tries to propagate out of the scope of the `enter`.

**Available since** RTSJ 2.0 extends StaticError

### 16.2.2.36.1 Constructors

**ThrowBoundaryError**

**Signature**

```java
public
ThrowBoundaryError()
```

**Description**

A constructor for `ThrowBoundaryError`, but application code should use `get()` instead.

### 16.2.2.36.2 Methods

**get**

**Signature**

```java
public static javax.realtime.ThrowBoundaryError
get()
```

**Description**

Gets the preallocated instance of this exception.

**Returns**

the preallocated instance of this exception.

944 RTSJ 2.0 (Draft 66)
16.2.2.37 UninitializedStateException

public class UninitializedStateException

Inheritance
java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      java.lang.RuntimeException
        StaticRuntimeException
          UninitializedStateException

Description
Thrown when a resource is not yet initialized, such as a Clock which cannot be
created yet because its data source is not yet available. This can happen when a
Java process starts early in the system startup process.

Available since RTSJ 2.0

16.2.2.37.1 Methods

get

Signature
  public static javax.realtime.UninitializedStateException
    get()

Description
  Gets the static instance of this class and initializes its stack trace.

Returns
  the static singleton of this class.

16.2.2.38 UnsupportedPhysicalMemoryException

public class UnsupportedPhysicalMemoryException

Inheritance
java.lang.Object
    java.lang.Throwable
    java.lang.Exception
    java.lang.RuntimeException
    StaticRuntimeException
    UnsupportedPhysicalMemoryException

Description

Thrown when the underlying hardware does not support the type of physical memory requested.

See Section PhysicalMemoryFactory

Available since RTSJ 1.0.1 became unchecked

Available since RTSJ 2.0 extends StaticRuntimeException

16.2.2.38.1 Constructors

UnsupportedPhysicalMemoryException

<table>
<thead>
<tr>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>public UnsupportedPhysicalMemoryException()</td>
</tr>
</tbody>
</table>

Description

A constructor for UnsupportedPhysicalMemoryException, but application code should use get() instead.

16.2.2.38.2 Methods

get

<table>
<thead>
<tr>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>946 RTSJ 2.0 (Draft 66)</td>
</tr>
</tbody>
</table>
public static javax.realtime.UnsupportedPhysicalMemoryException
get()

Description
Obtains the singleton of this static throwable. It is prepared for immediate throwing.

Returns
the single instance of this throwable.

Available since RTSJ 2.0

16.2.2.39 UnsupportedRawMemoryRegionException

public class UnsupportedRawMemoryRegionException

Inheritance
java.lang.Object
    java.lang.Throwable
        java.lang.Exception
            java.lang.RuntimeException
                StaticRuntimeException
                    UnsupportedRawMemoryRegionException

Description
Indicates an invalid raw memory region.

Available since RTSJ 2.0

16.2.2.39.1 Methods

get

Signature
public static javax.realtime.UnsupportedRawMemoryRegionException
get()

Description

RTSJ 2.0 (Draft 66) 947
Obtains the singleton of this static throwable. It is prepared for immediate throwing.

Returns
the single instance of this throwable.

16.2.2.40 UnsupportedRealtimeOperationException

public class UnsupportedRealtimeOperationException

Inheritance
java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      java.lang.RuntimeException
        java.lang.UnsupportedOperationException
          UnsupportedRealtimeOperationException

Interfaces
javax.realtime.StaticThrowable

Description
A version of UnsupportedOperationException to be thrown by RTSJ methods that does not require allocation.

Available since RTSJ 2.0

16.2.2.40.1 Methods

get

Signature
  public static javax.realtime.UnsupportedRealtimeOperationException get()

Description
Gets the preallocated version of this Throwable. Allocation is done in memory that acts like ImmortalMemory. The message and cause are cleared and the stack trace is filled out.
Returns
the preallocated exception.

initMessage(String)

Signature
public java.lang.Throwable
initMessage(String message)

Description
Store the message of the calling exception in the instance of StaticThrowableStorage associated with this task. This is the only method not defined in java.lang.Throwable.

Parameters
message—Text to be saved describing the exception’s cause.

Returns
this object.

Available since RTSJ 2.0 implemented by all static throwables

gMessage

Signature
public java.lang.String
getMessage()

Description
Gets the message describing the exception’s cause from thread local memory.

Returns
the message when this was the last method thrown and message was set for it or null.

getLocalizedMessage

Signature
public java.lang.String
getLocalizedMessage()
16 Exceptions

UnsupportedRealtimeOperationException

Description
Subclasses may override this message to get an error message that is localized to the default locale.
By default it returns `getMessage()`.

Returns
the value of `getMessage()`.

`initCause( Throwable )`

Signature
`public java.lang.Throwable
initCause( Throwable causingThrowable )`

Description
Store the cause of calling exception in the instance of `StaticThrowableStorage` associated with this task.

Parameters
`causingThrowable`—The reason why this `Throwable` gets thrown.

Throws
`IllegalRealtimeArgumentException`—when the cause is this `Throwable` itself.

Returns
the reference to this `Throwable`.

`getCause`

Signature
`public java.lang.Throwable
getCause()`

Description
Gets the cause from thread local memory of the calling exception or null when no cause was set. The cause is another exception that was caught before the current exception is to be thrown.

Returns
the cause when this was the last thrown exception and cause was set or null.
UnsupportedRealtimeOperationException  

Signature

```java
public java.lang.Throwable
fillInStackTrace()
```

Description

Calls into the virtual machine to capture the current stack trace in the instance of `StaticThrowableStorage` associated with this task.

Returns

a reference to this `Throwable`.

**setStackTrace(StackTraceElement)**

Signature

```java
public void
setStackTrace(java.lang.StackTraceElement[] trace)
throws NullPointerException
```

Description

This method enables 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.

Parameters

- `new_stackTrace`—the stack trace to be used as replace.

Throws

- `NullPointerException`—when `new_stackTrace` or any element of `new_stackTrace` is null.

**getStackTrace**

Signature

```java
public java.lang.StackTraceElement[]
getStackTrace()
```

Description
16 Exceptions

UnsupportedRealtimeOperationException

Gets the stack trace created by fillInStackTrace for this Throwable from the current thread local storage 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.

When memory areas of the RTSJ are used (see MemoryArea), and this Throwable was allocated in a different memory area than the current allocation context, the resulting stack trace will be allocated in either the same memory area this was allocated in or the current memory area, depending on which is the least deeply nested, thereby creating objects that are assignment compatible with both areas.

Returns
An array representing the stack trace when this was the last message thrown or StaticThrowableStorage.EMPTY_TRACE, but never null.

printStackTrace

Signature
public void
printStackTrace()

Description
Prints 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.

printStackTrace(PrintStream)

Signature
public void
printStackTrace(PrintStream stream)

Description
Prints 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.

Parameters

stream—The stream to print to.

printStackTrace(PrintWriter)

Signature

public void
printStackTrace(PrintWriter writer)

Description

Prints 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.

Parameters

s—The PrintWriter to write to.

16.3 Rationale

Given the new classes and semantics added to Java by this specification, the need for new exceptions is obvious; however, the need for static exceptions may be less clear. Exceptions, as defined in conventional Java, are heavy weight in that they cause a significant amount of memory to be allocated when created. Every throw requires the creation of an exception with a full backtrace. In scoped memory, this can dramatically increase the size of scopes necessary. Even with determinitic garbage collection, this is quite expensive in execution time. Ideally, one would not need to allocate some much memory and even be able to limit the amount of stack trace captured.

The concept of static exceptions was added to the specification to ameliorate this conflict. Conventional Java classes retain the “allocate on throw” paradigm, but exceptions thrown directly from classes in this specification use either their own exception or a subclass of an existing exception that implements the StaticThrowable interface, indicating that their exception information is stored in a
Exceptions

preallocated thread local structure. This structure’s size can be controlled via the
ConfigurationParameters (Section 5.3.2.1) class. The stacktrace is captured from
the deepest nesting up until there is no room left to store its elements.

This minimizes allocation and limits the amount of copying at throw time while
maintaining a high degree of compatibility with conventional Java exception semantics.
The difference being in how static exceptions are thrown and in what context their
information is valid. Since the main use of the backtrace is for debugging, it seems
reasonable to limit the validity of the information to the thread in which it was
created.

There is one part of the specification that require special exception support:
asynchronous transfer of control (ATC). This requires the addition of an asynchronous
exceptions: AsynchronouslyInterruptedException. Almost all conventional Java
exceptions are thrown synchronously. The notable exception is the ThreadDeath error,
which is thrown when Thread.stop() is called. The problem with ThreadDeath is
cleaning up resources. AsynchronouslyInterruptedException addresses this by
extending InterruptedException, so as to be able to break out of I/O exceptions
and clean up resources. In addition, ATC only applies to well-defined code blocks
giving further control over cleanup.
Appendix A

Bibliography


Appendix B

Deprecated APIs

Since modules are new in RTSJ 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 old classes, including those that have copies in new subpackages, are deprecated and appear only here. Other deprecated methods, constructors, and fields, including those for feasibility analysis, are also here. The feasibility methods are the only ones without replacement, as they where insufficient for admission control, so where of little utility. Only full implementation of the RTSJ should implement them.

B.1 Semantics

Implementations of the deprecated interfaces, classes, constructors, methods, and field given below are optional. In some cases, classes have been moved to a new package. In this case, the class appears here in its old place and in the documentation above in its new place. The deprecated elements are only needed for backward compatibility. They should not be included in implementations that do not include all modules.
B Deprecated APIs

B.2 javax.realtime

B.2.1 Interfaces

B.2.1.1 Interruptible

public interface Interruptible

 Interfaces
javax.realtime.control.Interruptible

Description

 Interruptible is an interface implemented by classes that will be used as arguments on the method doInterruptible() of AsynchronouslyInterruptedException and its subclasses. doInterruptible() invokes the implementations of the methods in this interface.

Deprecated in RTSJ 2.0; moved to package javax.realtime.control

B.2.1.1.1 Methods

 run(AsynchronouslyInterruptedException)

Signature

 public void run(AsynchronouslyInterruptedException exception)
  throws AsynchronouslyInterruptedException

Description

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.

Parameters

 exception—The AIE object whose doInterruptible method is calling the run method. Used to invoke methods on AsynchronouslyInterruptedException from within the run() method.
interruptAction(AsynchronouslyInterruptedException)

*Signature*

```java
public void interruptAction(AsynchronouslyInterruptedException exception)
```

*Description*

This method is called by the system when the `run()` method is interrupted. By using this, the program logic can determine when the `run()` method completed normally or had its control asynchronously transferred to its caller.

*Parameters*

- `exception` — The currently pending AIE. Used to invoke methods on `AsynchronouslyInterruptedException` from within the `interruptAction()` method.

### B.2.1.2 PhysicalMemoryTypeFilter

```java
public interface PhysicalMemoryTypeFilter
```

*Description*

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`, not directly from application code.

*Deprecated* as of RTSJ 2.0

#### B.2.1.2.1 Methods

`contains(long, long)`

*Signature*

```java
public boolean contains(long base, long size)
```
**B Deprecated APIs**

**Description**

Queries the system about whether the specified range of memory contains any of this type.

**Parameters**

- **base**—The physical address of the beginning of the 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.

**Returns**

`true`, when the specified range contains ANY of this type of memory.

See Section PhysicalMemoryManager.isRemovable

**find(long, long)**

**Signature**

```java
public long find(long base,
                 long size)
```

**Description**

Search for physical memory of the right type.

**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

Signature

public int
getVMAttributes()

Description

 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.

Returns

the virtual memory attributes as an integer.

getVMFlags

Signature

public int
getVMFlags()

Description

 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.

Returns

the virtual memory flags as an integer.

initialize(long, long, long)

Signature

public void
initialize(long base,
        long vBase,
        long size)

Description

RTSJ 2.0 (Draft 66)
When configuration is required for memory to fit the attribute of this object, do the configuration here.

**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)**

**Signature**
```java
public boolean isPresent(long base,
                        long size)
```

**Description**
Queries the system about the existence of the specified range of physical memory.

**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.

See Section PhysicalMemoryManager.isRemoved
isRemovable

Signature

public boolean
isRemovable()

Description

Queries the system about the removability of this memory.

Returns

ture, when this type of memory is removable.

onInsertion(long, long, AsyncEvent)

Signature

public void
onInsertion(long base,
            long size,
            AsyncEvent ae)

Description

Register the specified AsyncEvent to fire when any memory of this type in the range is added to the system.

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.

Available since RTSJ 1.0.1
onRemoval(long, long, AsyncEvent)

Signature
public void
    onRemoval(long base,
              long size,
              AsyncEvent ae)

Description
Registers the specified AE to fire when any memory in the range is removed from
the system.

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 mem-
ory 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.

Available since RTSJ 1.0.1

unregisterInsertionEvent(long, long, AsyncEvent)

Signature
public boolean
    unregisterInsertionEvent(long base,
                              long size,
                              AsyncEvent ae)

Description
Unregisters 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 that this method has no effect on handlers registered directly as async
event handlers.
PhysicalMemoryTypeFilter

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.

Available since RTSJ 1.0.1

unregisterRemovalEvent(long, long, AsyncEvent)

Signature
```java
public boolean unregisterRemovalEvent(long base,
                                      long size,
                                      AsyncEvent ae)
```

Description
Unregisters 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 that this method has no effect on handlers registered directly as async event handlers.

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.
**B Deprecated APIs**

**Schedulable**

Returns

true, when at least one event matched the pattern, false when no such event was found.

Available since RTSJ 1.0.1

---

**vFind(long, long)**

**Signature**

```java
public long vFind(long base, long size)
```

**Description**

Searches for virtual memory of the right type. This is important for systems where attributes are associated with particular ranges of virtual memory.

**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.

---

**B.2.1.3 Schedulable**

---

public interface Schedulable

The following elements of Schedulable are deprecated. The required elements are documented in Section 6.3.1.2 above.

---

**B.2.1.3.1 Methods**

---

966 RTSJ 2.0 (Draft 66)
setScheduler(Scheduler, SchedulingParameters, ReleaseParameters, MemoryParameters, ProcessingGroupParameters)

**Signature**

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

**Description**

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`.

**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` 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`.
- **release**—A reference to the `ReleaseParameters` 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`.)
- **memoryParameters**—A reference to the `MemoryParameters` 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`.)
- **group**—A reference to the `ProcessingGroupParameters` 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`.)

**Throws**

- `IllegalRealtimeArgumentException`—when `scheduler` is null or the parameter values are not compatible with `scheduler`. Also thrown when this schedulable may not use the heap and `scheduler`, scheduling `release`, memory `parameters`, 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`. 

RTSJ 2.0 (Draft 66) 967
IllegalThreadStateException—when scheduler prohibits the changing of the scheduler or a parameter at this time due to the state of the schedulable.

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

Deprecated since RTSJ 2.0

getProcessingGroupParameters

Signature
public javax.realtime.ProcessingGroupParameters getProcessingGroupParameters()

Description
Gets a reference to the ProcessingGroupParameters object for this schedulable.

Returns
A reference to the current ProcessingGroupParameters object.

Deprecated since RTSJ 2.0

setProcessingGroupParameters(ProcessingGroupParameters)

Signature
public void setProcessingGroupParameters(ProcessingGroupParameters group)

Description
Sets the ProcessingGroupParameters 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.

Parameters
group—A ProcessingGroupParameters object which will 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.)
Throws

- `IllegalRealtimeArgumentException`—when `group` is not compatible with the scheduler for this schedulable object. Also when this schedulable may not use the 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.

`Deprecated` since RTSJ 2.0; see `ProcessingGroup`

`setProcessingGroupParametersIfFeasible` (ProcessingGroupParameters)

**Signature**

```java
public boolean setProcessingGroupParametersIfFeasible(ProcessingGroupParameters group)
```

**Description**

This method first performs a feasibility analysis using the proposed parameter object as replacement for the current parameter of `this`. When the resulting system is feasible, this method replaces the current parameter of `this` with the proposed one.

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.

This method does not require that the schedulable be in the feasibility set before it is called. When it is not initially a member of the feasibility set it will be added when the resulting system is feasible.

**Parameters**

- `group`—The processing group parameters to use. 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`.)

**Throws**

- `IllegalRealtimeArgumentException`—when the parameter value is not compatible with the schedulable’s scheduler. Also when this schedulable may not use the heap and the specified parameter object is located in heap memory.
IllegalAssignmentError—when this cannot hold a reference to the specified parameter object, or the parameter object 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.

Returns
true, when the resulting system is feasible and the changes are made. False, when the resulting system is not feasible and no changes are made.

Deprecated as of RTSJ 2.0 The framework for feasibility analysis is inadequate

addIfFeasible

Signature
public boolean addIfFeasible()

Description
This method first performs a feasibility analysis with this added to the system. When the resulting system is feasible, informs the scheduler and cooperating facilities that this instance of Schedulable should be considered in feasibility analysis until further notified. When the analysis shows that the system including this would not be feasible, this method does not admit this to the feasibility set.

When the object is already included in the feasibility set, does nothing.

Returns
true when inclusion of this in the feasibility set yields a feasible system, and false otherwise. When true is returned then this is known to be in the feasibility set. When false is returned, this was not added to the feasibility set, but it may already have been present.

Available since RTSJ 1.0.1 Promoted to the Schedulable interface

Deprecated as of RTSJ 2.0, because the framework for feasibility analysis is inadequate.

addToFeasibility

Signature

970 RTSJ 2.0 (Draft 66)
public boolean
addToFeasibility()

Description
Informs the scheduler and cooperating facilities that this instance of `Schedulable`
should be considered in feasibility analysis until further notified.
When the object is already included in the feasibility set, does nothing.

Returns
true, when the resulting system is feasible. False, when not.

Deprecated as of RTSJ 2.0 The framework for feasibility analysis is inadequate.

removeFromFeasibility

Signature
public boolean
removeFromFeasibility()

Description
Informs the scheduler and cooperating facilities that this instance of `Schedulable`
should not be considered in feasibility analysis until it is further notified.

Returns
true when the removal was successful. false when the schedulable cannot be
removed from the scheduler’s feasibility set; e.g., the schedulable is not part of
the scheduler’s feasibility set.

Deprecated as of RTSJ 2.0 The framework for feasibility analysis is inadequate

setIfFeasible(ReleaseParameters, MemoryParameters)

Signature
public boolean
setIfFeasible(ReleaseParameters release,
MemoryParameters memory)
throws IllegalRealtimeArgumentException,
IllegalAssignmentError,
IllegalThreadStateException
Description

This method first performs a feasibility analysis using the proposed parameter objects as replacements for the current parameters of this. When the resulting system is feasible, this method replaces the current parameters of this with the proposed ones.

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.

This method does not require that the schedulable be in the feasibility set before it is called. When it is not initially a member of the feasibility set it will be added when the resulting system is feasible.

Parameters

release—The release parameters to use. 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.)

memory—The memory parameters to use. 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.)

Throws

IllegalRealtimeArgumentException—when the parameter values are not compatible with the schedulable’s scheduler. Also when this schedulable may not use the heap and any of the specified parameter objects are located in heap memory.

IllegalAssignmentError—when this cannot hold references to the specified parameter objects, or the parameter objects cannot hold a reference to this.

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

Returns

true, when the resulting system is feasible and the changes are made. False, when the resulting system is not feasible and no changes are made.

Available since RTSJ 1.0.1 Promoted to the Schedulable interface.

Deprecated as of RTSJ 2.0 The framework for feasibility analysis is inadequate

setIfFeasible(ReleaseParameters, MemoryParameters, ProcessingGroupParameters)
**Signature**

```java
public boolean setIfFeasible(ReleaseParameters release,
              MemoryParameters memory,
              ProcessingGroupParameters group)
throws IllegalRealtimeArgumentException,
IllegalAssignmentError,
IllegalThreadStateException
```

**Description**

This method first performs a feasibility analysis using the proposed parameter objects as replacements for the current parameters of this. When the resulting system is feasible, this method replaces the current parameters of this with the proposed ones.

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.

This method does not require that the schedulable be in the feasibility set before it is called. When it is not initially a member of the feasibility set it will be added when the resulting system is feasible.

**Parameters**

- **release**—The release parameters to use. 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`.)

- **memory**—The memory parameters to use. 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`.)

- **group**—The processing group parameters to use. 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`.)

**Throws**

- `IllegalRealtimeArgumentException`—when the parameter values are not compatible with the schedulable’s scheduler. Also when this schedulable may not use the heap and any of the specified parameter objects are located in heap memory.

- `IllegalAssignmentError`—when this cannot hold references to the specified parameter objects, or the parameter objects cannot hold a reference to this.
setIfFeasible(ReleaseParameters, ProcessingGroupParameters)

Signature

public boolean
    setIfFeasible(ReleaseParameters release,
                   ProcessingGroupParameters group)
        throws IllegalRealtimeArgumentException,
               IllegalAssignmentError,
               IllegalThreadStateException

Description

This method first performs a feasibility analysis using the proposed parameter objects as replacements for the current parameters of this. When the resulting system is feasible, this method replaces the current parameters of this with the proposed ones.

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.

This method does not require that the schedulable be in the feasibility set before it is called. When it is not initially a member of the feasibility set it will be added when the resulting system is feasible.

Parameters

release—The release parameters to use. 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.)
group—The processing group parameters to use. 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.)

Throws

IllegalRealtimeArgumentException—when the parameter values are not compatible with the schedulable’s scheduler. Also when this schedulable may not use the heap and any of the specified parameter objects are located in heap memory.

IllegalAssignmentError—when this cannot hold references to the specified parameter objects, or the parameter objects cannot hold a reference to this.

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

Returns

true, when the resulting system is feasible and the changes are made. False, when the resulting system is not feasible and no changes are made.

Available since RTSJ 1.0.1 Promoted to the Schedulable interface.

Deprecated as of RTSJ 2.0 The framework for feasibility analysis is inadequate

setIfFeasible(SchedulingParameters, ReleaseParameters, MemoryParameters)

Signature

public boolean
setIfFeasible(SchedulingParameters scheduling,
ReleaseParameters release,
MemoryParameters memory)
throws IllegalRealtimeArgumentException,
IllegalAssignmentError,
IllegalThreadStateException

Description

This method first performs a feasibility analysis using the proposed parameter objects as replacements for the current parameters of this. When the resulting system is feasible, this method replaces the current parameters of this with the proposed ones.

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.

This method does not require that the schedulable be in the feasibility set before it is called. When it is not initially a member of the feasibility set it will be added when the resulting system is feasible.

Parameters

- **scheduling**—The scheduling parameters to use. 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.)
- **release**—The release parameters to use. 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.)
- **memory**—The memory parameters to use. 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.)

Throws

- **IllegalRealtimeArgumentException**—when the parameter values are not compatible with the schedulable’s scheduler. Also when this schedulable may not use the heap and any of the specified parameter objects are located in heap memory.
- **IllegalAssignmentError**—when this cannot hold references to the specified parameter objects, or the parameter objects cannot hold a reference to this.
- **IllegalThreadStateException**—when the schedulable’s scheduler prohibits this parameter change at this time due to the state of the schedulable.

Returns

true, when the resulting system is feasible and the changes are made. False, when the resulting system is not feasible and no changes are made.

Available since RTSJ 1.0.1

Deprecated as of RTSJ 2.0 The framework for feasibility analysis is inadequate
Schedulable

```
  ReleaseParameters release,
  MemoryParameters memory,
  ProcessingGroupParameters group)
```

**Description**

This method first performs a feasibility analysis using the proposed parameter objects as replacements for the current parameters of this. When the resulting system is feasible, this method replaces the current parameters of this with the proposed ones.

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.

This method does not require that the schedulable be in the feasibility set before it is called. When it is not initially a member of the feasibility set it will be added when the resulting system is feasible.

**Parameters**

- **scheduling**—The scheduling parameters to use. 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.)

- **release**—The release parameters to use. 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.)

- **memory**—The memory parameters to use. 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.)

- **group**—The processing group parameters to use. 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.)

**Throws**

- **IllegalRealtimeArgumentException**—when the parameter values are not compatible with the schedulable’s scheduler. Also when this schedulable may not use the heap and any of the specified parameter objects are located in heap memory.

- **IllegalAssignmentError**—when this cannot hold references to the specified parameter objects, or the parameter objects cannot hold a reference to this.

- **IllegalThreadStateException**—when the schedulable’s scheduler prohibits this parameter change at this time due to the state of the schedulable.

**Returns**
true, when the resulting system is feasible and the changes are made. False, when the resulting system is not feasible and no changes are made.

Available since RTSJ 1.0.1

Deprecated as of RTSJ 2.0 The framework for feasibility analysis is inadequate

setMemoryParametersIfFeasible(MemoryParameters)

Signature
public boolean setMemoryParametersIfFeasible(MemoryParameters memory)

Description
This method first performs a feasibility analysis using the proposed parameter object as replacement for the current parameter of this. When the resulting system is feasible, this method replaces the current parameter of this with the proposed one.

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.

This method does not require that the schedulable be in the feasibility set before it is called. When it is not initially a member of the feasibility set it will be added when the resulting system is feasible.

Parameters
memory—The memory parameters to use. 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.)

Throws
IllegalRealtimeArgumentException—when the parameter value is not compatible with the schedulable’s scheduler. Also when this schedulable may not use the heap and the specified parameter object is located in heap memory.

IllegalAssignmentError—when this cannot hold a reference to the specified parameter object, or the parameter object cannot hold a reference to this.

IllegalThreadStateException—when the schedulable’s scheduler prohibits the changing of the memory parameter at this time due to the state of the schedulable.

Returns
true, when the resulting system is feasible and the changes are made. False, when the resulting system is not feasible and no changes are made.

**Deprecated** as of RTSJ 2.0 The framework for feasibility analysis is inadequate

**setReleaseParametersIfFeasible(ReleaseParameters)**

**Signature**

```java
public boolean setReleaseParametersIfFeasible(ReleaseParameters release)
```

**Description**

This method first performs a feasibility analysis using the proposed parameter object as replacement for the current parameter of this. When the resulting system is feasible, this method replaces the current parameter of this with the proposed one.

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.

This method does not require that the schedulable be in the feasibility set before it is called. When it is not initially a member of the feasibility set it will be added when the resulting system is feasible.

**Parameters**

- `release`—The release parameters to use. 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.)

**Throws**

- `IllegalRealtimeArgumentException`—when the parameter value is not compatible with the schedulable’s scheduler. Also when this schedulable may not use the heap and the specified parameter object is located in heap memory.
- `IllegalAssignmentError`—when this cannot hold a reference to the specified parameter object, or the parameter object cannot hold a reference to this.
- `IllegalThreadStateException`—when the schedulable’s scheduler prohibits the changing of the release parameter at this time due to the state of the schedulable.

**Returns**

- true, when the resulting system is feasible and the changes are made. False, when the resulting system is not feasible and no changes are made.
**Deprecated** as of RTSJ 2.0 The framework for feasibility analysis is inadequate.

**setSchedulingParametersIfFeasible(SchedulingParameters)**

*Signature*

```java
public boolean setSchedulingParametersIfFeasible(SchedulingParameters scheduling)
```

*Description*

This method first performs a feasibility analysis using the proposed parameter object as replacement for the current parameter of `this`. When the resulting system is feasible, this method replaces the current parameter of `this` with the proposed one.

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.

This method does not require that the schedulable be in the feasibility set before it is called. When it is not initially a member of the feasibility set it will be added when the resulting system is feasible.

*Parameters*

- `scheduling`—The scheduling parameters to use. 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`.)

*Throws*

- `IllegalRealtimeArgumentException`—when the parameter value is not compatible with the schedulable’s scheduler. Also when this schedulable may not use the heap and the specified parameter object is located in heap memory.
- `IllegalAssignmentError`—when `this` cannot hold a reference to the specified parameter object, or the parameter object 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.

*Returns*

- `true`, when the resulting system is feasible and the changes are made. False, when the resulting system is not feasible and no changes are made.

**Deprecated** as of RTSJ 2.0 The framework for feasibility analysis is inadequate
B.2.2 Classes

B.2.2.1 AbsoluteTime

public class AbsoluteTime

The following elements of AbsoluteTime are deprecated. The required elements are documented in Section 9.3.1.1 above.

B.2.2.1.1 Constructors

AbsoluteTime(long, int, Clock)

Signature

public

AbsoluteTime(long millis,

int nanos,

Clock clock)

throws IllegalRealtimeArgumentException

Description

Superceeded by and equivalent to AbsoluteTime(long, int, Chronograph)

Available since RTSJ 1.0.1

Deprecated since version 2.0

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.

clock—The clock providing the association for the newly constructed object.

Throws

IllegalRealtimeArgumentException—when there is an overflow in the millisecond component when normalizing.
AbsoluteTime(AbsoluteTime, Clock)

Signature
public
   AbsoluteTime(AbsoluteTime time,
               Clock clock)
   throws IllegalRealtimeArgumentException

Description
Equivalent to AbsoluteTime(long, int, Chronograph) with the arguments
   time.getMilliseconds(), time.getNanoseconds(), clock().

Available since RTSJ 1.0.1

 Deprecated since version 2.0

Parameters
time—The AbsoluteTime object which is the source for the copy.
clock—The clock providing the association for the newly constructed object.

Throws
IllegalRealtimeArgumentException—when the time parameter is null.

AbsoluteTime(Date, Clock)

Signature
public
   AbsoluteTime(Date date,
               Clock clock)
   throws IllegalRealtimeArgumentException

Description
Superceeded by and equivalent to AbsoluteTime(Date, Chronograph)

Available since RTSJ 1.0.1

 Deprecated since version 2.0

Parameters
date—The java.util.Date representation of the time past the Epoch.
clock—The clock providing the association for the newly constructed object.

Throws
IllegalRealtimeArgumentException—when the date parameter is null.

AbsoluteTime(Clock)

Signature
public AbsoluteTime(Clock clock)

Description
Superceeded by and equivalent to AbsoluteTime(Chronograph)

Available since RTSJ 1.0.1

Deprecated since version 2.0

Parameters
clock—The clock providing the association for the newly constructed object.

B.2.2.1.2 Methods

absolute(Clock)

Signature
public javax.realtime.AbsoluteTime absolute(Clock clock)

Description
Superceeded by and equivalent to absolute(Chronograph).

Parameters
clock—The clock parameter is used only as the new clock association with the result, since no conversion is needed.

Returns
the copy of this in a newly allocated AbsoluteTime object, associated with the clock parameter.

**Deprecated** since version 2.0

### absolute(Clock, AbsoluteTime)

**Signature**

```java
public javax.realtime.AbsoluteTime absolute(Clock clock,
                                           AbsoluteTime dest)
```

**Description**

Superceeded by and equivalent to `absolute(Chronograph, AbsoluteTime)`.

**Parameters**

- **clock**—The clock parameter is used only as the new clock association with the result, since no conversion is needed.
- **dest**—When `dest` is not `null`, the result is placed in it and returned.

**Returns**

the copy of this in `dest` when `dest` is not `null`, otherwise the result is returned in a newly allocated object. It is associated with the `clock` parameter.

**Deprecated** since version 2.0

### relative(Clock)

**Signature**

```java
public javax.realtime.RelativeTime relative(Clock clock)
```

**Description**

Superceeded by and equivalent to `relative(Chronograph)`.

**Parameters**

- **clock**—The instance of `Clock` 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.

Deprecated since version 2.0

relative(Clock, RelativeTime)

Signature
public javax.realtime.RelativeTime relative(Clock clock,
       RelativeTime dest)
       throws ArithmeticException

Description
Superceded by and equivalent to relative(Chronograph, RelativeTime).

Parameters
clock—The instance of Clock 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 in it and returned.

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 clock parameter.

Deprecated since version 2.0

B.2.2.2 AperiodicParameters

public class AperiodicParameters

The following elements of AperiodicParameters are deprecated. The required elements are documented in Section 6.3.3.2 above.

B.2.2.2.1 Fields
arrivalTimeQueueOverflowExcept

public static final java.lang.String arrivalTimeQueueOverflowExcept

Description

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 ArrivalTimeQueueOverflowExcept. 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 is bound then the arrival time is ignored.

Available since RTSJ 1.0.1 Moved here from SporadicParameters.

Deprecated since RTSJ 2.0

arrivalTimeQueueOverflowIgnore

public static final java.lang.String arrivalTimeQueueOverflowIgnore

Description

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.

Deprecated since RTSJ 2.0

arrivalTimeQueueOverflowReplace

public static final java.lang.String arrivalTimeQueueOverflowReplace

Description

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.

Deprecated since RTSJ 2.0

arrivalTimeQueueOverflowSave

public static final java.lang.String arrivalTimeQueueOverflowSave

Description

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;” 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.

Deprecated since RTSJ 2.0

B.2.2.2.2 Methods

getInitialArrivalTimeQueueLength

Signature

public int getInitialArrivalTimeQueueLength()

Description

RTSJ 2.0 (Draft 66)
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.

Returns
The initial length of the queue.

Available since RTSJ 1.0.1 Moved here from SporadicParameters.

Deprecated since RTSJ 2.0 replaced by ReleaseParameters.

getInitialQueueLength().

setInitialArrivalTimeQueueLength(int)

Signature
public void
setInitialArrivalTimeQueueLength(int initial)

Description
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.

Parameters
initial—The initial length of the queue.

Throws
IllegalRealtimeArgumentException—when initial is less than zero.

Available since RTSJ 1.0.1 Moved here from SporadicParameters.

Deprecated since RTSJ 2.0 replaced by ReleaseParameters.

setInitialQueueLength(int initial).

getArrivalTimeQueueOverflowBehavior

Signature
public java.lang.String
getArrivalTimeQueueOverflowBehavior()
Description

Gets the behavior of the arrival time queue in the event of an overflow.

Returns

The behavior of the arrival time queue as a string.

Available since RTSJ 1.0.1 Moved from SporadicParameters

Deprecated since RTSJ 2.0 and replaced by ReleaseParameters.

setArrivalTimeQueueOverflowBehavior(String)

Signature

public void
setArrivalTimeQueueOverflowBehavior(String behavior)

Description

Sets the behavior of the arrival time queue for the case where the insertion of a new element makes the queue size greater than the initial size given when this object was constructed.

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

Parameters

behavior—A string representing the behavior.

Throws

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

Available since RTSJ 1.0.1 Moved here from SporadicParameters.

Deprecated Since RTSJ 2.0

setIfFeasible(RelativeTime, RelativeTime)

Signature

public boolean
setIfFeasible(RelativeTime cost,
               RelativeTime deadline)

RTSJ 2.0 (Draft 66)
B Deprecated APIs

ArrivalTimeQueueOverflowException

Description
This method first performs a feasibility analysis using the new cost and deadline as replacements for the matching attributes of this. When the resulting system is feasible, the method replaces the current scheduling characteristics of this with the new scheduling characteristics.

Parameters
cost—The proposed cost used to determine when any particular object exceeds cost. When null, the default value is a new instance of RelativeTime(0, 0).
deadline—The proposed deadline. When null, the default value is a new instance of RelativeTime(Long.MAX_VALUE, 999999).

Throws
IllegalRealtimeArgumentException—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 values are incompatible with the scheduler for any of the schedulables which are presently using this parameter object.

IllegalAssignmentError—when cost or deadline cannot be stored in this.

Returns
false. Aperiodic parameters never yield a feasible system. (Subclasses of AperiodicParameters, such as SporadicParameters, need not return false.)

Deprecated as of RTSJ 2.0

B.2.2.3 ArrivalTimeQueueOverflowException

public class ArrivalTimeQueueOverflowException

The following elements of ArrivalTimeQueueOverflowException are deprecated. The required elements are documented in Section 16.2.2.2 above.

B.2.2.3.1 Constructors

ArrivalTimeQueueOverflowException(String)

Signature

990 RTSJ 2.0 (Draft 66)
public ArrivalTimeQueueOverflowException(String description)

Description

A descriptive constructor for ArrivalTimeQueueOverflowException.

Deprecated since RTSJ 2.0; application code should use get() instead.

Parameters

description—A description of the exception.

B.2.2.4 AsyncEvent

public class AsyncEvent

The following elements of AsyncEvent are deprecated. The required elements are documented in Section 8.3.2.4 above.

B.2.2.4.1 Methods

handledBy(AsyncEventHandler)

Signature

public boolean handledBy(AsyncEventHandler handler)

Description

Replaced by AsyncBaseEvent.handledBy(AsyncBaseEventHandler)

Parameters

handler—For querying its association with this.

Returns

true when and only when handler is associated with this event.

Deprecated since RTSJ 2.0
addHandler(AsyncEventHandler)

Signature
   public void
   addHandler(AsyncEventHandler handler)

Description
   Replaced by AsyncBaseEvent.addHandler(AsyncBaseEventHandler)

Parameters
   handler—For adding to this.
   Deprecated since RTSJ 2.0

setHandler(AsyncEventHandler)

Signature
   public void
   setHandler(AsyncEventHandler handler)

Description
   Replaced by AsyncBaseEvent.setHandler(AsyncBaseEventHandler)

Parameters
   handler—For becoming the sole handler for this.
   Deprecated since RTSJ 2.0

removeHandler(AsyncEventHandler)

Signature
   public void
   removeHandler(AsyncEventHandler handler)

Description
   Replaced by AsyncBaseEvent.removeHandler(AsyncBaseEventHandler)

Parameters
   handler—For removal.
   Deprecated since RTSJ 2.0
**bindTo(String)**

*Signature*

```java
public void
bindTo(String happening)
```

*Description*

Binds this to an external event, a *happening*. The meaningful values of *happening* are implementation dependent. This instance of `AsyncEvent` is considered to have occurred whenever the happening is triggered. More than one happening can be bound to the same `AsyncEvent`. However, binding a happening to an event has no effect when the happening is already bound to the event.

When an event, which is declared in a scoped memory area, is bound to an external happening, the reference count of that scoped memory area is incremented (as if there is an external realtime thread accessing the area). The reference count is decremented when the event is unbound from the happening.

*Parameters*

- `happening`—An implementation-dependent value that binds this instance of `AsyncEvent` to a happening.

*Throws*

- `UnknownHappeningException`—when the String value is not supported by the implementation.
- `IllegalRealtimeArgumentException`—when `happening` is null.

*Deprecated* since RTSJ 2.0

**unbindTo(String)**

*Signature*

```java
public void
unbindTo(String happening)
```

*Description*

Removes a binding to an external event, a *happening*. The meaningful values of *happening* are implementation dependent. When the associated event is declared in a scoped memory area, the reference count for the memory area is decremented.

*Parameters*

- `happening`—An implementation-dependent value representing some external event to which this instance of `AsyncEvent` is bound.
B Deprecated APIs

Throws

- `UnknownHappeningException`—when this instance of `AsyncEvent` is not bound to the given `happening` or the given `happening` is not supported by the implementation.
- `IllegalRealtimeArgumentException`—when `happening` is null.

 Deprecated since RTSJ 2.0

B.2.2.5 AsyncEventHandler

```java
public class AsyncEventHandler
```

The following elements of AsyncEventHandler are deprecated. The required elements are documented in Section 8.3.2.5 above.

B.2.2.5.1 Constructors

```java
AsyncEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, ProcessingGroupParameters, boolean, Runnable)
```

Signature

```java
public
AsyncEventHandler(SchedulingParameters scheduling,
ReleaseParameters release,
MemoryParameters memory,
MemoryArea area,
ProcessingGroupParameters group,
boolean nonheap,
Runnable logic)
```

Description

Calling this constructor is equivalent to calling `AsyncEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, ReleaseRunner, Runnable)` with arguments `(scheduling, release, memory.clone(!nonheap), null, logic)`.

994 RTSJ 2.0 (Draft 66)
AsyncEventHandler

**Deprecated** in version 2.0.

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

**Signature**

public

AsyncEventHandler(SchedulingParameters scheduling,
ReleaseParameters release,
MemoryParameters memory,
MemoryArea area,
ProcessingGroupParameters group,
Runnable logic)

**Description**

Calling this constructor is equivalent to calling AsyncEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, ReleaseRunner, Runnable) with arguments (scheduling, release, memory, area, group, null, logic).

**Deprecated** in version 2.0.

AsyncEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, ProcessingGroupParameters, boolean)

**Signature**

public

AsyncEventHandler(SchedulingParameters scheduling,
ReleaseParameters release,
MemoryParameters memory,
MemoryArea area,
ProcessingGroupParameters group,
boolean nonheap)

**Description**

RTSJ 2.0 (Draft 66)
Calling this constructor is equivalent to calling \texttt{AsyncEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, ReleaseRunner, Runnable)} with arguments (scheduling, release, memory, area, null, null).

\textbf{Deprecated} in version 2.0.

\textbf{AsyncEventHandler(boolean, Runnable)}

\textit{Signature}

\begin{verbatim}
public
AsyncEventHandler(boolean nonheap,
    Runnable logic)
\end{verbatim}

\textit{Description}

Calling this constructor is equivalent to calling \texttt{AsyncEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, ReleaseRunner, Runnable)} with arguments (null, null, new MemoryParameters(0), null, null, logic).

\textbf{Deprecated} in version 2.0.

\textbf{AsyncEventHandler(boolean)}

\textit{Signature}

\begin{verbatim}
public
AsyncEventHandler(boolean nonheap)
\end{verbatim}

\textit{Description}

Calling this constructor is equivalent to calling \texttt{AsyncEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, ReleaseRunner, Runnable)} with arguments (null, null, new MemoryParameters(!nonheap), null, null, null).

\textbf{Deprecated} in version 2.0.
B.2.2.5.2 Methods

getAndIncrementPendingFireCount

Signature
protected int
getAndIncrementPendingFireCount()

Description
This is an accessor method for fireCount. This method atomically increments, by one, the value of fireCount and returns the value from before the increment.

Calling this method is effectively the same as firing an event that is associated with this handler. When called from outside the handler’s control flow, call it is effectively the same as firing an event that is associated with this handler, except that it does not constitute a release event.

Throws
MITViolationException—when this AEH is controlled by sporadic scheduling parameters under the base scheduler, the parameters specify the mitViolationExcept policy, and this method would introduce a release that would violate the specified minimum interarrival time.

ArrivalTimeQueueOverflowException—when this AEH is controlled by aperiodic scheduling parameters under the base scheduler, the release parameters specify the arrivalTimeQueueOverflowExcept policy, and this method would cause the arrival time queue to overflow.

Returns
the value held by fireCount prior to incrementing it by one.

Deprecated as of RTSJ 2.0 Use ae.fire()

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

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

Description
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.

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 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.
release—A reference to the ReleaseParameters 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.)
memoryParameters—A reference to the MemoryParameters 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.)
group—A reference to the ProcessingGroupParameters 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.)

Throws
IllegalRealtimeArgumentException—when scheduler is null or the parameter values are not compatible with scheduler. Also thrown when this schedulable may not use the 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.
RealtimeSecurityException—when the caller is not permitted to set the scheduler for this schedulable.

Deprecated since RTSJ 2.0

getProcessingGroupParameters

Signature

998                RTSJ 2.0 (Draft 66)
public javax.realtime.ProcessingGroupParameters
getProcessingGroupParameters()

Description
Gets a reference to the ProcessingGroupParameters object for this schedulable.

Returns
A reference to the current ProcessingGroupParameters object.

Deprecated since RTSJ 2.0

setProcessingGroupParameters(ProcessingGroupParameters)

Signature
public void
setProcessingGroupParameters(ProcessingGroupParameters group)

Description
Sets the ProcessingGroupParameters 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.

Parameters
group—A ProcessingGroupParameters object which will 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.)

Throws
IllegalRealtimeArgumentException—when group is not compatible with the
scheduler for this schedulable object. Also when this schedulable may not use
the 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.

Deprecated since RTSJ 2.0

RTSJ 2.0 (Draft 66)
**addToFeasibility**

**Signature**

```java
public boolean addToFeasibility()
```

**Description**

Informs the scheduler and cooperating facilities that this instance of `Schedulable` should be considered in feasibility analysis until further notified.

When the object is already included in the feasibility set, does nothing.

**Returns**

- `true`, when the resulting system is feasible. False, when not.

**Deprecated** as of RTSJ 2.0 The framework for feasibility analysis is inadequate

---

**addIfFeasible**

**Signature**

```java
public boolean addIfFeasible()
```

**Description**

This method first performs a feasibility analysis with `this` added to the system. When the resulting system is feasible, informs the scheduler and cooperating facilities that this instance of `Schedulable` should be considered in feasibility analysis until further notified. When the analysis shows that the system including `this` would not be feasible, this method does not admit `this` to the feasibility set.

When the object is already included in the feasibility set, does nothing.

**Returns**

- `true` when inclusion of `this` in the feasibility set yields a feasible system, and false otherwise. When `true` is returned then `this` is known to be in the feasibility set. When false is returned, `this` was not added to the feasibility set, but it may already have been present.

**Deprecated** as of RTSJ 2.0 The framework for feasibility analysis is inadequate
removeFromFeasibility

Signature

public boolean
removeFromFeasibility()

Description

Informs the scheduler and cooperating facilities that this instance of Schedulable should not be considered in feasibility analysis until it is further notified.

Returns

ture when the removal was successful. false when the schedulable cannot be removed from the scheduler’s feasibility set; e.g., the schedulable is not part of the scheduler’s feasibility set.

Deprecated as of RTSJ 2.0 The framework for feasibility analysis is inadequate

setIfFeasible(ReleaseParameters, MemoryParameters)

Signature

public boolean
setIfFeasible(ReleaseParameters release,
MemoryParameters memory)

Description

This method first performs a feasibility analysis using the proposed parameter objects as replacements for the current parameters of this. When the resulting system is feasible, this method replaces the current parameters of this with the proposed ones.

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.

This method does not require that the schedulable be in the feasibility set before it is called. When it is not initially a member of the feasibility set it will be added when the resulting system is feasible.

Parameters

release—The release parameters to use. 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.)
memory—The memory parameters to use. 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.)

Throws

IllegalRealtimeArgumentException—when the parameter values are not compatible with the schedulable’s scheduler. Also when this schedulable may not use the heap and any of the specified parameter objects are located in heap memory.

IllegalAssignmentError—when this cannot hold references to the specified parameter objects, or the parameter objects cannot hold a reference to this.

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

Returns

true, when the resulting system is feasible and the changes are made. False, when the resulting system is not feasible and no changes are made.

Deprecated as of RTSJ 2.0 The framework for feasibility analysis is inadequate.

setIfFeasible(SchedulingParameters, ReleaseParameters, MemoryParameters)

Signature

public boolean setIfFeasible(SchedulingParameters scheduling,
ReleaseParameters release,
MemoryParameters memory)

Description

This method first performs a feasibility analysis using the proposed parameter objects as replacements for the current parameters of this. When the resulting system is feasible, this method replaces the current parameters of this with the proposed ones.

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.

This method does not require that the schedulable be in the feasibility set before it is called. When it is not initially a member of the feasibility set it will be added when the resulting system is feasible.
Params

**scheduling**—The scheduling parameters to use. 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`.)

**release**—The release parameters to use. When `null`, the default value is governed by the associated scheduler (a new object is created when the default value is `null`). (See `PriorityScheduler`.)

**memory**—The memory parameters to use. When `null`, the default value is governed by the associated scheduler (a new object is created when the default value is `null`). (See `PriorityScheduler`.)

**Throws**

- `IllegalRealtimeArgumentException`—when the parameter values are not compatible with the schedulable’s scheduler. Also when this schedulable may not use the heap and any of the specified parameter objects are located in heap memory.
- `IllegalAssignmentError`—when `this` cannot hold references to the specified parameter objects, or the parameter objects cannot hold a reference to `this`.
- `IllegalThreadStateException`—when the schedulable’s scheduler prohibits this parameter change at this time due to the state of the schedulable.

**Returns**

- `true`, when the resulting system is feasible and the changes are made. `false`, when the resulting system is not feasible and no changes are made.

**Deprecated** as of RTSJ 2.0 The framework for feasibility analysis is inadequate

```java
setIfFeasible(ReleaseParameters, MemoryParameters, ProcessingGroupParameters)
```

**Signature**

```java
public boolean setIfFeasible(ReleaseParameters release, MemoryParameters memory, ProcessingGroupParameters group)
```

**Description**

This method first performs a feasibility analysis using the proposed parameter objects as replacements for the current parameters of `this`. When the resulting system is feasible, this method replaces the current parameters of `this` with the proposed ones.
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.

This method does not require that the schedulable be in the feasibility set before it is called. When it is not initially a member of the feasibility set it will be added when the resulting system is feasible.

**Parameters**

- **release**—The release parameters to use. 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`.)
- **memory**—The memory parameters to use. 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`.)
- **group**—The processing group parameters to use. 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`.)

**Throws**

- `IllegalRealtimeArgumentException`—when the parameter values are not compatible with the schedulable’s scheduler. Also when this schedulable may not use the heap and any of the specified parameter objects are located in heap memory.
- `IllegalAssignmentError`—when `this` cannot hold references to the specified parameter objects, or the parameter objects cannot hold a reference to `this`.
- `IllegalThreadStateException`—when the schedulable’s scheduler prohibits this parameter change at this time due to the state of the schedulable.

**Returns**

- `true`, when the resulting system is feasible and the changes are made. False, when the resulting system is not feasible and no changes are made.

**Deprecated** as of RTSJ 2.0 The framework for feasibility analysis is inadequate

```java
setIfFeasible(SchedulingParameters, ReleaseParameters, MemoryParameters, ProcessingGroupParameters)
```

**Signature**

```java
public boolean setIfFeasible(SchedulingParameters scheduling,
```
AsyncEventHandler

```
ReleaseParameters release,
MemoryParameters memory,
ProcessingGroupParameters group)
```

**Description**

This method first performs a feasibility analysis using the proposed parameter objects as replacements for the current parameters of `this`. When the resulting system is feasible, this method replaces the current parameters of `this` with the proposed ones.

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.

This method does not require that the schedulable be in the feasibility set before it is called. When it is not initially a member of the feasibility set it will be added when the resulting system is feasible.

**Parameters**

- **scheduling**—The scheduling parameters to use. 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`.)
- **release**—The release parameters to use. 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`.)
- **memory**—The memory parameters to use. 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`.)
- **group**—The processing group parameters to use. 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`.)

**Throws**

- `IllegalRealtimeArgumentException`—when the parameter values are not compatible with the schedulable's scheduler. Also when this schedulable may not use the heap and any of the specified parameter objects are located in heap memory.
- `IllegalAssignmentError`—when `this` cannot hold references to the specified parameter objects, or the parameter objects cannot hold a reference to `this`.
- `IllegalThreadStateException`—when the schedulable’s scheduler prohibits this parameter change at this time due to the state of the schedulable.

**Returns**
true, when the resulting system is feasible and the changes are made. False, when the resulting system is not feasible and no changes are made.

**Deprecated** as of RTSJ 2.0 The framework for feasibility analysis is inadequate

### setReleaseParametersIfFeasible(ReleaseParameters)

**Signature**

```java
public boolean setReleaseParametersIfFeasible(ReleaseParameters release)
```

**Description**

This method first performs a feasibility analysis using the proposed parameter object as replacement for the current parameter of this. When the resulting system is feasible, this method replaces the current parameter of this with the proposed one.

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.

This method does not require that the schedulable be in the feasibility set before it is called. When it is not initially a member of the feasibility set it will be added when the resulting system is feasible.

**Parameters**

- `release`—The release parameters to use. 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`.)

**Throws**

- `IllegalRealtimeArgumentException`—when the parameter value is not compatible with the schedulable’s scheduler. Also when this schedulable may not use the heap and the specified parameter object is located in heap memory.
- `IllegalAssignmentError`—when `this` cannot hold a reference to the specified parameter object, or the parameter object cannot hold a reference to `this`.
- `IllegalThreadStateException`—when the schedulable’s scheduler prohibits the changing of the release parameter at this time due to the state of the schedulable.

**Returns**

true, when the resulting system is feasible and the changes are made. False, when the resulting system is not feasible and no changes are made.
AsyncEventHandler  

**Deprecated** as of RTSJ 2.0 The framework for feasibility analysis is inadequate

**setProcessingGroupParametersIfFeasible**  
(ProcessingGroupParameters)

**Signature**

```java
public boolean setProcessingGroupParametersIfFeasible(ProcessingGroupParameters group)
```

**Description**

This method first performs a feasibility analysis using the proposed parameter object as replacement for the current parameter of this. When the resulting system is feasible, this method replaces the current parameter of this with the proposed one.

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.

This method does not require that the schedulable be in the feasibility set before it is called. When it is not initially a member of the feasibility set it will be added when the resulting system is feasible.

**Parameters**

- **group**—The processing group parameters to use. 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.)

**Throws**

- **IllegalRealtimeArgumentException**—when the parameter value is not compatible with the schedulable’s scheduler. Also when this schedulable may not use the heap and the specified parameter object is located in heap memory.
- **IllegalAssignmentError**—when this cannot hold a reference to the specified parameter object, or the parameter object 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.

**Returns**

true, when the resulting system is feasible and the changes are made. False, when the resulting system is not feasible and no changes are made.
Deprecated as of RTSJ 2.0 The framework for feasibility analysis is inadequate.

**setIfFeasible(ReleaseParameters, ProcessingGroupParameters)**

**Signature**

```java
public boolean setIfFeasible(ReleaseParameters release,
                            ProcessingGroupParameters group)
```

**Description**

This method first performs a feasibility analysis using the proposed parameter objects as replacements for the current parameters of this. When the resulting system is feasible, this method replaces the current parameters of this with the proposed ones.

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.

This method does not require that the schedulable be in the feasibility set before it is called. When it is not initially a member of the feasibility set it will be added when the resulting system is feasible.

**Parameters**

- `release`—The release parameters to use. 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`.)

- `group`—The processing group parameters to use. 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`.)

**Throws**

- `IllegalRealtimeArgumentException`—when the parameter values are not compatible with the schedulable’s scheduler. Also when this schedulable may not use the heap and any of the specified parameter objects are located in heap memory.

- `IllegalAssignmentError`—when this cannot hold references to the specified parameter objects, or the parameter objects cannot hold a reference to this.

- `IllegalThreadStateException`—when the schedulable’s scheduler prohibits this parameter change at this time due to the state of the schedulable.
Returns
ture, when the resulting system is feasible and the changes are made. False, when
the resulting system is not feasible and no changes are made.

Deprecated as of RTSJ 2.0 The framework for feasibility analysis is inadequate

setMemoryParametersIfFeasible(MemoryParameters)

Signature
public boolean
setMemoryParametersIfFeasible(MemoryParameters memory)

Description
This method first performs a feasibility analysis using the proposed parameter
object as replacement for the current parameter of this. When the resulting
system is feasible, this method replaces the current parameter of this with the
proposed one.

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.

This method does not require that the schedulable be in the feasibility set
before it is called. When it is not initially a member of the feasibility set it will
be added when the resulting system is feasible.

Parameters
memory—The memory parameters to use. 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.)

Throws
IllegalRealtimeArgumentException—when the parameter value is not compati-
ble with the schedulable’s scheduler. Also when this schedulable may not use
the heap and the specified parameter object is located in heap memory.

IllegalAssignmentError—when this cannot hold a reference to the specified
parameter object, or the parameter object cannot hold a reference to this.

IllegalThreadStateException—when the schedulable’s scheduler prohibits the
changing of the memory parameter at this time due to the state of the schedul-
able.

Returns

RTSJ 2.0 (Draft 66) 1009
true, when the resulting system is feasible and the changes are made. False, when
the resulting system is not feasible and no changes are made.

**Deprecated** as of RTSJ 2.0 The framework for feasibility analysis is inadequate

**setSchedulingParametersIfFeasible(SchedulingParameters)**

**Signature**

```java
public boolean setSchedulingParametersIfFeasible(SchedulingParameters scheduling)
```

**Description**

This method first performs a feasibility analysis using the proposed parameter
object as replacement for the current parameter of this. When the resulting
system is feasible, this method replaces the current parameter of this with the
proposed one.

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.

This method does not require that the schedulable be in the feasibility set
before it is called. When it is not initially a member of the feasibility set it will
be added when the resulting system is feasible.

**Parameters**

- **scheduling**—The scheduling parameters to use. 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**.)

**Throws**

- **IllegalRealtimeArgumentException**—when the parameter value is not compati-
ble with the schedulable’s scheduler. Also when this schedulable may not use
the heap and the specified parameter object is located in heap memory.

- **IllegalAssignmentError**—when this cannot hold a reference to the specified
parameter object, or the parameter object 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.

**Returns**

true, when the resulting system is feasible and the changes are made. False, when
the resulting system is not feasible and no changes are made.
AsynchronouslyInterruptedException

**Deprecated** as of RTSJ 2.0 The framework for feasibility analysis is inadequate

### B.2.2.6 AsynchronouslyInterruptedException

public class AsynchronouslyInterruptedException

*Inheritance*
- java.lang.Object
- java.lang.Throwable
- java.lang.Exception
  - java.lang.InterruptedException
  - AsynchronouslyInterruptedException

*Description*
A special exception that is thrown in response to an attempt to asynchronously transfer the locus of control of a schedulable.

A **Scheduled** that is executing a method or constructor, which is declared with an `AsynchronouslyInterruptedException` 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 **Scheduled** 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 **Scheduled**, `s`, when the `s.interrupt()` method is called or the `fire` method is called of an AIE for which `s` has a `doInterruptible` method call in progress.

When an asynchronous interrupt is generated when the target **Scheduled** is executing within an ATC-deferred section, the asynchronous interrupt becomes pending. A pending asynchronous interrupt is delivered when the target **Scheduled** 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 `java.lang.Thread.interrupted()`.

When **Scheduled**.interrupt or `AsynchronouslyInterruptedException.fire()` is called, the `AsynchronouslyInterruptedException` is compared against any currently pending `AsynchronouslyInterruptedException` on the **Scheduled**. 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 `AsynchronouslyInterruptedException` 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.

`Schedulable.interrupt()` generates the 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-instance of `Schedulable`.

Other sources (e.g., `AsynchronouslyInterruptedException.fire()` and `Timed`) will generate specific instances of `AsynchronouslyInterruptedException` which applications can identify and thus limit propagation.

**Deprecated** in RTSJ 2.0; moved to package `javax.realtime.control`

### B.2.2.6.1 Constructors

**AsynchronouslyInterruptedException**

**Signature**

```java
public AsynchronouslyInterruptedException()
```

**Description**

Creates an instance of `AsynchronouslyInterruptedException`.

**AsynchronouslyInterruptedException(String)**

**Signature**

```java
1012 RTSJ 2.0 (Draft 66)
```
AsynchronouslyInterruptedException

class AsynchronouslyInterruptedException

public
    AsynchronouslyInterruptedException(String message)

Description
    Creates an instance of AsynchronouslyInterruptedException.

Parameters
    message—A message to identify this instance.

B.2.2.6.2 Methods

generic

Signature
    public static javax.realtime.AsynchronouslyInterruptedException
        getGeneric()
        throws IllegalTaskStateException

Description
    Gets the singleton system generic AsynchronouslyInterruptedException that
    is generated when Schedulable.interrupt() is invoked.

Throws
    IllegalTaskStateException—when the current thread context is not an instance
    of Schedulable.

Returns
    the generic AsynchronouslyInterruptedException.

enable

Signature
    public boolean
        enable()

Description
    Enables 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.
Returns

true, when this was disabled before the method was called and the call was invoked whilst the associated doInterruptible was in progress, and false otherwise.

disable

Signature

public synchronized boolean disable()

Description

Disables the throwing of this exception. When the fire 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.

Returns

true, when this was enabled before the method was called and the call was invoked with the associated doInterruptible in progress, and false otherwise.

isEnabled

Signature

public boolean isEnabled()

Description

Queries 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.

Returns

true, when this is enabled and the method call was invoked in the context of the associated doInterruptible, and false otherwise.
fire

Signature

public boolean
    fire()

Description
Generates 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) 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.

Behaves as if Thread.interrupt() were called on the task currently operating within this exception’s doInterruptible.

Returns
true, when this is not disabled and it has an invocation of a doInterruptible in progress and there is no outstanding fire request, and false otherwise.

doInterruptible(Interruptible)

Signature

public boolean
    doInterruptible(Interruptible logic)

Description
Executes the run() method of the given Interruptible. This method may be on the stack in exactly one Schedulable 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 the given Interruptible is always entered with the exception in the enabled state, but that state can be modified with enable() and disable(), and the state can be observed with isEnabled().

This AIE is cleared on return from doInterruptible.

Parameters
logic—An instance of an Interruptible whose run() method will be called.

Throws
IllegalTaskStateException—when called on the generic AsynchronouslyInterruptedException.

RTSJ 2.0 (Draft 66)
IllegalRealtimeArgumentException—when logic is null.

Returns
true, when the method call completed normally, and false, when another call to doInterruptible has not completed.

Available since RTSJ 2.0 no longer throws an exception when called from a Java thread.

clear

Signature
public boolean clear()

Description
Atomically checks whether or not this is pending on the currently executing schedulable, and when so, makes it non-pending.
This method may be called at any time, and in particular need not be called in a try or catch block.

Returns
true, when this was pending, and false, when this was not pending.

Available since RTSJ 1.0.1

Available since RTSJ 2.0 no longer throws an exception when called from a task that is not an instance of Schedulable.

throwPending

Signature
public static void throwPending()
throws AsynchronouslyInterruptedException

Description
Causes a pending AsynchronouslyInterruptedException to be thrown as a synchronous exception in an AI-deferred region if one exists.

Throws

1016 RTSJ 2.0 (Draft 66)
**AsynchronouslyInterruptedException**—if an AIE is pending.

**Available since** RTSJ 2.0

---

**fillInStackTrace**

**Signature**

```java
public java.lang.Throwable
fillInStackTrace()
```

**Description**

Does nothing, since no stacktrace is kept.

**Returns**

this instance.

---

**setStackTrace(StackTraceElement)**

**Signature**

```java
public void
setStackTrace(java.lang.StackTraceElement[] new_stackTrace)
throws NullPointerException
```

**Description**

Does nothing, since no stacktrace is kept.

---

**getStackTrace**

**Signature**

```java
public java.lang.StackTraceElement[]
getStackTrace()
```

**Description**

No stacktrace is kept, so none can be returned.

**Returns**

an empty array.
B Deprecated APIs

printStackTrace

Signature
public void
printStackTrace()

Description
No stacktrace is kept, so a message to that effect is printed.

printStackTrace(PrintStream)

Signature
public void
printStackTrace(PrintStream stream)

Description
No stacktrace is kept, so a message to that effect is printed.

Parameters
stream—A PrintStream for printing

printStackTrace(PrintWriter)

Signature
public void
printStackTrace(PrintWriter writer)

Description
No stacktrace is kept, so a message to that effect is printed.

Parameters
writer—A PrintWriter for printing

B.2.2.7 BoundAsyncEventHandler

public class BoundAsyncEventHandler

The following elements of BoundAsyncEventHandler are deprecated. The required elements are documented in Section 8.3.2.11 above.
B.2.2.7.1 Constructors

BoundAsyncEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, ProcessingGroupParameters, boolean, Runnable)

*Signature*

```java
class BoundAsyncEventHandler {
    public BoundAsyncEventHandler(SchedulingParameters scheduling,
                                    ReleaseParameters release,
                                    MemoryParameters memory,
                                    MemoryArea area,
                                    ProcessingGroupParameters group,
                                    boolean nonheap,
                                    Runnable logic)
}
```

*Description*

Creates an instance of `BoundAsyncEventHandler` with the specified parameters. The newly-created handler inherits the affinity of its creator.

*Deprecated* since RTSJ 2.0

*Parameters*

- **scheduling**—A `SchedulingParameters` object which will be associated with the constructed instance. When null, and the creator is not an instance of `Schedulable`, a `SchedulingParameters` object is created which has the default `SchedulingParameters` for the scheduler associated with the current thread. When null, and the creator is an instance of `Schedulable`, the `SchedulingParameters` are inherited from the current schedulable (a new `SchedulingParameters` object is cloned).
- **release**—A `ReleaseParameters` object which will be associated with the constructed instance. When null, this will have default `ReleaseParameters` for the BAEH’s scheduler.
- **memory**—A `MemoryParameters` object which will be associated with the constructed instance. When null, this will have no `MemoryParameters`.
- **area**—The `MemoryArea` for this. When null, the memory area will be that of the current thread/schedulable.
group—A `ProcessingGroupParameters` object which will be associated with the constructed instance. When null, this will not be associated with any processing group.

logic—The `Runnable` object whose `run()` method is executed by `AsyncEventHandler.handleAsyncEvent()`. When null, the default `handleAsyncEvent()` method invokes nothing.

nonheap—When `true`, the code executed by this handler may not reference or store objects in `HeapMemory`; otherwise, that code may do so. When `true` and the current handler tries to reference or store objects in `HeapMemory` or enter the `HeapMemory` an `IllegalRealtimeArgumentException` is thrown.

`Throws`

- `IllegalRealtimeArgumentException`—when `nonheap` is `false` and `logic`, any parameter object, or this is in heap memory. Also when `nonheap` is `true` and `area` is heap memory.

- `IllegalAssignmentError`—when the new `AsyncEventHandler` instance cannot hold a reference to non-null values of `scheduling release Memory` and `group`, or when those parameters cannot hold a reference to the new `AsyncEventHandler`. Also when the new `AsyncEventHandler` instance cannot hold a reference to non-null values of `area` and `logic`.

### B.2.2.8 DuplicateFilterException

```java
class DuplicateFilterException
```

**Inheritance**

- `java.lang.Object`
- `java.lang.Throwable`
- `java.lang.Exception`
- `DuplicateFilterException`

**Description**

`PhysicalMemoryManager` 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.

**Deprecated** since RTSJ 2.0

### B.2.2.8.1 Constructors

1020 RTSJ 2.0 (Draft 66)
DuplicateFilterException(String)

Signature
   public
   DuplicateFilterException(String description)

Description
   A descriptive constructor for DuplicateFilterException.

Parameters
   description—Description of the error.

DuplicateFilterException

Signature
   public
   DuplicateFilterException()

Description
   A constructor for DuplicateFilterException.

B.2.2.9  HighResolutionTime

public abstract class HighResolutionTime<T extends HighResolutionTime<T>>

The following elements of HighResolutionTime are deprecated. The required elements are documented in Section 9.3.1.2 above.

B.2.2.9.1 Methods

absolute(Clock, AbsoluteTime)

Signature
public abstract javax.realtime.AbsoluteTime
absolute(Clock clock,
    AbsoluteTime dest)

Description
Equivalent to and superseded by absolute(Chronograph, AbsoluteTime). When dest is not null, the result is placed in it and returned. Otherwise, a new object is allocated for the result. The clock association of the result is the clock passed as a parameter. See the subclass comments for more specific information.

Parameters
clock—The instance of Clock used to convert the time of this into absolute time, and the new clock association for the result.
dest—when dest is not null, the result is placed it and returned.

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 clock parameter.

Deprecated since version 2.0

absolute(Clock)

Signature
public abstract javax.realtime.AbsoluteTime
absolute(Clock clock)

Description
Equivalent to and superseded by absolute(Chronograph).

Parameters
clock—The instance of Clock used to convert the time of this into absolute time, and the new clock association for the result.

Returns
the AbsoluteTime conversion in a newly allocated object, associated with the clock parameter.

Deprecated since version 2.0
relative(Clock, RelativeTime)

**Signature**

```java
public abstract javax.realtime.RelativeTime relative(Clock clock,
                                                   RelativeTime dest)
```

**Description**

Equivalent to and superseded by `relative(Chronograph, RelativeTime)`

**Parameters**
- **clock**—The instance of `Clock` 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 in it and returned.

**Returns**
- the `RelativeTime` conversion in `dest` when `dest` is not `null`, otherwise the result is returned in a newly allocated object.

**Deprecated** since version 2.0

relative(Clock)

**Signature**

```java
public abstract javax.realtime.RelativeTime relative(Clock clock)
```

**Description**

Equivalent to and superseded by `relative(Chronograph)`

**Parameters**
- **clock**—The instance of `Clock` used to convert the time of this into relative time, and the new clock association for the result.

**Returns**
- the `RelativeTime` conversion in a newly allocated object, associated with the `clock` parameter.

**Deprecated** since version 2.0
B Deprecated APIs

B.2.2.10 *IllegalAssignmentError*

public class IllegalAssignmentError

The following elements of IllegalAssignmentError are deprecated. The required elements are documented in Section 16.2.2.8 above.

B.2.2.10.1 Constructors

IllegalAssignmentError(String)

*Signature*

```
public
IllegalAssignmentError(String description)
```

*Description*

A descriptive constructor for IllegalAssignmentError.

*Deprecated* since RTSJ 2.0; application code should use `get()` instead.

*Parameters*

description—The reason for throwing the error.

B.2.2.11 *ImmortalPhysicalMemory*

public class ImmortalPhysicalMemory

*Inheritance*

java.lang.Object

MemoryArea

PerennialMemory

ImmortalPhysicalMemory

*Description*

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 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.

Available since RTSJ 2.0 extends PerennialMemory instead of MemoryArea directly.

Deprecated since RTSJ 2.0

B.2.2.11.1 Constructors

ImmortalPhysicalMemory(Object, long, long, Runnable)

Signature
public
ImmortalPhysicalMemory(Object type,
    long base,
    long size,
    Runnable logic)

Description
Creates an instance with the given parameters.

Parameters
type—An instance of Object or an array of objects 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() is called. When logic is null, logic must be supplied when the memory area is entered.

**Throws**

RealtimeSecurityException—when the application doesn’t have permissions to access physical memory or the given type of memory.

OffsetOutOfBoundsException—when the base address is invalid.

SizeOutOfBoundsException—when size extends into an invalid range of memory.

UnsupportedPhysicalMemoryException—when the underlying hardware does not support the given type, or when no matching PhysicalMemoryTypeFilter has been registered with the PhysicalMemoryManager.

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 negative. IllegalArgumentException may also be when base plus size would be greater than the maximum physical address supported by the processor.

MemoryInUseException—when the specified memory is already in use.

OutOfRangeMemoryStaticError—when there is insufficient memory for the ImmortalPhysicalMemory object or for its allocation area in its backing store.

IllegalAssignmentError—when storing logic in this would violate the assignment rules.

**ImmortalPhysicalMemory(Object, long, SizeEstimator, Runnable)**

**Signature**

```
public
 ImmortalPhysicalMemory(Object type,
             long base,
             SizeEstimator size,
             Runnable logic)
```

**Description**

Creates an instance with the given parameters.

**Parameters**

1026       RTSJ 2.0 (Draft 66)
type—An instance of Object or an array of objects 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() is called. When logic is null, logic must be supplied when the memory area is entered.

Throws

RealtimeSecurityException—when the application doesn’t have permissions to access physical memory or the given type of memory.

OffsetOutOfBoundsException—when the base address is invalid.

SizeOutOfBoundsException—when the size estimate from size extends into an invalid range of memory.

UnsupportedPhysicalMemoryException—when the underlying hardware does not support the given type, or when no matching PhysicalMemoryTypeFilter has been registered with the PhysicalMemoryManager.

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. IllegalArgumentException may also be when base plus the size indicated by size would be greater than the maximum physical address supported by the processor.

MemoryInUseException—when the specified memory is already in use.

OutOfMemoryStaticError—when there is insufficient memory for the ImmortalPhysicalMemory object or for the backing memory.

IllegalAssignmentError—when storing logic in this would violate the assignment rules.

ImmortalPhysicalMemory(Object, long, Runnable)

Signature

public
ImmortalPhysicalMemory(Object type,
long size,
Runnable logic)

Description
Creates an instance with the given parameters.

Parameters
type—An instance of Object or an array of objects 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() is called. When logic is null, logic must be supplied when the memory area is entered.

Throws
RealtimeSecurityException—when the application doesn’t have permissions to access physical memory or the given type of memory.

SizeOutOfBoundsException—when size extends into an invalid range of memory.

UnsupportedPhysicalMemoryException—when the underlying hardware does not support the given type, or when no matching PhysicalMemoryTypeFilter has been registered with the PhysicalMemoryManager.

IllegalArgumentException—when size is negative.

MemoryTypeConflictException—when the specified base does not point to memory that matches the requested type, or when type specifies incompatible memory attributes.

OutOfMemoryStaticError—when there is insufficient memory for the ImmortalPhysicalMemory object or for its allocation area in its backing store.

IllegalAssignmentError—when storing logic in this would violate the assignment rules.

ImmortalPhysicalMemory(Object, SizeEstimator, Runnable)

Signature

1028 RTSJ 2.0 (Draft 66)
public
ImmortalPhysicalMemory(Object type,
    SizeEstimator size,
    Runnable logic)

Description

Creates an instance with the given parameters.

Parameters

- **type**—An instance of Object or an array of objects 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() is called. When logic is null, logic must be supplied when the memory area is entered.

Throws

- RealtimeSecurityException—when the application doesn’t have permissions to access physical memory or the given type of memory.

- SizeOutOfBoundsException—when the size extends into an invalid range of memory.

- UnsupportedPhysicalMemoryException—when the underlying hardware does not support the given type, or when no matching PhysicalMemoryTypeFilter has been registered with the PhysicalMemoryManager.

- IllegalArgumentException—when size is null, or size.getEstimate() is negative.

- MemoryTypeConflictException—when type specifies incompatible memory attributes.

- OutOfMemoryStaticError—when there is insufficient memory for the ImmortalPhysicalMemory object or for its allocation area in its backing store.

- IllegalAssignmentError—when storing logic in this would violate the assignment rules.

ImmortalPhysicalMemory(Object, long, long)
**Signature**

```java
public ImmortalPhysicalMemory(Object type,
                               long base,
                               long size)
```

**Description**

Creates an instance with the given parameters.

**Parameters**

- `type`—An instance of `Object` or an array of objects 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**

- `RealtimeSecurityException`—when the application doesn’t have permissions to access physical memory or the given range of memory.
- `OffsetOutOfBoundsException`—when the `base` address is invalid.
- `SizeOutOfBoundsException`—when the `size` extends into an invalid range of memory.
- `UnsupportedPhysicalMemoryException`—when the underlying hardware does not support the given type, or when no matching `PhysicalMemoryTypeFilter` has been registered with the `PhysicalMemoryManager`.
- `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. `IllegalArgumentException` may also be when `base` plus `size` would be greater than the maximum physical address supported by the processor.
- `MemoryInUseException`—when the specified memory is already in use.
- `OutOfMemoryStaticError`—when there is insufficient memory for the `ImmortalPhysicalMemory` object or for its allocation area in its backing store.

**ImmortalPhysicalMemory(Object, long, SizeEstimator)**
Signature

```java
public ImmortalPhysicalMemory(Object type,
                            long base,
                            SizeEstimator size)
```

Description

Creates an instance with the given parameters.

Parameters

- **type**—An instance of `Object` or an array of objects 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

- `RealtimeSecurityException`—when the application doesn’t have permissions to access physical memory or the given type of memory.

- `OffsetOutOfBoundsException`—when the `base` address is invalid.

- `SizeOutOfBoundsException`—when the size estimate from `size` extends into an invalid range of memory.

- `UnsupportedPhysicalMemoryException`—when the underlying hardware does not support the given type, or when no matching `PhysicalMemoryTypeFilter` has been registered with the `PhysicalMemoryManager`.

- `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. `IllegalArgumentException` may also be when `base` plus the size indicated by `size` would be greater than the maximum physical address supported by the processor.

- `MemoryInUseException`—when the specified memory is already in use.

- `OutOfMemoryStaticError`—when there is insufficient memory for the `ImmortalPhysicalMemory` object or for its allocation area in its backing store.
ImmortalPhysicalMemory(Object, long)

Signature

```java
public
ImmortalPhysicalMemory(Object type,
long size)
```

Description

Creates an instance with the given parameters.

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

RealtimeSecurityException—when the application doesn’t have permissions to access physical memory or the given type of memory.

UnsupportedPhysicalMemoryException—when the underlying hardware does not support the given type, or when no matching PhysicalMemoryTypeFilter has been registered with the PhysicalMemoryManager.

MemoryTypeConflictException—when type specifies incompatible memory attributes.

IllegalArgumentException—when size is less than zero.

OutOfMemoryStaticError—when there is insufficient memory for the ImmortalPhysicalMemory object or for its allocation area in its backing store.

SizeOutOfBoundsException—when the size extends into an invalid range of memory.

ImmortalPhysicalMemory(Object, SizeEstimator)

Signature

```java
public
ImmortalPhysicalMemory(Object type,
SizeEstimator)
```
size)}

Description

Creates an instance with the given parameters.

Parameters

type—An instance of Object or an array of objects 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

RealtimeSecurityException—when the application doesn’t have permissions to access physical memory or the given type of memory.

SizeOutOfBoundsException—when the size estimate from size extends into an invalid range of memory.

UnsupportedPhysicalMemoryException—when the underlying hardware does not support the given type, or when no matching PhysicalMemoryTypeFilter has been registered with the PhysicalMemoryManager.

MemoryTypeConflictException—when type specifies incompatible memory attributes.

IllegalArgumentException—when size is null, or size.getEstimate() is negative.

OutOfMemoryStaticError—when there is insufficient memory for the ImmortalPhysicalMemory object or for its allocation area in its backing store.

B.2.2.12 ImportanceParameters

public class ImportanceParameters

Inheritance

java.lang.Object
    SchedulingParameters
    PriorityParameters
    ImportanceParameters

Description
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` must not consider importance.

**Deprecated** since RTSJ 2.0

### B.2.2.12.1 Constructors

#### ImportanceParameters(int, int)

**Signature**

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

**Description**

Creates an instance of `ImportanceParameters`.

**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.

### B.2.2.12.2 Methods
getImportance

**Signature**

```java
public int getImportance()
```

**Description**

Gets the importance value.

**Returns**

the value of importance for the associated instances of `Scheduled`.  

setImportance(int)

**Signature**

```java
public javax.realtime.ImportanceParameters setImportance(int importance)
```

**Description**

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

**Parameters**

- `importance`—The value to which importance is set.

**Throws**

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

**Returns**

- `this`  

**Available since** RTSJ 2.0 returns itself
**B. Deprecated APIs**

**toString**

*Signature*

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

*Description*

Prints the value of the priority and importance values of the associated instance of `Schedulable`

### B.2.2.13 LTMemory

```java
public class LTMemory
```

*Inheritance*

```java
java.lang.Object  
MemoryArea  
ScopedMemory  
LTMemory
```

*Description*

Equivalent to and superseded by `javax.realtime.memory.LTMemory`.

**Deprecated** since RTSJ 2.0; moved to package `javax.realtime.memory`.

### B.2.2.13.1 Constructors

**LTMemory(long, long, Runnable)**

*Signature*

```java
public LTMemory(long initial, long maximum, Runnable logic)
```

*Description*

1036 RTSJ 2.0 (Draft 66)
LTMemory

Creates an LTMemory of the given size.

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).

Throws

IllegalArgumentException—when initial is greater than maximum, or when initial or maximum is less than zero.

OutOfMemoryStaticError—when there is insufficient memory for the LTMemory object or for its allocation area in its backing store.

IllegalAssignmentError—when storing logic in this would violate the assignment rules.

LTMemory(SizeEstimator, SizeEstimator, Runnable)

Signature

public

LTMemory(SizeEstimator initial,

SizeEstimator maximum,

Runnable logic)

Description

Equivalent to LTMemory(long, long, Runnable) with the argument list (initial.getEstimate(), maximum.getEstimate(), logic).

Parameters

initial—An instance of SizeEstimator 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 used to give an estimate for the maximum bytes 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(SizeEstimator initial, SizeEstimator maximum).

Throws

RTSJ 2.0 (Draft 66)
**B Deprecated APIs**

**LTMemory**

IllegalArgumentException—when initial is null, maximum is null, initial. getEstimate() is greater than maximum.getEstimate(), or when initial. getEstimate() is less than zero.

OutOfMemoryStaticError—when there is insufficient memory for the LTMemory object or for its allocation area in its backing store.

IllegalAssignmentError—when storing logic in this would violate the assignment rules.

**LTMemory(long, long)**

**Signature**

public LTMemory(long initial, long maximum)

**Description**

Equivalent to LTMemory(long, long, Runnable) with the argument list (initial, maximum, null).

**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.

OutOfMemoryStaticError—when there is insufficient memory for the LTMemory object or for its allocation area in its backing store.

**LTMemory(SizeEstimator, SizeEstimator)**

**Signature**

public LTMemory(SizeEstimator initial, SizeEstimator maximum)

**Description**

1038 RTSJ 2.0 (Draft 66)
Equivalent to LTMemory(long, long, Runnable) with the argument list
(initial.getEstimate(), maximum.getEstimate(), null).

Parameters
initial—An instance of SizeEstimator 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 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.
OutOfMemoryStaticError—when there is insufficient memory for the LTMemory
object or for its allocation area in its backing store.

LTMemory(long, Runnable)

Signature
public
   LTMemory(long size,
           Runnable logic)

Description
Equivalent to LTMemory(long, long, Runnable) with the argument list (size,
size, logic).

Available since RTSJ 1.0.1

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 ini-
tial memory area. When logic is null, this constructor is equivalent to
LTMemory(long size).

Throws
IllegalArgumentException—when size is less than zero.
OutOfMemoryStaticError—when there is insufficient memory for the LTMemory
object or for its allocation area in its backing store.
B Deprecated APIs

IllegalAssignmentError—when storing logic in this would violate the assignment rules.

LTMemory(SizeEstimator, Runnable)

Signature
public
LTMemory(SizeEstimator size,
Runnable logic)

Description
Equivalent to LTMemory(long, long, Runnable) with the argument list (size.
getEstimate(), size.getEstimate(), logic).

Available since RTSJ 1.0.1

Parameters
size—An instance of SizeEstimator 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 size).

Throws
IllegalArgumentException—when size is null, or size.getEstimate() is less than zero.
OutOfMemoryStaticError—when there is insufficient memory for the LTMemory object or for its allocation area in its backing store.
IllegalAssignmentError—when storing logic in this would violate the assignment rules.

LTMemory(long)

Signature
public
LTMemory(long size)

Description
1040 RTSJ 2.0 (Draft 66)
LTMemory

Equivalent to LTMemory(long, long, Runnable) with the argument list (size, size, null).

Available since RTSJ 1.0.1

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.
OutOfMemoryStaticError—when there is insufficient memory for the LTMemory object or for its allocation area in its backing store.

LTMemory(SizeEstimator)

Signature
public
   LTMemory(SizeEstimator size)

Description
Equivalent to LTMemory(long, long, Runnable) with the argument list (size.getEstimate(), size.getEstimate(), null).

Available since RTSJ 1.0.1

Parameters
size—An instance of SizeEstimator used to give an estimate of the initial size. This memory must be committed before the completion of the constructor.

Throws
IllegalArgumentException—when size is null, or size.getEstimate() is less than zero.
OutOfMemoryStaticError—when there is insufficient memory for the LTMemory object or for its allocation area in its backing store.

B.2.2.13.2 Methods
**B Deprecated APIs**

**toString**

**Signature**

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

**Description**

Creates a string representation of this object. The string is of the form

```java
{@code (LTMemory) ScopedMemory#<num>}
```

where `<num>` uniquely identifies the `LTMemory` area.

**Returns**

a string representing the value of this.

**B.2.2.14 LTPhysicalMemory**

```java
public class LTPhysicalMemory
```

**Inheritance**

java.lang.Object

    MemoryArea

        ScopedMemory

        LTPhysicalMemory

**Description**

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` memory areas, and the same performance restrictions as `LTMemory`.

No provision is made for sharing objects 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`.

**Deprecated** since RTSJ 2.0

1042 RTSJ 2.0 (Draft 66)
B.2.2.14.1 Constructors

LTPhysicalMemory(Object, long, long, Runnable)

Signature
public LTPhysicalMemory(Object type,
            long base,
            long size,
            Runnable logic)

Description
Creates an instance of LTPhysicalMemory with the given parameters.

See Section PhysicalMemoryManager

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() is called. When logic is null, logic must be supplied when the memory area is entered.

Throws
- SizeOutOfBoundsException—when the implementation detects that base plus size extends beyond physically addressable memory.
- RealtimeSecurityException—when the application doesn’t have permissions to access physical memory or the given type of memory.
- IllegalArgumentException—when size is less than zero.
- OffsetOutOfBoundsException—when the address is invalid.
UnsupportedPhysicalMemoryException—when the underlying hardware does not support the given type, or when no matching PhysicalMemoryTypeFilter has been registered with the PhysicalMemoryManager.

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.

LTPhysicalMemory(Object, long, SizeEstimator, Runnable)

Signature
public
LTPhysicalMemory(Object type,
long base,
SizeEstimator size,
Runnable logic)

Description
Equivalent to LTPhysicalMemory(Object, long, long, Runnable) with the argument list (type, base, size.getEstimate(), logic).

See Section PhysicalMemoryManager

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() is called. When logic is null, logic must be supplied when the memory area is entered.
Throws

RealtimeSecurityException—when the application doesn’t have permissions to access physical memory or the given type of memory.

SizeOutOfBoundsException—when the implementation detects that \texttt{base} plus the size estimate extends beyond physically addressable memory.

OffsetOutOfBoundsException—when the address is invalid.

UnsupportedPhysicalMemoryException—when the underlying hardware does not support the given type, or when no matching \texttt{PhysicalMemoryTypeFilter} has been registered with the \texttt{PhysicalMemoryManager}.

MemoryTypeConflictException—when the specified base does not point to memory that matches the requested type, or when \texttt{type} specifies incompatible memory attributes.

MemoryInUseException—when the specified memory is already in use.

IllegalArgumentException—when \texttt{size} is null, or \texttt{size.getEstimate()} is negative.

IllegalAssignmentError—when storing logic in \texttt{this} would violate the assignment rules.

\texttt{LTPhysicalMemory(Object, long, long)}

\textit{Signature}

\begin{verbatim}
public LTPhysicalMemory(Object type,
   long base,
   long size)
\end{verbatim}

\textit{Description}

Equivalent to \texttt{LTPhysicalMemory(Object, long, long, Runnable)} with the the argument list \texttt{(type, base, size, null)}.

\textit{See Section PhysicalMemoryManager}

\textit{Parameters}

\texttt{type}—An instance of \texttt{Object} representing the type of memory required, e.g., \texttt{dma}, \texttt{shared}, used to define the base address and control the mapping. When the required memory has more than one attribute, \texttt{type} may be an array of objects. When \texttt{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**
- `RealtimeSecurityException`—when the application doesn’t have permissions to access physical memory or the given type of memory.
- `SizeOutofBoundsException`—when the size is less than zero, or the implementation detects that `base` plus `size` extends beyond physically addressable memory.
- `OffsetOutofBoundsException`—when the address is invalid.
- `UnsupportedPhysicalMemoryException`—when the underlying hardware does not support the given type, or when no matching `PhysicalMemoryTypeFilter` has been registered with the `PhysicalMemoryManager`.
- `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.
- `MemoryInUseException`—when the specified memory is already in use.

**LTPhysicalMemory**(Object, long, SizeEstimator)

**Signature**
```java
public LTPhysicalMemory(Object type, long base, SizeEstimator size)
```

**Description**
Equivalent to `LTPhysicalMemory(Object, long, long, Runnable)` with the argument list (type, base, size.getEstimate(), null).

See Section `PhysicalMemoryManager`

**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

RealtimeSecurityException—when the application doesn’t have permissions to access physical memory or the given type of memory.

SizeOutOfBoundsException—when the implementation detects that base plus the size estimate extends beyond physically addressable memory.

OffsetOutOfBoundsException—when the address is invalid.

UnsupportedPhysicalMemoryException—when the underlying hardware does not support the given type, or when no matching PhysicalMemoryTypeFilter has been registered with the PhysicalMemoryManager.

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.

LTPhysicalMemory( Object, long, Runnable)

Signature

public
LTPhysicalMemory(Object type,
long size,
Runnable logic)

Description

Equivalent to LTPhysicalMemory(Object, long, long, Runnable) with the argument list (type, next, size, logic), where next is the beginning of the next best fit in the physical memory range.

See Section PhysicalMemoryManager

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() is called. When logic is null, logic must be supplied when the memory area is entered.

**Throws**

- RealtimeSecurityException—when the application doesn’t have permissions to access physical memory or the given type of memory.
- IllegalArgumentException—when size is less than zero.
- 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 has been registered with the PhysicalMemoryManager.
- 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.

**LTPhysicalMemory(\text{Object, SizeEstimator, Runnable})**

**Signature**

```
public LTPhysicalMemory(Object type, SizeEstimator size, Runnable logic)
```

**Description**

Equivalent to LTPhysicalMemory(\text{Object, long, long, Runnable}) with the argument list (type, next, size.getEstimate(), logic), where next is the beginning of the next best fit in the physical memory range.

See Section PhysicalMemoryManager

1048 RTSJ 2.0 (Draft 66)
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()` is called. When *logic* is `null`, *logic* must be supplied when the memory area is entered.

**Throws**

- `RealtimeSecurityException`—when the application doesn’t have permissions to access physical memory or the given type of memory.
- `SizeOutOfBoundsException`—when the implementation detects that `base` plus the size estimate extends beyond physically addressable memory.
- `UnsupportedPhysicalMemoryException`—when the underlying hardware does not support the given type, or when no matching `PhysicalMemoryTypeFilter` has been registered with the `PhysicalMemoryManager`.
- `MemoryTypeConflictException`—when the specified base does not point to memory that matches the request type, or when *type* specifies attributes with a conflict.
- `IllegalArgumentException`—when *size* is `null`, or `size.getEstimate()` is negative.
- `IllegalAssignmentError`—when storing *logic* in this violates the assignment rules.

**LTPhysicalMemory**(`Object, long`)

**Signature**

```java
public LTPhysicalMemory(Object type,
long size)
```

**Description**

Equivalent to `LTPhysicalMemory(Object, long, long, Runnable)` with the argument list `(type, next, size, null)`, where `next` is the beginning of the next best fit in the physical memory range.
See Section PhysicalMemoryManager

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

- `RealtimeSecurityException`—when the application doesn’t have permissions to access physical memory or the given type of memory.

- `IllegalArgumentException`—when `size` is less than zero.

- `SizeOutOfBoundsException`—when the implementation detects `size` extends beyond physically addressable memory.

- `UnsupportedPhysicalMemoryException`—when the underlying hardware does not support the given type, or when no matching `PhysicalMemoryTypeFilter` has been registered with the `PhysicalMemoryManager`.

- `MemoryTypeConflictException`—when `type` specifies incompatible memory attributes.

`LTPhysicalMemory(Object, SizeEstimator)`

Signature

```java
public LTPhysicalMemory(Object type,
                       SizeEstimator size)
```

Description

Equivalent to `LTPhysicalMemory(Object, long, long, Runnable)` with the argument list `(type, next, size.getEstimate(), null)`, where `next` is the beginning of the next best fit in the physical memory range.

See Section PhysicalMemoryManager

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

RealtimeSecurityException—when the application doesn’t have permissions to access physical memory or the given type 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 has been registered with the PhysicalMemoryManager.

MemoryTypeConflictException—when type specifies incompatible memory attributes.

IllegalArgumentException—when size is null, or size.getEstimate() is negative.

### B.2.2.14.2 Methods

**toString**

**Signature**

public java.lang.String toString()

**Description**

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.

**Returns**

A string representing the value of this.
**B Deprecated APIs**

**B.2.2.15 MemoryAccessError**

```java
public class MemoryAccessError {
    The following elements of MemoryAccessError are deprecated. The required elements are documented in Section 16.2.2.15 above.
}
```

**B.2.2.15.1 Constructors**

**MemoryAccessError(String)**

*Signature*
```java
public MemoryAccessError(String description)
```

*Description*
A descriptive constructor for `MemoryAccessError`.

*Deprecated* since RTSJ 2.0; application code should use `get()` instead.

*Parameters*
- `description`—The reason for throwing this error.

**B.2.2.16 MemoryParameters**

```java
public class MemoryParameters {
    The following elements of MemoryParameters are deprecated. The required elements are documented in Section 11.3.2.4 above.
}
```

**B.2.2.16.1 Fields**

1052 RTSJ 2.0 (Draft 66)
NO_MAX

public static final long NO_MAX

Description
Specifies no maximum limit.

Deprecated since RTSJ 2.0.

B.2.2.16.2 Methods

getMaxMemoryArea

Signature
public long getMaxMemoryArea()

Description
Gets the limit on the amount of memory the schedulable may allocate in its initial memory area. Units are in bytes.

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 then there is no limit for allocation in the initial memory area.

Deprecated since RTSJ 2.0, replace by getMaxInitialArea.

setAllocationRate(long)

Signature
public void setAllocationRate(long allocationRate)

Description
Sets the limit on the rate of allocation in the heap.
Changes to this parameter take place at the next object allocation for each associated schedulable, on an individual basis. Schedulables which are in current
violation of the newly configured value will simply receive an `OutOfMemoryStaticError` on violating allocations. Because this `MemoryParameters` may be associated with more than one schedulable, on a multiprocessor system there may be some implementation-defined delay before executing schedulables detect the parameter changes.

**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**

`IllegalRealtimeArgumentException`—when any value other than positive, zero, or `NO_MAX` is passed as the value of `allocationRate`.

**Deprecated** RTSJ 2.0

### setAllocationRateIfFeasible(long)

**Signature**

```java
public boolean setAllocationRateIfFeasible(long allocationRate)
```

**Description**

Sets the limit on the rate of allocation in the heap. When this `MemoryParameters` object is currently associated with one or more schedulables that have been passed admission control, this change in allocation rate will be submitted to admission control. The scheduler (in conjunction with the garbage collector) will either admit all the effected threads with the new allocation rate, or leave the allocation rate unchanged and cause `setAllocationRateIfFeasible` to return `false`.

Changes to this parameter take place at the next object allocation for each associated schedulable, on an individual basis. Schedulables which are in current violation of the newly configured value will simply receive an `OutOfMemoryStaticError` on violating allocations. Because this `MemoryParameters` may be associated with more than one schedulable, on a multiprocessor system there may be some implementation-defined delay before executing schedulables detect the parameter changes.

**Parameters**

1054 RTSJ 2.0 (Draft 66)
allocationRate—Units 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. 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
IllegalRealtimeArgumentException—when any value other than positive, zero, or NO_MAX is passed as the value of allocationRate.

Returns
true when the request was fulfilled.

Deprecated as of RTSJ 2.0 The framework for feasibility analysis is inadequate.

setMaxImmortalIfFeasible(long)

Signature
public boolean
setMaxImmortalIfFeasible(long maximum)

Description
Sets the limit on the amount of memory the schedulable may allocate in the immortal area.

Changes to this parameter take place at the next object allocation for each associated schedulable, on an individual basis. Schedulables which are in current violation of the newly configured value will simply receive an OutOfMemoryStaticError on violating allocations. Because this MemoryParameters may be associated with more than one schedulable, on a multiprocessor system there may be some implementation-defined delay before executing schedulables detect the parameter changes.

Parameters
maximum—Units in bytes. When zero, no allocation allowed in immortal. To specify no limit, use NO_MAX.

Throws
IllegalRealtimeArgumentException—when any value other than positive, zero, or NO_MAX is passed as the value of maximum.

Returns
true when the value is set, false when any of the schedulables have already allocated more than the given value. In this case the call has no effect.
**Deprecated** as of RTSJ 2.0 The framework for feasibility analysis is inadequate

**setMaxMemoryAreaIfFeasible(long)**

*Signature*

```java
public boolean setMaxMemoryAreaIfFeasible(long maximum)
```

*Description*

Sets the limit on the amount of memory the schedulable may allocate in its initial memory area.

Changes to this parameter take place at the next object allocation for each associated schedulable, on an individual basis. Schedulables which are in current violation of the newly configured value will simply receive an `OutOfMemoryStatistics` on violating allocations. Because this `MemoryParameters` may be associated with more than one schedulable, on a multiprocessor system there may be some implementation-defined delay before executing schedulables detect the parameter changes.

*Parameters*

- `maximum`—Units in bytes. When zero, no allocation allowed in the initial memory area. To specify no limit, use `UNLIMITED`.

*Throws*

- `IllegalRealtimeArgumentException`—when any value other than positive, zero, or `NO_MAX` is passed as the value of `maximum`.

*Returns*

- `true` when the value is set, false when any of the schedulables have already allocated more than the given value. In this case the call has no effect.

**Deprecated** as of RTSJ 2.0, since the framework for feasibility analysis is inadequate.

### B.2.2.17 MemoryScopeException

```java
public class MemoryScopeException
```

The following elements of `MemoryScopeException` are deprecated. The required elements are documented in Section 16.2.2.17 above.
B.2.2.17.1  Constructors

MemoryScopeException(String)

Signature
public
   MemoryScopeException(String description)

Description
A descriptive constructor for MemoryScopeException.

Deprecated since RTSJ 2.0; application code should use get() instead.

Parameters
description—The reason for throwing this exception.

B.2.2.18  NoHeapRealtimeThread

public class NoHeapRealtimeThread

Inheritance
   java.lang.Object
   java.lang.Thread
      RealtimeThread
         NoHeapRealtimeThread

Description
A NoHeapRealtimeThread is a specialized form of RealtimeThread. 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 this 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

RTSJ 2.0 (Draft 66) 1057
and join, complete normally, except where execution would cause access violations.
The constructors of NoHeapRealtimeThread require a reference to ScopedMemory or ImmortalMemory.

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 2.0

B.2.2.18.1 Constructors

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

Signature

public

NoHeapRealtimeThread(SchedulingParameters scheduling,
ReleaseParameters release,
MemoryParameters memory,
MemoryArea area,
ProcessingGroupParameters group,
Runnable logic)

Description

Creates a realtime thread with the given characteristics and a Runnable. The thread group of the new thread is (effectively) null. The newly-created realtime thread which may not use the heap is associated with the scheduler in effect during execution of the constructor.

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 default is a copy of the creator’s
NoHeapRealtimeThread

NoHeapRealtimeThread(SchedulingParameters, ReleaseParameters, MemoryArea)

Signature
public
   NoHeapRealtimeThread(SchedulingParameters scheduling,
                        ReleaseParameters release,
                        MemoryArea area)

Description
   Creates a realtime thread which may not use the heap with the given Sched-
   ulingParameters, ReleaseParameters and MemoryArea, and default values for
   all other parameters. This constructor is equivalent to NoHeapRealtime-
   Thread(scheduling, release, null, area, null, null, null).

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 NoHeapRealtime-
   Thread.
**B Deprecated APIs**

## NoHeapRealtimeThread

### NoHeapRealtimeThread(SchedulingParameters, MemoryArea)

**Signature**

```
public
NoHeapRealtimeThread(SchedulingParameters scheduling,
MemoryArea area)
```

**Description**

Creates a realtime thread with the given `SchedulingParameters` and `MemoryArea` and default values for all other parameters.

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

### B.2.2.18.2 Methods

#### start

**Signature**

```
public void
start()
```

**Description**

Sets up the realtime thread’s environment and starts it. The set up might include delaying it until the assigned start time and initializing the thread’s memory area stack. (See `ScopedMemory`.)

**Throws**

`IllegalRealtimeStateException`—when the configured `Scheduler` and `SchedulingParameters` for this `RealtimeThread` are not compatible.

**Available since** RTSJ 2.0 adds new exception

#### startPeriodic(PhasingPolicy)

**Signature**

```
1060  RTSJ 2.0 (Draft 66)
```
public void
startPeriodic(PhasingPolicy phasingPolicy)
throws LateStartException

*Description*

Starts the thread with the specified phasing policy.

*Available since* RTSJ 2.0

B.2.2.19  *OffsetOutOfBoundsException*

public class OffsetOutOfBoundsException

The following elements of OffsetOutOfBoundsException are deprecated. The required elements are documented in Section 16.2.2.19 above.

**B.2.2.19.1 Constructors**

**OffsetOutOfBoundsException(String)**

*Signature*

```java
public OffsetOutOfBoundsException(String description)
```

*Description*

A descriptive constructor for OffsetOutOfBoundsException.

*Deprecated* since RTSJ 2.0; application code should use `get()` instead.

*Parameters*

description—The reason for throwing the exception.
B Deprecated APIs

B.2.2.20 POSIXSignalHandler

public class POSIXSignalHandler

Inheritance
java.lang.Object
   POSIXSignalHandler

Description
This class enables 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.

Deprecated since RTSJ 2.0

B.2.2.20.1 Fields

SIGHUP

   public static final int SIGHUP

Description
Hangup (POSIX).

SIGINT

   public static final int SIGINT

Description
interrupt (ANSI)
SIGQUIT

public static final int SIGQUIT

Description
quit (POSIX)

SIGILL

public static final int SIGILL

Description
illegal instruction (ANSI)

SIGTRAP

public static final int SIGTRAP

Description
trace trap (POSIX), optional signal.

SIGABRT

public static final int SIGABRT

Description
Abort (ANSI).

SIGBUS

public static final int SIGBUS

Description
BUS error (4.2 BSD), optional signal.

SIGFPE

public static final int SIGFPE

Description
floating point exception
SIGKILL

public static final int SIGKILL

Description
Kill, unblockable (POSIX).

SIGUSR1

public static final int SIGUSR1

Description
User-defined signal 1 (POSIX).

SIGSEGV

public static final int SIGSEGV

Description
Segmentation violation (ANSI).

SIGUSR2

public static final int SIGUSR2

Description
User-defined signal 2 (POSIX).

SIGPIPE

public static final int SIGPIPE

Description
Broken pipe (POSIX).

SIGALRM

public static final int SIGALRM

Description
Alarm clock (POSIX).
SIGTERM

public static final int SIGTERM

*Description*
Termination (ANSI).

SIGCHLD

public static final int SIGCHLD

*Description*
Child status has changed (POSIX).

SIGCONT

public static final int SIGCONT

*Description*
Continue (POSIX), optional signal.

SIGSTOP

public static final int SIGSTOP

*Description*
Stop, unblockable (POSIX), optional signal.

SIGTSTP

public static final int SIGTSTP

*Description*
Keyboard stop (POSIX), optional signal.

SIGTTIN

public static final int SIGTTIN

*Description*
Background read from tty (POSIX), optional signal.
B Deprecated APIs

### SIGTTOU

```java
public static final int SIGTTOU
```

**Description**

Background write to tty (POSIX), optional signal.

### SIGSYS

```java
public static final int SIGSYS
```

**Description**

Bad system call, optional signal.

### SIGIOT

```java
public static final int SIGIOT
```

**Description**

IOT instruction (4.2 BSD), optional signal.

### SIGCLD

```java
public static final int SIGCLD
```

**Description**

Same as SIGCHLD (System V), optional signal.

### SIGEMT

```java
public static final int SIGEMT
```

**Description**

EMT instruction, optional signal.

---

#### B.2.2.20.2 Methods

1066 RTSJ 2.0 (Draft 66)
addHandler(int, AsyncEventHandler)

Signature

public static void
addHandler(int signal,
            AsyncEventHandler handler)

Description

Adds the handler provided to the set of handlers that will be released on the provided signal.

Parameters

signal—The POSIX signal as defined in the constants SIG*.
handler—The handler to be released on the given signal.

Throws

IllegalArgumentException—when signal is not defined by any of the constants in this class or handler is null.

removeHandler(int, AsyncEventHandler)

Signature

public static void
removeHandler(int signal,
              AsyncEventHandler handler)

Description

Removes a handler that was added for a given signal.

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—when signal is not defined by any of the constants in this class.

setHandler(int, AsyncEventHandler)

Signature

RTSJ 2.0 (Draft 66) 1067
public static void
setHandler(int signal,
        AsyncEventHandler handler)

Description
Sets the set of handlers that will be released on the provided signal to the set
with the provided handler being the single element.

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—when signal is not defined by any of the constants
        in this class.

B.2.2.21  PeriodicParameters

public class PeriodicParameters

The following elements of PeriodicParameters are deprecated. The required
elements are documented in Section 6.3.3.6 above.

B.2.2.21.1  Methods

setIfFeasible(RelativeTime, RelativeTime, RelativeTime)

Signature
public boolean
    setIfFeasible(RelativeTime period,
                RelativeTime cost,
                RelativeTime deadline)

Description
This method first performs a feasibility analysis using the new period, cost and
deadline attributes as replacements for the matching attributes of this. When
the resulting system is feasible the method replaces the current attributes of
PhysicalMemoryManager

Inheritance
java.lang.Object
   PhysicalMemoryManager

Description
The PhysicalMemoryManager is not ordinarily used by applications, except that the implementation may require the application to use the registerFilter 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**, **LTPhysicalMemory**, and **ImmortalPhysicalMemory**) 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 address space is linear but not necessarily contiguous. That is, addresses range from 0 .. **MAX_LONG**, but there may be gaps in the memory space. Some intervals in the range may be filled with removable memory as well.

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 whether or not 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 and locks the virtual memory to the memory chunk, on systems that support virtual memory.

Examples of characteristics that might be specified are DMA memory, hardware byte swapping, and non-cached access to memory. 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**.

Original Equipment Manufacturers (OEMs) or other interested parties may provide **PhysicalMemoryTypeFilter** classes that allow additional characteristics of memory devices to be specified.
**PhysicalMemoryManager**  

*javax.realtime B.2*

**Deprecated** as of RTSJ 2.0

---

### B.2.22.1 Fields

**ALIGNED**

```java
public static final java.lang.Object ALIGNED
```

**Description**

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.

See Section `javax.realtime.device.RawMemory`

**BYTESWAP**

```java
public static final java.lang.Object BYTESWAP
```

**Description**

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.

See Section `javax.realtime.device.RawMemory`

**DMA**

```java
public static final java.lang.Object DMA
```

**Description**

RTSJ 2.0 (Draft 66) 1071
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.

See Section `javax.realtime.device.RawMemory`

**IO_PAGE**

```java
public static final java.lang.Object IO_PAGE
```

**Description**

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**

```java
public static final java.lang.Object SHARED
```

**Description**

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.
B.2.2.22.2 Methods

isRemovable(long, long)

Signature

    public static boolean
    isRemovable(long base,
                long size)

Description

Queries the system about the removability of the specified range of memory.

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)

Signature

    public static boolean
    isRemoved(long base,
              long size)

Description

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).

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)

**Signature**
```java
class PhysicalMemoryManager {
    public static void onInsertion(long base,
                                    long size,
                                    AsyncEvent ae)
}
```

**Description**
Registers the specified `AsyncEvent` 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.

**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.

**Available since** RTSJ 1.0.1
public static void
onInsertion(long base,
    long size,
    AsyncEventHandler aeh)

Description
Registers the specified AsyncEventHandler 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, unregisters all "onInsertion"
references to the handler.

Note that 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.

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 con-
tains 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)

Signature
public static void
onRemoval(long base,
    long size,
    AsyncEvent ae)

Description
Registers 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.

Parameters
**B Deprecated APIs**

**PhysicalMemoryManager**

- **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)**

**Signature**

```java
public static void onRemoval(long base, long size, AsyncEventHandler aeh)
```

**Description**

Registers 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, unregisters all "onRemoval" references to the handler parameter.

Note that 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.

**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)**

*Signature*

```java
public static final void registerFilter(Object name,
    PhysicalMemoryTypeFilter filter)
    throws DuplicateFilterException
```

*Description*

Registers a memory type filter with the physical memory manager.
Values of *name* are compared using reference equality (==) not value equality (equals()).

*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.
- **RealtimeSecurityException**—when this operation is not permitted.

**removeFilter(Object)**

*Signature*

```java
public static final void removeFilter(Object name)
```

*Description*

Removes the identified filter from the set of registered filters. When the filter is not registered, silently does nothing.
Values of *name* are compared using reference equality (==) not value equality (equals()).

*Parameters*

- **name**—The identifying object for this memory attribute.

*Throws*
IllegalArgument Exception—when \texttt{name} is null.

RealtimeSecurityException—when this operation is not permitted.

\textbf{unregisterInsertionEvent}(long, long, AsyncEvent)

\textit{Signature}

\begin{verbatim}
public static boolean unregisterInsertionEvent(long base, long size, AsyncEvent ae)
\end{verbatim}

\textit{Description}

Unregisters the specified insertion event. The event is only unregistered when all three arguments match the arguments used to register the event, except that \texttt{ae} of \texttt{null} matches all values of \texttt{ae} and will unregister every \texttt{ae} that matches the address range.

Note that this method has no effect on handlers registered directly as async event handlers.

\textit{Parameters}

- \texttt{base}—The starting address in physical memory associated with \texttt{ae}.
- \texttt{size}—The size of the memory area associated with \texttt{ae}.
- \texttt{ae}—The event to unregister.

\textit{Throws}

- IllegalArgument Exception—when \texttt{size} is less than 0.
- OffsetOutOfBounds Exception—when \texttt{base} is less than zero.
- SizeOutOfBounds Exception—when \texttt{base} plus \texttt{size} would be greater than the physical addressing range of the processor.

\textit{Returns}

- \texttt{true}, when at least one event matched the pattern, \texttt{false}, when no such event was found.

Available since RTSJ 1.0.1

\textbf{unregisterRemovalEvent}(long, long, AsyncEvent)

\textit{Signature}

1078 RTSJ 2.0 (Draft 66)
public static boolean
unregisterRemovalEvent(long base,
    long size,
    AsyncEvent ae)

Description
Unregisters 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 that this method has no effect on handlers registered directly as async event handlers.

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.

Available since RTSJ 1.0.1

B.2.2.23 PriorityParameters

public class PriorityParameters

The following elements of PriorityParameters are deprecated. The required elements are documented in Section 6.3.3.7 above.

B.2.2.23.1 Methods
setPriority(int)

**Signature**

```java
public javax.realtime.PriorityParameters setPriority(int priority)
```

**Description**

Sets 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)`) the base priority of those schedulables is altered as specified by each schedulable’s scheduler.

**Parameters**

- **priority**—The value to which priority is set.

**Throws**

- `IllegalRealtimeArgumentException`—when the given priority value is incompatible with the scheduler for any of the schedulables which are presently using this parameter object.

**Returns**

- this

**Deprecated** since RTSJ 2.0

---

**B.2.2.24  PriorityScheduler**

```java
public abstract class PriorityScheduler
```

The following elements of PriorityScheduler are deprecated. The required elements are documented in Section 6.3.3.8 above.

**B.2.2.24.1  Methods**

**getMaxPriority(Thread)**

**Signature**

```
1080                  RTSJ 2.0 (Draft 66)
```
public static int getMaxPriority(Thread thread)

Description

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 not an instance of Schedulable, the maximum priority of its thread group is returned. Otherwise an exception is thrown.

Parameters

thread—An instance of Thread. When null, the maximum priority of this scheduler is returned.

Throws

IllegalRealtimeArgumentException—when thread is a realtime thread that is not scheduled by an instance of PriorityScheduler.

Returns

the maximum priority for thread

Deprecated since RTSJ 2.0

getMinPriority(Thread)

Signature

public static int getMinPriority(Thread thread)

Description

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 not an instance of Schedulable, Thread.MIN_PRIORITY is returned. Otherwise an exception is thrown.

Parameters

thread—An instance of Thread. When null, the minimum priority of this scheduler is returned.

Throws

IllegalRealtimeArgumentException—when thread is a realtime thread that is not scheduled by an instance of PriorityScheduler.
Returns
the minimum priority for thread

Deprecated since RTSJ 2.0

getNormPriority(Thread)

Signature
public static int
getNormPriority(Thread thread)

Description
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 not an instance of Schedulable, Thread.NORM_PRIORITY is returned. Otherwise an exception is thrown.

Parameters
thread—An instance of Thread. When null, the norm priority for this scheduler is returned.

Throws
IllegalRealtimeArgumentException—when thread is a realtime thread that is not scheduled by an instance of PriorityScheduler.

Returns
The norm priority for thread

Deprecated since RTSJ 2.0

instance

Signature
public static javax.realtime.PriorityScheduler
instance()

Description
Obtains a reference to the distinguished instance of PriorityScheduler, which is the system’s base scheduler.

Returns

1082 RTSJ 2.0 (Draft 66)
A reference to the distinguished instance `PriorityScheduler`.

**Deprecated** since RTSJ 2.0

### isFeasible

**Signature**

```java
public boolean isFeasible()
```

**Description**

Queries this `Scheduler` about the feasibility of the set of schedulables currently in the feasibility set.

**Implementation Notes**

The default feasibility test for the `PriorityScheduler` considers a set of schedulables with bounded resource requirements, to always be feasible. This covers all schedulable objects with release parameters of types `PeriodicParameters` and `SporadicParameters`.

When any schedulable within the feasibility set has release parameters of the exact type `AperiodicParameters` (not a subclass thereof), then the feasibility set is not feasible, as aperiodic release characteristics require unbounded resources. In that case, this method will return `false` and all methods in the `setIfFeasible` family of methods will also return `false`. Consequently, any call to a `setIfFeasible` method that passes a schedulable which has release parameters of type `AperiodicParameters`, or passes proposed release parameters of type `AperiodicParameters`, will return `false`. The only time a `setIfFeasible` method can return `true`, when there exists in the feasibility set a schedulable with release parameters of type `AperiodicParameters`, is when the method will change those release parameters to not be `AperiodicParameters`.

Implementations may provide a feasibility test other than the default test just described. In which case the details of that test should be documented here in place of this description of the default implementation.

**Deprecated** as of RTSJ 2.0 The framework for feasibility analysis is inadequate

### setIfFeasible(Schedulable, ReleaseParameters, MemoryParameters)

**Signature**

```java
RTSJ 2.0 (Draft 66)
```
public boolean setIfFeasible(Schedulable schedulable,
    ReleaseParameters release,
    MemoryParameters memory)

Description

This method first performs a feasibility analysis using the proposed parameter objects as replacements for the current parameters of Schedulable. When the resulting system is feasible, this method replaces the current parameters of Schedulable with the proposed ones. This method does not require that the schedulable be in the feasibility set before it is called. When it is not initially a member of the feasibility set it will be added when the resulting system is feasible.

Parameters

schedulable—The schedulable for which the changes are proposed.
release—The proposed release parameters. When null, the default value of this scheduler is used (a new object is created when the default value is not null). (See PriorityScheduler.)
memory—The proposed memory parameters. When null, the default value of this scheduler is used (a new object is created when the default value is not null). (See PriorityScheduler.)

Throws

IllegalRealtimeArgumentException—when Schedulable is null, or Schedulable is not associated with this scheduler, or the proposed parameters are not compatible with this scheduler.

IllegalAssignmentError—when Schedulable cannot hold references to the proposed parameter objects, or the parameter objects cannot hold a reference to Schedulable.

IllegalThreadStateException—when the new release parameters change Schedulable from periodic scheduling to some other protocol and Schedulable is currently waiting for the next release in RealtimeThread.waitForNextPeriod() or RealtimeThread.waitForNextPeriodInterruptible().

Returns

true, when the resulting system is feasible and the changes are made. False, when the resulting system is not feasible and no changes are made.

Deprecated as of RTSJ 2.0 The framework for feasibility analysis is inadequate
setIfFeasible(Schedulable, ReleaseParameters, MemoryParameters, ProcessingGroupParameters)

Signature
   public boolean
   setIfFeasible(Schedulable schedulable,
                   ReleaseParameters release,
                   MemoryParameters memory,
                   ProcessingGroupParameters group)

Description
This method first performs a feasibility analysis using the proposed parameter objects as replacements for the current parameters of Schedulable. When the resulting system is feasible, this method replaces the current parameters of Schedulable with the proposed ones.

This method does not require that the schedulable be in the feasibility set before it is called. When it is not initially a member of the feasibility set it will be added when the resulting system is feasible.

Parameters
   schedulable—The schedulable for which the changes are proposed.
   release—The proposed release parameters. When null, the default value of this scheduler is used (a new object is created when the default value is not null). (See PriorityScheduler.)
   memory—The proposed memory parameters. When null, the default value of this scheduler is used (a new object is created when the default value is not null). (See PriorityScheduler.)
   group—The proposed processing group parameters. When null, the default value of this scheduler is used (a new object is created when the default value is not null). (See PriorityScheduler.)

Throws
   IllegalRealtimeArgumentException—when Schedulable is null, or Schedulable is not associated with this scheduler, or the proposed parameters are not compatible with this scheduler.
   IllegalAssignmentError—when Schedulable cannot hold references to the proposed parameter objects, or the parameter objects cannot hold a reference to Schedulable.
   IllegalThreadStateException—when the new release parameters change Schedulable from periodic scheduling to some other protocol and Schedulable is currently waiting for the next release in RealtimeThread.waitForNextPeriod()
or `RealtimeThread.waitForNextPeriodInterruption()`.

Returns

- **true**, when the resulting system is feasible and the changes are made. **False**, when the resulting system is not feasible and no changes are made.

**Deprecated** as of RTSJ 2.0 The framework for feasibility analysis is inadequate

```java
setIfFeasible(Schedulable schedulable, SchedulingParameters scheduling, ReleaseParameters release, MemoryParameters memory, ProcessingGroupParameters group)
```

**Signature**

```java
public boolean setIfFeasible(Schedulable schedulable, SchedulingParameters scheduling, ReleaseParameters release, MemoryParameters memory, ProcessingGroupParameters group)
```

**Description**

This method first performs a feasibility analysis using the proposed parameter objects as replacements for the current parameters of `Schedulable`. When the resulting system is feasible, this method replaces the current parameters of `Schedulable` with the proposed ones.

This method does not require that the schedulable be in the feasibility set before it is called. When it is not initially a member of the feasibility set it will be added when the resulting system is feasible.

**Parameters**

- **schedulable**—The schedulable for which the changes are proposed.
- **scheduling**—The proposed scheduling parameters. When `null`, the default value of this scheduler is used (a new object is created when the default value is not `null`). (See `PriorityScheduler`.)
- **release**—The proposed release parameters. When `null`, the default value of this scheduler is used (a new object is created when the default value is not `null`). (See `PriorityScheduler`.)
- **memory**—The proposed memory parameters. When `null`, the default value of this scheduler is used (a new object is created when the default value is not `null`). (See `PriorityScheduler`.)
PriorityScheduler

group—The proposed processing group parameters. When null, the default value of this scheduler is used (a new object is created when the default value is not null). (See PriorityScheduler.)

Throws

IllegalRealtimeArgumentException—when Schedulable is null, or Schedulable is not associated with this scheduler, or the proposed parameters are not compatible with this scheduler.

IllegalAssignmentError—when Schedulable cannot hold references to the proposed parameter objects, or the parameter objects cannot hold a reference to Schedulable.

IllegalThreadStateException—when the new release parameters change Schedulable from periodic scheduling to some other protocol and Schedulable is currently waiting for the next release in RealtimeThread.waitForNextPeriod() or RealtimeThread.waitForNextPeriodInterruptible().

Returns

true, when the resulting system is feasible and the changes are made. False, when the resulting system is not feasible and no changes are made.

Deprecated as of RTSJ 2.0 The framework for feasibility analysis is inadequate

addToFeasibility(Schedulable)

Signature

protected boolean
addToFeasibility(Schedulable schedulable)

Description

Informs this scheduler and cooperating facilities that the resource demands of the given instance of Schedulable will be considered in the feasibility analysis of the associated Scheduler until further notice. Whether the resulting system is feasible or not, the addition is completed. When the object is already included in the feasibility set, does nothing.

Parameters

schedulable—A reference to the given instance of Schedulable

Throws

IllegalRealtimeArgumentException—when schedulable is null, or when schedulable is not associated with this; that is schedulable.getScheduler() != this.
**B Deprecated APIs**

PriorityScheduler

*Returns*

true, when the system is feasible after the addition, otherwise False.

**Deprecated** as of RTSJ 2.0 The framework for feasibility analysis is inadequate

**removeFromFeasibility(Schedulable)**

*Signature*

protected boolean

removeFromFeasibility(Schedulable schedulable)

*Description*

Informs this scheduler and cooperating facilities that the resource demands of the
given instance of Schedulable should no longer be considered in the feasibility
analysis of the associated Scheduler. Whether the resulting system is feasible or
not, the removal is completed.

*Parameters*

schedulable—A reference to the given instance of Schedulable

*Throws*

IllegalRealtimeArgumentException—when schedulable is null.

*Returns*

true, when the removal was successful. False, when the schedulable cannot be
removed from the scheduler’s feasibility set; e.g., the schedulable is not part of
the scheduler’s feasibility set.

**Deprecated** as of RTSJ 2.0 The framework for feasibility analysis is inadequate

**fireSchedulable(Schedulable)**

*Signature*

public void

fireSchedulable(Schedulable schedulable)

*Description*

Triggers the execution of a schedulable (like an AsyncEventHandler).

*Parameters*

schedulable—The schedulable to make active. When null, nothing happens.

*Throws*

1088 RTSJ 2.0 (Draft 66)
UnsupportedRealtimeOperationException—Thrown in all cases by instance of PriorityScheduler.

Deprecated RTSJ 2.0

B.2.2.25 ProcessingGroupParameters

public class ProcessingGroupParameters

Description

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 (budget). 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.

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 object’s affinity set and its processing group’s affinity set, then the schedulable execution is constrained by the default processing group’s affinity 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 parameter, and (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 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 values for cost, deadline, period and start time. Since those times are expressed as a HighResolutionTime, 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 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 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.

**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
Table B.1: ProcessingGroupParameter Default Values

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>new RelativeTime(0,0)</td>
</tr>
<tr>
<td>period</td>
<td>No default. A value must be supplied</td>
</tr>
<tr>
<td>cost</td>
<td>No default. A value must be supplied</td>
</tr>
<tr>
<td>deadline</td>
<td>new RelativeTime(period)</td>
</tr>
<tr>
<td>overrunHandler</td>
<td>None</td>
</tr>
<tr>
<td>missHandler</td>
<td>None</td>
</tr>
</tbody>
</table>

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.

Deprecated as of RTSJ 2.0; replaced by ProcessingGroup.

### B.2.2.25.1 Constructors

**ProcessingGroupParameters(HighResolutionTime, RelativeTime, RelativeTime, RelativeTime, AsyncEventHandler, AsyncEventHandler)**

**Signature**

```java
public ProcessingGroupParameters(HighResolutionTime<?> start,
                                RelativeTime period,
                                RelativeTime cost,
                                RelativeTime deadline,
                                AsyncEventHandler overrunHandler,
                                AsyncEventHandler missHandler)
```

throws IllegalArgumentException,
IllegalAssignmentError

RTSJ 2.0 (Draft 66)
Description

Creates a ProcessingGroupParameters object.

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.

B.2.2.25.2 Methods
clone

Signature

```java
public java.lang.Object
clone()
throws CloneNotSupportedException
```

Description

Creates 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.)

Throws

`CloneNotSupportedException`—never

Returns

the clone of this

Available since RTSJ 1.0.1

getCost

Signature

```java
public javax.realtime.RelativeTime
getCost()
```

Description

Gets the value of `cost`.

Returns

a reference to the value of `cost`.
**getCostOverrunHandler**

*Signature*

```java
class javax.realtime.AsyncEventHandler
getCostOverrunHandler()
```

*Description*

Gets the cost overrun handler.

*Returns*

a reference to an instance of `AsyncEventHandler` that is cost overrun handler of this.

**getDeadline**

*Signature*

```java
class javax.realtime.RelativeTime
getDeadline()
```

*Description*

Gets the value of `deadline`.

*Returns*

a reference to an instance of `RelativeTime` that is the deadline of this.

**getDeadlineMissHandler**

*Signature*

```java
class javax.realtime.AsyncEventHandler
getDeadlineMissHandler()
```

*Description*

Gets the deadline miss handler.

*Returns*

a reference to an instance of `AsyncEventHandler` that is deadline miss handler of this.
getPeriod

Signature
   public javax.realtime.RelativeTime
   getPeriod()

Description
   Gets the value of period.

Returns
   a reference to an instance of RelativeTime that represents the value of period.

getStart

Signature
   public javax.realtime.HighResolutionTime<?>
   getStart()

Description
   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.

Returns
   a reference to an instance of HighResolutionTime that represents the value of
   start.

setCost(RelativeTime)

Signature
   public void
   setCost(RelativeTime cost)
   throws IllegalArgumentException,
         IllegalAssignmentError

Description
   Sets the value of cost.

Parameters
   cost—The new value for cost. When null, an exception is thrown.

RTSJ 2.0 (Draft 66) 1095
**Throws**

*IllegalArgumentException*—when *cost* is *null* or its time value is less than zero.

*IllegalAssignmentError*—when *cost* cannot be stored in *this*.

### setCostOverrunHandler(AsyncEventHandler)

**Signature**

```java
public void setCostOverrunHandler(AsyncEventHandler handler)
throws IllegalAssignmentError
```

**Description**

Sets the cost overrun handler.

**Parameters**

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)

**Signature**

```java
public void setDeadline(RelativeTime deadline)
throws IllegalArgumentException, IllegalAssignmentError
```

**Description**

Sets the value of *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)

Signature
public void
setDeadlineMissHandler(AsyncEventHandler handler)
throws IllegalAssignmentError

Description
Sets the deadline miss 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.

setIfFeasible(RelativeTime, RelativeTime, RelativeTime)

Signature
public boolean
setIfFeasible(RelativeTime period,
RelativeTime cost,
RelativeTime deadline)
throws IllegalArgumentException,
IllegalAssignmentError

Description
This method first performs a feasibility analysis using the period, cost and deadline attributes as replacements for the matching attributes this. When the resulting system is feasible the method replaces the current attributes of this with the new attributes.

Parameters
period—The proposed period. There is no default value. When period is null an exception is thrown.
cost—The proposed cost. When null, an exception is thrown.
**Deprecated APIs**

`ProcessingGroupParameters`

**deadline**—The proposed deadline. When `null`, the default value is new instance of `RelativeTime(period)`.

**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.
- `IllegalAssignmentError`—when `period`, `cost`, or `deadline` cannot be stored in this.

**Returns**

- `true`, when the resulting system is feasible and the changes are made. `False`, when the resulting system is not feasible and no changes are made.

**setPeriod(RelativeTime)**

**Signature**

```java
public void setPeriod(RelativeTime period)
throws IllegalArgumentException,
        IllegalAssignmentError
```

**Description**

Sets the value of `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(HighResolutionTime)**

**Signature**

```java
public void setStart(HighResolutionTime<?> start)
```

1098 

RTSJ 2.0 (Draft 66)
throws IllegalArgumentException, IllegalAssignmentError

Description
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.

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.

B.2.2.26 RawMemoryAccess

public class RawMemoryAccess

Inheritance
java.lang.Object
    RawMemoryAccess

Description
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 static byte variable in class RealtimeSystem. 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 each case the documentation for the PhysicalMemoryTypeFilter 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**. 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**. 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 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

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Values</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access type</td>
<td>read, write</td>
<td></td>
</tr>
<tr>
<td>Data type</td>
<td>byte, short, int, long, float, double</td>
<td></td>
</tr>
<tr>
<td>Alignment</td>
<td>0 aligned</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 to one less than data type size</td>
<td>the first byte of the data is alignment bytes away from natural alignment.</td>
</tr>
<tr>
<td>Atomicity</td>
<td>processor</td>
<td>means access is atomic with respect to other tasks on processor.</td>
</tr>
<tr>
<td></td>
<td>smp</td>
<td>means access is processor atomic, and atomic with respect to all processors in an SMP.</td>
</tr>
<tr>
<td></td>
<td>memory</td>
<td>means that access is smp atomic, and atomic with respect to all access to the memory including DMA.</td>
</tr>
</tbody>
</table>

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 `get<type>` or `put<type>` method, where `type` is defined in the table
above, and to complete the reads and writes in the order they appear in the
program order.

**Deprecated** as of RTSJ 2.0. Use `javax.realtime.device.RawMemoryFactory` to
create the appropriate `javax.realtime.device.RawMemory` object.

### B.2.2.26.1 Constructors

#### RawMemoryAccess(Object, long, long)

**Signature**

```java
class RawMemoryAccess {
    public RawMemoryAccess(Object type,
                           long base,
                           long size)
}
```

**Description**

Constructs an instance of `RawMemoryAccess` with the given parameters, and sets
the object to the mapped state. When the platform supports virtual memory,
maps 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` and `PhysicalMemoryTypeFilter.getVMFlags`.

**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**

1102 RTSJ 2.0 (Draft 66)
RealtimeSecurityException—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 has been registered with the PhysicalMemoryManager.

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.

RawMemoryAccess(Object, long)

Signature
public
    RawMemoryAccess(Object type,
        long size)

Description
Constructs an instance of RawMemoryAccess with the given parameters, and sets the object to the mapped state. When the platform supports virtual memory, maps 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 vMAtributes of the PhysicalMemoryTypeFilter objects that matched this object’s type parameter. (See PhysicalMemoryTypeFilter.getVMAttributes and PhysicalMemoryTypeFilter.getVMFlags.

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

RealtimeSecurityException—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 has been registered with the PhysicalMemoryManager.

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.

RealtimeSecurityException—when the application doesn’t have permissions to access physical memory or the given range of memory.

B.2.2.26.2 Methods

getByte(long)

Signature

public byte getByte(long offset)

Description

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.)

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* somewhat overlaps this exception since it is when the offset is within the object but outside the mapped area.

*RealtimeSecurityException*—when this access is not permitted by the security manager.

Returns

the byte from raw memory.

See Section `RawMemoryAccess.map(long,long)`

**getBytes(long, byte, int, int)**

**Signature**

```java
public void getBytes(long offset,
                     byte[] bytes,
                     int low,
                     int number)
```

**Description**

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 occurs in the same order as in the program. Multiple writes to the same location may not be coalesced.)

**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 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 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 byte 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.

RealtimeSecurityException—when this access is not permitted by the security manager.

See Section RawMemoryAccess.map(long,long)

**getInt(long)**

**Signature**

public int
getInt(long offset)

**Description**

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 occurs in the same order as in the program. Multiple writes to the same location may not be coalesced.)

**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** 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.

**RealtimeSecurityException**—when this access is not permitted by the security manager.

Returns

the integer from raw memory.

See Section `RawMemoryAccess.map(long, long)`

getInts(long, int, int, int)

Signature

```java
public void getInts(long offset,
                    int[] ints,
                    int low,
                    int number)
```

Description

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.)

Parameters

**offset**—The offset in bytes from the beginning of the raw memory area at which to start loading.
**Deprecated APIs**

**RawMemoryAccess**

- **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 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.
- **SizeOutOfBoundsException**—when the object is not mapped, or when an integer 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 **int** 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**.
- **RealtimeSecurityException**—when this access is not permitted by the security manager.

See Section **RawMemoryAccess.map(long,long)**

**getLong(long)**

**Signature**

```java
public long getLong(long offset)
```

**Description**

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.)

**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 long falls in an invalid address range.
- `RealtimeSecurityException`—when this access is not permitted by the security manager.

Returns

the long from raw memory.

**getLongs(long, long, int, int)**

**Signature**

```java
public void getLongs(long offset,
                     long[] longs,
                     int low,
                     int number)
```

**Description**

Gets `number` longs starting at the given offset in the memory area associated with this object and assign them to the `longs` 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.)

**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`
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 long 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.

**RealtimeSecurityException**—when this access is not permitted by the security manager.

See Section `RawMemoryAccess.map(long,long)`

### getMappedAddress

**Signature**

```java
public long
getMappedAddress()
```

**Description**

Gets the virtual memory location at which the memory region is mapped.

**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. When virtual memory is not supported, this is the same as the physical base address.

### getShort(long)

**Signature**

```java
public short
getShort(long offset)
```

**Description**

1110 RTSJ 2.0 (Draft 66)
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 occurs in the same order as in the program. Multiple writes to the same location may not be coalesced.)

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 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.
RealtimeSecurityException—when this access is not permitted by the security manager.

Returns
the short loaded from raw memory.

See Section RawMemoryAccess.map(long,long)

getShorts(long, short, int, int)

Signature
public void
getShorts(long offset,
       short[] shorts,
       int low,
       int number)

Description
 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.)

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 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 short 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.

RealtimeSecurityException—when this access is not permitted by the security manager.

See Section RawMemoryAccess.map(long,long)

map

Signature

1112 RTSJ 2.0 (Draft 66)
public long map()

Description
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 vMAtributes of the PhysicalMemoryTypeFilter objects that matched this object’s type parameter. (See PhysicalMemoryTypeFilter.getVMAttributes and PhysicalMemoryTypeFilter.getVMFlags.

When the object is already mapped into virtual memory, this method does not change anything.

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)

Signature
public long map(long base)

Description
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 vMAtributes of the PhysicalMemoryTypeFilter objects that matched this object’s type parameter. (See PhysicalMemoryTypeFilter.getVMAttributes and PhysicalMemoryTypeFilter.getVMFlags.

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.
B Deprecated APIs

RawMemoryAccess

Parameters

base—The location to map at the virtual memory space.

Throws

OutOfMemoryError—when there is insufficient free virtual memory at the specified address.

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)

Signature

public long
map(long base,
long size)

Description

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 vMAtributes of the PhysicalMemoryTypeFilter objects that matched this object’s type parameter. (See PhysicalMemoryTypeFilter.getVMAttributes and PhysicalMemoryTypeFilter.getVMFlags.

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.

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. 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

1114 RTSJ 2.0 (Draft 66)
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)**

*Signature*
```java
public void setByte(long offset,
byte value)
```

*Description*
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.)

*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 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.
- RealtimeSecurityException—when this access is not permitted by the security manager.

*See Section RawMemoryAccess.map(long,long)*
setBytes(long, byte, int, int)

Signature

```java
public void setBytes(long offset,
                     byte[] bytes,
                     int low,
                     int number)
```

Description

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.)

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` 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 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 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`. 

1116 RTSJ 2.0 (Draft 66)
length.

RealtimeSecurityException—when this access is not permitted by the security manager.

See Section RawMemoryAccess.map(long,long)

setInt(long, int)

Signature

public void
  setInt(long offset,
           int value)

Description

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.)

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

OutOfBoundsException—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.

SizeOutOfBoundsException—when the object is not mapped, or when the integer falls in an invalid address range.

RealtimeSecurityException—when this access is not permitted by the security manager.
See Section RawMemoryAccess.map(long,long)

**setInts(long, int, int, int)**

_Signature_

```java
public void setInts(long offset,
                   int[] ints,
                   int low,
                   int number)
```

_Description_

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.

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.)

_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_ 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 integer 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.

RealtimeSecurityException—when this access is not permitted by the security manager.

See Section RawMemoryAccess.map(long,long)

setLong(long, long)

Signature

    public void
    setLong(long offset,
            long value)

Description

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.)

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 somewhat overlaps this exception since it is when the offset is within the object but outside the mapped area.
Deprecated APIs

- SizeOutOfBoundsException—when the object is not mapped, or when the long falls in an invalid address range.
- RealtimeSecurityException—when this access is not permitted by the security manager.

See Section RawMemoryAccess.map(long,long)

setLongs(long, long, int, int)

**Signature**

```java
public void setLongs(long offset,
long[] longs,
int low,
int number)
```

**Description**

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 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.)

**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`
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 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.

RealtimeSecurityException—when this access is not permitted by the security manager.

See Section RawMemoryAccess.map(long,long)

setShort(long, short)

Signature

public void
setShort(long offset,
    short value)

Description

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.

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.)

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
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.

**RealtimeSecurityException**—when this access is not permitted by the security manager.

See Section RawMemoryAccess.map(long,long)

### setShorts(long, short, int, int)

**Signature**

```java
class RawMemoryAccess {
  public void setShorts(long offset, short[] shorts, int low, int number)
}
```

**Description**

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.)

**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`
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 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`.

**RealtimeSecurityException**—when this access is not permitted by the security manager.

See Section `RawMemoryAccess.map(long,long)`

### unmap

**Signature**

```java
class RawMemoryFloatAccess
public void unmap()
```

**Description**

Unmaps 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**.

An unmapped raw memory object can be returned to mapped state with any of the object’s **map** methods.

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.

### B.2.2.27 RawMemoryFloatAccess

```java
class RawMemoryFloatAccess
```
**B Deprecated APIs**

### Inheritance

java.lang.Object
  RawMemoryAccess
  RawMemoryFloatAccess

### Description

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 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 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 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 optional system properties that identify a platform’s level of support.
for atomic raw put and get. (See `RawMemoryAccess`.) 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.

Many of the constructors and methods in this class throw `OffsetOutOfBoundsException`. 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`. 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** as of RTSJ 2.0. Use `javax.realtime.device.RawMemory`.

### B.2.2.27.1 Constructors

**RawMemoryFloatAccess(Object, long, long)**

**Signature**

```java
public
RawMemoryFloatAccess(Object type,
    long base,
    long size)
```

**Description**

Constructs an instance of `RawMemoryFloatAccess` with the given parameters,
and sets the object to the mapped state. When the platform supports virtual
memory, maps 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.getVMAtributes` and `PhysicalMemoryTypeFilter.getVMFlags`.

**Parameters**

RTSJ 2.0 (Draft 66)
type—An instance of Object representing the type of memory required, e.g., dma, shared, for defining the base address and controlling 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
  RealtimeSecurityException—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 has been registered with the PhysicalMemoryManager.
  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.

RawMemoryFloatAccess(Object, long)

Signature
  public
  RawMemoryFloatAccess(Object type, long size)

Description
  Constructs an instance of RawMemoryFloatAccess with the given parameters, and sets the object to the mapped state. When the platform supports virtual memory, maps the raw memory into virtual memory.

  The runtime 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 vMAT-
tributes of the `PhysicalMemoryTypeFilter` objects that matched this object’s `type` parameter. (See `PhysicalMemoryTypeFilter.getVMAttributes` and `PhysicalMemoryTypeFilter.getVMFlags`.

**Parameters**

type—An instance of `Object` representing the type of memory required, e.g., `dma`, `shared`, for defining the base address and controlling 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**

`RealtimeSecurityException`—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` has been registered with the `PhysicalMemoryManager`.

`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.

### B.2.2.27.2 Methods

#### getDouble(long)

**Signature**

```
public double getDouble(long offset)
```

**Description**

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.)

**Parameters**
- **offset**—The offset in bytes from the beginning of the raw memory area from which to load the double.

**Throws**
- **OffsetOutOfBoundsException**—when the offset is invalid.
- **SizeOutOfBoundsException**—when the object is not mapped, or when the double falls in an invalid address range.
- **RealtimeSecurityException**—when this access is not permitted by the security manager.

**Returns**
- the double from raw memory.

**getDoubles(long, double, int, int)**

**Signature**

```java
public void getDoubles(long offset,
            double[] doubles,
            int low,
            int number)
```

**Description**

Gets number doubles starting at the given offset in the memory area associated with this object and assigns 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.)
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)).

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 or remapped. The double 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.

RealtimeSecurityException—when this access is not permitted by the security manager.

getFloat(long)

Signature

public float
getFloat(long offset)

Description

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.)
**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)`).

`SizeOutOfBoundsException`—when the object is not mapped, or when the float falls in an invalid address range.

`RealtimeSecurityException`—when this access is not permitted by the security manager.

**Returns**

the float from raw memory.

getFloats(long, float, int, int)

**Signature**

```java
public void
getFloats(long offset,
        float[] floats,
        int low,
        int number)
```

**Description**

Gets `number` floats starting at the given offset in the memory area associated with this object and assign them to the `floats` 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.)

**Parameters**

1130

RTSJ 2.0 (Draft 66)
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 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)).

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 float 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.

RealtimeSecurityException—when this access is not permitted by the security manager.

setDouble(long, double)

Signature

public void
setDouble(long offset,
            double value)

Description

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.)

Parameters
**Deprecated APIs**

**RawMemoryFloatAccess**

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)`).
- SizeOutOfBoundsException—when the object is not mapped, or when the double falls in an invalid address range.
- RealtimeSecurityException—when this access is not permitted by the security manager.

**setDoubles(long, double, int, int)**

**Signature**

```java
public void setDoubles(long offset,
        double[] doubles,
        int low,
        int number)
```

**Description**

Sets number doubles starting at the given offset in the memory area associated with this object from the doubles 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.)

**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)`).

**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 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**.

**RealtimeSecurityException**—when this access is not permitted by the security manager.

**setFloat(long, float)**

**Signature**

```java
public void setFloat(long offset,
                      float value)
```

**Description**

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.)

**Parameters**

- **offset**—The offset in bytes from the beginning of the raw memory area at which to write the float.
- **value**—The float to write.

**Throws**

RTSJ 2.0 (Draft 66) 1133
**B Deprecated APIs**

**RawMemoryFloatAccess**

**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)**).

**SizeOutOfBoundsException**—when the object is not mapped, or when the float falls in an invalid address range.

**RealtimeSecurityException**—when this access is not permitted by the security manager.

**setFloats(long, float, int, int)**

**Signature**

```java
public void setFloats(long offset,
                     float[] floats,
                     int low,
                     int number)
```

**Description**

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.)

**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**

**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)**).
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 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.

RealtimeSecurityException—when this access is not permitted by the security manager.

B.2.2.28 RealtimeSecurity

```java
public class RealtimeSecurity
```

Inheritance

- java.lang.Object
- RealtimeSecurity

Description

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 at run-time, RealtimeSystem.setSecurityManager. 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.

Deprecated since RTSJ 2.0

B.2.2.28.1 Constructors

**RealtimeSecurity**

Signature

RTSJ 2.0 (Draft 66)
public
RealtimeSecurity()

Description
Create an RealtimeSecurity object.

B.2.2.28.2 Methods

checkAccessPhysical

Signature
public void
checkAccessPhysical()
throws RealtimeSecurityException

Description
Check whether the application is allowed to access physical memory.

Throws
RealtimeSecurityException—The application doesn’t have permission to access physical memory.

checkAccessPhysicalRange(long, long)

Signature
public void
checkAccessPhysicalRange(long base, long size)
throws RealtimeSecurityException

Description
Checks whether the application is allowed to access physical memory within the specified range.

Parameters
base—The beginning of the address range.
size—The size of the address range.
Thros

**RealtimeSecurityException**—The application doesn’t have permission to access the memory in the given range.

### checkSetFilter

**Signature**
```
public void checkSetFilter()
throws RealtimeSecurityException
```

**Description**
Checks whether the application is allowed to register `PhysicalMemoryTypeFilter` objects with the `PhysicalMemoryManager`.

**Thros**

**RealtimeSecurityException**—The application doesn’t have permission to register filter objects.

### checkSetMonitorControl(MonitorControl)

**Signature**
```
public void checkSetMonitorControl(MonitorControl policy)
throws RealtimeSecurityException
```

**Description**
Checks whether the application is allowed to set the default monitor control policy.

**Parameters**
- `policy`—The new policy

**Thros**

**RealtimeSecurityException**—when the application doesn’t have permission to change the default monitor control policy to `policy`.

**Available since** RTSJ 1.0.1
B Deprecated APIs

checkAEHSetDaemon

Signature
   public void
   checkAEHSetDaemon()
   throws RealtimeSecurityException

Description
   Checks whether the application is allowed to set the daemon status of an AEH.

Throws
   RealtimeSecurityException—when the application is not permitted to alter the
daemon status.

Available since RTSJ 1.0.1

checkSetScheduler

Signature
   public void
   checkSetScheduler()
   throws RealtimeSecurityException

Description
   Checks whether the application is allowed to set the scheduler.

Throws
   RealtimeSecurityException—The application doesn’t have permission to set the
scheduler.

B.2.2.29   RealtimeSystem

public class RealtimeSystem
   The following elements of RealtimeSystem are deprecated. The required elements
are documented in Section 15.2.2.5 above.

B.2.2.29.1   Methods
RealtimeThread

getSecurityManager

Signature
public static javax.realtime.RealtimeSecurity
getSecurityManager()

Description
Gets a reference to the security manager used to control access to realtime system
features such as access to physical memory.

Returns
a RealtimeSecurity object representing the default realtime security manager.

Deprecated since RTSJ 2.0

B.2.2.30 RealtimeThread

public class RealtimeThread

The following elements of RealtimeThread are deprecated. The required elements
are documented in Section 5.3.2.2 above.

B.2.2.30.1 Constructors

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

Signature
public
RealtimeThread(SchedulingParameters scheduling,
ReleaseParameters release,
MemoryParameters memory,
MemoryArea area,
ProcessingGroupParameters group,
Runnable logic)
B Deprecated APIs

RealtimeThread

Description

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

Deprecated since RTSJ 2.0

B.2.2.30.2 Methods

getInitialMemoryAreaIndex

Signature

public static int getInitialMemoryAreaIndex()
throws IllegalRealtimeStateException, ClassCastException

Description

Gets the position of the initial memory area for the current Schedulable in the memory area stack. Memory area stacks may include inherited stacks from parent threads. The initial memory area of a RealtimeThread or an AsyncBaseEventHandler 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.

Throws

IllegalRealtimeStateException—when the memory area stack of the current Schedulable has changed from its initial configuration and the memory area at the originally specified initial memory area index is not the initial memory area, thus the index is invalid.

This can only happen when the application uses the alternate memory module and the initial memory area is a scoped memory area. The following is an example of an event handler that will throw this exception when its initial memory area is a scoped memory area.

    public void handleAsyncEvent() {
    
    }
MemoryArea current = RealtimeThread.getCurrentMemoryArea();
if (current instanceof ScopedMemory)
{
    MemoryArea parent =
        ((ScopedMemory) current).getParent();
    parent.executeInArea(() ->
    {
        ScopedMemory scope = new LTMemory(1000);
        scope.enter(() ->
        {
            System.out.println("Initial Memory Area Index = " +
            RealtimeThread.getInitialMemoryAreaIndex());
        });
    });
}

ClassCastException—when the current execution context is not an instance of
Scheduled. An exception will be thrown on line 12, where the first opening
bracket is line one, of the handler above.

Returns
the index into the initial memory area stack of the initial memory area of the current
Scheduled.

Deprecated as of RTSJ 2.0.

gMemoryAreaStackDepth

Signature
public static int
gMemoryAreaStackDepth()
throws ClassCastException

Description
Gets the size of the memory area stack of MemoryArea instances to which the
current schedulable has access. For a realtime thread started with immortal
or heap as its explicit initial memory area, the initial size is one. The current
memory area \((\text{getCurrentMemoryArea}())\) is found at memory area stack index of \(\text{getMemoryAreaStackDepth}() - 1\).

**Throws**

ClassCastException—when the current execution context is not an instance of Schedulable.

**Returns**

the size of the stack of MemoryArea instances.

**Deprecated** as of RTSJ 2.0

### getOuterMemoryArea(int)

**Signature**

```java
public static javax.realtime.MemoryArea getOuterMemoryArea(int index)
throws ClassCastException,
MemoryAccessError
```

**Description**

Gets the instance of MemoryArea in the memory area stack at the index given. When the given index does not exist in the memory area stack, then null is returned. For a thread started with immortal or heap as its explicit initial memory area, the index of that area is zero. The current memory area \((\text{getCurrentMemoryArea}())\) is found at memory area stack index \(\text{getMemoryAreaStackDepth}() - 1\), so \(\text{getCurrentMemoryArea}() == \text{getOuterMemoryArea}(\text{getMemoryAreaStackDepth}() - 1)\).

Note that accessing the stack should have a maximum complexity of \(O(n)\), where \(n\) is the stack depth. This means the memory stack need not be backed by an array.

**Parameters**

- **index**—The offset into the memory area stack.

**Throws**

ClassCastException—when the current execution context is not an instance of Schedulable.

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

**Returns**

1142 RTSJ 2.0 (Draft 66)
the instance of MemoryArea at index or null when the given index does not correspond to a position in the stack.

sleep(Clock, HighResolutionTime)

**Signature**

```java
public static void sleep(Clock clock,
    HighResolutionTime<?> time)
throws InterruptedException,
    ClassCastException,
    UnsupportedRealtimeOperationException,
    IllegalRealtimeArgumentException
```

**Description**

A sleep method that is controlled by a generalized clock. Since the time is expressed as a HighResolutionTime, this method is an accurate timer with nanosecond granularity. The actual resolution available for the clock and even the quantity it measures depends on clock associated with time. The time base is the given Clock associated with time. The sleep time may be relative or absolute. When relative, then the calling thread is blocked for the amount of time given by time, and measured by clock. When absolute, then the calling thread is blocked until the indicated value is reached by clock. When the given absolute time is less than or equal to the current value of clock, the call to sleep returns immediately.

Calling sleep is permissible when control is in an AsyncEventHandler. The method causes the handler to sleep.

This method must not throw IllegalAssignmentError. It must tolerate time instances that may not be stored in this.

**Parameters**

clock—The instance of Clock used as the base. When clock is null the realtime clock (see Clock.getRealtimeClock) is used. When time uses a time-base other than clock, time is reassociated with clock for purposes of this method.

time—The amount of time to sleep or the point in time at which to awaken.

**Throws**

InterruptedException—when the thread is interrupted by interrupt() or AsynchronouslyInterruptedException.fire() during the time between calling this method and returning from it.
ClassCastException—when the current execution context is not an instance of Schedulable.

UnsupportedRealtimeOperationException—when the sleep operation is not supported by clock.

IllegalRealtimeArgumentException—when time is null, or when time is a relative time less than zero.

Deprecated in RTSJ 2.0

waitForNextPeriod

Signature

```java
public static boolean waitForNextPeriod()
throws ClassCastException,
   IllegalThreadStateException
```

Description

Causes the current realtime thread to delay until the beginning of the next period. Used by threads that have a reference to a ReleaseParameters type of PeriodicParameters 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.)

Throws

IllegalThreadStateException—when this does not have a reference to a ReleaseParameters type of PeriodicParameters.

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

Returns

either false when the thread is in a deadline miss condition or true otherwise.

When a deadline miss condition occurs is defined by its thread’s scheduler.

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

Deprecated RTSJ 2.0 Replaced by waitForNextRelease()
waitForNextPeriodInterruptible

Signature

```java
public static boolean waitForNextPeriodInterruptible()
throws InterruptedException,
ClassCastException,
IllegalThreadStateException
```

Description

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

Used by threads that have a reference to a `ReleaseParameters` type of `PeriodicParameters` 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 `waitForNextPeriodInterruptible()` is controlled by this thread’s scheduler. (See `PriorityScheduler`)

Throws

- `InterruptedException`—when the thread is interrupted by `interrupt()` or `AsynchronouslyInterruptedException.fire()` 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` type of `PeriodicParameters`.

Returns

either `false`, when the thread is in a deadline miss condition, or `true`, otherwise.

The time at which a deadline miss condition occurs is defined by its thread’s scheduler.

Available since RTSJ 1.0.1

Deprecated RTSJ 2.0 Replaced by `waitForNextReleaseInterruptible`
addIfFeasible

Signature

\[
\text{public boolean addIfFeasible()}
\]

Description

This method first performs a feasibility analysis with this added to the system. When the resulting system is feasible, informs the scheduler and cooperating facilities that this instance of \texttt{Schedulable} should be considered in feasibility analysis until further notified. When the analysis shows that the system including this would not be feasible, this method does not admit this to the feasibility set.

When the object is already included in the feasibility set, does nothing.

Returns

true when inclusion of this in the feasibility set yields a feasible system, and false otherwise. When true is returned then this is known to be in the feasibility set. When false is returned, this was not added to the feasibility set, but it may already have been present.

Deprecated as of RTSJ 2.0 The framework for feasibility analysis is inadequate

addToFeasibility

Signature

\[
\text{public boolean addToFeasibility()}
\]

Description

Informs the scheduler and cooperating facilities that this instance of \texttt{Schedulable} should be considered in feasibility analysis until further notified.

When the object is already included in the feasibility set, does nothing.

Returns

true, when the resulting system is feasible. False, when not.

Deprecated as of RTSJ 2.0 The framework for feasibility analysis is inadequate
deschedulePeriodic

Signature

public void
deschedulePeriodic()

Description
When the ReleaseParameters object associated with this RealtimeThread is an instance of PeriodicParameters, performs any deschedulePeriodic actions specified by this thread’s scheduler. When the type of the associated instance of ReleaseParameters is not PeriodicParameters nothing happens.

Deprecated since RTSJ 2.0

getProcessingGroupParameters

Signature

public javax.realtime.ProcessingGroupParameters
getProcessingGroupParameters()

Description
Gets a reference to the ProcessingGroupParameters object for this schedulable.

Returns
A reference to the current ProcessingGroupParameters object.

Deprecated since RTSJ 2.0

removeFromFeasibility

Signature

public boolean
removeFromFeasibility()

Description
Informs the scheduler and cooperating facilities that this instance of Schedulable should not be considered in feasibility analysis until it is further notified.

Returns
true when the removal was successful. false when the schedulable cannot be
removed from the scheduler’s feasibility set; e.g., the schedulable is not part of
the scheduler’s feasibility set.

Deprecated as of RTSJ 2.0 The framework for feasibility analysis is inadequate

schedulePeriodic

Signature
public void
schedulePeriodic()

Description
Begins unblocking RealtimeThread.waitForNextPeriod() for a periodic thread. When
deadline miss detection is disabled, enables it. Typically used when a
periodic schedulable is in a deadline miss condition. The details of the interaction
of this method with deschedulePeriodic() and waitForNextPeriod() are
dictated by this thread’s scheduler.

When this RealtimeThread does not have a type of PeriodicParameters as
its ReleaseParameters nothing happens.

Deprecated since RTSJ 2.0

setIfFeasible(ReleaseParameters, MemoryParameters)

Signature
public boolean
setIfFeasible(ReleaseParameters release,
MemoryParameters memory)

Description
This method first performs a feasibility analysis using the proposed parameter
objects as replacements for the current parameters of this. When the resulting
system is feasible, this method replaces the current parameters of this with the
proposed ones.

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.
This method does not require that the schedulable be in the feasibility set before it is called. When it is not initially a member of the feasibility set it will be added when the resulting system is feasible.

**Parameters**

**release**—The release parameters to use. 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`.)

**memory**—The memory parameters to use. 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`.)

**Throws**

- **IllegalRealtimeArgumentException**—when the parameter values are not compatible with the schedulable’s scheduler. Also when this schedulable may not use the heap and any of the specified parameter objects are located in heap memory.

- **IllegalAssignmentError**—when this cannot hold references to the specified parameter objects, or the parameter objects cannot hold a reference to this.

- **IllegalThreadStateException**—when the schedulable’s scheduler prohibits this parameter change at this time due to the state of the schedulable.

**Returns**

true, when the resulting system is feasible and the changes are made. False, when the resulting system is not feasible and no changes are made.

**Deprecated** as of RTSJ 2.0 The framework for feasibility analysis is inadequate

### setIfFeasible(ReleaseParameters, MemoryParameters, ProcessingGroupParameters)

**Signature**

public boolean setIfFeasible(ReleaseParameters release, MemoryParameters memory, ProcessingGroupParameters group)

**Description**

This method first performs a feasibility analysis using the proposed parameter objects as replacements for the current parameters of this. When the resulting system is feasible, this method replaces the current parameters of this with the proposed ones.
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.

This method does not require that the schedulable be in the feasibility set before it is called. When it is not initially a member of the feasibility set it will be added when the resulting system is feasible.

**Parameters**

release—The release parameters to use. 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`.)

memory—The memory parameters to use. 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`.)

group—The processing group parameters to use. 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`.)

**Throws**

`IllegalRealtimeArgumentException`—when the parameter values are not compatible with the schedulable’s scheduler. Also when this schedulable may not use the heap and any of the specified parameter objects are located in heap memory.

`IllegalAssignmentError`—when this cannot hold references to the specified parameter objects, or the parameter objects cannot hold a reference to this.

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

**Returns**

true, when the resulting system is feasible and the changes are made. False, when the resulting system is not feasible and no changes are made.

**Deprecated** as of RTSJ 2.0 The framework for feasibility analysis is inadequate
RealtimeThread

javax.realtime B.2

ProcessingGroupParameters group)

Description
This method first performs a feasibility analysis using the proposed parameter objects as replacements for the current parameters of this. When the resulting system is feasible, this method replaces the current parameters of this with the proposed ones.

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.

This method does not require that the schedulable be in the feasibility set before it is called. When it is not initially a member of the feasibility set it will be added when the resulting system is feasible.

Parameters
release—The release parameters to use. 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.)
group—The processing group parameters to use. 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.)

Throws
 IllegalRealtimeArgumentException—when the parameter values are not compatible with the schedulable’s scheduler. Also when this schedulable may not use the heap and any of the specified parameter objects are located in heap memory.
 IllegalAssignmentError—when this cannot hold references to the specified parameter objects, or the parameter objects cannot hold a reference to this.
 IllegalThreadStateException—when the schedulable’s scheduler prohibits this parameter change at this time due to the state of the schedulable.

Returns
true, when the resulting system is feasible and the changes are made. False, when the resulting system is not feasible and no changes are made.

Deprecated as of RTSJ 2.0 The framework for feasibility analysis is inadequate.

setIfFeasible(SchedulingParameters, ReleaseParameters, MemoryParameters)
Signature

public boolean setIfFeasible(SchedulingParameters scheduling,
ReleaseParameters release,
MemoryParameters memory)

Description

This method first performs a feasibility analysis using the proposed parameter objects as replacements for the current parameters of this. When the resulting system is feasible, this method replaces the current parameters of this with the proposed ones.

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.

This method does not require that the schedulable be in the feasibility set before it is called. When it is not initially a member of the feasibility set it will be added when the resulting system is feasible.

Parameters

scheduling—The scheduling parameters to use. 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.)

release—The release parameters to use. 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.)

memory—The memory parameters to use. 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.)

Throws

IllegalArgumentException—when the parameter values are not compatible with the schedulable’s scheduler. Also when this schedulable may not use the heap and any of the specified parameter objects are located in heap memory.

IllegalAssignmentError—when this cannot hold references to the specified parameter objects, or the parameter objects cannot hold a reference to this.

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

Returns
true, when the resulting system is feasible and the changes are made. False, when
the resulting system is not feasible and no changes are made.

Deprecated as of RTSJ 2.0 The framework for feasibility analysis is inadequate

`setIfFeasible(SchedulingParameters, ReleaseParameters, MemoryParameters, ProcessingGroupParameters)`

**Signature**

```java
public boolean setIfFeasible(SchedulingParameters scheduling,
ReleaseParameters release,
MemoryParameters memory,
ProcessingGroupParameters group)
```

**Description**

This method first performs a feasibility analysis using the proposed parameter
objects as replacements for the current parameters of this. When the resulting
system is feasible, this method replaces the current parameters of this with the
proposed ones.

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.

This method does not require that the schedulable be in the feasibility set
before it is called. When it is not initially a member of the feasibility set it will
be added when the resulting system is feasible.

**Parameters**

- `scheduling`—The scheduling parameters to use. 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`.)

- `release`—The release parameters to use. 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`.)

- `memory`—The memory parameters to use. 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`.)

- `group`—The processing group parameters to use. 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.)

**Throws**

*IllegalRealtimeArgumentException*—when the parameter values are not compatible with the schedulable’s scheduler. Also when this schedulable may not use the heap and any of the specified parameter objects are located in heap memory.

*IllegalAssignmentError*—when *this* cannot hold references to the specified parameter objects, or the parameter objects cannot hold a reference to *this*.

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

**Returns**

true, when the resulting system is feasible and the changes are made. False, when the resulting system is not feasible and no changes are made.

**Deprecated** as of RTSJ 2.0 The framework for feasibility analysis is inadequate

### setMemoryParametersIfFeasible(MemoryParameters)

**Signature**

```java
public boolean setMemoryParametersIfFeasible(MemoryParameters memory)
```

**Description**

This method first performs a feasibility analysis using the proposed parameter object as replacement for the current parameter of *this*. When the resulting system is feasible, this method replaces the current parameter of *this* with the proposed one.

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.

This method does not require that the schedulable be in the feasibility set before it is called. When it is not initially a member of the feasibility set it will be added when the resulting system is feasible.

**Parameters**

*memory*—The memory parameters to use. 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.)

**Throws**
RealtimeThread

IllegalRealtimeArgumentException—when the parameter value is not compatible with the schedulable’s scheduler. Also when this schedulable may not use the heap and the specified parameter object is located in heap memory.

IllegalAssignmentError—when this cannot hold a reference to the specified parameter object, or the parameter object cannot hold a reference to this.

IllegalThreadStateException—when the schedulable’s scheduler prohibits the changing of the memory parameter at this time due to the state of the schedulable.

Returns

true, when the resulting system is feasible and the changes are made. False, when the resulting system is not feasible and no changes are made.

Deprecated as of RTSJ 2.0 The framework for feasibility analysis is inadequate

setProcessingGroupParameters(ProcessingGroupParameters)

Signature

public void setProcessingGroupParameters(ProcessingGroupParameters group)

Description

Sets the ProcessingGroupParameters 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.

Parameters

group—A ProcessingGroupParameters object which will 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.)

Throws

IllegalRealtimeArgumentException—when group is not compatible with the scheduler for this schedulable object. Also when this schedulable may not use the 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.

Deprecated

setProcessingGroupParametersIfFeasible (ProcessingGroupParameters)

Signature
   public boolean
   setProcessingGroupParametersIfFeasible(ProcessingGroupParameters group)

Description
   This method first performs a feasibility analysis using the proposed parameter object as replacement for the current parameter of this. When the resulting system is feasible, this method replaces the current parameter of this with the proposed one.
   
   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.
   
   This method does not require that the schedulable be in the feasibility set before it is called. When it is not initially a member of the feasibility set it will be added when the resulting system is feasible.

Parameters
   group—The processing group parameters to use. 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.)

Throws
   IllegalRealtimeArgumentException—when the parameter value is not compatible with the schedulable’s scheduler. Also when this schedulable may not use the heap and the specified parameter object is located in heap memory.
   
   IllegalAssignmentError—when this cannot hold a reference to the specified parameter object, or the parameter object 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.

Returns
true, when the resulting system is feasible and the changes are made. False, when the resulting system is not feasible and no changes are made.

Deprecated as of RTSJ 2.0 The framework for feasibility analysis is inadequate

`setReleaseParametersIfFeasible(ReleaseParameters)`

**Signature**

```java
public boolean setReleaseParametersIfFeasible(ReleaseParameters release)
```

**Description**

This method first performs a feasibility analysis using the proposed parameter object as replacement for the current parameter of this. When the resulting system is feasible, this method replaces the current parameter of this with the proposed one.

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.

This method does not require that the schedulable be in the feasibility set before it is called. When it is not initially a member of the feasibility set it will be added when the resulting system is feasible.

**Parameters**

- `release`—The release parameters to use. 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`.)

**Throws**

- `IllegalRealtimeArgumentException`—when the parameter value is not compatible with the schedulable’s scheduler. Also when this schedulable may not use the heap and the specified parameter object is located in heap memory.
- `IllegalAssignmentError`—when `this` cannot hold a reference to the specified parameter object, or the parameter object cannot hold a reference to `this`.
- `IllegalThreadStateException`—when the schedulable’s scheduler prohibits the changing of the release parameter at this time due to the state of the schedulable.

**Returns**

true, when the resulting system is feasible and the changes are made. False, when the resulting system is not feasible and no changes are made.
Deprecated as of RTSJ 2.0 The framework for feasibility analysis is inadequate

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

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

**Description**
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`.

**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` 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`.
- `release`—A reference to the `ReleaseParameters` 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`.)
- `memoryParameters`—A reference to the `MemoryParameters` 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`.)
- `group`—A reference to the `ProcessingGroupParameters` 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`.)

**Throws**
- `IllegalRealtimeArgumentException`—when `scheduler` is null or the parameter values are not compatible with `scheduler`. Also thrown when this schedulable may not use the 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.

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

Deprecated since RTSJ 2.0

setSchedulingParametersIfFeasible(SchedulingParameters)

Signature

public boolean setSchedulingParametersIfFeasible(SchedulingParameters scheduling)

Description

This method first performs a feasibility analysis using the proposed parameter object as replacement for the current parameter of this. When the resulting system is feasible, this method replaces the current parameter of this with the proposed one.

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.

This method does not require that the schedulable be in the feasibility set before it is called. When it is not initially a member of the feasibility set it will be added when the resulting system is feasible.

Parameters

scheduling—The scheduling parameters to use. 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.)

Throws

IllegalRealtimeArgumentException—when the parameter value is not compatible with the schedulable’s scheduler. Also when this schedulable may not use the heap and the specified parameter object is located in heap memory.

IllegalAssignmentError—when this cannot hold a reference to the specified parameter object, or the parameter object 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.

Returns
ture, when the resulting system is feasible and the changes are made. False, when the resulting system is not feasible and no changes are made.

Deprecated as of RTSJ 2.0 The framework for feasibility analysis is inadequate

B.2.2.31 RelativeTime

public class RelativeTime

    The following elements of RelativeTime are deprecated. The required elements are documented in Section 9.3.1.3 above.

B.2.2.31.1 Constructors

RelativeTime(long, int, Clock)

Signature

    public
    RelativeTime(long millis,
                 int nanos,
                 Clock clock)
    throws IllegalRealtimeArgumentException

Description

Constructs a RelativeTime object representing an interval based on the parameter millis plus the parameter nanos. The construction is subject to millis and nanos parameter normalization. When there is an overflow in the millisecond component when normalizing then an IllegalRealtimeArgumentException will be thrown.

The clock association is made with the clock parameter. When clock is null the association is made with the default realtime clock.
Available since RTSJ 1.0.1

Deprecated since version 2.0

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.
- **clock**—The clock providing the association for the newly constructed object.

Throws

- **IllegalRealtimeArgumentException**—when there is an overflow in the millisecond component when normalizing.

**RelativeTime(RelativeTime, Clock)**

Signature

```java
public RelativeTime(RelativeTime time,
                    Clock clock)
    throws IllegalRealtimeArgumentException
```

Description

Makes a new *RelativeTime* object from the given *RelativeTime* object.

The clock association is made with the *clock* parameter. When *clock* is null the association is made with the default realtime clock.

Available since RTSJ 1.0.1

Deprecated since version 2.0

Parameters

- **time**—The *RelativeTime* object which is the source for the copy.
- **clock**—The clock providing the association for the newly constructed object.

Throws

- **IllegalRealtimeArgumentException**—when the *time* parameter is null.
RelativeTime(Clock)

Signature
public RelativeTime(Clock clock)

Description
Equivalent to new RelativeTime(0,0,clock).
The clock association is made with the clock parameter. When clock is null the association is made with the default realtime clock.

Available since RTSJ 1.0.1

Deprecated since version 2.0

Parameters
clock—The clock providing the association for the newly constructed object.

B.2.2.31.2 Methods

absolute(Clock)

Signature
public javax.realtime.AbsoluteTime absolute(Clock clock)
throws ArithmeticException

Description
Converts the time of this to an absolute time, using the given instance of Clock to determine the current time. The calculation is the current time indicated by the given instance of Clock plus the interval given by this. When clock is null the realtime clock is assumed. A destination object is allocated for the result. The clock association of the result is with the clock passed as a parameter.

Parameters
clock—The instance of Clock used to convert the time of this into absolute time, and the new clock association for the result.

Throws
1162 RTSJ 2.0 (Draft 66)
AbsoluteTime conversion in a newly allocated object, associated with the
clock parameter.

Deprecated since version 2.0

absolute(Clock, AbsoluteTime)

Signature
    public javax.realtime.AbsoluteTime
    absolute(Clock clock,
             AbsoluteTime dest)
    throws ArithmeticException

Description
    Converts the time of this to an absolute time, using the given instance of Clock
to determine the current time. The calculation is the current time indicated by
the given instance of Clock plus the interval given by this. When clock is null
the default realtime clock is assumed. When dest is null, a destination object
is allocated for the result. The clock association of the result is with the clock
passed as a parameter.

Parameters
    clock—The instance of Clock used to convert the time of this into absolute time,
            and the new clock association for the result.
    dest—When dest is not null, the result is placed there and returned.

Throws
    ArithmeticException—when the result does not fit in the normalized format.

Returns
    the AbsoluteTime conversion in dest when dest is not null, otherwise the result
    is returned in a newly allocated object. The result is associated with the clock
    parameter.

Deprecated since version 2.0

relative(Clock)
public javax.realtime.RelativeTime
relative(Clock clock)

Description
Returns a copy of this. 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 RelativeTime is needed in this case. The clock association of the result is with the clock passed as a parameter. When clock is null the association is made with the realtime clock.

Parameters
clock—The clock parameter is used only as the new clock association with the result, since no conversion is needed.

Returns
the copy of this in a newly allocated RelativeTime object, associated with the clock parameter.

Deprecated since version 2.0

relative(Clock, RelativeTime)

Signature
public javax.realtime.RelativeTime
relative(Clock clock,
   RelativeTime dest)

Description
Returns a copy of this. When dest is not null, the result is placed there and returned. Otherwise, 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 RelativeTime is needed in this case. The clock association of the result is with the clock passed as a parameter. When clock is null the association is made with the realtime clock.

Parameters
clock—The clock parameter is used only as the new clock association with the result, since no conversion is needed.
dest—When dest is not null, the result is placed there and returned.

Returns
the copy of this in dest when dest is not null, otherwise the result is returned in a newly allocated object. It is associated with the clock parameter.
Deprecated since version 2.0

B.2.2.32 ReleaseParameters

public abstract class ReleaseParameters

The following elements of ReleaseParameters are deprecated. The required elements are documented in Section 6.3.3.11 above.

B.2.2.32.1 Methods

setIfFeasible(RelativeTime, RelativeTime)

Signature

public boolean
    setIfFeasible(RelativeTime cost,
                   RelativeTime deadline)

Description

This method first performs a feasibility analysis using the new cost, and deadline as replacements for the matching attributes of all schedulables associated with this release parameters object. When the resulting system is feasible, the method replaces the current scheduling characteristics of this release parameters object with the new scheduling characteristics. The change in the release characteristics, including the timing of the change, of any associated schedulables will take place under the control of their schedulers.

Parameters

cost—The proposed cost. Equivalent to RelativeTime(0,0) when null. (A new instance of RelativeTime is created in the memory area containing this ReleaseParameters instance). When null, the default value is a new instance of RelativeTime(0,0).

deadline—The proposed deadline. There is no default for deadline in this class. The default must be determined by the subclasses.

Throws

RTSJ 2.0 (Draft 66)
**IllegalRealtimeArgumentException**—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** or **deadline** cannot be stored in **this**.

**Returns**

true, when the resulting system is feasible and the changes are made, and false, when the resulting system is not feasible and no changes are made.

**Deprecated** as of RTSJ 2.0

### B.2.2.33 Scheduler

```java
public abstract class Scheduler

The following elements of Scheduler are deprecated. The required elements are documented in Section 6.3.3.14 above.

#### B.2.2.33.1 Methods

**addToFeasibility(Schedulable)**

**Signature**

```java
protected abstract boolean addToFeasibility(Schedulable schedulable)
```

**Description**

Informs this scheduler and cooperating facilities that the resource demands of the given instance of **Schedulable** will be considered in the feasibility analysis of the associated **Scheduler** until further notice. Whether the resulting system is feasible or not, the addition is completed. When the object is already included in the feasibility set, does nothing.

**Parameters**

- schedulable—A reference to the given instance of **Schedulable**

**Throws**

- **IllegalRealtimeArgumentException**—when **schedulable** is null, or when **schedulable** is not associated with **this**; that is **schedulable.getScheduler() != this**.
Returns

true, when the system is feasible after the addition, and false, when not.

Deprecated as of RTSJ 2.0 The framework for feasibility analysis is inadequate

**isFeasible**

**Signature**

```
public abstract boolean isFeasible()
```

**Description**

Queries the system about the feasibility of the system currently being considered. The definitions of "feasible" and "system" are the responsibility of the feasibility algorithm of the actual Scheduler subclass.

Returns

true, when the system is feasible; false, when not.

Deprecated as of RTSJ 2.0 The framework for feasibility analysis is inadequate

**setIfFeasible(Schedulable, ReleaseParameters, MemoryParameters)**

**Signature**

```
public abstract boolean setIfFeasible(Schedulable schedulable,
                                      ReleaseParameters release,
                                      MemoryParameters memory)
```

**Description**

This method first performs a feasibility analysis using the proposed parameter objects as replacements for the current parameters of Schedulable. When the resulting system is feasible, this method replaces the current parameters of Schedulable with the proposed ones. This method does not require that the schedulable be in the feasibility set before it is called. When it is not initially a member of the feasibility set it will be added when the resulting system is feasible.

**Parameters**
schedulable—The schedulable for which the changes are proposed.
release—The proposed release parameters. When null, the default value of this scheduler is used (a new object is created when the default value is not null). (See PriorityScheduler.)
memory—The proposed memory parameters. When null, the default value of this scheduler is used (a new object is created when the default value is not null). (See PriorityScheduler.)

Throws
IllegalRealtimeArgumentException—when Schedulable is null, or Schedulable is not associated with this scheduler, or the proposed parameters are not compatible with this scheduler.
IllegalAssignmentError—when Schedulable cannot hold references to the proposed parameter objects, or the parameter objects cannot hold a reference to Schedulable.
IllegalThreadStateException—when the new ReleaseParameters changes Schedulable from periodic scheduling to some other protocol and Schedulable is currently waiting for the next release in RealtimeThread.waitForNextPeriod() or RealtimeThread.waitForNextPeriodInterruptible().

Returns
true, when the resulting system is feasible and the changes are made; false, when the resulting system is not feasible and no changes are made.

Deprecated as of RTSJ 2.0 The framework for feasibility analysis is inadequate

setIfFeasible(Schedulable, ReleaseParameters, MemoryParameters, ProcessingGroupParameters)

Signature
public abstract boolean
setIfFeasible(Schedulable schedulable,
             ReleaseParameters release,
             MemoryParameters memory,
             ProcessingGroupParameters group)

Description
This method first performs a feasibility analysis using the proposed parameter objects as replacements for the current parameters of Schedulable. When the resulting system is feasible, this method replaces the current parameters of Schedulable with the proposed ones.
This method does not require that the schedulable be in the feasibility set before it is called. When it is not initially a member of the feasibility set it will be added when the resulting system is feasible.

**Parameters**

- `schedulable`—The schedulable for which the changes are proposed.
- `release`—The proposed release parameters. When `null`, the default value of this scheduler is used (a new object is created when the default value is not `null`). (See `PriorityScheduler`.)
- `memory`—The proposed memory parameters. When `null`, the default value of this scheduler is used (a new object is created when the default value is not `null`). (See `PriorityScheduler`.)
- `group`—The proposed processing group parameters. When `null`, the default value of this scheduler is used (a new object is created when the default value is not `null`). (See `PriorityScheduler`.)

**Throws**

- `IllegalRealtimeArgumentException`—when `Schedulable` is `null`, or `Schedulable` is not associated with this scheduler, or the proposed parameters are not compatible with this scheduler.
- `IllegalAssignmentError`—when `Schedulable` cannot hold references to the proposed parameter objects, or the parameter objects cannot hold a reference to `Schedulable`.
- `IllegalThreadStateException`—when the new release parameters change `Schedulable` from periodic scheduling to some other protocol and `Schedulable` is currently waiting for the next release in `RealtimeThread.waitForNextPeriod()` or `RealtimeThread.waitForNextPeriodInterruptible()`.

**Returns**

- `true`, when the resulting system is feasible and the changes are made; `false`, when the resulting system is not feasible and no changes are made.

**Deprecated** as of RTSJ 2.0 The framework for feasibility analysis is inadequate
Scheduler

SchedulingParameters scheduling,
ReleaseParameters release,
MemoryParameters memory,
ProcessingGroupParameters group)

Description

This method first performs a feasibility analysis using the proposed parameter objects as replacements for the current parameters of Schedulable. When the resulting system is feasible, this method replaces the current parameters of Schedulable with the proposed ones.

This method does not require that the schedulable be in the feasibility set before it is called. When it is not initially a member of the feasibility set it will be added when the resulting system is feasible.

Parameters

schedulable—The schedulable for which the changes are proposed.

scheduling—The proposed scheduling parameters. When null, the default value of this scheduler is used (a new object is created when the default value is not null). (See PriorityScheduler.)

release—The proposed release parameters. When null, the default value of this scheduler is used (a new object is created when the default value is not null). (See PriorityScheduler.)

memory—The proposed memory parameters. When null, the default value of this scheduler is used (a new object is created when the default value is not null). (See PriorityScheduler.)

group—The proposed processing group parameters. When null, the default value of this scheduler is used (a new object is created when the default value is not null). (See PriorityScheduler.)

Throws

IllegalRealtimeArgumentException—when Schedulable is null, or Schedulable is not associated with this scheduler, or the proposed parameters are not compatible with this scheduler.

IllegalAssignmentError—when Schedulable cannot hold references to the proposed parameter objects, or the parameter objects cannot hold a reference to Schedulable.

IllegalThreadStateException—when the new release parameters change Schedulable from periodic scheduling to some other protocol and Schedulable is currently waiting for the next release in RealtimeThread.waitForNextPeriod() or RealtimeThread.waitForNextPeriodInterruptible().

Returns
true, when the resulting system is feasible and the changes are made; false, when the resulting system is not feasible and no changes are made.

**Deprecated** as of RTSJ 2.0 The framework for feasibility analysis is inadequate

### removeFromFeasibility(Schedulable)

**Signature**

```
protected abstract boolean removeFromFeasibility(Schedulable schedulable)
```

**Description**

Informs this scheduler and cooperating facilities that the resource demands of the given instance of Schedulable should no longer be considered in the feasibility analysis of the associated Scheduler. Whether the resulting system is feasible or not, the removal is completed.

**Parameters**

- `schedulable`—A reference to the given instance of Schedulable

**Throws**

- `IllegalRealtimeArgumentException`—when `schedulable` is null.

**Returns**

- true, when the removal was successful, and false, when the schedulable cannot be removed from the scheduler’s feasibility set; e.g., the schedulable is not part of the scheduler’s feasibility set.

**Deprecated** as of RTSJ 2.0 The framework for feasibility analysis is inadequate

### fireSchedulable(Schedulable)

**Signature**

```
public abstract void fireSchedulable(Schedulable schedulable)
```

**Description**

Triggers the execution of a schedulable (like an AsyncEventHandler).

**Parameters**

- `schedulable`—The schedulable to make active. When `null`, nothing happens.

**Throws**

- RTSJ 2.0 (Draft 66) 1171
B Deprecated APIs

ScopedMemory

UnsupportedRealtimeOperationException—when the scheduler cannot release schedulable for execution.

Deprecated RTSJ 2.0

B.2.2.34 ScopedMemory

public abstract class ScopedMemory

Inheritance
java.lang.Object
MemoryArea
ScopedMemory

Description
Equivalent to and superseded by javax.realtime.memory.ScopedMemory.

Deprecated in RTSJ 2.0; moved to package javax.realtime.memory

B.2.2.34.1 Constructors

ScopedMemory(long, Runnable)

Signature
public
ScopedMemory(long size,
Runnable logic)

Description
Creates a new ScopedMemory area with the given parameters.

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.
ScopedMemory

Throws
- IllegalArgumentException—when size is less than zero.
- IllegalAssignmentError—when storing logic in this would violate the assignment rules.
- OutOfMemoryStaticError—when there is insufficient memory for the ScopedMemory object or for its allocation area in its backing store.

ScopedMemory(SizeEstimator, Runnable)

Signature
public ScopedMemory(SizeEstimator size, Runnable logic)

Description
Equivalent to ScopedMemory(long, Runnable) with the argument list (size.getEstimate(), logic).

Parameters
- size—The size of the new ScopedMemory area estimated by an instance of SizeEstimator.
- 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.
- OutOfMemoryStaticError—when there is insufficient memory for the ScopedMemory object or for its allocation area in its backing store.
- IllegalAssignmentError—when storing logic in this would violate the assignment rules.

ScopedMemory(long)

Signature
public ScopedMemory(long size)

RTSJ 2.0 (Draft 66) 1173
**B Deprecated APIs**

**ScopedMemory**

*Description*

Equivalent to `ScopedMemory(long, Runnable)` with the argument list `(size, null)`.

*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 its allocation area in its backing store.

**ScopedMemory(SizeEstimator)**

*Signature*

```java
public ScopedMemory(SizeEstimator size)
```

*Description*

Equivalent to `ScopedMemory(long, Runnable)` with the argument list `(size.getEstimate(), null)`.

*Parameters*

- `size` — The size of the new `ScopedMemory` area estimated by an instance of `SizeEstimator`.

*Throws*

- `IllegalArgumentException` — when `size` is `null`, or `size.getEstimate()` is negative.
- `OutOfMemoryError` — when there is insufficient memory for the `ScopedMemory` object or for its allocation area in its backing store.

**B.2.2.34.2 Methods**

**enter**

*Signature*

```java
1174 RTSJ 2.0 (Draft 66)
```
public void enter()

Description

Associates 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) or the enter method exits.

Throws

ScopedCycleException—when this invocation would break the single parent rule.

ThrowBoundaryError—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, so the JVM cannot be permitted to deliver the exception. The ThrowBoundaryError 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—null

MemoryAccessError—when caller is a schedulable that may not use the heap and this memory area’s logic value is allocated in heap memory.

enter(Runnable)

Signature

public void enter(Runnable logic)

Description

Associates 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) or the enter method exits.
B Deprecated APIs

ScopedMemory

Parameters
logic—The Runnable object whose run() method should be invoked.

Throws
ScopedCycleException—when this invocation would break the single parent rule.

ThrowBoundaryError—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, so the JVM cannot be permitted to deliver the exception. The ThrowBoundaryError 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.

IllegalStateException—null

eexecuteInArea(Runnable)

Signature
public void executeInArea(Runnable logic)

Description
Executes 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.

Parameters
logic—The runnable object whose run() method should be executed.

Throws
IllegalThreadStateException—when the caller context in not an instance of Schedulable.

InaccessibleAreaException—when the memory area is not in the schedulable’s scope stack.

IllegalStateException—when the caller is a schedulable and logic is null.

getMaximumSize

1176 RTSJ 2.0 (Draft 66)
ScopedMemory

javax.realtime B.2

**Signature**

```java
public long getMaximumSize()
```

**Description**

Gets the maximum size this memory area can attain. If this is a fixed size memory area, the returned value will be equal to the initial size.

**Returns**

the maximum size attainable.

**getPortal**

**Signature**

```java
public java.lang.Object getPortal()
```

**Description**

Obtains 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.

**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 context in not an instance of Schedulable.

**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.

**getReferenceCount**

**Signature**

```java
public int getReferenceCount()
```

RTSJ 2.0 (Draft 66) 1177
Description

Returns the reference count of this ScopedMemory. Note that 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.

Returns

the reference count of this ScopedMemory.

join

Signature

public void
join()
throws InterruptedException

Description

Waits until the reference count of this ScopedMemory goes down to zero. Returns immediately when the memory is unreferenced.

Throws

InterruptedException—When this schedulable is interrupted by RealtimeThread.interrupt() or AsynchronouslyInterruptedException. fire() while waiting for the reference count to go to zero.

IllegalThreadStateException—when the caller context in not an instance of Schedulable.

join(HighResolutionTime)

Signature

public void
join(HighResolutionTime<?> time)
throws InterruptedException

Description

Waits at most until the time designated by the time parameter for the reference count of this ScopedMemory to drop to zero. Returns immediately when the memory area is unreferenced.

Since the time is expressed as a HighResolutionTime, 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 \texttt{time}. The delay time may be relative or absolute. When relative, then the delay is the amount of time given by \texttt{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 \texttt{join} returns immediately.

\textit{Parameters}

\texttt{time}—When this time is an absolute time, the wait is bounded by that point in time. When the time is a relative time, the wait is bounded by a the specified interval from some time between the time \texttt{join} is called and the time it starts waiting for the reference count to reach zero.

\textit{Throws}

\texttt{InterruptedException}—When this schedulable is interrupted by \texttt{RealtimeThread.interrupt()} or \texttt{AsynchronouslyInterruptedException.fire()} while waiting for the reference count to go to zero.

\texttt{IllegalThreadStateException}—when the caller context in not an instance of \texttt{Schedulable}.

\texttt{IllegalArgumentException}—when the caller is a schedulable and \texttt{time} is \texttt{null}.

\texttt{UnsupportedOperationException}—when the wait operation is not supported using the clock associated with \texttt{time}.

\textbf{joinAndEnter}

\textit{Signature}

\begin{verbatim}
public void joinAndEnter()
    throws InterruptedException
\end{verbatim}

\textit{Description}

In the error-free case, \texttt{joinAndEnter} combines \texttt{join();enter();} such that no \texttt{enter()} from another schedulable can intervene between the two method invocations. The resulting method will wait for the reference count on this \texttt{ScopedMemory} to reach zero, then enter the \texttt{ScopedMemory} and execute the \texttt{run} method from \texttt{logic} passed in the constructor. When no instance of \texttt{Runnable} was passed to the memory area’s constructor, the method throws \texttt{IllegalArgumentException} immediately.

When multiple threads are waiting in \texttt{joinAndEnter} family methods for a memory area, at most \textit{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.

**Throws**

- **InterruptedException**—When this schedulable is interrupted by `RealtimeThread.interrupt()` or `AsynchronouslyInterruptedException.fire()` 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**—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`, so the JVM cannot be permitted to deliver the exception. The `ThrowBoundaryError` 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(HighResolutionTime)`

**Signature**

```java
public void joinAndEnter(HighResolutionTime<?> time)
throws InterruptedException
```

**Description**

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 `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, 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.

Note that expiration of time may cause control to enter the memory area
before its reference count has gone to zero.

Parameters

- time—The time that bounds the wait.

Throws

- ThrowBoundaryError—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, so the JVM cannot
  be permitted to deliver the exception. The ThrowBoundaryError is allocated
  in the current allocation context and contains information about the exception
  it replaces.

- InterruptedException—When this schedulable is interrupted by
  RealtimeThread.interrupt() or AsynchronouslyInterruptedException.
  fire() while waiting for the reference count to go to zero.

- IllegalThreadStateException—when the caller context is not an instance of
  Schedulable, or when this method is invoked during finalization of objects in
  scoped memory and entering this scoped memory area would force deletion of
  the instance that triggered finalization. This would include the scope containing
  the instance, and the scope (if any) containing the scope containing the instance.

- 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 null was supplied as logic value to the memory area’s constructor.

- UnsupportedOperationException—when the wait operation is not supported us-
  ing the clock associated with time.
MemoryAccessError—when the calling schedulable may not use the heap and this memory area’s logic value is allocated in heap memory.

### joinAndEnter(Runnable)

**Signature**

```java
public void joinAndEnter(Runnable logic)
throws InterruptedException
```

**Description**

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.

**Parameters**

- `logic`—The `Runnable` object which contains the code to execute.

**Throws**

- `InterruptedException`—When this schedulable is interrupted by `RealtimeThread.interrupt()` or `AsynchronouslyInterruptedException.fire()` 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`—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`, so the JVM cannot be permitted to deliver the exception. The `ThrowBoundaryError` 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(Runnable, HighResolutionTime)

**Signature**

```java
public void joinAndEnter(Runnable logic,
                          HighResolutionTime<?> time)
throws InterruptedException
```

**Description**

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, 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)`.

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.

**Parameters**

- logic—The Runnable object which contains the code to execute.
- time—The time that bounds the wait.

**Throws**

RTSJ 2.0 (Draft 66)
**InterruptedException**—When this schedulable is interrupted by `RealtimeThread.interrupt()` or `AsynchronouslyInterruptedException.fire()` while waiting for the reference count to go to zero.

**IllegalThreadStateException**—When the execution context in not an instance of `Schedulable`, or when this method is invoked during finalization of objects in scoped memory and entering this scoped memory area would force deletion of the task that triggered finalization. This would include the scope containing the task, and the scope (if any) containing the scope containing the task.

**ThrowBoundaryError**—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`, so the JVM cannot be permitted to deliver the exception. The `ThrowBoundaryError` 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`.

### newArray(Class, int)

**Signature**

```java
public java.lang.Object newArray(java.lang.Class<?> type, int number)
```

**Description**

Allocates an array of the given type in this memory area. This method may be concurrently used by multiple threads.

**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`—null
ScopedMemory

**OutOfMemoryStaticError**—when space in the memory area is exhausted.
**IllegalThreadStateException**—when the caller context is not an instance of `Schedulable`.
**InaccessibleAreaException**—when the memory area is not in the schedulable’s scope stack.

**Returns**
a new array of class type, of number elements.

**newInstance(Class)**

**Signature**
```java
public T
newInstance(java.lang.Class<T> type)
throws IllegalAccessException,
InstantiationException
```

**Description**
Allocates an object in this memory area. This method may be concurrently used by multiple threads.

**Parameters**
`type`—The class of which to create a new instance.

**Throws**
**IllegalAccessException**—The class or initializer is inaccessible.
**IllegalArgumentException**—null
**ExceptionInInitializerError**—when an unexpected exception has occurred in a static initializer.
**OutOfMemoryStaticError**—when space in the memory area is exhausted.
**InstantiationException**—when the specified class object could not be instantiated. Possible causes are it is an interface, it is abstract, or it is an array.
**IllegalThreadStateException**—when the caller context is not an instance of `Schedulable`.
**InaccessibleAreaException**—when the memory area is not in the schedulable’s scope stack.

**Returns**
a new instance of class type.
**newInstance(Constructor, Object)**

*Signature*

```java
public T newInstance(java.lang.reflect.Constructor<T> c,
                   java.lang.Object[] args)
throws IllegalAccessException,
       InstantiationException,
       InvocationTargetException
```

*Description*

Allocates an object in this memory area. This method may be concurrently used
by multiple threads.

*Parameters*

- `c`—The constructor for the new instance.
- `args`—An array of arguments to pass to the constructor.

*Throws*

- `IllegalAccessException`—when the class or initializer is inaccessible under Java access control.
- `InstantiationException`—when the specified class object could not be instantiated. Possible causes are it is an interface, it is abstract, it is an array.
- `OutOfMemoryStaticError`—when space in the memory area is exhausted.
- `IllegalThreadStateException`—when the caller context is not an instance of `Scheduled`. 
- `InvocationTargetException`—when the underlying constructor throws an exception.
- `InaccessibleAreaException`—when the memory area is not in the schedulable’s scope stack.

*Returns*

A new instance of the object constructed by `c`.

**setPortal(Object)**

*Signature*

```java
public void setPortal(Object object)
```

1186 RTSJ 2.0 (Draft 66)
SporadicParameters

Description

Sets the portal object of the memory area represented by this instance of Scoped-Memory to the given object. The object must have been allocated in this Scoped-Memory instance.

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 context in not an instance of Schedulable.

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

Signature

public java.lang.String toString()

Description

Returns a user-friendly representation of this ScopedMemory of the form Scoped-Memory#<num> where <num> is a number that uniquely identifies this scoped memory area.

Returns

The string representation

B.2.2.35 SporadicParameters

public class SporadicParameters

The following elements of SporadicParameters are deprecated. The required elements are documented in Section 6.3.3.16 above.
B.2.2.35.1 Fields

mitViolationExcept

```java
public static final java.lang.String mitViolationExcept
```

**Description**

Represents the “EXCEPT” policy for dealing with minimum interarrival time violations. Under this policy, when an arrival time for any instance of `Schedulable` which has `this` as its instance of `ReleaseParameters` occurs at a time less then the minimum interarrival time defined here then the `fire()` method shall throw `MITViolationException`. 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` is bound then the arrival time is ignored.

**Deprecated** since RTSJ 2.0

mitViolationIgnore

```java
public static final java.lang.String mitViolationIgnore
```

**Description**

Represents the “IGNORE” policy for dealing with minimum interarrival time violations. Under this policy, when an arrival time for any instance of `Schedulable` which has `this` as its instance of `ReleaseParameters` 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.

**Deprecated** since RTSJ 2.0

mitViolationSave

```java
public static final java.lang.String mitViolationSave
```

**Description**

RTSJ 2.0 (Draft 66)
SporadicParameters

Represents the “SAVE” policy for dealing with minimum interarrival time violations. Under this policy the arrival time for any instance of Schedulable which has this as its instance of ReleaseParameters 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.

**Deprecated** since RTSJ 2.0

mitViolationReplace

```java
public static final java.lang.String mitViolationReplace
```

**Description**

Represents the “REPLACE” policy for dealing with minimum interarrival time violations. Under this policy when an arrival time for any instance of Schedulable which has this as its instance of ReleaseParameters 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.

**Deprecated** since RTSJ 2.0

---

### B.2.2.35.2 Methods

**setMitViolationBehavior(String)**

**Signature**

```java
public void setMitViolationBehavior(String behavior)
```

**Description**

Sets the behavior of the arrival time queue for 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()).

**Parameters**
behavior—A string representing the behavior.

Throws

IllegalRealtimeArgumentException—when behavior is not one of the final MIT violation behavior values defined in this class.

Deprecated since RTSJ 2.0 and replaced by setMinimumInterarrivalPolicy.

getMitViolationBehavior

Signature

public java.lang.String
getMitViolationBehavior()

Description

Gets the arrival time queue behavior in the event of a minimum interarrival time violation.

Returns

the minimum interarrival time violation behavior as a string.

Deprecated since RTSJ 2.0 and replaced by getMinimumInterarrivalPolicy.

setIfFeasible(RelativeTime, RelativeTime)

Signature

public boolean
setIfFeasible(RelativeTime cost,
RelativeTime deadline)

Description

This method first performs a feasibility analysis using the new cost, and deadline as replacements for the matching attributes of this. When the resulting system is feasible, the method replaces the current scheduling characteristics, of this with the new scheduling characteristics.

Parameters

cost—The proposed cost used to determine when any particular object exceeds cost. When null, the default value is a new instance of RelativeTime(0,0).
deadline—The proposed deadline. When null, the default value is a new instance of RelativeTime(mit).
SporadicParameters

Throws

**IllegalRealtimeArgumentException**—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 values are incompatible with the scheduler for any of the schedulables which are presently using this parameter object.

**IllegalAssignmentError**—when `cost` or `deadline` cannot be stored in this.

Returns

`true`, when the resulting system is feasible and the changes are made; `false`, when the resulting system is not feasible and no changes are made.

Deprecated as of RTSJ 2.0 The framework for feasibility analysis is inadequate.

**setIfFeasible(RelativeTime, RelativeTime, RelativeTime)**

Signature

```java
public boolean setIfFeasible(RelativeTime interarrival,
                             RelativeTime cost,
                             RelativeTime deadline)
```

Description

This method first performs a feasibility analysis using the new interarrival, cost and deadline attributes as replacements for the matching attributes of this. When the resulting system is feasible the method replaces the current attributes with the new ones.

Changes to a `SporadicParameters` instance effect subsequent arrivals.

Parameters

- **interarrival**—The proposed interarrival time. There is no default value. When `minInterarrival` is `null` an illegal argument exception is thrown.
- **cost**—The proposed cost. When `null`, the default value is a new instance of `RelativeTime(0,0)`.
- **deadline**—The proposed deadline. When `null`, the default value is a new instance of `RelativeTime(mit)`.

Throws

**IllegalRealtimeArgumentException**—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.

**IllegalAssignmentError**—when `interarrival`, `cost` or `deadline` cannot be stored in this.
Returns

true, when the resulting system is feasible and the changes are made; false, when
the resulting system is not feasible and no changes are made.

Deprecated as of RTSJ 2.0 The framework for feasibility analysis is inadequate

B.2.2.36  ThrowBoundaryError

public class ThrowBoundaryError

The following elements of ThrowBoundaryError are deprecated. The required
elements are documented in Section 16.2.2.36 above.

B.2.2.36.1  Constructors

ThrowBoundaryError(String)

Signature

public

ThrowBoundaryError(String description)

Description

A descriptive constructor for ThrowBoundaryError.

Deprecated since RTSJ 2.0; application code should use get() instead.

Parameters

description—The reason for throwing this error.

B.2.2.37  Timed

public class Timed

Inheritance

java.lang.Object

1192  RTSJ 2.0 (Draft 66)
Description

Creates a scope in a Schedulable 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 when 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.

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);

Deprecated in RTSJ 2.0; moved to package javax.realtime.control

B.2.2.37.1 Constructors

Timed(HighResolutionTime)

Signature

public
Timed(HighResolutionTime<? super T> time)
throws IllegalRealtimeArgumentException, UnsupportedRealtimeOperationException

Description

Creates an instance of Timed with a timer set to time. When the time is in the past the AsynchronouslyInterruptedException mechanism is activated immediately after or when the doInterruptible method is called.
Parameters
time—When time is a RelativeTime value, it is the interval of time between the invocation of doInterruptible and the time when the schedulable is asynchronously interrupted. When time is an AbsoluteTime value, the timer asynchronously interrupts at this time (assuming the timer has not been cancelled).

Throws
IllegalRealtimeArgumentException—when time is null.
UnsupportedRealtimeOperationException—when time is not based on a Clock.

B.2.2.37.2 Methods

doInterruptible(Interruptible)

Signature
public boolean doInterruptible(Interruptible logic)

Description
Executes a timeout method by starting the timer and executing the run() method of the given Interruptible object.

Parameters
logic—An instance of an Interruptible whose run() method will be called.

Throws
IllegalRealtimeArgumentException—when logic is null.
IllegalThreadStateException—null

Returns
true, when the method call completed normally, and false, when another call to doInterruptible has not completed.

resetTime(HighResolutionTime)

Signature
public void resetTime(HighResolutionTime<?> time)
**UnknownHappeningException**  

*Description*

Sets the timeout for the next invocation of `doInterruptible`.

*Parameters*

`time`—This can be an absolute time or a relative time. When `null` or not based on a `Clock`, the timeout is not changed.

**restart(HighResolutionTime)**

*Signature*

```java
public void restart(HighResolutionTime<?> time)
```

*Description*

Resets the timeout. When this `Timed` instance is executing, it adjusts the timeout to `time` and restarts the timer. When the instance is not executing, it adjusts the timeout for the next invocation.

*Parameters*

`time`—The new timeout.

*Throws*

- `IllegalRealtimeArgumentException`—when `time` is `null` or a relative time less than zero.
- `UnsupportedRealtimeOperationException`—when `time` is not based on a `Clock`

**Available since** RTSJ 2.0

---

**B.2.2.38 UnknownHappeningException**

```java
public class UnknownHappeningException
```

*Inheritance*

```
java.lang.Object  
java.lang.Throwable  
java.lang.Exception  
java.lang.RuntimeException  
UnknownHappeningException
```

*Description*

This exception is used to indicate a situation where an instance of `AsyncEvent` attempts to bind to a happening that does not exist.
**Deprecated APIs**

**UnsupportedPhysicalMemoryException**

**Deprecated** since RTSJ 2.0

**B.2.2.38.1 Constructors**

**UnknownHappeningException**

**Signature**

```java
public
UnknownHappeningException()
```

**Description**

A constructor for UnknownHappeningException.

**UnknownHappeningException(String)**

**Signature**

```java
public
UnknownHappeningException(String description)
```

**Description**

A descriptive constructor for UnknownHappeningException.

**Parameters**

description—The reason for throwing the exception.

**B.2.2.39 UnsupportedPhysicalMemoryException**

```java
public class UnsupportedPhysicalMemoryException

The following elements of UnsupportedPhysicalMemoryException are deprecated. The required elements are documented in Section 16.2.2.38 above.
```

1196 RTSJ 2.0 (Draft 66)
B.2.2.39.1 Constructors

UnsupportedPhysicalMemoryException(String)

Signature
public
UnsupportedPhysicalMemoryException(String description)

Description
A descriptive constructor for UnsupportedPhysicalMemoryException.

Deprecated since RTSJ 2.0; application code should use get() instead.

Parameters
description—The reason for throwing the exception.

B.2.2.40 VTMemory

public class VTMemory

Inheritance
java.lang.Object
  MemoryArea
    ScopedMemory
      VTMemory

Description
VTMemory is similar to LTMemory 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 as of RTSJ 2.0

B.2.2.40.1 Constructors
VTMemory(long, long, Runnable)

Signature
   public
   VTMemory(long initial,
            long maximum,
            Runnable logic)

Description
   Creates a VTMemory with the given parameters.

Parameters
   initial—The size in bytes of the memory to initially allocate for this area.
   maximum—The maximum size in bytes to which this memory area’s 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).

Throws
   IllegalArgumentException—when initial is greater than maximum, or when initial or maximum is less than zero.
   OutOfMemoryStaticError—when there is insufficient memory for the VTMemory object or for its allocation area in its backing store.
   IllegalAssignmentError—when storing logic in this would violate the assignment rules.

VTMemory(SizeEstimator, SizeEstimator, Runnable)

Signature
   public
   VTMemory(SizeEstimator initial,
            SizeEstimator maximum,
            Runnable logic)

Description
   Equivalent to VTMemory(long, long, Runnable) with the argument list (initial.getEstimate(), maximum.getEstimate(), logic).

Parameters
initial—The size in bytes of the memory to initially allocate for this area estimated by an instance of SizeEstimator.

maximum—The maximum size in bytes to which this memory area’s size may grow estimated by an instance of SizeEstimator.

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).

Throws

IllegalArgumentException—when initial is null, maximum is null, initial.getEstimate() is greater than maximum.getEstimate(), or when initial.getEstimate() is less than zero.

OutOfMemoryStaticError—when there is insufficient memory for the VTMemory object or for its allocation area in its backing store.

IllegalAssignmentError—when storing logic in this would violate the assignment rules.

VTMemory(long, long)

Signature

public
VTMemory(long initial,
long maximum)

Description

Equivalent to VTMemory(long, long, Runnable) with the argument list (initial, maximum, null).

Parameters

initial—The size in bytes of the memory to initially allocate for this area.

maximum—The maximum size in bytes to which this memory area’s size may grow.

Throws

IllegalArgumentException—when initial is greater than maximum or when initial or maximum is less than zero.

OutOfMemoryStaticError—when there is insufficient memory for the VTMemory object or for its allocation area in its backing store.
VTMemory(SizeEstimator, SizeEstimator)

**Signature**

```java
public VTMemory(SizeEstimator initial,
                SizeEstimator maximum)
```

**Description**

Equivalent to `VTMemory(long, long, Runnable)` with the argument list `(initial.getEstimate(), maximum.getEstimate(), null)`.

**Parameters**

- `initial`—The size in bytes of the memory to initially allocate for this area estimated by an instance of `SizeEstimator`.
- `maximum`—The maximum size in bytes to which this memory area’s size may grow estimated by an instance of `SizeEstimator`.

**Throws**

- `IllegalArgumentException`—when `initial` is `null`, `maximum` is `null`, `initial.getEstimate()` is greater than `maximum.getEstimate()`, or when `initial.getEstimate()` is less than zero.
- `OutOfMemoryStaticError`—when there is insufficient memory for the `VTMemory` object or for its allocation area in its backing store.

VTMemory(long, Runnable)

**Signature**

```java
public VTMemory(long size,
                Runnable logic)
```

**Description**

Equivalent to `VTMemory(long, long, Runnable)` with the argument list `(size, size, logic)`.

**Available since** RTSJ 1.0.1

**Parameters**

<table>
<thead>
<tr>
<th>Size</th>
<th>RTSJ Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>1200</td>
<td>RTSJ 2.0 (Draft 66)</td>
</tr>
</tbody>
</table>
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).

**Throws**

IllegalArgumentException—when size is less than zero.

OutOfMemoryStaticError—when there is insufficient memory for the VTMemory object or for its allocation area in its backing store.

IllegalAssignmentError—when storing logic in this would violate the assignment rules.

**VTMemory(SizeEstimator, Runnable)**

**Signature**

```java
class VTMemory
{
  public VTMemory(SizeEstimator size, Runnable logic)
```

**Description**

Equivalent to VTMemory(long, long, Runnable) with the argument list (size.getEstimate(), size.getEstimate(), logic).

**Available since** RTSJ 1.0.1

**Parameters**

size—An instance of SizeEstimator 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 size).

**Throws**

IllegalArgumentException—when size is null, or size.getEstimate() is less than zero.

OutOfMemoryStaticError—when there is insufficient memory for the VTMemory object or for its allocation area in its backing store.

IllegalAssignmentError—when storing logic in this would violate the assignment rules.
VTMemory(long)

Signature
   public
   VTMemory(long size)

Description
   Equivalent to VTMemory(long, long, Runnable) with the argument list (size, size, null).

Available since RTSJ 1.0.1

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.
   OutOfMemoryStaticError—when there is insufficient memory for the VTMemory object or for its allocation area in its backing store.

VTMemory(SizeEstimator)

Signature
   public
   VTMemory(SizeEstimator size)

Description
   Equivalent to VTMemory(long, long, Runnable) with the argument list (size.getEstimate(), size.getEstimate(), null).

Available since RTSJ 1.0.1

Parameters
   size—An instance of SizeEstimator used to give an estimate of the initial size. This memory must be committed before the completion of the constructor.

Throws
   IllegalArgumentException—when size is null, or size.getEstimate() is less than zero.
OutOfMemoryStaticError—when there is insufficient memory for the VTMemory object or for its allocation area in its backing store.

B.2.2.40.2 Methods

toString

Signature

public java.lang.String
toString()

Description

Creates a string representing this object. The string is of the form

{@code (VTMemory) ScopedMemory#<num>}

where <num> uniquely identifies the VTMemory area.

Returns

a string representing the value of this.

B.2.2.41 VTPhysicalMemory

public class VTPhysicalMemory

Inheritance

java.lang.Object
  MemoryArea
    ScopedMemory
      VTPhysicalMemory

Description

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 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

See Section RealtimeThread

See Section NoHeapRealtimeThread

**Deprecated** since RTSJ 2.0

### B.2.2.41.1 Constructors

**VTPhysicalMemory**(*Object, long, long, Runnable*)

**Signature**

```java
public VTPhysicalMemory(Object type,
                        long base,
                        long size,
                        Runnable logic)
```

**Description**

1204 RTSJ 2.0 (Draft 66)
Creates an instance of VTPhysicalMemory with the given parameters.

See Section PhysicalMemoryManager

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() 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.

RealtimeSecurityException—when the application does not 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 has been registered with the PhysicalMemoryManager.

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, long, SizeEstimator, Runnable)

Signature

class VTPhysicalMemory extends PhysicalMemory {  
  
  public VTPhysicalMemory(Object type,
  
  size, logic)

  return

  RTSJ 2.0 (Draft 66)
long base,
SizeEstimator size,
Runnable logic)

Description

Equivalent to VTPhysicalMemory(Object, long, long, Runnable) with the argument list (type, base, size.getEstimate(), logic).

See Section PhysicalMemoryManager

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() is called. When logic is null, logic must be supplied when the memory area is entered.

Throws

RealtimeSecurityException—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 has been registered with the PhysicalMemoryManager.

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.

IllegalAssignmentError—when storing logic in this would violate the assignment rules.
VTPhysicalMemory(Object, long, long)

Signature

public
   VTPhysicalMemory(Object type,
   long base,
   long size)

Description

Equivalent to VTPhysicalMemory(Object, long, long, Runnable) with the argument list (type, base, size, null).

See Section PhysicalMemoryManager

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

RealtimeSecurityException—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.

UnsupportedPhysicalMemoryException—when the underlying hardware does not support the given type, or when no matching PhysicalMemoryTypeFilter has been registered with the PhysicalMemoryManager.

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, long, SizeEstimator)**

**Signature**

```java
public VTPhysicalMemory(Object type,
                        long base,
                        SizeEstimator size)
```

**Description**

Equivalent to `VTPhysicalMemory(Object, long, long, Runnable)` with the argument list `(type, base, size.getEstimate(), null).

See Section PhysicalMemoryManager

**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**

- `RealtimeSecurityException`—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` has been registered with the `PhysicalMemoryManager`.
- `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.
VTPhysicalMemory(Object, long, Runnable)

Signature

```
public
VTPhysicalMemory(Object type,
    long size,
    Runnable logic)
```

Description

Equivalent to `VTPhysicalMemory(Object, long, long, Runnable)` with the argument list `(type, next, size, logic)`, where `next` is the beginning of the next best fit in the physical memory range.

See Section PhysicalMemoryManager

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()` is called. When `logic` is `null`, `logic` must be supplied when the memory area is entered.

Throws

- `RealtimeSecurityException`—when the application does not 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` has been registered with the `PhysicalMemoryManager`.
- `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, SizeEstimator, Runnable)

Signature

    public
    VTPhysicalMemory(Object type,
    SizeEstimator size,
    Runnable logic)

Description

    Equivalent to VTPhysicalMemory(Object, long, long, Runnable) with the
    argument list (type, next, size.getEstimate(), logic), where next is the
    beginning of the next best fit in the physical memory range.

See Section PhysicalMemoryManager

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() is called. When logic is null, logic must be supplied when the
    memory area is entered.

Throws

    RealtimeSecurityException—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 esti-
    mate from size extends beyond physically addressable memory.

    UnsupportedPhysicalMemoryException—when the underlying hardware does not
    support the given type, or when no matching PhysicalMemoryTypeFilter has
    been registered with the PhysicalMemoryManager.

    MemoryTypeConflictException—when the specified base does not point to mem-
    ory that matches the requested type, or when type specifies incompatible
    memory attributes.

    IllegalArgumentException—when size is null.
IllegalAssignmentError—when storing logic in this would violate the assignment rules.

VTPhysicalMemory(Object, long)

Signature

    public
    VTPhysicalMemory(Object type,
    long size)

Description

    Equivalent to VTPhysicalMemory(Object, long, long, Runnable) with the argument list (type, next, size, null), where next is the beginning of the next best fit in the physical memory range.

See Section PhysicalMemoryManager

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

    RealtimeSecurityException—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 has been registered with the PhysicalMemoryManager.
    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, SizeEstimator)

Signature

public VTPhysicalMemory(Object type,
                        SizeEstimator size)

Description

Equivalent to VTPhysicalMemory(Object, long, long, Runnable) with the argument list (type, next, size.getEstimate(), null), where next is the beginning of the next best fit in the physical memory range.

See Section PhysicalMemoryManager

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

RealtimeSecurityException—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 has been registered with the PhysicalMemoryManager.

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.

B.2.2.41.2 Methods
**to_string**

*Signature*

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

*Description*

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.

*Returns*

a string representing the value of *this*.

**B.3 Rationale**

These are interface and classes that have been shown to be less than ideal. Except for feasibility analysis, they have been replaced by elements that better fulfill the requirements. Compatibility can be provided by implementations that use existing facilities, so there is no reason to continue requiring their inclusion in new implementations.
Appendix C

Indices

C.1 Class Index

C.2 Class Index

AbsoluteTime, 347, 981
ActiveEvent, 265
ActiveEventDispatcher, 269
Affinity, 139
AffinityPermission, 806
AlignmentError, 861
AperiodicParameters, 147, 985
ArrivalTimeQueueOverflowException, 862, 990
AsyncBaseEvent, 276
AsyncBaseEventHandler, 281
AsyncEvent, 297, 991
AsyncEventHandler, 299, 994
AsynchronousControlGroup, 606
AsynchronouslyInterruptedException, 608, 1011
AsyncLongEvent, 306
AsyncLongEventHandler, 308
AsyncObjectEvent, 315
AsyncObjectEventHandler, 316
AsyncTimable, 396
BackgroundParameters, 153
BlockableReleaseRunner, 323
BoundAsyncEventHandler, 326, 1018
BoundAsyncLongEventHandler, 330
BoundAsyncObjectEventHandler, 333
BoundScheduled, 121
CeilingViolationException, 864
Chronograph, 397
Clock, 401
ConfigurationParameters, 55
ConstructorCheckedException, 866
CoreMemoryPermission, 808
DeregistrationException, 866
DirectMemoryBufferFactory, 711
DirectMemoryByteBuffer, 635
DirectMemoryPermission, 833
DirectMemoryRegion, 715
DuplicateFilterException, 1020
EnclosedType, 458
EventQueueOverflowException, 867
FirstInFirstOutParameters, 155
FirstInFirstOutScheduler, 156
FixedPriorityReleaseRunner, 337

1215
ForEachTerminationException, 868
GarbageCollector, 812
Happening, 717
HappeningDispatcher, 727
HappeningPermission, 836
HeapMemory, 460
HighResolutionTime, 361, 1021
IllegalAssignmentError, 869, 1024
IllegalRealtimeArgumentException, 870
IllegalRealtimeStateException, 876
IllegalTaskStateException, 881
ImmortalMemory, 465
ImmortalPhysicalMemory, 1024
ImportanceParameters, 1033
InaccessibleAreaException, 887
InterruptCeilingEmulation, 730
InterruptDescriptor, 731
Interruptible, 605, 958
InterruptInheritance, 734
InterruptMasking, 735
InterruptServiceRoutine, 736
InterruptUnmaskable, 740
LateStartException, 889
LTMemory, 497, 1036
LTPhysicalMemory, 1042
MemoryAccessError, 891, 1052
MemoryArea, 466
MemoryGroup, 500
MemoryInUseException, 892
MemoryParameters, 481, 1052
MemoryScopeException, 894, 1056
MemoryTypeConflictException, 895
MinimumInterarrivalPolicy, 133
MITViolationException, 889
MonitorControl, 235
NoHeapRealtimeThread, 1057
OffsetOutOfBoundsException, 897, 1061
OneShotTimer, 408
OutOfMemoryStaticError, 899
PerennialMemory, 486
PeriodicParameters, 160, 1068
PeriodicTimer, 410
PhasingPolicy, 52
PhysicalMemoryCharacteristic, 493
PhysicalMemoryFactory, 502
PhysicalMemoryManager, 1069
PhysicalMemoryPermission, 843
PhysicalMemoryRegion, 510
PhysicalMemorySelector, 511
PhysicalMemorySelector.CachingBehavior, 493
PhysicalMemorySelector.PagingBehavior, 495
PhysicalMemoryTypeFilter, 959
PinnableMemory, 514
POSIXException, 904
POSIXInvalidSignalException, 905
POSIXInvalidTargetException, 905
POSIXPermission, 849
POSIXSignalHandler, 1062
POSIXSignalPermissionException, 906
PriorityCeilingEmulation, 238
PriorityInheritance, 241
PriorityParameters, 168, 1079
PriorityScheduler, 170, 1080
ProcessingGroup, 173
ProcessingGroupParameters, 1089
ProcessorAffinityException, 907
QueueOverflowPolicy, 136
RangeOutOfBoundsException, 908
RawByte, 650
RawByteReader, 651
RawByteWriter, 654
RawDouble, 657
RawDoubleReader, 657
RawDoubleWriter, 661
RawFloat, 664
RawFloatReader, 664
RawFloatWriter, 667
RawInt, 670
RawIntReader, 671
RawIntWriter, 674
RawLong, 677
RawLongReader, 677
RawLongWriter, 680
RawMemory, 683
RawMemoryAccess, 1099
RawMemoryFactory, 741
RawMemoryFloatAccess, 1123
RawMemoryPermission, 839
RawMemoryRegion, 765
RawMemoryRegionFactory, 685
RawShort, 705
RawShortReader, 705
RawShortWriter, 708
RealtimePermission, 813
RealtimeSecurity, 1135
RealtimeSecurityException, 909
RealtimeSignal, 775
RealtimeSignalDispatcher, 781
RealtimeSystem, 817, 1138
RealtimeThread, 59, 1139
RealtimeThreadGroup, 186
RegistrationException, 914
RelativeTime, 371, 1160
Releasable, 268
ReleaseParameters, 192, 1165
ReleaseRunner, 339
ResourceLimitError, 915
RoundRobinParameters, 204
RoundRobinScheduler, 206
RTSJModule, 803
Schedulable, 121, 966
Scheduler, 210, 1166
SchedulingParameters, 214
SchedulingPermission, 825
ScopedConfigurationParameters, 535
ScopedCycleException, 917
ScopedMemory, 537, 1172
ScopedMemoryParameters, 564
ScopedMemoryPermission, 846
Signal, 785
SignalDispatcher, 792
SizeEstimator, 486
SizeOutOfBoundsException, 919
SporadicParameters, 217, 1187
StackedMemory, 568
StaticCheckedException, 921
StaticError, 926
StaticRuntimeException, 932
StaticThrowable, 856
StaticThrowableStorage, 937
Subsumable, 268
TaskPermission, 829
ThrowBoundaryError, 943, 1192
Timable, 400
Timed, 616, 1192
TimeDispatcher, 419
Timer, 422
UninitializedStateException, 945
UnknownHappeningException, 1195
UnsupportedPhysicalMemoryException, 945, 1196
UnsupportedRawMemoryRegionException, 947
UnsupportedRealtimeOperationException, 948
VTMemory, 1197
VTPhysicalMemory, 1203
WaitFreeReadQueue, 242
WaitFreeWriteQueue, 249
C.3 Method Index

C.4 Method Index

abort, 608
absolute, 351, 352, 368, 369, 374, 983, 984, 1021, 1022, 1162, 1163
add, 354–356, 366–378
addHandler, 278, 412, 726, 780, 791, 992, 1067
addIfFeasible, 970, 1000, 1146
addressOf, 714
addToFeasibility, 970, 1000, 1087, 1146, 1166
allocateByteBuffer, 713
associate, 504, 505
attach, 326, 338, 341
awaken, 82, 132, 297
bindTo, 993
canEnforceAllocationRate, 824
canEnforceCost, 823
checkAccessPhysical, 1136
checkAccessPhysicalRange, 1136
checkAEHSetDaemon, 1138
checkSetFilter, 1137
checkSetMonitorControl, 1137
checkSetScheduler, 1138
clear, 246, 252, 491, 613, 1016
clearAlarm, 407
clon, 195, 215, 367, 484, 1093
cmpTo, 367, 381
cmpToZero, 381
contains, 959
create, 732
createrd, 720
createImmortalMemory, 506
createLMemory, 507
createPinnableMemory, 508
createRawByte, 686, 746
createRawByteReader, 687, 747
createRawByteWriter, 688, 748
createRawDouble, 701, 762
createRawDoubleReader, 702, 763
createRawDoubleWriter, 703, 764
createRawFloat, 698, 759
createRawFloatReader, 699, 760
createRawFloatWriter, 700, 761
createRawInt, 692, 752
createRawIntReader, 693, 753
createRawIntWriter, 694, 754
createRawLong, 695, 755
createRawLongReader, 696, 756
createRawLongWriter, 697, 758
createRawShort, 689, 759
createRawShortReader, 690, 750
createRawShortWriter, 691, 751
createReleaseParameters, 280, 415, 431
createStackedMemory, 509
currentGC, 819
currentRealtimeThread, 65
currentSchedulable, 66, 212
deregister, 275, 421, 729, 746, 784, 795
deschedule, 76
deschedulePeriodic, 1147
destroy, 275, 422, 730, 784, 795
detach, 326, 338, 341
disable, 267, 277, 432, 611, 1014
doInterruptible, 612, 617, 1015, 1194
duplicate, 635
enable, 267, 277, 431, 611, 1013
enforceCost, 184, 202
enforcingCost, 183, 202
enter, 461, 469–473, 540–544, 578, 579, 1174, 1175
equals, 362, 368, 807, 810, 816, 827, 831, 834, 838, 841, 844, 847, 851
executeInArea, 462, 466, 478–480, 544–547, 1176
fillInStackTrace, 614, 859, 873, 879, 884, 902, 912, 924, 929, 935, 939, 951, 1017
find, 960
fire, 81, 298, 307, 315, 397, 410, 419, 612, 1015
fireSchedulable, 1088, 1171
flip, 649
force, 254
free, 713
generate, 142
getActions, 807, 811, 816, 828, 831, 834, 838, 841, 844, 847, 851
getAddress, 684
getAffinity, 217, 737
getAllocationRate, 484
getAndClearPendingFireCount, 283, 306, 314, 322
getAndDecrementPendingFireCount, 284, 305, 313, 321
getAndIncrementPendingFireCount, 997
getArrivalTimeQueueOverflowBehavior, 988
getAvailableProcessors, 143
getBase, 511
getByte, 651, 652, 1104
getByteOrder, 819
getBytes, 1105
getCachingBehavior, 514
getCallerPriority, 865
getCause, 858, 873, 878, 884, 901, 911, 923, 929, 934, 940, 950
getCeiling, 240, 865
getChar, 637
getChronograph, 363
getClock, 363, 415, 425
getConcurrentLocksUsed, 820
getConfigurationParameters, 73, 128, 287, 325, 337, 340
getCost, 196, 1093
getCostOverrunHandler, 183, 197, 1094
getCostUnderrunHandler, 182
gGetCurrent, 938
gGetCurrentConsumption, 67, 282, 283
gGetCurrentCost, 184
gGetCurrentMemoryArea, 67
gGetCurrentReleaseTime, 66, 281, 282
gDate, 356
gGetDeadline, 197, 1094
gGetDeadlineMissHandler, 198, 1094
gGetDefault, 57, 504
gGetDefaultConfiguration, 824
gGetDefaultFactory, 745
gGetDefaultRunner, 58, 536
gGetDefaultScheduler, 211
gGetDispatcher, 81, 268, 292, 400, 429, 725, 777, 788
gGetDouble, 637, 638, 658, 1127
gGetDoubles, 1128
gGetDrivePrecision, 406
gGetEffectiveStart, 177, 178
gGetEffectiveStartTime, 78, 426, 427
gGetEpochOffset, 398, 403
gGetEstimate, 491
gGetEventQueueOverflowPolicy, 203
gGetFireTime, 416, 417, 428
gGetFloat, 638, 665, 1129
gGetFloats, 1130
gGetGeneric, 610, 1013
gGetGranularity, 185
gGetHandler, 737
gGetHappening, 719
gGetID, 734
gGetId, 721, 722, 776, 777, 786, 787
gGetImportance, 1035
gGetInitialArrivalTimeQueueLength, 987
gGetInitialMemoryAreaIndex, 1140
gGetInitialMonitorControl, 822
gGetInitialQueueLength, 203
gGetInt, 639, 671, 672, 1106
getInterval, 418
getInts, 1107
getLastCost, 184
getLastThrowable, 939
getLength, 511
getLocalizedMessage, 858, 872, 878, 883, 900, 911, 922, 928, 933, 941, 949
getLong, 639, 640, 678, 1108
getLongs, 1109
getMappedAddress, 1110
getMaxBackingStore, 502
gsetMax, 241
getMaxConsumption, 80, 129, 130, 294
getMaxContainingArea, 567
gsetMaxEligibility, 188
getMaxGlobalBackingStore, 567
getMaxImmortal, 485
getMaximumConcurrentLocks, 820
getMaximumCost, 179, 180
getMaximumSize, 577, 1176
gsetMaxInitialArea, 485
gsetMaxInitialBackingStore, 567
ggetMaxMemoryArea, 1053
g getMaxPriority, 158, 172, 208, 1080
ggetMemoryArea, 72, 285, 473
ggetMemoryAreaStackDepth, 1141
ggetMemoryParameters, 73, 122, 285
ggetMessage, 857, 872, 877, 883, 900, 910, 922, 928, 933, 940, 949
getMaxLength, 58
getMilliseconds, 364
getMinConsumption, 79, 80, 128, 129, 293
getMinimumCost, 181
getMinimumInterarrival, 221, 222
getMinimumInterarrivalPolicy, 223
ggetMinPriority, 158, 172, 209, 1081
ggetMitViolationBehavior, 1190
ggetMonitorControl, 235, 236
ggetName, 685, 723, 733, 766, 777, 788
ggetNanoseconds, 364
getNormPriority, 158, 172, 209, 1082
ggetOuterMemoryArea, 1142
getPagingBehavior, 514
getPendingFireCount, 283, 304, 313, 321
ggetPeriod, 165, 178, 1095
ggetPinCount, 517
ggetPolicyName, 159, 171, 209, 213
ggetPortal, 547, 1177
ggetPredefinedAffinities, 141
ggetPredefinedAffinitiesCount, 141
ggetPreemptionLatency, 812
ggetPriority, 169
ggetProcessId, 787
ggetProcessingGroupParameters, 968, 998, 1147
ggetProcessorAddedEvent, 144
ggetProcessorCount, 146
ggetProcessorRemovedEvent, 145
ggetProcessors, 145, 146
ggetQuantum, 208
ggetQueryPrecision, 399, 405
ggetQueueLength, 293
ggetRealtimeClock, 402
ggetRealtimeThreadGroup, 73, 127, 274, 286, 340
ggetReferenceCount, 548, 1177
ggetRegion, 685
ggetRejectSet, 513
ggetReleaseParameters, 74, 123, 285
ggetRequestSet, 513
ggetScheduler, 74, 124, 188, 271, 286
ggetSchedulingParameters, 74, 126, 273, 286
ggetSecurityManager, 1139
ggetShort, 640, 706, 1110
ggetShorts, 1111
ggetSize, 684
ggetSizes, 59
ggetStackTrace, 615, 860, 874, 880, 885, 902, 913, 924, 930, 935, 941, 951, 1017
ggetStackTraceLength, 58
ggetStart, 165, 426, 1095
ggetStride, 684
ggetSupportedCachingBehavior, 512
getMethodIndex C.4

getSupportedPagingBehavior, 513
getTime, 398, 399, 404
getUniversalClock, 402
getVMAttributes, 961
getVMFlags, 961
globalBackingStoreConsumed, 539
globalBackingStoreRemaining, 538
globalBackingStoreSize, 538
handle, 739
handleAsyncEvent, 304, 311, 320
handledBy, 277, 430, 991
hasHandlers, 280
hashCode, 367, 808, 811, 816, 828, 831,
835, 838, 842, 845, 848, 851
hasRemaining, 650
hasUniversalClock, 823
hostBackingStoreConsumed, 578
hostBackingStoreRemaining, 577
hostBackingStoreSize, 577
implies, 808, 811, 817, 828, 832, 835, 839,
842, 845, 848, 852
in, 805
initCause, 858, 872, 878, 884, 901, 911,
923, 928, 934, 941, 950
initialize, 961
initMessage, 857, 871, 877, 882, 900, 910,
922, 927, 933, 940, 949
inRange, 714
inSchedulableExecutionContext, 212
instance, 157, 207, 240, 242, 462, 466,
731, 735, 740, 1082
interrupt, 75, 131, 295
interruptAction, 605, 959
isActive, 265, 429, 724, 778, 788
isAffinityChangeNotificationSupported,
143
isCompatible, 156, 169, 205, 216
isDaemon, 130, 292
isEmpty, 247, 253
isEnabled, 612, 1014
isFeasible, 1083, 1167
isFull, 247, 253
isHappening, 720
isInterrupt, 733
isInterrupted, 76, 132, 295
isPinned, 517
isPOSIXRealtimeSignal, 775
isPOSIXSignal, 785
isPresent, 962
isProcessorInSet, 146
isRawMemoryRegion, 766
isReadOnly, 635
isRegistered, 738
isRemovable, 963, 1073
isRemoved, 1073
isRousable, 201, 296
isRunning, 266, 276, 430, 725, 778, 789
isSetAffinitySupported, 142
isValid, 147
join, 518, 548, 549, 1178
joinAndEnter, 519–526, 550, 551,
553–559, 579, 581–583, 1179,
1180, 1182, 1183
joinAndEnterPinned, 528–535
joinPinned, 527
limit, 648
map, 1112–1114
mark, 648
mayHoldReferenceTo, 480, 481
mayUseHeap, 81, 131, 294
memoryConsumed, 473
memoryRemaning, 474
modules, 823
newArray, 463, 474, 560, 1184
newArrayInArea, 475
newInstance, 463, 464, 475, 476, 466,
731, 735, 740, 1082
newInstance, 463, 464, 475, 476, 561, 562,
1185, 1186
onInsertion, 963, 1074
onRemoval, 964, 1075, 1076
peekPending, 312, 320
physicalAddressOf, 717

RTSJ 2.0 (Draft 66)
pin, 516
position, 647
printStackTrace, 615, 616, 860, 861, 875, 880, 881, 886, 903, 904, 913, 914, 925, 926, 930, 931, 936, 937, 942, 943, 952, 953, 1018
put, 641
putChar, 641, 642
putDouble, 642, 643
putFloat, 643, 644
putInt, 644
putLong, 645
putShort, 646
read, 247, 253
readFence, 715
regionAddressOf, 716
register, 274, 421, 729, 738, 745, 783, 794
registerFilter, 1077
relative, 352, 353, 370, 375, 984, 985, 1023, 1163, 1164
release, 75, 325, 338, 341
remaining, 650
removeFilter, 1077
removeFromFeasibility, 971, 1001, 1088, 1147, 1171
removeHandler, 279, 726, 781, 792, 992, 1067
reschedule, 77, 159, 210, 213, 433
reserve, 487, 488
reserveArray, 488, 489
reserveLambda, 490, 491
reset, 649
resetTime, 618, 1194
resize, 576
restart, 618, 1195
run, 297, 304, 312, 321, 605, 958
scale, 380
schedulePeriodic, 1148
send, 780, 790
set, 357, 364–366, 655, 656, 662, 663, 669, 675, 676, 682, 710
setAlarm, 407
setAllocationRate, 1053
setAllocationRateIfFeasible, 1054
setArrivalTimeQueueOverflowBehavior, 989
setByte, 655, 1115
setBytes, 1116
setCost, 198, 1095
setCostOverrunHandler, 183, 199, 1096
setCostUnderrunHandler, 182
setDaemon, 130, 291
setDeadline, 152, 166, 199, 1096
setDeadlineMissHandler, 200, 1097
setDefault, 57
setDefaultConfiguration, 824
setDefaultDispatcher, 421, 728, 783, 794
setDefaultRunner, 57, 536
setDefaultScheduler, 212
setDouble, 661, 662, 1131
setDoubles, 1132
setEventQueueOverflowPolicy, 203, 223
setFloat, 668, 1133
setFloats, 1134
setGranularity, 185
setHandler, 279, 413, 726, 780, 792, 992, 1067
setIfFeasible, 971, 972, 974–976, 989, 1001–1004, 1008, 1068, 1083, 1085, 1086, 1097, 1148–1151, 1153, 1165, 1167–1169, 1190, 1191
setImportance, 1035
setInitialArrivalTimeQueueLength, 988
setInitialQueueLength, 204
setInt, 674, 675, 1117
setInterval, 418
setInts, 1118
setLastThrowable, 939
setLong, 681, 1119
setLongs, 1120
setMaxBackingStore, 502
setMaxEligibility, 188
setMaxImmortalIfFeasible, 1055
setMaximumConcurrentLocks, 821

1222

RTSJ 2.0 (Draft 66)
setMaximumCost, 180
setMaxMemoryAreaIfFeasible, 1056
setMemoryParameters, 82, 122, 287
setMemoryParametersIfFeasible, 978, 1009, 1154
setMinimumCost, 181
setMinimumInterarrival, 222
setMinimumInterarrivalPolicy, 222
setMitViolationBehavior, 1189
setMonitorControl, 236, 237
setPeriod, 166, 179, 1098
setPortal, 563, 1186
setPriority, 1080
setProcessingGroupParameters, 968, 999, 1155
setProcessingGroupParametersIfFeasible, 969, 1007, 1156
setProcessorAddedEvent, 144
setProcessorRemovedEvent, 145
setQuantum, 207
setRealtimeClock, 402
setReleaseParameters, 83, 123, 288
setReleaseParametersIfFeasible, 979, 1006, 1157
setRousable, 201, 296
setScheduler, 83, 84, 124, 125, 271, 272, 289, 290, 967, 997, 1158
setSchedulingParameters, 85, 126, 273, 291
setSchedulingParametersIfFeasible, 980, 1010, 1159
setSecurityManager, 822
setShort, 709, 1121
setShorts, 1122
setStackTrace, 614, 859, 874, 879, 885, 902, 912, 924, 929, 935, 942, 951, 1017
setStart, 167, 1098
setUniversalClock, 403
size, 248, 254, 477
sleep, 68, 1143
slice, 646
spin, 69, 70
start, 78, 266, 414, 432, 433, 723, 778, 779, 789, 1060
startPeriodic, 77, 1060
stop, 267, 433, 724, 779, 790
subsumes, 156, 170, 206, 216, 269
subtract, 357–359, 378, 379
supports, 822
suspend, 69
throwPending, 614, 1016
toString, 170, 360, 382, 563, 734, 766, 1036, 1042, 1051, 1187, 1203, 1213
trigger, 722, 725
triggerAlarm, 407
unbindTo, 993
unmap, 1123
unpin, 516
unregister, 739
unregisterInsertionEvent, 964, 1078
unregisterRemovalEvent, 965, 1078
value, 135, 138, 139, 805
valueOf, 54, 135, 138, 460, 495, 496, 805
values, 54, 134, 138, 460, 494, 496, 804
vFind, 966
visitChildren, 191, 192
visitConventionalThreads, 189, 190
visitInterrupts, 733
visitNestedScopes, 560
visitSchedulables, 190, 191
visitScopeRoots, 539
waitForData, 248
waitForNextPeriod, 1144
waitForNextPeriodInterruptible, 1145
waitForNextRelease, 71
waitForNextReleaseInterruptible, 71
waitForObject, 361
write, 249, 255
writeFence, 714