

Realtime and Embedded Specification for Java

Version 2.0

Draft 43
Eiffel Tower Edition
6th of May 2016

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	xiii
List of Tables	xiv
1 Introduction	1
1.1 Guiding Principles	2
1.1.1 Applicability to Particular Java Environments	2
1.1.2 Backward Compatibility	3
1.1.3 Write Once, Run Anywhere	3
1.1.4 Current Practice vs. Advanced Features	3
1.1.5 Predictable Execution	3
1.1.6 No Syntactic Extension	3
1.1.7 Allow Variation in Implementation Decisions	3
1.1.8 Interoperability	4
1.2 Areas of Enhancement	4
1.2.1 Thread Scheduling and Dispatching	4
1.2.2 Memory Management	5
1.2.3 Synchronization and Resource Sharing	5
1.2.4 Asynchronous Event Handling	5
1.2.5 Task Interruption	5
1.2.6 Raw Memory Access	6
1.2.7 Physical Memory Access	6
1.2.8 Modularization	6
2 Overview	7
2.1 Threads and Scheduling	7
2.2 Synchronization	9
2.2.1 Priority Inversion	9
2.2.2 Priority Inversion Avoidance	10
2.2.3 Execution Eligibility	11

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
2.3.3	Principles	13
2.3.3.1	Methodological Principles	13
2.3.3.2	Expressibility Principles	14
2.3.3.3	Semantic Principles	14
2.3.3.4	Pragmatic Principles	15
2.3.4	Asynchronous Realtime Thread Termination	15
2.4	Clocks, Time, and Timers	16
2.5	Memory Management	16
2.5.1	Memory Areas	16
2.5.2	Heap Memory	17
2.5.3	Immortal Memory	17
2.5.4	Scoped Memory	17
2.5.5	Physical Memory Areas	18
2.5.6	Budgeted Allocation	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	Base Module	25
3.2.2.2	Device Module	28
3.2.2.3	Alternative Memory Areas Module	29
3.2.3	POSIX module	29
3.2.4	Optional Features	30
3.2.5	Deprecated Classes	31
3.2.6	Implementation types Allowed	31
3.2.6.1	Realtime Deployment Implementation	32
3.2.6.2	Simulation Implementation	33
3.3	Required Documentation	34
3.4	Rationale	36

4	Realtime vs Conventional Java	37
4.1	Definitions	39
4.2	Semantics	39
4.2.1	Scheduling	39
4.2.1.1	Priority	40
4.2.1.2	Thread Groups	41
4.2.1.3	Current Thread	43
4.2.2	InterruptedException	43
4.2.3	Java Memory Model	43
4.2.4	Memory Management	44
4.2.4.1	Memory Areas	44
4.2.4.2	Garbage Collection	44
4.2.4.3	Realtime Garbage Collections	45
4.3	Rationale	46
5	Realtime Threads	47
5.1	Definitions	48
5.2	Semantics	48
5.3	javax.realtime	50
5.3.1	Enumerations	50
5.3.1.1	PhasingPolicy	50
5.3.2	Classes	53
5.3.2.1	ConfigurationParameters	53
5.3.2.2	RealtimeThread	56
5.4	Rationale	87
6	Scheduling	89
6.1	Definitions	91
6.2	Semantics	93
6.2.1	Schedulers	93
6.2.2	The Base Scheduler	94
6.2.2.1	Priorities	94
6.2.3	The Round-Robin Scheduler	95
6.2.3.1	Priorities	96
6.2.4	All Priority Schedulers	96
6.2.4.1	Dispatching	97
6.2.4.2	Parameter Values	98
6.2.4.3	Cost Monitoring and Cost Enforcement	100
6.2.4.4	Release of a Realtime Thread	102
6.2.4.5	General Release Control	107
6.2.4.6	Sporadic Release Control	111

TABLE OF CONTENTS

6.2.4.7	Release Control for Asynchronous Event Handlers	113
6.2.5	Associating Schedulables with Schedulers	115
6.2.6	Managing Groups of Schedulables	116
6.2.6.1	Scheduling Groups	116
6.2.6.2	Processing Groups	117
6.3	javax.realtime	120
6.3.1	Interfaces	120
6.3.1.1	BoundSchedulable	120
6.3.1.2	RealtimeExecutionContext	120
6.3.1.3	Schedulable	120
6.3.2	Enumerations	132
6.3.2.1	MinimumInterarrivalPolicy	132
6.3.2.2	QueueOverflowPolicy	135
6.3.3	Classes	138
6.3.3.1	Affinity	138
6.3.3.2	AperiodicParameters	149
6.3.3.3	BackgroundParameters	155
6.3.3.4	FirstInFirstOutScheduler	157
6.3.3.5	ImportanceParameters	160
6.3.3.6	PeriodicParameters	163
6.3.3.7	PriorityParameters	171
6.3.3.8	PriorityScheduler	173
6.3.3.9	ProcessingGroup	177
6.3.3.10	ReleaseParameters	190
6.3.3.11	RoundRobinScheduler	201
6.3.3.12	Scheduler	206
6.3.3.13	SchedulingGroup	209
6.3.3.14	SchedulingParameters	213
6.3.3.15	SporadicParameters	215
6.4	Rationale	222
6.4.1	SchedulingGroup and ProcessingGroup	223
6.4.2	Multiprocessor Support	223
6.4.3	Impact of Clock Granularity	225
6.4.4	Deadline Miss Detection	225
7	Synchronization	227
7.1	Definitions	227
7.2	Semantics	228
7.2.1	Monitor Control	228
7.2.2	Priority Schedulers	229
7.2.3	Additional Schedulers	231

7.3	javax.realtime	233
7.3.1	Classes	233
7.3.1.1	MonitorControl	233
7.3.1.2	PriorityCeilingEmulation	236
7.3.1.3	PriorityInheritance	239
7.3.1.4	WaitFreeReadQueue	240
7.3.1.5	WaitFreeWriteQueue	247
7.4	Rationale	253
8	Asynchrony	255
8.1	Definitions	257
8.2	Semantics	259
8.2.1	Asynchronous Events and their Handlers	259
8.2.2	Active Events and Dispatching	261
8.2.3	Asynchronous Transfer of Control	262
8.2.3.1	Summary of ATC Operation	265
8.3	javax.realtime	267
8.3.1	Interfaces	267
8.3.1.1	ActiveEvent	267
8.3.1.2	BoundAsyncBaseEventHandler	270
8.3.1.3	Interruptible	270
8.3.1.4	Releasable	271
8.3.2	Exceptions	272
8.3.2.1	AsynchronouslyInterruptedException	272
8.3.2.2	Timed	277
8.3.3	Classes	280
8.3.3.1	ActiveEventDispatcher	280
8.3.3.2	AsyncBaseEvent	284
8.3.3.3	AsyncBaseEventHandler	289
8.3.3.4	AsyncEvent	305
8.3.3.5	AsyncEventHandler	306
8.3.3.6	AsyncLongEvent	313
8.3.3.7	AsyncLongEventHandler	315
8.3.3.8	AsyncObjectEvent	320
8.3.3.9	AsyncObjectEventHandler	322
8.3.3.10	BoundAsyncEventHandler	328
8.3.3.11	BoundAsyncLongEventHandler	332
8.3.3.12	BoundAsyncObjectEventHandler	335
8.4	Rationale	338
9	Time	341

TABLE OF CONTENTS

9.1	Definitions	341
9.2	Semantics	342
9.3	javax.realtime	345
9.3.1	Classes	345
9.3.1.1	AbsoluteTime	345
9.3.1.2	HighResolutionTime	359
9.3.1.3	RelativeTime	368
9.4	Rationale	380
10	Clocks and Timers	381
10.1	Definitions	382
10.2	Semantics	383
10.2.1	Clock Model	383
10.2.2	Clocks and Timables	384
10.2.3	Timers	387
10.2.3.1	Counter Model	387
10.2.3.2	Comparator Model	388
10.2.3.3	Triggering	388
10.2.3.4	Behavior of Timers	388
10.2.3.5	Phasing	389
10.3	javax.realtime	390
10.3.1	Interfaces	390
10.3.1.1	AsyncTimable	390
10.3.1.2	Chronograph	391
10.3.1.3	Timable	394
10.3.2	Classes	395
10.3.2.1	Clock	395
10.3.2.2	OneShotTimer	400
10.3.2.3	PeriodicTimer	403
10.3.2.4	TimeDispatcher	412
10.3.2.5	TimeDispatcher.Runner	415
10.3.2.6	Timer	416
10.4	Rationale	429
11	Alternative Memory Areas	431
11.1	Definitions	433
11.2	Semantics	434
11.2.1	Allocation Execution Time	434
11.2.2	Allocation Context	435
11.2.3	The Parent Scope	436
11.2.4	Memory Areas and Schedulables	436

11.2.5	Scoped Memory Reference Counting	437
11.2.6	Immortal Memory	438
11.2.7	Maintaining Referential Integrity	439
11.2.8	Object Initialization	439
11.2.9	Maintaining the Scope Stack	440
11.2.10	The enter Method	441
11.2.11	The executeInArea or newInstance Methods	441
11.2.12	Constructor Methods for Schedulables	442
11.2.13	The Single Parent Rule	442
11.2.14	Scope Tree Maintenance	443
11.2.14.1	Pushing a MemoryArea onto the Scope Stack	443
11.2.14.2	Popping a MemoryArea off the Scope Stack	444
11.2.14.3	Reservation Management	444
11.2.15	Physical Memory	445
11.2.16	Stacked Memory	446
11.3	javax.realtime	449
11.3.1	Interfaces	449
11.3.1.1	MemoryAreaVisitor	449
11.3.2	Classes	450
11.3.2.1	HeapMemory	450
11.3.2.2	ImmortalMemory	455
11.3.2.3	MemoryArea	457
11.3.2.4	MemoryParameters	472
11.3.2.5	SizeEstimator	478
11.4	javax.realtime.memory	483
11.4.1	Interfaces	483
11.4.1.1	PhysicalMemoryCharacteristic	483
11.4.2	Enumerations	483
11.4.2.1	PhysicalMemorySelector.CachingBehavior	483
11.4.2.2	PhysicalMemorySelector.PagingBehavior	485
11.4.3	Classes	486
11.4.3.1	LTMemory	486
11.4.3.2	PhysicalMemoryFactory	490
11.4.3.3	PhysicalMemoryRegion	498
11.4.3.4	PhysicalMemorySelector	500
11.4.3.5	PinnableMemory	503
11.4.3.6	ScopedMemory	510
11.4.3.7	StackedMemory	534
11.5	The Rationale	550
11.5.1	The Scoped Memory Model	550

TABLE OF CONTENTS

11.5.2	The Physical Memory Model	551
11.5.2.1	The Original Physical Memory Framework	553
11.5.2.2	The RTSJ 2.0 Physical Memory Framework	554
11.5.2.3	An example	556
12	Devices and Triggering	559
12.1	Definitions	560
12.2	Semantics	561
12.2.1	Raw Memory	561
12.2.1.1	Raw Memory Region	564
12.2.1.2	Raw Memory Factory	564
12.2.1.3	Stride	564
12.2.2	Direct Memory Access Support	565
12.2.3	External Triggering	565
12.2.3.1	Happenings	566
12.2.4	Interrupt Service Routines	567
12.3	javax.realtime.device	572
12.3.1	Interfaces	572
12.3.1.1	RawByte	572
12.3.1.2	RawByteReader	572
12.3.1.3	RawByteWriter	575
12.3.1.4	RawDouble	578
12.3.1.5	RawDoubleReader	579
12.3.1.6	RawDoubleWriter	582
12.3.1.7	RawFloat	585
12.3.1.8	RawFloatReader	585
12.3.1.9	RawFloatWriter	588
12.3.1.10	RawInt	592
12.3.1.11	RawIntReader	592
12.3.1.12	RawIntWriter	595
12.3.1.13	RawLong	598
12.3.1.14	RawLongReader	599
12.3.1.15	RawLongWriter	602
12.3.1.16	RawMemory	605
12.3.1.17	RawMemoryRegionFactory	606
12.3.1.18	RawShort	627
12.3.1.19	RawShortReader	628
12.3.1.20	RawShortWriter	631
12.3.2	Classes	634
12.3.2.1	DMABufferFactory	634
12.3.2.2	DMARegion	638

12.3.2.3	Happening	640
12.3.2.4	HappeningDispatcher	648
12.3.2.5	InterruptServiceRoutine	651
12.3.2.6	RawMemoryFactory	656
12.3.2.7	RawMemoryRegion	681
12.4	Rationale	683
12.4.1	Raw Memory	683
12.4.1.1	Direct memory access	685
12.4.2	Interrupt Handling	686
12.4.3	An Illustrative Example	688
12.4.3.1	Software architecture	688
12.4.3.2	Device initialization	690
12.4.3.3	Responding to external happenings	692
12.4.3.4	Access to the flash controller's device registers	692
13	Interprocess Signalling	695
13.1	Definitions	695
13.2	Semantics	695
13.2.1	POSIX Signals	695
13.2.2	POSIX Realtime Signals	696
13.3	javax.realtime.posix	697
13.3.1	Classes	697
13.3.1.1	RealtimeSignal	697
13.3.1.2	RealtimeSignalDispatcher	702
13.3.1.3	Signal	705
13.3.1.4	SignalDispatcher	711
13.4	Rationale	714
14	System and Options	715
14.1	Semantics	715
14.1.1	RealtimeSystem	715
14.1.2	RealtimeSecurity	716
14.1.3	GarbageCollection	718
14.1.4	Compliance Version	718
14.2	javax.realtime	719
14.2.1	Enumerations	719
14.2.1.1	RTSJModule	719
14.2.2	Classes	721
14.2.2.1	GarbageCollector	721
14.2.2.2	RealtimeSecurity	723
14.2.2.3	RealtimeSystem	727

TABLE OF CONTENTS

14.3 Rationale	734
15 Exceptions	737
15.1 Semantics	737
15.2 javax.realtime	739
15.2.1 Interfaces	739
15.2.1.1 StaticThrowable	739
15.2.2 Exceptions	744
15.2.2.1 ArrivalTimeQueueOverflowException	744
15.2.2.2 CeilingViolationException	745
15.2.2.3 DeregistrationException	751
15.2.2.4 IllegalSchedulableStateException	752
15.2.2.5 InaccessibleAreaException	758
15.2.2.6 LateStartException	759
15.2.2.7 MITViolationException	760
15.2.2.8 MemoryInUseException	762
15.2.2.9 MemoryScopeException	764
15.2.2.10 MemoryTypeConflictException	765
15.2.2.11 OffsetOutOfBoundsException	767
15.2.2.12 POSIXException	768
15.2.2.13 POSIXInvalidSignalException	769
15.2.2.14 POSIXInvalidTargetException	770
15.2.2.15 POSIXSignalPermissionException	770
15.2.2.16 ProcessorAffinityException	771
15.2.2.17 RangeOutOfBoundsException	772
15.2.2.18 RegistrationException	773
15.2.2.19 ScopedCycleException	774
15.2.2.20 SizeOutOfBoundsException	775
15.2.2.21 StaticCheckedException	777
15.2.2.22 StaticRuntimeException	782
15.2.2.23 UnsupportedPhysicalMemoryException	788
15.2.2.24 UnsupportedRawMemoryRegionException	789
15.2.3 Classes	790
15.2.3.1 AlignmentError	790
15.2.3.2 IllegalAssignmentError	791
15.2.3.3 MemoryAccessError	792
15.2.3.4 ResourceLimitError	793
15.2.3.5 StaticError	795
15.2.3.6 StaticOutOfMemoryError	800
15.2.3.7 StaticThrowableStorage	805
15.2.3.8 ThrowBoundaryError	811

15.3 Rationale	812
Open Issues	813
A Deprecated APIs	815
A.1 Semantics	815
A.2 javax.realtime	816
A.2.1 Interfaces	816
A.2.1.1 PhysicalMemoryTypeFilter	816
A.2.1.2 <i>Schedulable</i>	824
A.2.2 Exceptions	840
A.2.2.1 <i>ArrivalTimeQueueOverflowException</i>	840
A.2.2.2 <i>AsynchronouslyInterruptedException</i>	840
A.2.2.3 DuplicateFilterException	842
A.2.2.4 <i>MemoryScopeException</i>	843
A.2.2.5 <i>OffsetOutOfBoundsException</i>	844
A.2.2.6 UnknownHappeningException	845
A.2.2.7 <i>UnsupportedPhysicalMemoryException</i>	846
A.2.3 Classes	846
A.2.3.1 <i>AbsoluteTime</i>	846
A.2.3.2 <i>AperiodicParameters</i>	851
A.2.3.3 <i>AsyncEvent</i>	856
A.2.3.4 <i>AsyncEventHandler</i>	859
A.2.3.5 <i>BoundAsyncEventHandler</i>	878
A.2.3.6 <i>Clock</i>	880
A.2.3.7 <i>GarbageCollector</i>	881
A.2.3.8 <i>HighResolutionTime</i>	882
A.2.3.9 <i>IllegalAssignmentError</i>	885
A.2.3.10 ImmortalPhysicalMemory	885
A.2.3.11 LTMemory	895
A.2.3.12 LTPhysicalMemory	901
A.2.3.13 <i>MemoryAccessError</i>	911
A.2.3.14 NoHeapRealtimeThread	912
A.2.3.15 <i>OneShotTimer</i>	916
A.2.3.16 POSIXSignalHandler	917
A.2.3.17 <i>PeriodicParameters</i>	927
A.2.3.18 <i>PeriodicTimer</i>	928
A.2.3.19 PhysicalMemoryManager	930
A.2.3.20 <i>PriorityCeilingEmulation</i>	940
A.2.3.21 <i>PriorityScheduler</i>	940
A.2.3.22 ProcessingGroupParameters	950

TABLE OF CONTENTS

A.2.3.23	<i>RationalTime</i>	960
A.2.3.24	<i>RawMemoryAccess</i>	965
A.2.3.25	<i>RawMemoryFloatAccess</i>	990
A.2.3.26	<i>RealtimeSystem</i>	1002
A.2.3.27	<i>RealtimeThread</i>	1002
A.2.3.28	<i>RelativeTime</i>	1022
A.2.3.29	<i>ReleaseParameters</i>	1029
A.2.3.30	<i>Scheduler</i>	1030
A.2.3.31	<i>ScopedMemory</i>	1036
A.2.3.32	<i>SporadicParameters</i>	1053
A.2.3.33	<i>ThrowBoundaryError</i>	1057
A.2.3.34	<i>Timer</i>	1058
A.2.3.35	<i>VTMemory</i>	1059
A.2.3.36	<i>VTPhysicalMemory</i>	1065
A.2.3.37	<i>WaitFreeDequeue</i>	1076
A.3	<i>Rationale</i>	1080
B	Bibliography	1083

List of Figures

6.1	Sequence Diagram of Some Example Realtime Thread Releases	108
6.2	A State Chart for a Realtime Thread without a Deadline Miss Handler	109
6.3	A State Chart for a Realtime Thread with a Deadline Miss Handler .	110
8.1	The Event Class Hierarchy	259
8.2	States of a Simple AsyncBaseEvent	261
8.3	States of an ActiveEvent	263
10.1	Sequence Diagram for Using a Timer	385
10.2	Sequence Diagram for Realtime Sleep	386
10.3	States of a Timer	390
11.1	Manipulation of StackedMemory Areas	447
12.1	Raw Memory Interface	562
12.2	Event Classes	563
12.3	Happening State Transition Diagram	566
12.4	Interrupt servicing	568
12.5	Creating Raw Memory Accessors	684
12.6	Flash memory device	686
12.7	Flash memory classes	690
12.8	Sequence diagram showing initialization operations	690
12.9	Sequence diagrams showing operations to initialize the hardware device	691
12.10	The FMSocketController.handleAsync method	692
12.11	Application usage	693

List of Tables

3.1	RTSJ Options	30
5.1	Effect of PhasingPolicy on the First Release of a RealtimeThread with PeriodicParameters	51
6.3	AperiodicParameters Default Values	150
6.4	FirstInFirstOut Default PriorityParameter Values	157
6.5	PeriodicParameter Default Values	164
6.6	PriorityScheduler Default PriorityParameter Values	174
6.7	ProcessingGroup Default Values	179
6.8	ReleaseParameter Default Values	191
6.9	SporadicParameters Default Values	216
8.1	Event to Handler Matrix	256
9.1	Examples of Normalized Times	343
9.2	Semantics of Time Conversion	344
11.1	Memory Area Referencing Restrictions	439
12.1	Properties Array	658
12.2	Device registers	689
A.1	ProcessingGroupParameter Default Values	952
A.2	Properties Array	968

Chapter 1

Introduction

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

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

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

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

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

Providing realtime semantics and the additional programming interfaces required

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

Realtime tasks can be modeled both with realtime threads and with event handlers. Realtime threads are much the same as conventional Java threads except for how they are scheduled. Event handlers encapsulate a bit of work that is done every time some event occurs. Events are referred to as asynchronous because they generally occur independent of program flow. Thus, a 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 set of programming interfaces requires some guiding principles around which to organize the RTSJ. These principles delimit the scope of the RTSJ and its compatibility requirements with conventional Java.

1.1.1 Applicability to Particular Java Environments

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

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

1.1.2 Backward Compatibility

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

1.1.3 Write Once, Run Anywhere

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

1.1.4 Current Practice vs. Advanced Features

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

1.1.5 Predictable Execution

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

1.1.6 No Syntactic Extension

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

1.1.7 Allow Variation in Implementation Decisions

Implementations of the RTSJ may vary in a number of implementation decisions, such as the use of efficient or inefficient algorithms, trade-offs between time and space efficiency, inclusion of scheduling algorithms not required in the minimum implementation, and variation in code path length for the execution of byte codes. The RTSJ should not mandate algorithms or specific time constants for such, but require that the semantics of the implementation be met and where necessary put

limits on execution time complexity. The RTSJ offers implementers the flexibility to create implementations suited to meet the requirements of their customers.

1.1.8 Interoperability

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

1.2 Areas of Enhancement

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

1.2.1 Thread Scheduling and Dispatching

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

To accommodate current practice, the RTSJ shall require a base scheduler in all implementations. The required base scheduler will be familiar to realtime system programmers. It is a priority preemptive, 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 serial access to resources, and realtime systems introduce an additional complexity: the need to minimize priority inversion and hence the excessive delay of more critical tasks. The least intrusive specification for enabling realtime safe synchronization is to require that implementations of the Java keyword `synchronized` implement one or more algorithms that prevent priority inversion among realtime Java tasks that share the serialized resource. In addition, the specification provides other data passing mechanisms to minimize the need for synchronization.

1.2.4 Asynchronous Event Handling

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

1.2.5 Task Interruption

Sometimes, the real world changes so drastically (and asynchronously) that the current point of logic execution should be immediately, efficiently, and safely ended and control transferred to another point of execution. The RTSJ provides a mechanism which

extends Java's interrupt and exception handling mechanisms to enable applications to programmatically change the locus of control of another Java task. This mechanism may restrict this asynchronous transfer of control to logic specifically written with the assumption that its locus of control may asynchronously change. Due to the inherent susceptibility to deadlock, the `Thread.stop` method cannot be used for this.

1.2.6 Raw Memory Access

Accessing device memory is not in and of itself a realtime issue; 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 of threads and scheduling, synchronization, asynchrony, clocks and timers, memory management, device access and raw memory, system options, and exceptions are all crucial to the acceptance of the RTSJ as a viable realtime development platform. Further details, exact requirements, class documentation, and rationale for these extensions are given in subsequent chapters.

2.1 Threads and Scheduling

One of the concerns of realtime programming is to ensure the timely and predictable execution of sequences of machine instructions. Various scheduling schemes name these sequences of instructions differently, for example, thread, task, module, or block. In Java, this computation is executed in the context of a thread. Since Java threads were designed for fair execution¹ 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

¹Actually, 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 format <string>Parameters such as ReleaseParameters, which provide parameters for resource management. An instance of one of these parameter classes holds a particular resource-demand characteristic for one or more schedulables. For example, the PriorityParameters subclass of SchedulingParameters contains the execution eligibility metric of the base scheduler, i.e., a priority. At some time (construction-time or later when the parameters are replaced using setter methods), instances of parameter classes are bound to a schedulable. The schedulable then assumes the characteristics of the values in the parameter object. For example, a PriorityParameters instance with its priority set to the value representing the highest priority available on a system is bound to a schedulable, then that schedulable will assume the characteristic that it will execute whenever it is ready in preference to all other schedulables (except, of course, those also with the same priority).

The RTSJ provides implementers with the flexibility to install arbitrary scheduling algorithms in an implementation of the specification. This is to support the widely

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

2.2 Synchronization

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

2.2.1 Priority Inversion

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

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

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

2.2.2 Priority Inversion Avoidance

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

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

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

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 (such as an execution-time clock of some granularity).

2.3.2 Asynchronous Transfer of Control

Many event-driven computer systems that tightly interact with external physical systems (e.g., humans, machines, control processes, etc.) may require mode changes in their computational behavior as a result of significant changes in the 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 in only the oracle and not in each thread, the overall system design is simpler.

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

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

2.3.3 Principles

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

2.3.3.1 Methodological Principles

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`).

2.3.3.2 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.

2.3.3.3 Semantic Principles

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

2.3.3.4 Pragmatic Principles

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

2.3.4 Asynchronous Realtime Thread Termination

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

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

1. make all of the application methods of the realtime thread asynchronously interruptible;
2. create an oracle which monitors the external world by setting up an asynchronous event with a number of asynchronous event handlers, which is fired when an appropriate mode change;
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 (if coded to be so) but orderly clean up and termination of the realtime thread. Note that the oracle can comprise as many or as few asynchronous event handlers as appropriate.

2.4 Clocks, Time, and Timers

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

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

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

2.7 System Options

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

2.8 Exceptions

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

Using exceptions is resource intensive, since a new exception is allocated for each throw. This is particularly a problem for scoped memory, since scopes may need to be sized much larger than otherwise necessary to hold exceptions and their stack traces. Additionally, the information they contain cannot be propagated beyond the scope in which they are allocated. To better support scoped, immortal, and physical memory, a new class of throwable has been included: `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 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, 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.

- 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.AbstractAsyncEventHandler` 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 `AbstractAsyncEventHandler`. 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 thread or schedulable, including Java threads.
- 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 `javafx.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 the implementation latitude in its implementation. For instance, the required priority scheduler requires at least 28 consecutively numbered realtime priorities. It does not, however, specify the numeric values of the maximum and minimum realtime priorities. Implementations are encouraged to offer as many realtime priority levels immediately above the conventional Java priorities as they can support.

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

3.2.2 Modules

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

Modules provide a means of grouping like functionality together in a way that promotes maximal adoption for various implementation classes. A conventional JVM may simply implement the Base 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 Base 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 implementation some choice over which modules and features to include and which not.

3.2.2.1 Base Module

The Base Module adds the concepts of processor affinity, threads with realtime scheduling, and asynchronous event handling. This includes the notion of executing code at a given time interval, providing a much more stable response than using sleep in a loop. These features should have no impact on the overall performance

of a system that implements them, but enrich the programming modules available to the programmer. The classes and interfaces required in this module are all in package `javax.realtime` and are listed below.

- `AbsoluteTime` (Section 9.3.1.1)
- `ActiveEvent` (Section 8.3.1.1)
- `ActiveEventDispatcher` (Section 8.3.3.1)
- `Affinity` (Section 6.3.3.1)
- `AperiodicParameters` (Section 6.3.3.2)
- `AsyncBaseEvent` (Section 8.3.3.2)
- `AsyncBaseEventHandler` (Section 8.3.3.3)
- `AsyncEvent` (Section 8.3.3.4)
- `AsyncEventHandler` (Section 8.3.3.5)
- `AsyncLongEvent` (Section 8.3.3.6)
- `AsyncLongEventHandler` (Section 8.3.3.7)
- `AsyncObjectEvent` (Section 8.3.3.8)
- `AsyncTimable` (Section 10.3.1.1)
- `AsyncObjectEventHandler` (Section 8.3.3.9)
- `BoundAsyncBaseEventHandler` (Section 8.3.1.2)
- `BoundAsyncEventHandler` (Section 8.3.3.10)
- `BoundAsyncLongEventHandler` (Section 8.3.3.11)
- `BoundAsyncObjectEventHandler` (Section 8.3.3.12)
- `Clock` (Section 10.3.2.1)
- `Chronograph` (Section 10.3.1.2)
- `ConfigurationParameters` (Section 5.3.2.1)
- `FirstInFirstOutScheduler` (Section 6.3.3.4)
- `GarbageCollector` (Section 14.2.2.1)
- `HeapMemory` (Section 11.3.2.1)
- `HighResolutionTime` (Section 9.3.1.2)
- `ImmortalMemory` (Section 11.3.2.2)
- `ImportanceParameters` (Section 6.3.3.5)
- `Interruptible` (Section 8.3.1.3)
- `MemoryArea` (Section 11.3.2.3)
- `MemoryAreaVisitor` (Section 11.3.1.1)
- `MemoryParameters` (Section 11.3.2.4)¹
- `MonitorControl` (Section 7.3.1.1)
- `OneShotTimer` (Section 10.3.2.2)
- `PeriodicParameters` (Section 6.3.3.6)
- `PeriodicTimer` (Section 10.3.2.3)
- `PhasingPolicy` (Section 5.3.1.1)

¹The `mayUseHeap` flag is present, but can only be set if the Memory Module is supported.

- [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)
- [RealtimeExecutionContext](#) (Section 6.3.1.2)
- [RealtimeSecurity](#) (Section 14.2.2.2)
- [RealtimeSystem](#) (Section 14.2.2.3)
- [RealtimeThread](#) (Section 5.3.2.2)
- [RelativeTime](#) (Section 9.3.1.3)
- [Releasable](#) (Section 8.3.1.4)
- [ReleaseParameters](#) (Section 6.3.3.10)
- [RoundRobinScheduler](#) (Section 6.3.3.11)
- [RTSJModule](#) (Section 14.2.1.1)
- [Schedulable](#) (Section 6.3.1.3)
- [Scheduler](#) (Section 6.3.3.12)
- [SchedulingParameters](#) (Section 6.3.3.14)
- [SizeEstimator](#) (Section 11.3.2.5)
- [SporadicParameters](#) (Section 6.3.3.15)
- [Timable](#) (Section 10.3.1.3)
- [Timed](#) (Section 8.3.2.2)
- [TimeDispatcher](#) (Section 10.3.2.4)
- [Timer](#) (Section 10.3.2.6)
- [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 15.2.3.1)
- [ArrivalTimeQueueOverflowException](#) (Section 15.2.2.1)
- [CeilingViolationException](#) (Section 15.2.2.2)
- [DeregistrationException](#) (Section 15.2.2.3)
- [IllegalAssignmentError](#) (Section 15.2.3.2)
- [InaccessibleAreaException](#) (Section 15.2.2.5)
- [LateStartException](#) (Section 15.2.2.6)
- [MemoryAccessError](#) (Section 15.2.3.3)
- [MemoryInUseException](#) (Section 15.2.2.8)
- [MemoryScopeException](#) (Section 15.2.2.9)
- [MemoryTypeConflictException](#) (Section 15.2.2.10)
- [MITViolationException](#) (Section 15.2.2.7)
- [OffsetOutOfBoundsException](#) (Section 15.2.2.11)

- [POSIXException](#) (Section 15.2.2.12)
- [POSIXInvalidSignalException](#) (Section 15.2.2.13)
- [POSIXInvalidTargetException](#) (Section 15.2.2.14)
- [POSIXSignalPermissionException](#) (Section 15.2.2.15)
- [ProcessorAffinityException](#) (Section 15.2.2.16)
- [RangeOutOfBoundsException](#) (Section 15.2.2.17)
- [RegistrationException](#) (Section 15.2.2.18)
- [ResourceLimitError](#) (Section 15.2.3.4)
- [ScopedCycleException](#) (Section 15.2.2.19)
- [StaticCheckedException](#) (Section 15.2.2.21)
- [StaticError](#) (Section 15.2.3.5)
- [StaticOutOfMemoryError](#) (Section 15.2.3.6)
- [StaticRuntimeException](#) (Section 15.2.2.22)
- [StaticThrowable](#) (Section 15.2.1.1)
- [StaticThrowableStorage](#) (Section 15.2.3.7)
- [SizeOutOfBoundsException](#) (Section 15.2.2.20)
- [ThrowBoundaryError](#) (Section 15.2.3.8)
- [UnsupportedPhysicalMemoryException](#) (Section 15.2.2.23)
- [UnsupportedRawMemoryRegionException](#) (Section 15.2.2.24)

3.2.2.2 Device Module

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

- [Happening](#) (Section 12.3.2.3)
- [HappeningDispatcher](#) (Section 12.3.2.4)
- [InterruptServiceRoutine](#) (Section 12.3.2.5)
- [DMABufferFactory](#) (Section 12.3.2.1)
- [RawMemory](#) (Section 12.3.1.16)
- [RawMemoryFactory](#) (Section 12.3.2.6)
- [RawMemoryRegion](#) (Section 12.3.2.7)
- [RawMemoryRegionFactory](#) (Section 12.3.1.17)
- [RawByte](#) (Section 12.3.1.1)
- [RawByteReader](#) (Section 12.3.1.2)
- [RawByteWriter](#) (Section 12.3.1.3)
- [RawDouble](#) (Section 12.3.1.4)
- [RawDoubleReader](#) (Section 12.3.1.5)

- [RawDoubleWriter](#) (Section 12.3.1.6)
- [RawFloat](#) (Section 12.3.1.7)
- [RawFloatReader](#) (Section 12.3.1.8)
- [RawFloatWriter](#) (Section 12.3.1.9)
- [RawInt](#) (Section 12.3.1.10)
- [RawIntReader](#) (Section 12.3.1.11)
- [RawIntWriter](#) (Section 12.3.1.12)
- [RawLong](#) (Section 12.3.1.13)
- [RawLongReader](#) (Section 12.3.1.14)
- [RawLongWriter](#) (Section 12.3.1.15)
- [RawShort](#) (Section 12.3.1.18)
- [RawShortReader](#) (Section 12.3.1.19)
- [RawShortWriter](#) (Section 12.3.1.20)

3.2.2.3 Alternative Memory Areas Module

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

- [LTMemory](#) (Section 11.4.3.1)
- [PhysicalMemoryCharacteristic](#) (Section 11.4.1.1)
- [PhysicalMemoryFactory](#) (Section 11.4.3.2)
- [PhysicalMemoryRegion](#) (Section 11.4.3.3)
- [PhysicalMemorySelector](#) (Section 11.4.3.4)
- [PinnableMemory](#) (Section 11.4.3.5)
- [ScopedMemory](#) (Section 11.4.3.6)
- [StackedMemory](#) (Section 11.4.3.7)

3.2.3 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 it 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 `javax.realtime.posix` and are listed below.

- [RealtimeSignal](#) (Section 13.3.1.1)
- [RealtimeSignalDispatcher](#) (Section 13.3.1.2)
- [Signal](#) (Section 13.3.1.3)

- [SignalDispatcher](#) (Section 13.3.1.4)

3.2.4 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 3.1: RTSJ Options

Hard cost enforcement	Provides an automatic means of controlling the processor usage of a task or group of tasks.
Processing Group deadline less than period	Enables the application to specify a processing group deadline less than the processing group period
Allocation-rate enforcement on heap allocation	Enables the application to limit the rate at which a schedulable creates objects in the heap.
Interrupt Service Routine	Provides first level interrupt processing in Java.

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`.

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.

3.2.5 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:

- `DuplicateFilterException` (Section A.2.2.3)
- `ImmortalPhysicalMemory` (Section A.2.3.10)
- `LTMemory` (Section A.2.3.11)
- `LTPhysicalMemory` (Section A.2.3.12)
- `NoHeapRealtimeThread` (Section A.2.3.14)
- `PhysicalMemoryManager` (Section A.2.3.19)
- `PhysicalMemoryTypeFilter` (Section A.2.1.1)
- `ProcessingGroupParameters` (Section A.2.3.22)
- `POSIXSignalHandler` (Section A.2.3.16)
- `RationalTime` (Section A.2.3.23)
- `RawMemoryAccess` (Section A.2.3.24)
- `RawMemoryFloatAccess` (Section A.2.3.25)
- `ScopedMemory` (Section A.2.3.31)
- `UnknownHappeningException` (Section A.2.2.6)
- `VTMemory` (Section A.2.3.35)
- `VTPhysicalMemory` (Section A.2.3.36)
- `WaitFreeDequeue` (Section A.2.3.37)

They are documented fully in Chapter A.

3.2.6 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 for Safety Critical Java, may subset classes and packages herein, but must implement the methods and classes defined in that specification.

3.2.6.1 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 and at least 28 unique priority levels. 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 unique priority levels are required to be unique to preclude implementations from using fewer priority levels of underlying systems to implement the required 28 by simplistic algorithms (such as lumping four RTSJ priorities into seven buckets for an underlying system that only supports seven priority levels). It is sufficient for systems with fewer than 28 priority levels to use more sophisticated algorithms to implement the required 28 unique levels as long as Schedulable behave as though there were at least 28 unique levels. (e.g. if there were 28 RealtimeThreads (t_1, \dots, t_{28}) with priorities (p_1, \dots, 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, \dots, t_{28} and thread t_2 would *always* execute in preference to threads t_3, \dots, 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.

All instances of Schedulable 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.

3.2.6.2 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 implemen-

tation 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 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 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 nonheap context, 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 a extraheap context, the documentation shall detail the effect of these restrictions for each RTSJ API.
 8. The worst-case response interval between firing an `AsyncEvent` because of a bound happening to releasing an associated `AsyncEventHandler` (assuming no higher-priority schedulables are runnable) shall be documented for 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-overflow) 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.

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 11.2.6.)
- Some Throwables, in particular, those implementing `StaticThrowable`, which includes `StaticOutOfMemoryError`, which an RTSJ VM throws 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 15.1.)

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 11.2.7.)
- 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 6.3.3.8.3.)
- 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 8.3.3.5.)

The third and final category is behaviors experienced 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 8.2.3.)
- 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 8.2.3.)
- Exceptions propagating into asynchronously interruptible regions of code will be lost if an `AsynchronouslyInterruptedException` is pending. (See Section 8.2.3.)
- Subclasses of `AsynchronouslyInterruptedException` indicated in the signature of a method do not indicate that the method is asynchronously interruptible. (See Section 8.2.3.)
- Catch clauses for `AsynchronouslyInterruptedException` or its subclasses in asynchronously interruptible methods will not catch an AIE. (See Section 8.2.3.)
- A Throwable crossing a `MemoryArea` boundary might be transformed into a `ThrowBoundaryError`, and the original exception may be lost. (See Section 15.2.3.8 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

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 `Thread.setPriority()` are governed by semantics from Chapter 7 on *Synchronization*.

2. Conventional Java threads may not use `setPriority` to apply the expanded range of priorities defined by this specification.
3. When `setPriority` is called on a realtime thread, that thread's `SchedulingParameters` are set to null and the thread is scheduled as if it were a Java thread.

4.2.1.1.2 Getting Priority

The `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 `SchedulingParameters` through a scheduler.
2. When called on a realtime thread with null `SchedulingParameters`, a value in the conventional Java priority range is returned.
3. When called on a realtime thread (`t`) with `PriorityParameters`, `getPriority` behaves effectively as if it included the following code snippet:

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

4. When the scheduling parameters are of a type other than `PriorityParameters`, a `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 `Schedulable`, it makes sense to have facilities for managing groups of these as well. The RTSJ provides an extension of `ThreadGroup` for this called `SchedulingGroup`.

Every instance of `ThreadGroup` holds a reference to every member thread and every subgroup instance of `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 `ThreadGroup` (and, in particular, its `enumerate` methods), the RTSJ requires that no `ThreadGroup` or Java thread is allocated in scoped memory, and that no thread allocated in `ScopedMemory` is referenced by a `ThreadGroup`. Instances of `SchedulingGroup` are

instead used for these purposes, and an alternative to enumerate is provided on `SchedulingGroup` in the form of a visitor.

Scheduling groups, i.e., instances of `SchedulingGroup` (a subclass of `ThreadGroup`, are designed to be able to reference threads, schedulables, and other scheduling groups, even when they are in scoped memory. These are only reachable using a visitor with a lambda expression. Consequently schedulables and scheduling groups are not part of any thread group and will hold a scheduling 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 `SchedulingGroup` 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 `SchedulingGroup`.
3. Call the `SchedulingGroup.enumerate(Thread[])` and `SchedulingGroup.enumerate(Thread[], boolean)` only return Java threads.
4. Call the `SchedulingGroup.enumerate(ThreadGroup[])` and `SchedulingGroup.enumerate(ThreadGroup[], boolean)` only return threads groups and scheduling 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 scheduling 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 scheduling group of the handler's dispatcher, when that dispatcher's scheduling 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 scheduling group, either explicitly 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` class.

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 12.2.1), and `DMABufferFactory` fence methods are all ordered in a way that is consistent with respect to native code or hardware devices that use platform-native memory coherence 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 improving determinism, its use complicates the logic of application and library code.

On systems that support memory areas other than heap and do not support realtime garbage collection, some global resources must be put in immortal memory. System properties and their String values allocated during system initialization shall be allocated in immortal memory. For such a system, class objects should also be stored there. Though this avoids priority inversion with the garbage collector, it can cause higher memory use than expected.

4.2.4.2 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 implementation meant as a development system or for systems that choose not to provide realtime guarantees. In any case, an implementation must document what realtime guarantees it gives and which method it uses to do so.

4.2.4.3 Realtime Garbage Collections

Industrial realtime garbage collectors are available with varying approaches to providing realtime response. Though new collectors will undoubtedly 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.

4.2.4.3.4 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, amongst other things, the size of Java 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 facilities are also available to asynchronous event handlers. In particular,

- the default scheduler must support the scheduling of both realtime threads and asynchronous event handlers;
- realtime threads and asynchronous event handlers are allowed to enter into memory areas and consequently they have associated scope stacks; 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.

5.1 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 detecting 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 `SchedulingGroup`.
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 `java.lang.Thread.setPriority(int)`, which causes it to be treated as a conventional java thread until a new `SchedulingParameters` object is set.

5.3 javax.realtime

5.3.1 Enumerations

5.3.1.1 PhasingPolicy

Inheritance

java.lang.Object
java.lang.Enum
 javax.realtime.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 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 ADJUST_FORWARD

Description

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

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

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

```
public static final ADJUST_BACKWARD
```

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 STRICT_PHASING
```

Description

Indicates that a periodic thread started after the absolute time given for its start time should throw the [LateStartException](#)¹ exception instead of being released.

5.3.1.1.2 Methods

values*Signature*

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

*Description***valueOf(String)***Signature*

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

Description

¹Section [15.2.2.6](#)

5.3.2 Classes

5.3.2.1 ConfigurationParameters

Inheritance

java.lang.Object
 [javafx.runtime.ConfigurationParameters](#)

Description

Configuration parameters provide a way to specify various implementation-dependent parameters such as the Java stack and native stack sizes, and to configure the statically allocated [ThrowBoundaryError](#)² associated with a [Schedulable](#)³.

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 IllegalStateException
```

Description

Creates a parameter object for initializing the state of a [Schedulable](#)⁴. The parameters provide the data for this initialization. For [RealtimeThread](#)⁵ and bound versions of [AsyncBaseEventHandler](#)⁶, the stack and message buffers can

²Section [15.2.3.8](#)

³Section [6.3.1.3](#)

⁴Section [6.3.1.3](#)

⁵Section [5.3.2.2](#)

⁶Section [8.3.3.3](#)

be set exactly, but for the unbound event handlers, the system cannot give any guarantees to allow thread sharing.

Parameters

`messageLength` is 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](#)⁷ 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` 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](#)⁸ 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

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

Description

Same as [ConfigurationParameters\(int,int,long\[\]\)](#)⁹ with arguments -1, -1, sizes.

5.3.2.1.2 Methods

⁷Section [6.3.1.3](#)

⁸Section [6.3.1.3](#)

⁹Section [5.3.2.1.1](#)

getMessageLength

Signature

```
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 [Schedulable](#)¹⁰ created with these parameters. The value 0 indicates that no message will be stored.

Returns

Reserved memory size in units of char.

getStackTraceLength

Signature

```
public int  
getStackTraceLength()
```

Description

Gets the length of the stack trace buffer dedicated to [Schedulable](#)¹¹ 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

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

Description

¹⁰Section [6.3.1.3](#)

¹¹Section [6.3.1.3](#)

Gets the array of implementation-specific sizes associated with [Schedulable](#)¹² 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

Inheritance

java.lang.Object
 java.lang.Thread
[javax.realtime.RealtimeThread](#)

Interfaces

[javax.realtime.BoundSchedulable](#)
[javax.realtime.AsyncTimable](#)

Description

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

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

5.3.2.2.1 Constructors

RealtimeThread(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, ConfigurationParameters, TimeDispatcher, SchedulingGroup, Runnable)

Signature

¹²Section [6.3.1.3](#)

```
public
RealtimeThread(SchedulingParameters scheduling,
                javax.realtime.ReleaseParameters<?> release,
                MemoryParameters memory,
                MemoryArea area,
                ConfigurationParameters config,
                TimeDispatcher dispatcher,
                SchedulingGroup group,
                Runnable logic)
```

Description

Create a realtime thread with the given characteristics and a specified Runnable. The scheduling 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](#)¹³ associated with this (And possibly other instances of [Schedulable](#)¹⁴). When scheduling is null and the creator is a schedulable, [SchedulingParameters](#)¹⁵ 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](#)¹⁶ associated with this (and possibly other instances of [Schedulable](#)¹⁷). 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](#)¹⁸ associated with this (and possibly other instances of [Schedulable](#)¹⁹). 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.

¹³Section [6.3.3.14](#)

¹⁴Section [6.3.1.3](#)

¹⁵Section [6.3.3.14](#)

¹⁶Section [6.3.3.10](#)

¹⁷Section [6.3.1.3](#)

¹⁸Section [11.3.2.4](#)

¹⁹Section [6.3.1.3](#)

area the initial memory area of this handler.

config The [ConfigurationParameters](#)²⁰ associated with this (and possibly other instances of [Schedulable](#)²¹). 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](#)²² to use for realtime sleep and determining the period of a periodic thread.

group The SchedulingGroup of the newly created realtime thread or the parent's scheduling 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

[IllegalArgumentException](#) when the parameters are not compatible with the associated scheduler or the current thread group is not a [SchedulingGroup](#) and group is null.

[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)

²⁰Section [5.3.2.1](#)

²¹Section [6.3.1.3](#)

²²Section [10.3.2.4](#)

Signature

```
public
RealtimeThread(SchedulingParameters scheduling,
               javax.realtime.ReleaseParameters<?> release,
               MemoryParameters memory,
               MemoryArea area,
               ConfigurationParameters config,
               Runnable logic)
```

Description

Create a realtime thread with the given [SchedulingParameters](#)²³, [ReleaseParameters](#)²⁴, [MemoryParameters](#)²⁵, [ConfigurationParameters](#)²⁶, 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

```
public
RealtimeThread(SchedulingParameters scheduling,
               javax.realtime.ReleaseParameters<?> release,
               ConfigurationParameters config,
               Runnable logic)
```

Description

Create a realtime thread with the given [SchedulingParameters](#)²⁷, [ReleaseParam-](#)

²³Section [6.3.3.14](#)

²⁴Section [6.3.3.10](#)

²⁵Section [11.3.2.4](#)

²⁶Section [5.3.2.1](#)

²⁷Section [6.3.3.14](#)

eters²⁸, [MemoryArea](#)²⁹ and a specified Runnable and default values for all other parameters.

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

Available since RTSJ 2.0

RealtimeThread(SchedulingParameters, ReleaseParameters, ConfigurationParameters)

Signature

```
public
RealtimeThread(SchedulingParameters scheduling,
               javax.realtime.ReleaseParameters<?> release,
               ConfigurationParameters config)
```

Description

Create a realtime thread with the given [SchedulingParameters](#)³⁰, [ReleaseParameters](#)³¹ and [MemoryArea](#)³² 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,
               javax.realtime.ReleaseParameters<?> release,
               Runnable logic)
```

²⁸Section [6.3.3.10](#)

²⁹Section [11.3.2.3](#)

³⁰Section [6.3.3.14](#)

³¹Section [6.3.3.10](#)

³²Section [11.3.2.3](#)

Description

Create a realtime thread with the given [SchedulingParameters](#)³³, [ReleaseParameters](#)³⁴ 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)

Signature

```
public  
RealtimeThread(SchedulingParameters scheduling,  
                javax.runtime.ReleaseParameters<?> release)
```

Description

Create a realtime thread with the given [SchedulingParameters](#)³⁵ and [ReleaseParameters](#)³⁶ and default values for all other parameters.

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

RealtimeThread(SchedulingParameters, TimeDispatcher)

Signature

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

Description

³³Section [6.3.3.14](#)

³⁴Section [6.3.3.10](#)

³⁵Section [6.3.3.14](#)

³⁶Section [6.3.3.10](#)

Create a realtime thread with the given [SchedulingParameters](#)³⁷ and [TimeDispatcher](#)³⁸ 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)

Signature

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

Description

Create a realtime thread with the given [SchedulingParameters](#)³⁹ and default values for all other parameters. This constructor is equivalent to `RealtimeThread(scheduling, null, null, null, null, null, null, null)`.

RealtimeThread

Signature

```
public  
RealtimeThread()
```

Description

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

5.3.2.2.2 Methods

³⁷Section [6.3.3.14](#)

³⁸Section [10.3.2.4](#)

³⁹Section [6.3.3.14](#)

currentRealtimeThread

Signature

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

Description

Gets a reference to the current instance of RealtimeThread.

It is permissible to call currentRealtimeThread when control is in an [Async-EventHandler](#)⁴⁰. 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 that of a Java thread.

Returns

A reference to the current instance of RealtimeThread.

currentSchedulable

Signature

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

Description

Gets a reference to the current instance of Schedulable. It behaves the same when the current thread is an instance of java.lang.Thread, but otherwise it returns an instance of [AsyncBaseEventHandler](#)⁴¹.

Throws

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

Returns

A reference to the current instance of Schedulable.

⁴⁰Section [8.3.3.5](#)

⁴¹Section [8.3.3.3](#)

getCurrentMemoryArea

Signature

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

Description

Return a reference to the [MemoryArea](#)⁴² object representing the current allocation context.

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

Returns

A reference to the [MemoryArea](#)⁴³ object representing the current allocation context.

getInitialMemoryAreaIndex

Signature

```
public static int  
getInitialMemoryAreaIndex()  
throws IllegalStateException,  
ClassCastException
```

Description

Gets the position of the initial memory area for the current [Schedulable](#)⁴⁴ in the memory area stack. Memory area stacks may include inherited stacks from parent threads. The initial memory area of a [RealtimeThread](#)⁴⁵ or an [AsyncBaseEventHandler](#)⁴⁶ 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

[IllegalThreadStateException](#) when the memory area stack of the current [RealtimeThread](#) has changed from its initial configuration and the memory area at the originally specified initial memory area index is not the initial memory area, thus the index is invalid.

⁴²Section [11.3.2.3](#)

⁴³Section [11.3.2.3](#)

⁴⁴Section [6.3.1.3](#)

⁴⁵Section [5.3.2.2](#)

⁴⁶Section [8.3.3.3](#)

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 that of a Java thread. 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 RealtimeThread.

getMemoryAreaStackDepth

Signature

```
public static int
getMemoryAreaStackDepth()
throws ClassCastException
```

Description

Gets the size of the stack of [MemoryArea](#)⁴⁷ instances to which the current schedulable has access.

Note, the current memory area ([getCurrentMemoryArea\(\)](#)⁴⁸) is found at memory area stack index of [getMemoryAreaStackDepth\(\)](#) - 1.

Throws

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

Returns

The size of the stack of [MemoryArea](#)⁴⁹ instances.

getOuterMemoryArea(int)

Signature

```
public static javax.realtime.MemoryArea
getOuterMemoryArea(int index)
throws ClassCastException,
        MemoryAccessError
```

Description

Gets the instance of [MemoryArea](#)⁵⁰ in the memory area stack at the index given. When the given index does not exist in the memory area scope stack then null is returned.

Note, the current memory area ([getCurrentMemoryArea\(\)](#)⁵¹) is found at memory area stack index [getMemoryAreaStackDepth\(\)](#) - 1, so [getCurrentMemoryArea\(\)](#) == [getOutMemoryArea\(getMemoryAreaStackDepth\(\) - 1\)](#).

Parameters

index The offset into the memory area stack.

Throws

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

[MemoryAccessError](#) when the memory area is allocate in heap memory and the caller is a schedulable that may not use the heap.

Returns

The instance of [MemoryArea](#)⁵² at index or null when the given value does not correspond to a position in the stack.

⁴⁷Section [11.3.2.3](#)

⁴⁸Section [5.3.2.2.2](#)

⁴⁹Section [11.3.2.3](#)

⁵⁰Section [11.3.2.3](#)

⁵¹Section [5.3.2.2.2](#)

⁵²Section [11.3.2.3](#)

sleep(HighResolutionTime)

Signature

```
public static void  
sleep(javafx.runtime.HighResolutionTime<?> time)  
throws InterruptedException,  
    ClassCastException,  
    IllegalArgumentException
```

Description

A sleep method that is controlled by a generalized clock. Since the time is expressed as a [HighResolutionTime](#)⁵³, 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](#)⁵⁴. 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.

It is permissible to call sleep when control is in an [AsyncEventHandler](#)⁵⁵. The method cause 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()`⁵⁶ or `AsynchronouslyInterruptedE`
`fire()`⁵⁷ during the time between calling this method and returning from it.

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

`IllegalArgumentException` when time is null, when time is a relative time less than zero, or when the [Chronograph](#)⁵⁸ of time is not a [Clock](#)⁵⁹.

⁵³Section [9.3.1.2](#)

⁵⁴Section [10.3.2.1](#)

⁵⁵Section [8.3.3.5](#)

⁵⁶Section [5.3.2.2.2](#)

⁵⁷Section [8.3.2.1.2](#)

⁵⁸Section [10.3.1.2](#)

⁵⁹Section [10.3.2.1](#)

suspend(HighResolutionTime)

Signature

```
public static void  
suspend(javax.realtime.HighResolutionTime<?> time)  
throws ClassCastException,  
        IllegalArgumentException
```

Description

The same as [sleep\(HighResolutionTime\)](#)⁶⁰ 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 that of a Java thread.

IllegalArgumentException when time is null, when time is a relative time less than zero, or when the [Chronograph](#)⁶¹ of time is not a [Clock](#)⁶².

Available since RTSJ 2.0

spin(HighResolutionTime)

Signature

```
public static void  
spin(javax.realtime.HighResolutionTime<?> time)  
throws InterruptedException,  
        ClassCastException,  
        IllegalArgumentException
```

Description

Similar to [sleep\(HighResolutionTime\)](#)⁶³ except it performs a busy wait by polling on the [Chronograph](#)⁶⁴ associated with time until time has been reached. Note that interaction with other tasks, scheduling considerations, and other effects 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.

⁶⁰Section [5.3.2.2.2](#)

⁶¹Section [10.3.1.2](#)

⁶²Section [10.3.2.1](#)

⁶³Section [5.3.2.2.2](#)

⁶⁴Section [10.3.1.2](#)

Parameters

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

Throws

InterruptedException when the thread is interrupted by `interrupt()`⁶⁵ or `AsynchronouslyInterruptedE`
`fire()`⁶⁶ during the time between calling this method and returning from it.

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

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

Available since RTSJ 2.0

spin(int)*Signature*

```
public static void  
spin(int nanos)  
throws InterruptedException,  
    ClassCastException,  
    IllegalArgumentException
```

Description

The same as calling `spin(HighResolutionTime)`⁶⁷ 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()`⁶⁸ or `AsynchronouslyInterruptedE`
`fire()`⁶⁹ during the time between calling this method and returning from it.

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

IllegalArgumentException when nanos is less than zero.

Available since RTSJ 2.0

⁶⁵Section 5.3.2.2.2

⁶⁶Section 8.3.2.1.2

⁶⁷Section 5.3.2.2.2

⁶⁸Section 5.3.2.2.2

⁶⁹Section 8.3.2.1.2

waitForNextRelease

Signature

```
public static boolean
waitForNextRelease()
throws AsynchronouslyInterruptedException,
        IllegalStateException,
        ClassCastException
```

Description

Causes the current realtime thread to delay until the next release. (See [release\(\)](#)⁷⁰.) Used by threads that have a reference to either periodic or aperiodic [ReleaseParameters](#)⁷¹. The first release starts when this thread is released as a consequence of the action of one of the [start\(\)](#)⁷² 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](#)⁷³.)

Throws

[AsynchronouslyInterruptedException](#) when the thread is interrupted by [interrupt\(\)](#)⁷⁴ or [AsynchronouslyInterruptedException.fire\(\)](#)⁷⁵ during the time between calling this method and returning from it and the [ReleaseParameters.isRousable\(\)](#)⁷⁶ 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.

[IllegalStateException](#) when this does not have a reference to a [ReleaseParameters](#)⁷⁷ type of either [PeriodicParameters](#)⁷⁸ or [AperiodicParameters](#)⁷⁹.

[ClassCastException](#) when the current thread is not an instance of `RealtimeThread`.

⁷⁰Section [5.3.2.2.2](#)

⁷¹Section [6.3.3.10](#)

⁷²Section [5.3.2.2.2](#)

⁷³Section [6.3.3.8](#)

⁷⁴Section [5.3.2.2.2](#)

⁷⁵Section [8.3.2.1.2](#)

⁷⁶Section [6.3.3.10.2](#)

⁷⁷Section [6.3.3.10](#)

⁷⁸Section [6.3.3.6](#)

⁷⁹Section [6.3.3.2](#)

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 javafx.realtime.MemoryArea  
getMemoryArea()
```

Description

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

Returns

A reference to the initial memory area for this thread.

Available since RTSJ 1.0.1

getMemoryParameters

Signature

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

Description

Gets a reference to the [MemoryParameters](#)⁸⁰ object for this schedulable.

Returns

A reference to the current [MemoryParameters](#)⁸¹ object.

⁸⁰Section [11.3.2.4](#)

⁸¹Section [11.3.2.4](#)

getSchedulingGroup

Signature

```
public javax.realtime.SchedulingGroup  
getSchedulingGroup()
```

Description

Gets a reference to the [SchedulingGroup](#)⁸² instance of this schedulable.

Returns

A reference to the current [SchedulingGroup](#)⁸³ object.

Available since since RTSJ 2.0

getConfigurationParameters

Signature

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

Description

Gets a reference to the [ConfigurationParameters](#)⁸⁴ object for this schedulable.

Returns

A reference to the associated [ConfigurationParameters](#)⁸⁵ object.

Available since RTSJ 2.0

getReleaseParameters

Signature

```
public javax.realtime.ReleaseParameters<?>  
getReleaseParameters()
```

Description

⁸²Section [6.3.3.13](#)

⁸³Section [6.3.3.13](#)

⁸⁴Section [5.3.2.1](#)

⁸⁵Section [5.3.2.1](#)

Gets a reference to the [ReleaseParameters](#)⁸⁶ object for this schedulable.

Returns

A reference to the current [ReleaseParameters](#)⁸⁷ object.

getScheduler

Signature

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

Description

Gets a reference to the [Scheduler](#)⁸⁸ object for this schedulable.

Returns

A reference to the associated [Scheduler](#)⁸⁹ object.

getSchedulingParameters

Signature

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

Description

Gets a reference to the [SchedulingParameters](#)⁹⁰ object for this schedulable.

Returns

A reference to the current [SchedulingParameters](#)⁹¹ object.

release

Signature

```
public void  
release()
```

⁸⁶Section [6.3.3.10](#)

⁸⁷Section [6.3.3.10](#)

⁸⁸Section [6.3.3.12](#)

⁸⁹Section [6.3.3.12](#)

⁹⁰Section [6.3.3.14](#)

⁹¹Section [6.3.3.14](#)

Description

Generate 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.

Throws

IllegalStateException when this does not have a reference to a [ReleaseParameters](#)⁹² type of [AperiodicParameters](#)⁹³.

Available since RTSJ 2.0

interrupt

Signature

```
public void  
interrupt()
```

Description

Make the generic [AsynchronouslyInterruptedException](#)⁹⁴ 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 cause an early release. In any case, AsynchronouslyInterruptedException is thrown once a method is entered that implements AsynchronouslyInterruptedException.

Available since RTSJ 2.0

isInterrupted

Signature

```
public boolean  
isInterrupted()
```

Description

Determines whether or not the generic [AsynchronouslyInterruptedException](#)⁹⁵ is pending.

⁹²Section [6.3.3.10](#)

⁹³Section [6.3.3.2](#)

⁹⁴Section [8.3.2.1](#)

⁹⁵Section [8.3.2.1](#)

Returns

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

Available since RTSJ 2.0

deschedule

Signature

```
public void  
deschedule()
```

Description

Perform any *deschedule* actions specified by this thread's scheduler, either immediately when in `waitForNextRelease()`⁹⁶ or the next time the thread enters `waitForNextRelease()`.

Available since RTSJ 2.0

reschedule

Signature

```
public void  
reschedule()  
throws InterruptedException
```

Description

Returns the thread to the blocked-for-next-release state. This causes the next event release the thread and `waitForNextRelease`⁹⁷ to return. Deadline miss and cost enforcement are re-enabled.

The details of the interaction of this method with `deschedule`⁹⁸, `waitForNextRelease`⁹⁹ and `release`¹⁰⁰ are dictated by this thread's scheduler.

Throws

⁹⁶Section 5.3.2.2.2

⁹⁷Section 5.3.2.2.2

⁹⁸Section 5.3.2.2.2

⁹⁹Section 5.3.2.2.2

¹⁰⁰Section 5.3.2.2.2

IllegalStateException 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
```

Description

Start 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](#)¹⁰¹ 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](#)¹⁰².

IllegalArgumentException when the thread is not periodic, or when its start time is not absolute.

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

Available since RTSJ 2.0

start

Signature

```
public void  
start()
```

Description

¹⁰¹Section [6.3.3.6](#)

¹⁰²Section [5.3.1.1.1](#)

Set up the realtime thread's environment and start it. The set up might include delaying it until the assigned start time and initializing the thread's scope stack. (See [ScopedMemory](#)¹⁰³.)

Throws

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

Available since RTSJ 2.0 adds new exception

getLastReleaseTime

Signature

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

Description

Equivalent to `getLastReleaseTime(null)`

Available since RTSJ 2.0

getLastReleaseTime(AbsoluteTime)

Signature

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

Description

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

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

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

¹⁰³Section [A.2.3.31](#)

getEffectiveStartTime

Signature

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

Description

Equivalent to `getEffectiveStartTime(null)`.

Available since RTSJ 2.0

getEffectiveStartTime(AbsoluteTime)

Signature

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

Description

Determine 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, return the effective start time in an [AbsoluteTime](#)¹⁰⁴ instance created in the current memory area.

Available since RTSJ 2.0

getCurrentConsumption(RelativeTime)

Signature

¹⁰⁴Section [9.3.1.1](#)


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

Description

Determine the CPU consumption for this release.

Throws

IllegalStateException when the caller is not a [Schedulable](#)¹⁰⁵.

Returns

When dest is null, return the CPU consumption in a [RelativeTime](#)¹⁰⁶ instance created in the current execution context. When dest is not null, return 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

getMinConsumption(RelativeTime)

Signature

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

Description

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

Throws

¹⁰⁵Section [6.3.1.3](#)

¹⁰⁶Section [9.3.1.3](#)

IllegalStateException when the caller is not a [Schedulable](#)¹⁰⁷.

Returns

the minimum CPU consumption in dest. When dest is null return the minimum CPU consumption in a [RelativeTime](#)¹⁰⁸ instance created in the current memory area.

Available since RTSJ 2.0

getMinConsumption

Signature

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

Description

Equivalent to getMinConsumption(null).

Available since RTSJ 2.0

getMaxConsumption(RelativeTime)

Signature

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

Description

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

Throws

IllegalStateException when the caller is not a [Schedulable](#)¹⁰⁹.

Returns

the maximum CPU consumption in dest. When dest is null return the maximum CPU consumption in a [RelativeTime](#)¹¹⁰ instance created in the current memory area.

¹⁰⁷Section [6.3.1.3](#)

¹⁰⁸Section [9.3.1.3](#)

¹⁰⁹Section [6.3.1.3](#)

¹¹⁰Section [9.3.1.3](#)

Available since RTSJ 2.0

getMaxConsumption

Signature

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

Description

Equivalent to getMaxConsumption(null).

Available since RTSJ 2.0

getDispatcher

Signature

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

Description

Get 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](#)¹¹¹ infrastructure to cause a call to [waitForNextRelease](#)¹¹² to return.

¹¹¹Section [10.3.2.1](#)

¹¹²Section [5.3.2.2.2](#)

See [Section AsyncTimable.fire\(\)](#)

Available since RTSJ 2.0

mayUseHeap

Signature

```
public boolean  
mayUseHeap()
```

Description

Determine 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 [Clock¹¹³](#) infrastructure to cause a call to [sleep¹¹⁴](#) to return.

See [Section Schedulable.awaken\(\)](#)

Available since RTSJ 2.0

setMemoryParameters(MemoryParameters)

Signature

¹¹³Section [10.3.2.1](#)

¹¹⁴Section [5.3.2.2.2](#)

```
public javax.realtime.RealtimeThread  
setMemoryParameters(MemoryParameters memory)
```

Description

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

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

Parameters

memory memory A [MemoryParameters](#)¹¹⁵ 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](#)¹¹⁶.)

Throws

`IllegalArgumentException` `IllegalArgumentException` 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](#) `IllegalAssignmentError` when the schedulable cannot hold a reference to memory, or when memory cannot hold a reference to this schedulable instance.

`IllegalStateException` null

Returns

this

setReleaseParameters(ReleaseParameters)

Signature

```
public javax.realtime.RealtimeThread  
setReleaseParameters(javax.realtime.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

¹¹⁵Section [11.3.2.4](#)

¹¹⁶Section [6.3.3.8](#)

the release parameters may take effect at different times. See the documentation for the scheduler for details.

Parameters

release A [ReleaseParameters](#)¹¹⁷ 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](#)¹¹⁸.)

Throws

`IllegalArgumentException` Thrown 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.

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

Returns

this

setScheduler(Scheduler)

Signature

```
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\(\)](#)¹¹⁹ has been called on it and `RealtimeThread.join()` would block.

Parameters

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

¹¹⁷Section [6.3.3.10](#)

¹¹⁸Section [6.3.3.8](#)

¹¹⁹Section [5.3.2.2.2](#)

Throws

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

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

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

Returns

this

setScheduler(Scheduler, SchedulingParameters, ReleaseParameters, MemoryParameters)*Signature*

```
public javafx.runtime.RealtimeThread  
setScheduler(Scheduler scheduler,  
             SchedulingParameters scheduling,  
             javafx.runtime.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`¹²⁰ 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`¹²¹.)

¹²⁰Section 6.3.3.14

¹²¹Section 6.3.3.8

release A reference to the [ReleaseParameters](#)¹²² 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](#)¹²³.)

memoryParameters A reference to the [MemoryParameters](#)¹²⁴ 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](#)¹²⁵.)

Throws

`IllegalArgumentException` Thrown 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.

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

Returns

this

setSchedulingParameters(SchedulingParameters)

Signature

```
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

¹²²Section [6.3.3.10](#)

¹²³Section [6.3.3.8](#)

¹²⁴Section [11.3.2.4](#)

¹²⁵Section [6.3.3.8](#)

scheduling A reference to the [SchedulingParameters](#)¹²⁶ 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](#)¹²⁷.)

Throws

`IllegalArgumentException` Thrown 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.

`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

this

5.4 Rationale

Realtime programming requires a schedule method radically different than what a conventional Java programmer would expect, but most other aspects of thread behavior is the same, it is reasonable to model a realtime thread as a `java.lang.Thread`. The main additions that were needed are for adding additional 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 realtime operating systems. The `ReleaseParameters` and `MemoryParameters` provided to the `RealtimeThread` constructor provide a number of common realtime thread types, including periodic threads. However, conventional Java thread scheduling is supported. The realtime priorities are all above the conventional Java priorities to ensure the realtime 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.

¹²⁶Section [6.3.3.14](#)

¹²⁷Section [6.3.3.8](#)

Chapter 6

Scheduling

Scheduling is a key differentiator between a conventional Java implementation and a realtime Java implementation. Whereas conventional Java implementations relies 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. 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, affinity, 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. 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: `SchedulingGroup` and `ProcessingGroup`. These classes provide additional means of managing tasks.

An instance of the `SchedulingGroup` 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 `SchedulingGroup` or its subclass. Therefore the root thread group and the thread group of the initial thread must both be scheduling groups in an RTSJ implementation.

The `ProcessingGroup` class is a subclass of `SchedulingGroup`. 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 three execution states: *executing*, *blocked*, and *eligible-for-execution*.

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-overflow.
3. *Eligible-for-execution* refers to the state where the schedulable could be selected to have its state changed to executing.

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

Release — 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. (Some actions implicitly clear any pending releases.)

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 , $R_{i+1} - R_i \geq T$. 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 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 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, a detail semantic applicable across the classes 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.

The scheduler can be changed independently of the `SchedulingParameters` and vice versa only when the `Schedulable` in question is descheduled. Rescheduling will throw an `IllegalSchedulableStateException` when called on a `Schedulable` scheduling parameters that are inconsistent with its scheduler. Trying to add a handler with `SchedulingParameters` that do not match its scheduler to an event will also result in an `IllegalSchedulableStateException` being thrown.

6.2.2 The Base Scheduler

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

6.2.2.1 Priorities

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

1. The base scheduler must support at least 28 distinct values (realtime priorities) that can be stored in an instance of `PriorityParameters` in addition to the values 1 through 10 required to support the priorities defined by `java.lang.Thread`.
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 `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()`, 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 Chapter 7.
 12. When schedulable *A* managed by the base scheduler creates a 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

$$\frac{((\text{PriorityScheduler.getMaxPriority}) - \text{PriorityScheduler.getMinPriority}) / 3 + \text{PriorityScheduler.getMinPriority}}{1}$$

 14. Hardware priorities, where supported, have values above the base scheduler's priority range (see Section 12.2.4).

6.2.3 The Round-Robin Scheduler

The round-robin scheduler specified by the RTSJ is a fixed-quantum, fixed-priority priority-preemptive scheduler that interacts predictably with the base scheduler: [RoundRobinScheduler](#). The time at which a quantum expires may be calculated either from last task switch or on a heartbeat. It uses the `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 either until it blocks (as on a monitor or for some I/O operation), voluntarily relinquishes the CPU (as for sleep), is priority preempted, or 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.

6.2.3.1 Priorities

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 `ImportanceParameters` subclass of `PriorityParameters`.
4. `RoundRobinScheduler.getNormPriority()` shall be set to

$$\frac{1 \text{ ((RoundRobinScheduler.getMaxPriority() - } \\ 2 \text{ RoundRobinScheduler.getMinPriority()) / 3) + } \\ 3 \text{ RoundRobinScheduler.getMinPriority()}}{}$$

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.

6.2.4 All Priority Schedulers

Both the base (FIFO) scheduler and the round-robin scheduler share the semantics for dispatching, parameter values, cost monitoring and enforcement, and release semantics and control, as well as the same base class: `PriorityScheduler`.

6.2.4.1 Dispatching

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

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

1. A Schedulable can become a running schedulable only when it is ready and one of the processors in its requested 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 `PriorityParameters setPriority()` method, the `RealtimeThread setSchedulingParameters()` method, or `Thread's 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.

6.2.4.2 Parameter Values

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

1. The default values for the base scheduler are as follows.
 - (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 priority parameters value.
 - (b) The default for release depend on the type of schedulable:
 - i. for instance of `@code RealtimeThread` the default is an instance of background parameters with default values (see [AperiodicParameters](#)), and
 - ii. for instance of `@code AsyncBaseEventHandler` the default is an instance of aperiodic parameters with default values (see [AperiodicParameters](#)).
 - (c) Memory parameters default to null which signifies that memory allocation by the schedulable is not constrained by the scheduler.
 - (d) The default scheduling parameter values for parameter objects created by a schedulable controlled by the base scheduler are given by the following table (see `FirstInFirstOutScheduler`).

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

2. All numeric or `RelativeTime` attributes in parameter values must be greater than or equal to zero.
3. Values of period must be greater than zero.
4. Changes to scheduling, release, memory, and processing group parameters (by methods on the schedulables bound to the parameters or by altering the parameter objects themselves) 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.
5. 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 immediately except when constrained by priority inversion avoidance algorithms.
 - (b) Changes to release parameters depend on the parameter being changed, the type of release parameter object, and the type of schedulable.
 - i. Changes to the deadline and the deadline miss handler take effect at each release event as follows: when the i^{th} release event occurred at a time t_i , then the i^{th} deadline is the time $t_i + D_i$, where D_i is the

- value of the deadline stored in the schedulable's release parameters object at the time t_i . 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.

6.2.4.3 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 now 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 6.2.4.4 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.

6.2.4.4 Release of a Realtime Thread

The repeated release of a realtime thread is achieved by executing in a loop and invoking the `RealtimeThread.waitForNextRelease`¹ 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

¹The method `RealtimeThread.waitForNextPeriod` has been replaced by `RealtimeThread.waitForNextRelease` as of RTSJ 2.0. The same goes for its interruptible equivalent.

-
- (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.
 - 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 set the thread's state to `Descheduled`.
 - ii. When the current state is not a blocked state, it just sets the value of `deschedule` to true.
- (c) The function of the `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 `deschedule` to false, sets the values of `pendingReleases` and `missCount` to zero, changes the thread's state to blocked-for-release-event, and tell the cost monitoring and enforcement system to reset for this thread.
 - ii. When the realtime thread is *not* in the `Descheduled` state, it just sets the value of `deschedule` to 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 `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 `wait-ForNextRelease`, increment the value of `pendingReleases`, inform cost monitoring and enforcement that the next release event has occurred, and notify the thread to make it eligible for execution;
 - iii. Otherwise, when the thread is in a release, increment the value of `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 `deschedule` is set to true, the handler is atomically released with its `fireCount` increased by the value of `missCount` + 1, and zero for `missCount`;
 - ii. otherwise, one is added to the `missCount` value.
- (g) When the `waitForNextRelease` method is invoked by the current realtime thread there are three possible behaviors depending on the value of

missCount and lastReturn.

- i. When missCount is zero, any pending parameter changes are applied, cost monitoring and enforcement are informed of completion, and then the thread waits while deschedule is true, or pendingReleases is zero. Then the lastReturn value is set to true, pendingReleases is decremented, and true is returned.
 - ii. When missCount is greater than zero and the lastReturn value is false, completion occurs: the missCount value is decremented; then any pending parameter changes are applied, 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 waitNextRelease method with release parameters where ReleaseParameters.isRousable return 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) When an instance of AIE becomes pending on the realtime thread while it is blocked-for-release-event, and the realtime thread is descheduled, then the AIE remains pending until the realtime thread is no longer descheduled. The associated reschedule acts as a release event. Execution then continues as in 7d where the time value used as t_{int} is the time at which the schedulable was rescheduled.
 - (c) When an instance of AIE becomes pending on the realtime thread while it is blocked-for-release-event and it is not descheduled, then this acts as a release event. Execution then continues as in 7d, where the time value used as t_{int} is the time at which the AIE becomes pending.
 - (d)
 - i. The realtime thread is made eligible for execution.
 - ii. Upon execution, the invocation completes abruptly by throwing the pending AIE instance as an InterruptedException. When the pending instance is the generic AIE instance then the interrupt state of the realtime thread is cleared.
 - iii. The deadline associated with this release is the time $t_{int} + D_{int}$, where D_{int} is the value of the deadline stored in the realtime thread's release

- parameters object at the time t_{int} .
 - iv. The next release time for the realtime thread will be $t_{int} + T_{int}$, where T_{int} is the value of the period stored in the realtime thread's release parameters object at the time t_{int} .
 - v. Cost monitoring and enforcement is informed of the release event.
- When the thrown AIE instance is caught, the AIE becomes pending again (as per the usual semantics for AIE) until it is explicitly cleared.
8. Changes to release parameter types are treated as a pseudo 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 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 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 new parameters are aperiodic, the release is immediate, and when 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.

6.2.4.4.1 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 example Realtime Thread releases. The second is a UML state chart of the 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 finish. 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.

6.2.4.5 General 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.

The base 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

Figure 6.1: Sequence Diagram of Some Example Realtime Thread Releases

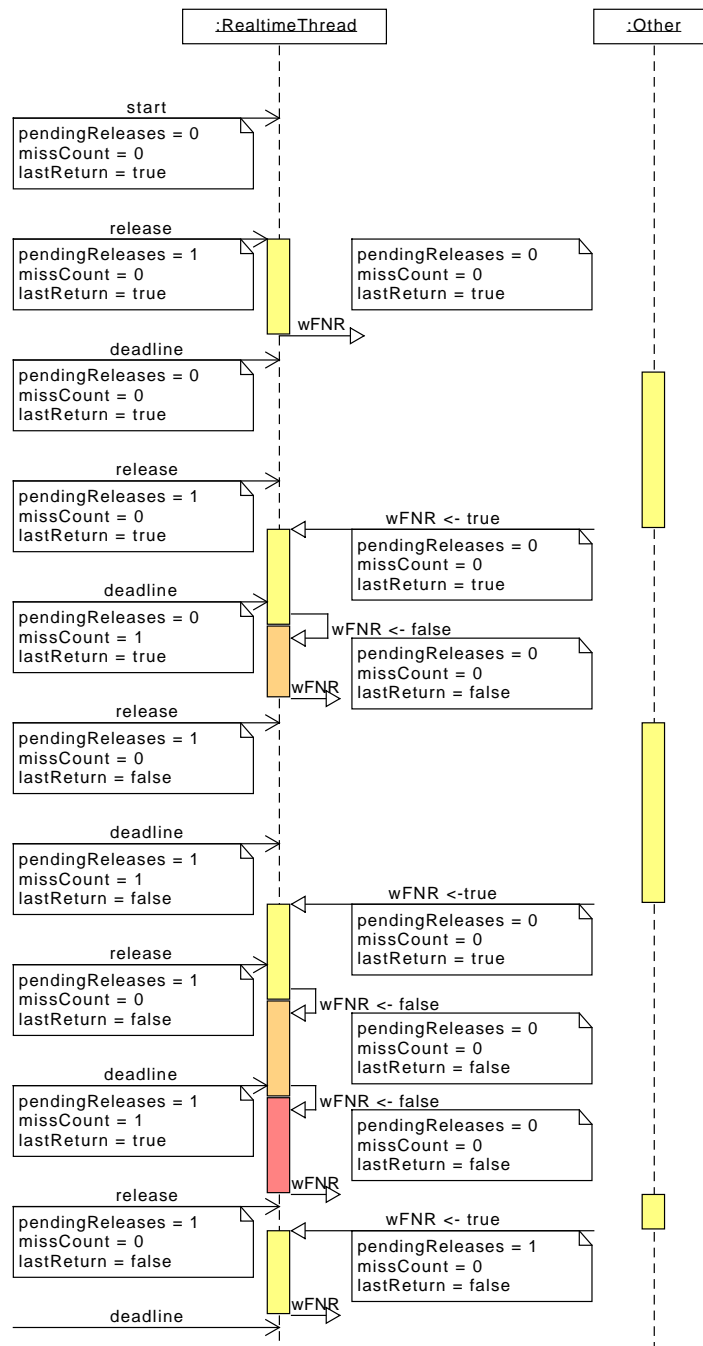


Figure 6.2: A State Chart for a Realtime Thread without a Deadline Miss Handler

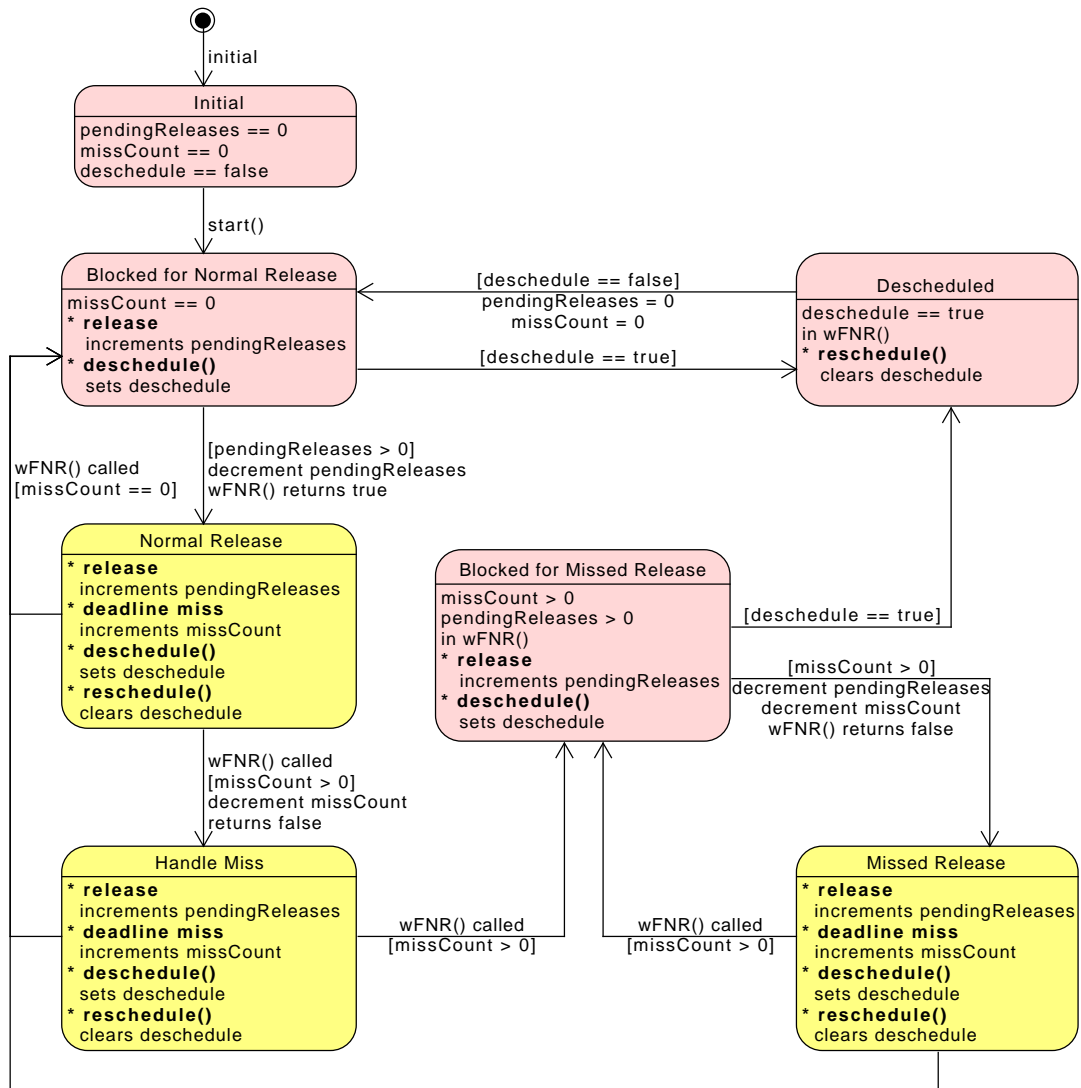
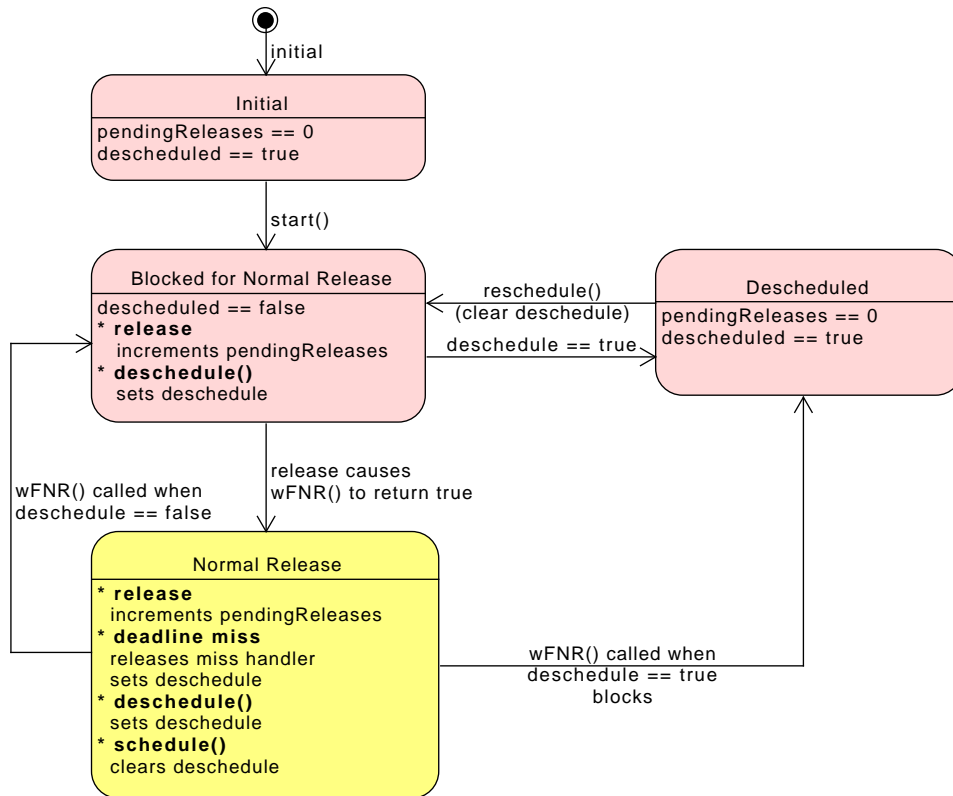


Figure 6.3: A State Chart for a Realtime Thread with a Deadline Miss Handler



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.

Policy	Action on Overflow
IGNORE	Silently ignore the arrival. The arrival is not accepted, no release event occurs, and, when the arrival was caused programmatically, such as by invoking <code>fire</code> on an asynchronous event, the caller is not informed that the arrival has been ignored.

EXCEPT	Throw an <code>ArrivalTimeQueueOverflowException</code> . The arrival is not accepted, and no release event occurs, but when the arrival was caused programmatically, the caller will have <code>ArrivalTimeQueueOverflowException</code> thrown.
REPLACE	The arrival replaces the latest release in the queue, when there is one, but no new 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, 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 deadline has already been missed or the queue length is zero, the behavior of the REPLACE policy is equivalent to the IGNORE policy.
SAVE	Behave effectively as if the queue were expanded as necessary to accommodate the new arrival. This expansion is permanent. The arrival is accepted and a release event occurs.
DISABLE	No queuing takes place. All incoming events increment the pending fire or release count. I may only be used where there is no payload and the release parameters are not sporadic.

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.

6.2.4.6 Sporadic Release Control

“Sporadic Release Control” is a special case of “General 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 schedulable objects 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.

<i>Arrival-Time Regulation</i>	
Policy	Action on Violation
IGNORE	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 <code>fire</code> on an asynchronous event), the caller is not informed that the arrival has been ignored.
EXCEPT	Throw a <code>MITViolationException</code> . The arrival is not accepted, and no release event occurs, but when the arrival was caused programmatically, the caller will have <code>MITViolationException</code> thrown.
REPLACE	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.

<i>Arrival-Time Regulation</i>	
Policy	Action on Violation
SAVE	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 async event handler until after its deadline has passed. In this case, the deadline miss handler is released at the deadline time even though the related async event has not yet reached its effective release. 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.

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.

6.2.4.7 Release Control for Asynchronous Event Handlers

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

1. Each firing of an associated asynchronous event is an arrival. 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 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, then the cost monitoring and enforcement system is told to reset for this handler.
9. The value of `ReleaseParameters.isRousable` controls whether a call to `Schedulable.interrupt` causes a premature release or only affects a running schedulable.
 - (a) When interrupt is called on an instance of Schedulable and the schedulable is running, the interrupt is made pending and as soon as AI code is entered, an AIE is thrown.

- (b) When `interrupt` is called on an instance of `RealtimeThread` that is waiting for its next release or its start time, the interrupt is made pending.
- (c) 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.
- (d) 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 immediately or not, but in both cases an AIE will be pending until the next AI method.

6.2.5 Associating Schedulables with Schedulers

The `Scheduler` associated with a `Schedulable` at the time it is started is derived from its configuration and the configuration of the task (an 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 `si` 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 `SchedulingGroup` instance associated with `si`. The `Scheduler` for `si` is determined as follows and in the order stated.

1. When `Scheduler.setScheduler(ns)` has been used to explicitly configure a scheduler for `si`, that scheduler will be the scheduler associated with `si`.
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 `si` 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 `si` will use the scheduler currently associated with `parent`.
4. When the default scheduler is an instance of the class returned by `sg.getScheduler()`, then `si` will use the default scheduler.
5. When none of these conditions hold, a scheduler cannot be determined for `si` and an `IllegalStateException` will be thrown.

Schedulables must always have a compatible `Scheduler` and `SchedulingParameters` any time these are explicitly configured. 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.

6.2.6 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`.

6.2.6.1 Scheduling Groups

The `SchedulingGroup` 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 tasks in a scheduling group may use a realtime scheduler,
2. instances of `Schedulable` may only be created in a scheduling group,
3. the root `ThreadGroup` instance must be an instance of `SchedulingGroup`,
4. the `ThreadGroup` instance of the initial thread must be an instance of `SchedulingGroup`,
5. an instance of `SchedulingGroup` may not have a parent that is not an instance of `SchedulingGroup`, and
6. all children of a `SchedulingGroup` allocated in a `ScopedMemory` must be instances of `SchedulingGroup`.

Furthermore, the enumeration methods on a scheduling group are aware of scoped memory and the referential integrity restrictions discussed in Chapter 11, Alternative Memory Areas. The enumeration methods of `SchedulingGroup` will not return references to any descendants allocated in a `ScopedMemory` to which references may not be made from the current allocation context. That is, if a newly allocated object in the current allocation context could not safely hold a reference to a descendant of the `ScopedMemory`, that descendant will not be included in the array returned by `enumerate()`. For processing such `SchedulingGroups`, a visitor must be used.

The maximum priority and scheduler restrictions on `SchedulingGroup` 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`.

Changing the maximum eligibility allowed to tasks in a `SchedulingGroup` (via the `SchedulingGroup.setMaxEligibility(SchedulingParameters)` method) takes effect immediately, and will do the following.

1. For any task *t* in the affected *SchedulingGroup* 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 *SchedulingGroup* child *sg* of the affected *SchedulingGroup* 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 *SchedulingGroup* children in *sg*.

6.2.6.2 Processing Groups

A processing group is defined by an instance of the *ProcessingGroup* subclass of *SchedulingGroup* and each schedulable that is bound to that parameter 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*, *SchedulingGroup* instances are members of the thread group hierarchy of thread groups in the system. Since a *SchedulingGroup* may have another *SchedulingGroup* instance as its ancestor, a task might be in more than one scheduling group.

6.2.6.2.1 Semantics for Processing Groups Open issue 6.2.1

Hierarchy not clear from points 10 and 11 (see above. –jjh)

End of issue 6.2.1

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 release 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) **period** can change at each release, so the next period is set based on the current value of the processing group's period;
 - (c) **deadline** can change at each release, so the next deadline is set based on the current value of the processing group's deadline;
 - (d) **OverflowHandler** can change at each release, so the overflowHandler is set based on the current value of the processing group's overflowHandler;
 - (e) **MissHandler** can change at each release, so the missHandler is set based on the current value of the processing group's missHandler; and
 - (f) **UnderrunHandler** can change 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 SchedulingGroup, such as affinity and scheduling parameters.

Open issue 6.2.2

API Issues:

- When is the memory not compatible with the scheduler?
- When is release not compatible with the scheduler?
- Why does a scheduler refuse to accept a schedulable due to its state?

End of issue 6.2.2

6.3 javax.realtime

6.3.1 Interfaces

6.3.1.1 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](#)² is by definition bound.

6.3.1.2 RealtimeExecutionContext

Description

All RTSJ objects that encapsulate execution. This type includes [Schedulable](#) and [javax.realtime.device.InterruptServiceRoutine](#). It is used by [Affinity](#) to remove the need to have a reference into the [javax.realtime.device](#) package.

6.3.1.3 Schedulable

Interfaces

[Runnable](#)

[javax.realtime.Timable](#)

[javax.realtime.RealtimeExecutionContext](#)

Description

Handlers and other objects can be dispatched by a [Scheduler](#)³ when they provide a `run()` method and the methods defined below. The [Scheduler](#)⁴ uses this information to create a suitable context to execute the `run()` method.

²Section [5.3.2.2](#)

³Section [6.3.3.12](#)

⁴Section [6.3.3.12](#)

6.3.1.3.1 Methods

getMemoryParameters

Signature

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

Description

Gets a reference to the [MemoryParameters](#)⁵ object for this schedulable.

Returns

A reference to the current [MemoryParameters](#)⁶ object.

setMemoryParameters(MemoryParameters)

Signature

```
public T extends javax.realtime.Schedulable<T>  
setMemoryParameters(MemoryParameters memory)
```

Description

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

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

Parameters

memory A [MemoryParameters](#)⁷ 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.)

Throws

⁵Section [11.3.2.4](#)

⁶Section [11.3.2.4](#)

⁷Section [11.3.2.4](#)

⁸Section [6.3.3.8](#)

`IllegalArgumentException` 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

getReleaseParameters

Signature

```
public javax.realtime.ReleaseParameters<?>
getReleaseParameters()
```

Description

Gets a reference to the `ReleaseParameters`⁹ object for this schedulable.

Returns

A reference to the current `ReleaseParameters`¹⁰ object.

setReleaseParameters(ReleaseParameters)

Signature

```
public T extends javax.realtime.Schedulable<T>
setReleaseParameters(javax.realtime.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

⁹Section 6.3.3.10

¹⁰Section 6.3.3.10

release A [ReleaseParameters](#)¹¹ 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](#)¹².)

Throws

`IllegalArgumentException` Thrown 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.

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

Returns

this

getScheduler

Signature

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

Description

Gets a reference to the [Scheduler](#)¹³ object for this schedulable.

Returns

A reference to the associated [Scheduler](#)¹⁴ object.

setScheduler(Scheduler)

Signature

```
public T extends javafx.realtime.Schedulable<T>  
setScheduler(Scheduler scheduler)
```

Description

¹¹Section [6.3.3.10](#)

¹²Section [6.3.3.8](#)

¹³Section [6.3.3.12](#)

¹⁴Section [6.3.3.12](#)

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

`IllegalArgumentException` Thrown 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.

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

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

Returns

this

setScheduler(Scheduler, SchedulingParameters, ReleaseParameters, MemoryParameters)

Signature

```
public T extends javax.realtime.Schedulable<T>
    setScheduler(Scheduler scheduler,
                SchedulingParameters scheduling,
                javax.realtime.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](#)¹⁵ 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](#)¹⁶.)

release A reference to the [ReleaseParameters](#)¹⁷ 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](#)¹⁸.)

memoryParameters A reference to the [MemoryParameters](#)¹⁹ 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](#)²⁰.)

Throws

[IllegalArgumentException](#) Thrown 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.

[SecurityException](#) when the caller is not permitted to set the scheduler for this schedulable.

Returns

this

getSchedulingParameters

Signature

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

Description

Gets a reference to the [SchedulingParameters](#)²¹ object for this schedulable.

Returns

¹⁵Section [6.3.3.14](#)

¹⁶Section [6.3.3.8](#)

¹⁷Section [6.3.3.10](#)

¹⁸Section [6.3.3.8](#)

¹⁹Section [11.3.2.4](#)

²⁰Section [6.3.3.8](#)

²¹Section [6.3.3.14](#)

A reference to the current [SchedulingParameters²²](#) object.

setSchedulingParameters(SchedulingParameters)

Signature

```
public T extends javax.realtime.Schedulable<T>  
    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 [SchedulingParameters²³](#) 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 [PriorityScheduler²⁴](#).)

Throws

`IllegalArgumentException` Thrown 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.

`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

this

getSchedulingGroup

Signature

²²Section [6.3.3.14](#)

²³Section [6.3.3.14](#)

²⁴Section [6.3.3.8](#)


```
public javax.realtime.SchedulingGroup  
getSchedulingGroup()
```

Description

Gets a reference to the [SchedulingGroup](#)²⁵ instance of this schedulable.

Returns

A reference to the current [SchedulingGroup](#)²⁶ object.

Available since since RTSJ 2.0

getConfigurationParameters

Signature

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

Description

Gets a reference to the [ConfigurationParameters](#)²⁷ object for this schedulable.

Returns

A reference to the associated [ConfigurationParameters](#)²⁸ object.

Available since RTSJ 2.0

getMinConsumption(RelativeTime)

Signature

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

Description

Determine the minimum CPU consumption for this schedulable in any single release. When this method is called on the current schedulable, the CPU consumption of the current release is not considered. When dest is null, return the

²⁵Section [6.3.3.13](#)

²⁶Section [6.3.3.13](#)

²⁷Section [5.3.2.1](#)

²⁸Section [5.3.2.1](#)

minimum consumption in a [RelativeTime](#)²⁹ instance from the current allocation context. When dest is not null, return the minimum consumption in dest

Parameters

dest when not null is 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

getMaxConsumption(RelativeTime)

Signature

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

Description

Determine the maximum CPU consumption for this schedulable in any single release. When this method is called on the current schedulable, the CPU consumption of the current release is not considered. When dest is null, return the maximum consumption in a [RelativeTime](#)³⁰ instance from the current allocation context. When dest is not null, return the maximum consumption in dest

Parameters

²⁹Section [9.3.1.3](#)

³⁰Section [9.3.1.3](#)

dest when not null is the object in which to return the result.

Returns

the maximum time consumed in any release.

Available since RTSJ 2.0

getMaxConsumption

Signature

```
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

```
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 daemon. 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

[IllegalSchedulableStateException](#) when this schedulable is active.

SecurityException when the current schedulable cannot modify this event handler.

Available since RTSJ 2.0

isDaemon

Signature

```
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

Determine 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()
```

Description

Make the generic [AsynchronouslyInterruptedException](#)³¹ 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 cause an

³¹Section [8.3.2.1](#)

early release. In any case, `AsynchronouslyInterruptedException` is thrown once a method is entered that implements `AsynchronouslyInterruptedException`.

Available since RTSJ 2.0

isInterrupted

Signature

```
public boolean  
isInterrupted()
```

Description

Determines whether or not the generic `AsynchronouslyInterruptedException`³² is pending.

Returns

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

Available since RTSJ 2.0

awaken

Signature

```
public void  
awaken()  
throws IllegalStateException
```

Description

Provides a means for a `Clock`³³ to end a sleep.

Throws

`IllegalStateException` when called from user code.

Available since RTSJ 2.0

³²Section 8.3.2.1

³³Section 10.3.2.1

6.3.2 Enumerations

6.3.2.1 MinimumInterarrivalPolicy

Inheritance

```
java.lang.Object
  java.lang.Enum
    javax.realtime.MinimumInterarrivalPolicy
```

Description

Defines the set of policies for handling interarrival time violations in [SporadicParameters](#)³⁴. Each policy governs every instance of [Schedulable](#)³⁵ which has [SporadicParameters](#)³⁶ with that minimum interarrival time policy.

Available since RTSJ 2.0

6.3.2.1.1 Enumeration Constants

EXCEPT

```
public static final 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](#)³⁷.

IGNORE

```
public static final IGNORE
```

Description

³⁴Section [6.3.3.15](#)

³⁵Section [6.3.1.3](#)

³⁶Section [6.3.3.15](#)

³⁷Section [15.2.2.7](#)

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

public static final 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

public static final 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

```
public static javafx.realtime.MinimumInterarrivalPolicy[]  
values()
```

Description

valueOf(String)*Signature*

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

*Description***value***Signature*

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

Description

Determine the string corresponding to this value.

Returns

the corresponding string.

value(String)*Signature*

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

Description

Convert a string into a policy type.

Parameters

value is the string to convert.

Returns

the corresponding policy type.

6.3.2.2 QueueOverflowPolicy

Inheritance

java.lang.Object
 java.lang.Enum
 javax.realtime.QueueOverflowPolicy

Description

Defines the set of policies for handling overflow on event queues used by [ReleaseParameters](#)³⁸. 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 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](#)³⁹ with no payload and instances of [RealtimeThread](#)⁴⁰ with [PeriodicParameters](#)⁴¹. In contrast to [IGNORE](#)⁴², 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](#)⁴³ with [SporadicParameters](#)⁴⁴. This policy is also the default for

³⁸Section 6.3.3.10

³⁹Section 8.3.1.1

⁴⁰Section 5.3.2.2

⁴¹Section 6.3.3.6

⁴²Section 6.3.2.2.1

⁴³Section 6.3.1.3

⁴⁴Section 6.3.3.15

[PeriodicParameters](#)⁴⁵. Instances of `RealtimeThread` without with null release parameters have this policy implicitly, as they do not have an event queue either.

EXCEPT

public static final 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](#)⁴⁶. When `fire` is used within a [Timer](#)⁴⁷, the exception is ignored and the `fire` does nothing, i.e., it acts the same as "IGNORE".

IGNORE

public static final 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 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.

⁴⁵Section [6.3.3.6](#)

⁴⁶Section [15.2.2.1](#)

⁴⁷Section [10.3.2.6](#)

SAVE

public static final 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](#)⁴⁸.

6.3.2.2.2 Methods

values

Signature

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

Description

valueOf(String)

Signature

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

Description

⁴⁸Section [6.3.3.2](#)

value

Signature

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

Description

Determine the string corresponding to this value.

Returns

the corresponding string.

value(String)

Signature

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

Description

Convert 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

Inheritance

```
java.lang.Object  
  javax.realtime.Affinity
```

Description

This is the API for all processor-affinity-related aspects of the RTSJ. It includes a factory that generates Affinity objects, and methods that control the CPU affinity used by java.lang.ThreadGroup to control the affinity of all its tasks. With it, the affinity of every task in the JVM can be controlled.

An affinity is a set of processors that can be associated with certain types of tasks. Each task (`java.lang.Thread` and `RealtimeExecutionContext`⁴⁹) can be associated with an affinity. Groups of these can be assigned an affinity through their `java.lang.ThreadGroup`.

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

The processor membership of an affinity set is immutable. The tasks associations of an affinity set 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 set factory only generates usable Affinity instances; i.e., affinity sets that (at least when they are created) can be used with `set(Affinity, RealtimeExecutionContext)`⁵⁰, `set(Affinity, Thread)`⁵¹, and `set(Affinity, ThreadGroup)`⁵². The factory cannot create an affinity set with more than one processor member, but such affinity sets are supported. They may be internally created by the RTSJ runtime at startup time.

The set of affinity sets created at startup (the predefined set) is visible through the `getPredefinedAffinities(Affinity[])`⁵³ method. The affinity set factory may be used to create affinity sets 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 set with the schedulable's affinity set must contain at least one entry.

⁴⁹Section 6.3.1.2

⁵⁰Section 6.3.3.1.1

⁵¹Section 6.3.3.1.1

⁵²Section 6.3.3.1.1

⁵³Section 6.3.3.1.1

Trying to set a task's affinity outside its thread group always fails. Trying to setting the affinity of a thread group that does not intersect with the thread group of its tasks will also fail.

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

- Java threads do not inherit affinity from [Schedulable](#)⁵⁴s,
- instances of [AsyncBaseEventHandler](#)⁵⁵ that are not bound do not inherit affinity, and
- Schedulingables do not inherit affinity from Java threads.

When a task does not inherit its creator's affinity set, its initial affinity set is set to all processors and is thus only limited by its thread group.

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

Available since RTSJ 2.0

6.3.3.1.1 Methods

getPredefinedAffinitiesCount

Signature

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

Description

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

Returns

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

getPredefinedAffinities

Signature

⁵⁴Section [6.3.1.3](#)

⁵⁵Section [8.3.3.3](#)

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

Description

Equivalent to invoking `getPredefinedAffinitySets(null)`.

Returns

an array of the predefined affinity sets.

getPredefinedAffinities(Affinity)*Signature*

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

Description

Determine what affinity sets are predefined by the Java runtime.

Parameters

`dest` The destination array, or null.

Throws

`IllegalArgumentException` when `dest` is not large enough.

Returns

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

isSetAffinitySupported*Signature*

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

Description

Determine whether or not affinity control is supported.

Returns

true when the `set(Affinity, Thread)`⁵⁶ family of methods is supported.

⁵⁶Section 6.3.3.1.1

generate(BitSet)

Signature

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

Description

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

Platforms that support specific affinity sets will register those Affinity instances with [Affinity](#)⁵⁷. They appear in the arrays returned by [getPredefinedAffinities\(\)](#)⁵⁸ and [getPredefinedAffinities\(Affinity\[\]\)](#)⁵⁹.

Parameters

set is the BitSet to convert into an Affinity.

Throws

NullPointerException when set is null.

IllegalArgumentException when set does not refer to a valid set of processors, where “valid” is defined as the bitset from a predefined affinity set, or a bitset of cardinality one containing a processor from the set returned by [getAvailableProcessors\(\)](#). The definition of “valid set of processors” is system dependent; however, every set consisting of one valid processor makes up a valid bit set, and every bit set correspond to a predefined affinity set is valid.

Returns

The resulting Affinity.

getAvailableProcessors

Signature

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

Description

This method is equivalent to [getAvailableProcessors\(BitSet\)](#)⁶⁰ with a null argument.

⁵⁷Section [6.3.3.1](#)

⁵⁸Section [6.3.3.1.1](#)

⁵⁹Section [6.3.3.1.1](#)

⁶⁰Section [6.3.3.1.1](#)

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\(BitSet\)](#)⁶¹.

get(RealtimeExecutionContext)*Signature*

```
public static final javafx.runtime.Affinity  
get(RealtimeExecutionContext task)
```

Description

Determine the affinity set instance associated with task.

Parameters

task is the execution context to query.

Returns

The associated affinity.

⁶¹Section [6.3.3.1.1](#)

set(Affinity, RealtimeExecutionContext)

Signature

```
public static final void  
set(Affinity set,  
    RealtimeExecutionContext task)  
throws ProcessorAffinityException
```

Description

Set the processor affinity of a task.

Parameters

set is the processor affinity
task is the execution context whose affinity will be set.

Throws

IllegalArgumentException when the intersection of set the affinity of any Thread-Group instance containing task is empty.

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

NullPointerException when set or task is null.

get(Thread)

Signature

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

Description

Determine the affinity set instance associated with thread.

Parameters

thread a Java thread, or one of its subclasses (including **RealtimeThread**⁶²).

Returns

The associated affinity set.

⁶²Section 5.3.2.2

set(Affinity, Thread)

Signature

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

Description

Set the processor affinity of a Java thread or [RealtimeThread](#)⁶³ to set.

Parameters

set The processor affinity set
thread The thread or realtime thread.

Throws

[IllegalArgumentException](#) when the intersection of set and the affinity of any [ThreadGroup](#) instance containing thread is empty.
[ProcessorAffinityException](#) when the runtime fails to set the affinity for platform-specific reasons.
[NullPointerException](#) when set or thread is null.

get(ThreadGroup)

Signature

```
public static final javax.realtime.Affinity  
get(ThreadGroup group)
```

Description

Determine the affinity set instance associated with group.

Parameters

group An instance of `java.lang.ThreadGroup`

Returns

The associated affinity set.

⁶³Section [5.3.2.2](#)

set(Affinity, ThreadGroup)

Signature

```
public static final void  
set(Affinity set,  
    ThreadGroup group)  
throws ProcessorAffinityException
```

Description

Set the processor affinity of group to set with immediate effect.

Parameters

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

Throws

IllegalArgumentException when the intersection of set and the affinity of any task in group is empty, or when the disjunction of set and the affinity of any ThreadGroup containing group is non-empty.

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

NullPointerException when set or group is null.

getProcessors

Signature

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

Description

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

Returns

A newly created BitSet representing this Affinity.

getProcessors(BitSet)

Signature

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

Description

Determine 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

Ask 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.

applyTo(BoundAsyncEventHandler)*Signature*

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

Description

Set the processor affinity of a bound AEH to this.

Parameters

aeh The bound async event handler

Throws

IllegalArgumentException when intersection of this with the affinity of any group containing aeh is empty.

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

NullPointerException when aeh is null.

applyTo(Thread)

Signature

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

Description

Set the processor affinity of a Java thread or [RealtimeThread](#)⁶⁴ to this.

Parameters

thread The thread or realtime thread.

Throws

[IllegalArgumentException](#) when intersection of this with the affinity of any group containing thread is empty.

[ProcessorAffinityException](#) when the runtime fails to set the affinity for platform-specific reasons.

[NullPointerException](#) when thread is null.

applyTo(ThreadGroup)

Signature

```
public final void  
applyTo(ThreadGroup group)  
throws ProcessorAffinityException
```

Description

Set the processor affinity of group to this.

Parameters

group The processing group parameters instance.

Throws

[IllegalArgumentException](#) when the intersection of this and the affinity of any task in group is empty, or when the disjunction of this and the affinity of any [ThreadGroup](#) containing group is non-empty.

[ProcessorAffinityException](#) when the runtime fails to set the affinity for platform-specific reasons or group contains more than one processor.

[NullPointerException](#) when group is null.

⁶⁴Section [5.3.2.2](#)

applyTo(ActiveEventDispatcher)

Signature

```
public final void  
applyTo(javafx.runtime.ActiveEventDispatcher<?, ?> dispatcher)  
throws ProcessorAffinityException
```

Description

Set the processor affinity of dispatcher to this.

Parameters

dispatcher is the dispatcher instance.

Throws

IllegalArgumentException when intersection of this with the affinity of any group containing dispatcher is empty.

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

NullPointerException when dispatcher is null.

6.3.3.2 AperiodicParameters

Inheritance

java.lang.Object

javafx.runtime.ReleaseParameters

javafx.runtime.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 that the scheduler that the parameter object has changed. At that point the object is reconsidered for every schedulable that uses it. Invoking a

method on the `RelativeTime` object that is the cost for this object may change the cost but it does not pass the change to the scheduler at that time. That change must not change the behavior of the schedulable's that use the parameter object until a setter method on the `AperiodicParameters` object is invoked, or the parameter object is used in `setReleaseParameters()` or a constructor for a schedulable.

The implementation must use modified copy semantics for each `HighResolutionTime`⁶⁵ 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`⁶⁶ the arrival time is the time at which the `start()` is invoked. For other instances of `Schedulable`⁶⁷, the required behaviors may require the implementation to behave 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()`⁶⁸ method is called.

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

Table 6.3: `AperiodicParameters` Default Values

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

⁶⁵Section 9.3.1.2

⁶⁶Section 5.3.2.2

⁶⁷Section 6.3.1.3

⁶⁸Section 5.3.2.2.2

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

6.3.3.2.1 Fields

6.3.3.2.2 Constructors

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

Signature

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

Description

Create 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.

roustable determines whether or not an instance of `Schedulable` can be prematurely released by a thread interrupt.

Throws

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

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

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)`⁶⁹ 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)`.

⁶⁹Section 6.3.3.2.2

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

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

Throws

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

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

AperiodicParameters(RelativeTime, AsyncEventHandler, boolean)

Signature

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

Description

Equivalent to `AperiodicParameters(RelativeTime, RelativeTime, AsyncEventHandler, AsyncEventHandler, boolean)`⁷⁰ with the argument list (null, deadline, null, missHandler, rousable).

Available since RTSJ 2.0

AperiodicParameters(RelativeTime)

Signature

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

Description

⁷⁰Section 6.3.3.2.2

Equivalent to `AperiodicParameters(RelativeTime, RelativeTime, AsyncEventHandler, AsyncEventHandler, boolean)`⁷¹ with the argument list (null, deadline, null, null, false).

Available since RTSJ 2.0

AperiodicParameters

Signature

```
public  
AperiodicParameters()
```

Description

Equivalent to `AperiodicParameters(RelativeTime, RelativeTime, AsyncEventHandler, AsyncEventHandler, boolean)`⁷² with the argument list (null, null, null, null, false).

Available since RTSJ 1.0.1

6.3.3.2.3 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 (by being passed through the schedulable's constructor or set with a method such as `RealtimeThread.setReleaseParameters(ReleaseParameters)`⁷³) the deadline of those schedulables is altered as specified by each schedulable's respective scheduler.

⁷¹Section 6.3.3.2.2

⁷²Section 6.3.3.2.2

⁷³Section 5.3.2.2.2

Parameters

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

Throws

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

`IllegalAssignmentError` `IllegalAssignmentError` when deadline cannot be stored in this.

Returns

this

6.3.3.3 BackgroundParameters**Inheritance**

java.lang.Object
 javafx.runtime.ReleaseParameters
 javafx.runtime.BackgroundParameters

Description

Parameters for realtime threads that are only released once. A thread using this release parameters may not use `RealtimeThread.waitForNextRelease()`⁷⁴ or have its `RealtimeThread.release()`⁷⁵ methods called. Calling these methods results in an `IllegalThreadStateException`. Event handlers may not use this type of `ReleaseParameters`.

6.3.3.3.1 Constructors**BackgroundParameters(RelativeTime, RelativeTime, AsyncEventHandler, AsyncEventHandler)***Signature*

⁷⁴Section 5.3.2.2.2

⁷⁵Section 5.3.2.2.2

```

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. **Available since** RTSJ 2.0

Parameters

cost is the maximum cost for the initial release
 deadline is the deadline for the initial release
 overrunHandler is the handler to call on cost overrun.
 missHandler is the handler to call on deadline miss.

Throws

`IllegalArgumentException` when the time value of cost is less than zero, or the time value of deadline is less than or equal to zero, or the chronograph associated with the cost or deadline parameters is not an instance of `Clock`⁷⁶.

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

BackgroundParameters(RelativeTime, AsyncEventHandler)

Signature

```

public
BackgroundParameters(RelativeTime deadline,
                     AsyncEventHandler missHandler)

```

Description

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

⁷⁶Section 10.3.2.1

Available since RTSJ 2.0

BackgroundParameters

Signature

```
public
BackgroundParameters()
```

Description

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

6.3.3.4 FirstInFirstOutScheduler

Inheritance

```
java.lang.Object
  javafx.runtime.Scheduler
    javafx.runtime.PriorityScheduler
      javafx.runtime.FirstInFirstOutScheduler
```

Description

A version of [PriorityScheduler⁷⁷](#) 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

Attribute	Default Value
Priority	norm priority

⁷⁷Section [6.3.3.8](#)

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()`⁷⁸ method is the *base scheduler* and is returned by `Scheduler.getDefaultScheduler()`⁷⁹ unless the default scheduler is reset with `Scheduler.setDefaultScheduler(Scheduler)`⁸⁰.

Available since RTSJ 2.0

6.3.3.4.1 Methods

instance

Signature

```
public static javax.realtime.FirstInFirstOutScheduler  
instance()
```

Description

Obtain 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

```
public int  
getMaxPriority()
```

Description

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

Returns

The value of the maximum priority.

⁷⁸Section 6.3.3.4.1

⁷⁹Section 6.3.3.12.2

⁸⁰Section 6.3.3.12.2

getMinPriority

Signature

```
public int  
getMinPriority()
```

Description

Obtain 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

Obtain 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

Obtain the policy name of this.

Returns

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

reschedule(Thread, int)*Signature*

```
public void
reschedule(Thread thread,
            int priority)
```

Description

Promotes a `java.lang.Thread` to realtime priority under this scheduler. The affected thread will be scheduled as if it were a [RealtimeThread](#)⁸¹ of the given priority. This does not make the affected thread a `RealtimeThread`, however, and it will not have access to facilities reserved for instances of `RealtimeThread`.

Parameters

`thread` The thread to promote to realtime scheduling.

`priority` An integer priority equivalent to a priority set via [PriorityParameters](#)⁸² on a `RealtimeThread`.

Throws

`IllegalArgumentException` when `priority` is not between [getMinPriority\(\)](#)⁸³ and [getMaxPriority\(\)](#)⁸⁴, inclusive.

6.3.3.5 ImportanceParameters

Inheritance

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

⁸¹Section [5.3.2.2](#)

⁸²Section [6.3.3.7](#)

⁸³Section [6.3.3.4.1](#)

⁸⁴Section [6.3.3.4.1](#)

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](#)⁸⁵ must not consider importance.

6.3.3.5.1 Constructors

ImportanceParameters(int, int)

Signature

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

Description

Create 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.

6.3.3.5.2 Methods

getImportance

Signature

```
public int  
    getImportance()
```

Description

Gets the importance value.

⁸⁵Section [6.3.3.8](#)

Returns

The value of importance for the associated instances of [Schedulable](#)⁸⁶.

setImportance(int)*Signature*

```
public javax.realtime.ImportanceParameters  
    setImportance(int importance)
```

Description

Set the importance value. When this parameter object is associated with any schedulable (by being passed through the schedulable's constructor or set with a method such as [RealtimeThread.setSchedulingParameters\(SchedulingParameters\)](#)⁸⁷) 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

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

Returns

this

toString*Signature*

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

Description

Print the value of the priority and importance values of the associated instance of [Schedulable](#)⁸⁸

⁸⁶Section [6.3.1.3](#)

⁸⁷Section [5.3.2.2.2](#)

⁸⁸Section [6.3.1.3](#)

6.3.3.6 PeriodicParameters

Inheritance

```
java.lang.Object
  javafx.realtime.ReleaseParameters
    javafx.realtime.PeriodicParameters
```

Description

This release parameter indicates that the schedulable is released on a regular basis. For an [AsyncEventHandler](#)⁸⁹, this means that the handler is either released by a periodic timer, or the associated event occurs periodically. For a [RealtimeThread](#)⁹⁰, this means that the [RealtimeThread.waitForNextRelease](#)⁹¹ method will unblock the associated realtime thread at the start of each period.

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

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

Periodic parameters use [HighResolutionTime](#)⁹² values for period and start time. Since these times are expressed as a [HighResolutionTime](#)⁹³ 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

⁸⁹Section [8.3.3.5](#)

⁹⁰Section [5.3.2.2](#)

⁹¹Section [5.3.2.2.2](#)

⁹²Section [9.3.1.2](#)

⁹³Section [9.3.1.2](#)

with each time value.

The implementation must use modified copy semantics for each `HighResolutionTime`⁹⁴ 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

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

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

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

⁹⁴Section 9.3.1.2

Signature

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

Description

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

Available since RTSJ 2.0

Open issue 6.3.1

If an event handler is periodic but NOT released by a Timer, where is the deadline measured from? It cannot be from the (i - 1)th release + Period. JJH: deadline should always be measured starting at the time fire is called.

End of issue 6.3.1*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.

rousalable when true, and interrupt will cause an early release, otherwise not.

Throws

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

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

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

Signature

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

Description

Equivalent to `PeriodicParameters(HighResolutionTime, RelativeTime, RelativeTime, RelativeTime, AsyncEventHandler, AsyncEventHandler, boolean)`⁹⁵ with the argument list (start, period, cost, deadline, overrunHandler, missHandler, false);

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

Signature

⁹⁵Section 6.3.3.6.1


```
public
PeriodicParameters(javafx.realtime.HighResolutionTime<?> start,
                    RelativeTime period,
                    RelativeTime deadline,
                    AsyncEventHandler missHandler,
                    boolean rousable)
```

Description

Equivalent to `PeriodicParameters(HighResolutionTime, RelativeTime, RelativeTime, RelativeTime, AsyncEventHandler, AsyncEventHandler, boolean)`⁹⁶ with the argument list (start, period, deadline, null, null, missHandler, rousable);

Available since RTSJ 2.0

PeriodicParameters(HighResolutionTime, RelativeTime)

Signature

```
public
PeriodicParameters(javafx.realtime.HighResolutionTime<?> start,
                    RelativeTime period)
```

Description

Equivalent to `PeriodicParameters(HighResolutionTime, RelativeTime, RelativeTime, RelativeTime, AsyncEventHandler, AsyncEventHandler, boolean)`⁹⁷ with the argument list (start, period, null, null, null, null, false);

Available since RTSJ 1.0.1

PeriodicParameters(RelativeTime)

Signature

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

⁹⁶Section 6.3.3.6.1

⁹⁷Section 6.3.3.6.1

Description

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

Available since RTSJ 1.0.1

6.3.3.6.2 Methods

getPeriod

Signature

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

Description

Determine 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

Determine 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.

getStart

Signature

```
public javafx.runtime.HighResolutionTime<?>  
getStart()
```

Description

Determine the time used to start an instance of `Schedulable`, which is not necessarily the time at which it actually started.

Returns

the object last used to set the start containing the current value of start.

setDeadline(RelativeTime)

Signature

```
public javafx.runtime.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)`⁹⁸) the deadline of those schedulables is altered as specified by each schedulable's respective scheduler.

Parameters

deadline The latest permissible completion time measured from the release time of the associated invocation of the schedulable. When deadline is null, the deadline is set to a new instance of `RelativeTime` equal to period.

Throws

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

`IllegalAssignmentError` `IllegalAssignmentError` when deadline cannot be stored in this.

Returns

this

⁹⁸Section 5.3.2.2.2

setPeriod(RelativeTime)

Signature

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

Description

Sets the period.

Parameters

period The value to which period is set.

Throws

`IllegalArgumentException` when the given period is null or its time value is not greater than zero. Also when period is incompatible with the scheduler for any associated schedulable or when an associated [AsyncBaseEventHandler](#)⁹⁹ is associated with a [Timer](#)¹⁰⁰ whose period does not match period.

`IllegalAssignmentError` when period cannot be stored in this.

Returns

this

setStart(HighResolutionTime)

Signature

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

Description

Sets the start time.

The effect of changing the start time for any schedulables associated with this parameter object is determined by the scheduler associated with each schedulable.

Note: An instance of `PeriodicParameters` may be shared by several schedulables. A change to the start time may take effect on a subset of these schedulables. That leaves the start time returned by `getStart` unreliable as a way to determine the start time of a schedulable.

Parameters

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

⁹⁹Section [8.3.3.3](#)

¹⁰⁰Section [10.3.2.6](#)

Throws

`IllegalArgumentException` when the given start time is incompatible with the scheduler for any of the schedulable objects which are presently using this parameter object.

`IllegalAssignmentError` when start cannot be stored in this.

Returns

this

6.3.3.7 PriorityParameters

Inheritance

`java.lang.Object`

`javax.realtime.SchedulingParameters`

`javax.realtime.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`¹⁰¹ is such a scheduler.

6.3.3.7.1 Constructors

PriorityParameters(int)

Signature

```
public
PriorityParameters(int priority)
```

Description

Create an instance of `PriorityParameters`¹⁰² with the given priority.

Parameters

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

¹⁰¹Section 6.3.3.8

¹⁰²Section 6.3.3.7

6.3.3.7.2 Methods

isCompatible(Class)

Signature

```
public boolean  
isCompatible(java.lang.Class<javax.realtime.Scheduler> type)
```

Description

Determine whether this scheduling parameters can be used by tasks scheduled by instances of type.

Parameters

type of scheduler to check against

Returns

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

Available since RTSJ 2.0

getPriority

Signature

```
public int  
getPriority()
```

Description

Gets the priority value.

Returns

The priority.

setPriority(int)

Signature

```
public javax.realtime.PriorityParameters  
setPriority(int priority)
```

Description

Set the priority value. When this parameter object is associated with any schedulable (by being passed through the schedulable's constructor or set with a method such as `RealtimeThread.setSchedulingParameters(SchedulingParameters)`¹⁰³) 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

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

Returns

this

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

Inheritance

```
java.lang.Object  
  javax.realtime.Scheduler  
    javax.realtime.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.

¹⁰³Section 5.3.2.2.2

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

Table 6.6: PriorityScheduler Default PriorityParameter Values

Attribute	Default Value
Priority	norm priority

Note that the system contains one instance of the PriorityScheduler which is the system's base scheduler and is returned by `FirstInFirstOutScheduler.instance()`¹⁰⁴. 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()`¹⁰⁵ unless the default scheduler is changed with `Scheduler.setDefaultScheduler(Scheduler)`¹⁰⁶.

6.3.3.8.1 Fields

6.3.3.8.2 Constructors

PriorityScheduler

Signature

```
protected
PriorityScheduler()
```

Description

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

¹⁰⁴Section 6.3.3.4.1

¹⁰⁵Section 6.3.3.12.2

¹⁰⁶Section 6.3.3.12.2

6.3.3.8.3 Methods

getPolicyName

Signature

```
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  
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  
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  
getNormPriority()
```

Description

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

Returns

The value of the normal priority.

reschedule(Thread, int)

Signature

```
public abstract void  
reschedule(Thread thread,  
            int priority)
```

Description

Promotes a `java.lang.Thread` to realtime priority under this scheduler. The affected thread will be scheduled as if it were a [RealtimeThread](#)¹⁰⁷ of the given priority. This does not make the affected thread a `RealtimeThread`, however, and it will not have access to facilities reserved for instances of `RealtimeThread`.

Parameters

`thread` The thread to promote to realtime scheduling.

`priority` An integer priority equivalent to a priority set via [PriorityParameters](#)¹⁰⁸ on a `RealtimeThread`.

Throws

`IllegalArgumentException` when `priority` is not between [getMinPriority\(\)](#)¹⁰⁹ and [getMaxPriority\(\)](#)¹¹⁰, inclusive.

Available since RTSJ 2.0

¹⁰⁷Section [5.3.2.2](#)

¹⁰⁸Section [6.3.3.7](#)

¹⁰⁹Section [6.3.3.8.3](#)

¹¹⁰Section [6.3.3.8.3](#)

6.3.3.9 ProcessingGroup

Inheritance

```
java.lang.Object
  java.lang.ThreadGroup
    javafx.realtime.SchedulingGroup
      javafx.realtime.ProcessingGroup
```

Description

A descendant class of ThreadGroup for handling tasks (instances of [Schedulable](#)¹¹¹ and java.lang.Thread) as a group. As with ThreadGroup and [SchedulingGroup](#)¹¹², 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 SchedulingGroup.

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 groups 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 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

¹¹¹Section [6.3.1.3](#)

¹¹²Section [6.3.3.13](#)

when its deadline expires, any associated `missHandler` is released. When the server is logically executable when its next release time occurs, any associated `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 `HighResolutionTime`¹¹³ values for cost, deadline, period and start time. Since those times are expressed as a `HighResolutionTime`¹¹⁴, 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 `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 `ProcessingGroup` object becomes the processing group parameters object bound to that schedulable object. Changes to the values in the `ProcessingGroup` object affect that schedulable object. When bound to more than one schedulable then changes to the values in the `ProcessingGroup` 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 copy semantics for each `HighResolutionTime`¹¹⁵ 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 `ProcessingGroup` object caused by methods on that object are immediately visible to the scheduler. For instance, invoking `setPeriod()` on a `ProcessingGroup` object will make the change, then notify that the scheduler that the parameter object has changed. At that point the scheduler's view of the processing group parameters object is updated. Invoking a method on the `RelativeTime` object that is the period for this object may change the period but it does not pass the change to the scheduler at that time. That new value for period must not change the behavior of the SOs that use the parameter object until a setter method on the `ProcessingGroup` object is invoked or a constructor for an SO.

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 class who are mutating instances will be doing their own synchronization at a higher level.

¹¹³Section 9.3.1.2

¹¹⁴Section 9.3.1.2

¹¹⁵Section 9.3.1.2

Table 6.7: ProcessingGroup Default Values

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

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(SchedulingGroup, String, HighResolutionTime, RelativeTime, RelativeTime, AsyncEventHandler, RelativeTime, AsyncEventHandler)

Signature

```
public
ProcessingGroup(SchedulingGroup parent,
                String name,
                javafx.realtime.HighResolutionTime<?> start,
                RelativeTime period,
                RelativeTime cost,
                AsyncEventHandler overrun,
                RelativeTime minimum,
                AsyncEventHandler underrun)
```

Description

Create a ProcessingGroup

Parameters

parent is the parent [SchedulingGroup](#)¹¹⁶ of this ProcessingGroup.

name is a string identifier for this group.

start is when monitoring should begin.

period is an amount of time for cost and overrun monitoring and for cost enforcement.

cost is the maximum total execution time of all tasks in the group during a given period.

overrun is called when the the total execution of all tasks in the group exceeds cost for a given period.

minimum is the least amount of processing time for all the tasks in this group together.

underrun is called at the end of period when the total processing time of all tasks was less than minimum in the last period.

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

Signature

```
public
ProcessingGroup(SchedulingGroup parent,
                String name,
                java.xml.realtime.HighResolutionTime<?> start,
                RelativeTime period,
                RelativeTime cost,
                AsyncEventHandler overrun)
```

Description

Equivalent to [ProcessingGroup\(SchedulingGroup, String, HighResolutionTime, RelativeTime, RelativeTime, AsyncEventHandler, RelativeTime, AsyncEventHandler\)](#)¹¹⁷ with the argument list (parent, name, start, period, cost, overrun, null, null).

¹¹⁶Section [6.3.3.13](#)

¹¹⁷Section [6.3.3.9.1](#)

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

Signature

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

Description

Equivalent to `ProcessingGroup(SchedulingGroup, String, HighResolutionTime, RelativeTime, RelativeTime, AsyncEventHandler, RelativeTime, AsyncEventHandler)`¹¹⁸ with the argument list (Scheduler.currentSchedulable().getSchedulingGroup(), name, start, period, cost, overrun, null, null).

6.3.3.9.2 Methods

getEffectiveStart(AbsoluteTime)

Signature

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

Description

Obtain the actual time of the group started 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 is 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.

¹¹⁸Section 6.3.3.9.1

getEffectiveStart

Signature

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

Description

Obtain the actual time of the group started as recorded by the system.

Equivalent to `getEffectiveStart(AbsoluteTime)`¹¹⁹ where `dest` is set to null.

Returns

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

getPeriod(RelativeTime)

Signature

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

Description

Gets the value of period and returns it in the provided `RelativeTime`¹²⁰ 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.

getPeriod

Signature

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

Description

¹¹⁹Section 6.3.3.9.2

¹²⁰Section 9.3.1.3

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

Returns

A reference to a newly allocated instance of `RelativeTime`¹²¹ that represents the value of period.

setPeriod(RelativeTime)

Signature

```
public javax.realtime.ProcessingGroup  
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.

Returns

`this`

getMaxCost(RelativeTime)

Signature

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

Description

¹²¹Section 9.3.1.3

Gets the value of cost. and returns it in the provided [RelativeTime](#)¹²² 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.

getMaxCost

Signature

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

Description

Gets the value of cost.
Equivalent to `getMaxCost(null)`.

Returns

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

setMaxCost(RelativeTime)

Signature

```
public javax.realtime.ProcessingGroup  
setMaxCost(RelativeTime cost)  
throws IllegalArgumentException,  
        IllegalAssignmentError
```

Description

Sets the value of cost.

Parameters

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

Throws

`IllegalArgumentException` when **cost** is null or its time value is less than zero.

Returns

this

¹²²Section [9.3.1.3](#)

getMinimumCost(RelativeTime)

Signature

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

Description

Gets the value of minimum. and returns it in the provided [RelativeTime](#)¹²³ 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 IllegalArgumentException,  
IllegalAssignmentError
```

¹²³Section [9.3.1.3](#)

Description

Sets the value of minimum.

Parameters

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

Throws

IllegalArgumentException when minimum is null or its time value is less than zero.

Returns

this

getCostUnderrunHandler

Signature

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

Description

Gets the cost underrun handler.

Returns

A reference to an instance of [AsyncEventHandler](#)¹²⁴ that is cost overrun handler of this.

setCostUnderrunHandler(AsyncEventHandler)

Signature

```
public javax.realtime.ProcessingGroup  
setCostUnderrunHandler(AsyncEventHandler handler)  
throws IllegalArgumentException
```

Description

Sets the cost underrun 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

¹²⁴Section [8.3.3.5](#)

IllegalAssignmentError when handler cannot be stored in this.

Returns

this

getCostOverrunHandler

Signature

```
public javafx.runtime.AsyncEventHandler  
getCostOverrunHandler()
```

Description

Gets the cost overrun handler.

Returns

A reference to an instance of **AsyncEventHandler**¹²⁵ that is cost overrun handler of this.

setCostOverrunHandler(AsyncEventHandler)

Signature

```
public javafx.runtime.ProcessingGroup  
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.

Returns

this

¹²⁵Section 8.3.3.5

enforcingCost

Signature

```
public boolean  
enforcingCost()
```

Description

Determine whether or not cost is being enforced for releases.

Returns

true when enforcing code.

enforceCost

Signature

```
public void  
enforceCost()  
throws UnsupportedOperationException
```

Description

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

Throws

UnsupportedOperationException when cost enforcement is not supported.

getCurrentCost(RelativeTime)

Signature

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

Description

Get the cost used in the current period so far.

Parameters

dest 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 current period

getLastCost(RelativeTime)

Signature

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

Description

Get the total cost used in the last period.

Parameters

dest 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

getGranularity

Signature

```
public long  
getGranularity()
```

Description

Determine the measurement granularity of cost monitoring and cost enforcement.

Returns

the granularity in nanoseconds.

See Section `setGranularity` `setGranularity` for more details

setGranularity(long)

Signature

```
public javax.realtime.ProcessingGroup  
setGranularity(long nanos)  
throws IllegalArgumentException
```

Description

Set 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 is silently 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 measure at most once per the elapse of this granularity.

Parameters

nanos the new granularity in nanoseconds.

Throws

IllegalArgumentException when nanos is less than one.

Returns

this

6.3.3.10 ReleaseParameters

Inheritance

java.lang.Object

javax.realtime.ReleaseParameters

Interfaces

Cloneable

Serializable

Description

The top-level class for release characteristics used by [Schedulable](#)¹²⁶. When a reference to a ReleaseParameters object is given as a parameter to a constructor of a schedulable, the ReleaseParameters object becomes bound to the object being created. Changes to the values in the ReleaseParameters object affect the constructed object. When given to more than one constructor, then changes to the values in the ReleaseParameters object affect *all* of the associated objects. Note that this is a one-to-many relationship and *not* a many-to-many.

Only changes to an ReleaseParameters object caused by methods on that object cause the change to propagate to all schedulables using the object. For instance, invoking setDeadline on a ReleaseParameters instance will make the change, and then notify that the scheduler that the object has been changed. At that point the object is reconsidered for every SO that uses it. Invoking a method on the RelativeTime object that is the deadline for this object may change the

¹²⁶Section [6.3.1.3](#)

time value but it does not pass the new time value to the scheduler at that time. Even though the changed time value is referenced by ReleaseParameters objects, it will not change the behavior of the SOs that use the parameter object until a setter method on the ReleaseParameters object is invoked, or the parameter object is used in setReleaseParameters() or a constructor for a schedulable.

Release parameters use [HighResolutionTime](#)¹²⁷ values for cost, and deadline. Since the times are expressed as a [HighResolutionTime](#)¹²⁸ 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](#)¹²⁹ 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 6.8: ReleaseParameter Default Values

Attribute	Default Value
cost	new RelativeTime(0,0)
deadline	no default
overrunHandler	None
missHandler	None
rousable	false
initial event queue length	0

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

6.3.3.10.1 Constructors

¹²⁷Section [9.3.1.2](#)

¹²⁸Section [9.3.1.2](#)

¹²⁹Section [9.3.1.2](#)

ReleaseParameters(RelativeTime, RelativeTime, AsyncEventHandler, AsyncEventHandler)

Signature

```
protected
ReleaseParameters(RelativeTime cost,
                  RelativeTime deadline,
                  AsyncEventHandler overrunHandler,
                  AsyncEventHandler missHandler)
```

Description

Create 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 a schedulable object is executed, When null, the default value is a new instance of RelativeTime(0, 0).

deadline The latest permissible completion time measured from the release time of the associated invocation of the schedulable. There is no default for deadline in this class. The default must be determined by the subclasses.

overrunHandler This handler is invoked when an invocation of the schedulable exceeds cost. In the minimum implementation overrunHandler is ignored. When null, no application event handler is executed on cost overrun.

missHandler This handler is invoked when the run() method of the schedulable is still executing after the deadline has passed. When null, no application event handler is executed on the miss deadline condition.

Throws

IllegalArgumentException when the time value of cost is less than zero, or the time value of deadline is less than or equal to zero, or the chronograph associated with the cost or deadline parameters is not an instance of [Clock](#)¹³⁰.

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

ReleaseParameters

Signature

¹³⁰Section [10.3.2.1](#)

```
protected  
ReleaseParameters()
```

Description

Equivalent to `ReleaseParameters(RelativeTime, RelativeTime, AsyncEventHandler, AsyncEventHandler)`¹³¹ with the argument list (null, null, null, null).

6.3.3.10.2 Methods

clone

Signature

```
public java.lang.Object  
clone()
```

Description

Return a clone of this. This method should behave effectively as when it constructed a new object with clones of the high-resolution time values of this.

- The new object is in the current allocation context.
- clone does not copy any associations from this and it does not implicitly bind the new object to a SO.
- The new object has clones of all high-resolution time values (deep copy).
- References to event handlers are copied (shallow copy.)

Available since RTSJ 1.0.1

getCost

Signature

```
public javafx.runtime.RelativeTime  
getCost()
```

Description

Determine the current value of cost.

¹³¹Section 6.3.3.10.1

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

Determine the current value of cost.

Returns

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

getCostOverrunHandler*Signature*

```
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*

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

Description

Determine the current value of deadline.

Returns

the object last used to set the deadline containing the current value of deadline.

getDeadline(RelativeTime)

Signature

```
public javax.realtime.RelativeTime  
getDeadline(RelativeTime value)
```

Description

Determine the current value of deadline.

Returns

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

getDeadlineMissHandler

Signature

```
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

```
public T extends javax.realtime.ReleaseParameters<T>  
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\(ReleaseParameters\)](#)¹³²) the cost of those schedulables is altered as specified by each schedulable's respective scheduler.

Parameters

¹³²Section [5.3.2.2.2](#)

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

Throws

`IllegalArgumentException` when the time value of cost is less than zero, or the clock associated with the cost parameters is not the realtime clock.

`IllegalAssignmentError` when cost cannot be stored in this.

Returns

this

setCostOverrunHandler(AsyncEventHandler)

Signature

```
public T extends javax.realtime.ReleaseParameters<T>
setCostOverrunHandler(AsyncEventHandler handler)
throws UnsupportedOperationException,
    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)`¹³³) the cost overrun handler of those schedulables is altered as specified by each schedulable's respective scheduler.

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.

`UnsupportedOperationException` when cost enforcement is not supported.

Returns

this

¹³³Section 5.3.2.2.2

setDeadline(RelativeTime)

Signature

```
public T extends javax.realtime.ReleaseParameters<T>  
    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\(ReleaseParameters\)](#)¹³⁴) the deadline of those schedulables is altered as specified by each schedulable's respective scheduler.

Parameters

deadline The latest permissible completion time measured from the release time of the associated invocation of the schedulable. The default value of the deadline must be controlled by the classes that extend ReleaseParameters.

Throws

[IllegalArgumentException](#) when deadline is null, the time value of deadline is less than or equal to zero, or when the new value of this deadline is incompatible with the scheduler for any associated schedulable.

[IllegalAssignmentError](#) when deadline cannot be stored in this.

Returns

this

setDeadlineMissHandler(AsyncEventHandler)

Signature

```
public T extends javax.realtime.ReleaseParameters<T>  
    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\(ReleaseParameters\)](#)¹³⁵) the deadline miss handler of those schedulables is altered as specified by each schedulable's respective scheduler.

¹³⁴Section 5.3.2.2.2

¹³⁵Section 5.3.2.2.2

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

isRousable

Signature

```
public boolean  
isRousable()
```

Description

Determine whether or not a thread interrupt will cause instances of `Schedulable` associated with an instance of this class will be prematurely released.

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](#)¹³⁶. In the former case, there are no releases to interrupt and, in the case, the handler does not have a [ActiveEventDispatcher](#)¹³⁷ to release it.

Returns

true when rousable and false when not.

Available since RTSJ 2.0

setRousable(boolean)

Signature

```
public T extends javax.realtime.ReleaseParameters<T>  
setRousable(boolean value)
```

Description

¹³⁶Section [8.3.1.1](#)

¹³⁷Section [8.3.3.1](#)

Dictate whether or not a thread interrupt will cause instances of `Schedulable` associated with an instance of this class will be prematurely released.

Parameters

value is true when rousable and false when not.

Returns

this

Available since RTSJ 2.0

enforcingCost

Signature

```
public boolean  
enforcingCost()
```

Description

Determine 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 UnsupportedOperationException
```

Description

Set cost enforcement.

Parameters

value true when enforcing code.

Throws

`UnsupportedOperationException` 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 T extends javax.realtime.ReleaseParameters<T>  
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 is a queue overflow policy to use for handlers associated with this.

Returns

this

Available since RTSJ 2.0.

getInitialQueueLength

Signature

```
public int  
getInitialQueueLength()
```

Description

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 replacing `AperiodicParameters.getInitialArrivalTimeQueueLength()`¹³⁸.

setInitialQueueLength(int)

Signature

```
public T extends javafx.realtime.ReleaseParameters<T>
    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

`IllegalArgumentException` when initial is less than zero.

Returns

this

Available since RTSJ 2.0 replacing `AperiodicParameters.setInitialArrivalTimeQueueLength(int)`¹³⁹.

6.3.3.11 RoundRobinScheduler

Inheritance

java.lang.Object

¹³⁸Section ??

¹³⁹Section [A.2.3.2.2](#)

```
javax.realtime.Scheduler  
  javax.realtime.PriorityScheduler  
    javax.realtime.RoundRobinScheduler
```

Description

Class which represents a priority-based round-robin scheduler.

The default instance of this scheduler (returned by `instance()`¹⁴⁰) represents the RTSJ-specified round-robin scheduler.

Available since RTSJ 2.0

6.3.3.11.1 Methods

instance

Signature

```
public static javax.realtime.RoundRobinScheduler  
instance()
```

Description

Return a reference to the distinguished instance of `RoundRobinScheduler` which is the RTSJ-specified round-robin scheduler.

Throws

`UnsupportedOperationException` 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 UnsupportedOperationException,  
    IllegalArgumentException
```

¹⁴⁰Section 6.3.3.11.1

Description

Set 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

UnsupportedOperationException if this scheduler's quantum is not configurable at runtime.

IllegalArgumentException if the provided quantum is null, less than zero, or not appropriate for this platform.

Returns

this

getQuantum

Signature

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

Description

Get the quantum of this instance of RoundRobinScheduler.

Returns

a newly-allocated RelativeTime containing the currently-configured quantum of this scheduler.

getQuantum(RelativeTime)

Signature

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

Description

Get the quantum of this instance of RoundRobinScheduler.

Parameters

dest return the quantum in dest . When dest is null, allocate a new [RelativeTime](#)¹⁴¹

¹⁴¹Section [9.3.1.3](#)

instance to hold the returned value.

Returns

The currently-configured quantum of this scheduler.

getMaxPriority

Signature

```
public int  
getMaxPriority()
```

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.

reschedule(Thread, int)

Signature

```
public void  
reschedule(Thread thread,  
            int priority)
```

Description

Promotes a `java.lang.Thread` to realtime priority under this scheduler. The affected thread will be scheduled as if it were a `RealtimeThread`¹⁴² of the given priority. This does not make the affected thread a `RealtimeThread`, so it will not have access to facilities reserved for instances of `RealtimeThread`.

The method `Thread.setPriority(int)` can be used to reschedule back to the conventional Java priority levels.

Parameters

`thread` The thread to promote to realtime scheduling.

`priority` An integer priority equivalent to a priority set via `PriorityParameters`¹⁴³ on a `RealtimeThread`.

Throws

`IllegalArgumentException` when `thread` is null or `priority` is not between `getMinPriority()`¹⁴⁴ and `getMaxPriority()`¹⁴⁵, inclusive.

¹⁴²Section 5.3.2.2

¹⁴³Section 6.3.3.7

¹⁴⁴Section 6.3.3.11.1

¹⁴⁵Section 6.3.3.11.1

6.3.3.12 Scheduler

Inheritance

java.lang.Object
 javafx.runtime.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`¹⁴⁶ (e.g., `EDFScheduler`).

6.3.3.12.1 Constructors

Scheduler

Signature

protected
Scheduler()

Description

Create an instance of Scheduler.

6.3.3.12.2 Methods

getDefaultScheduler

Signature

¹⁴⁶Section 6.3.3.12.2


```
public static javax.realtime.Scheduler  
getDefaultScheduler()
```

Description

Gets a reference to the default scheduler.

Returns

A reference to the default scheduler.

setDefaultScheduler(Scheduler)

Signature

```
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](#)¹⁴⁷ at startup.

Parameters

scheduler The Scheduler that becomes the default scheduler assigned to new schedulables created by Java threads. When null nothing happens.

Throws

SecurityException when the caller is not permitted to set the default scheduler.

inSchedulableExecutionContext

Signature

```
public static boolean  
inSchedulableExecutionContext()
```

Description

Determine whether the current calling context is a [Schedulable](#)¹⁴⁸: [RealtimeThread](#)¹⁴⁹ or [AsyncBaseEventHandler](#)¹⁵⁰.

¹⁴⁷Section [6.3.3.8](#)

¹⁴⁸Section [6.3.1.3](#)

¹⁴⁹Section [5.3.2.2](#)

¹⁵⁰Section [8.3.3.3](#)

Returns

true when yes and false otherwise.

Available since RTSJ 2.0

currentSchedulable

Signature

```
public static javax.realtime.Schedulable<?>  
currentSchedulable()
```

Description

Get the current execution context when called from a [Schedulable](#)¹⁵¹ execution context.

Throws

ClassCastException when the caller is not a [Schedulable](#)¹⁵²

Returns

the current [Schedulable](#)¹⁵³.

Available since RTSJ 2.0

getPolicyName

Signature

```
public abstract java.lang.String  
getPolicyName()
```

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](#)¹⁵⁴ when accessed.

Returns

A String object which is the name of the scheduling policy used by this.

¹⁵¹Section [6.3.1.3](#)

¹⁵²Section [6.3.1.3](#)

¹⁵³Section [6.3.1.3](#)

¹⁵⁴Section [15.2.3.3](#)

6.3.3.13 SchedulingGroup

Inheritance

```
java.lang.Object
  java.lang.ThreadGroup
    javafx.runtime.SchedulingGroup
```

Description

An enhanced ThreadGroup in which a [Schedulable¹⁵⁵](#) may be started. Limits for what realtime scheduler and scheduling parameters can be enforced on all tasks in this group. A normal ThreadGroup may not contain instance of [Schedulable¹⁵⁶](#), but may contain other instances of SchedulingGroup to form a hierarchy. Every task is in some instance of ThreadGroup and every instance of Schedulable is in some instance of SchedulingGroup.

Available since RTSJ 2.0

6.3.3.13.1 Constructors

SchedulingGroup(SchedulingGroup, String)

Signature

```
public
  SchedulingGroup(SchedulingGroup parent,
                  String name)
```

Description

Create a new scheduling group.

Parameters

parent is the parent group of the new group
name is the name of the new group

Throws

¹⁵⁵Section [6.3.1.3](#)

¹⁵⁶Section [6.3.1.3](#)

`IllegalStateException` when the parent `ThreadGroup` instance is not an instance of `SchedulingGroup`.

`IllegalAssignmentError` when the parent `ThreadGroup` instance is not assignable to this.

SchedulingGroup(String)

Signature

```
public  
SchedulingGroup(String name)  
throws IllegalStateException,  
    IllegalAssignmentError
```

Description

Create a new group with the current `ThreadGroup` instance as its parent, so long as it is an instance of `SchedulingGroup`.

Parameters

`name` is the name of the new group

Throws

`IllegalStateException` when the parent `ThreadGroup` instance is not an instance of `SchedulingGroup`.

`IllegalAssignmentError` when the parent `ThreadGroup` instance is not assignable to this.

6.3.3.13.2 Methods

getMaxEligibility

Signature

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

Description

Find 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.SchedulingGroup  
setMaxEligibility(SchedulingParameters parameters)  
throws IllegalStateException
```

Description

Set 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 a child of this `SchedulingGroup` 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 `SchedulingGroup` or a child of this `SchedulingGroup` 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 scheduling parameter instance denoting the new upper bound on the scheduling eligibility of threads in this group.

Throws

`IllegalStateException` when parameters are not consistent with the scheduler type.
`IllegalArgumentException` when parameters is a higher eligibility than the max eligibility enforced by a `SchedulingParameters` above this in the hierarchy.

Returns

this

getScheduler*Signature*

```
public java.lang.Class<javax.realtime.Scheduler>  
getScheduler()
```

Description

Find 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

setScheduler(Class)*Signature*

```
public javax.realtime.SchedulingGroup  
setScheduler(java.lang.Class<javax.realtime.Scheduler> type)
```

Description

Limit the schedulers that may be used for tasks in this group.

Parameters

type is the type of scheduler of which the schedulers of all tasks must be instances.

Throws

`IllegalStateException` when a thread in the group has a scheduler that is not an instance of type or `getMaxEligibility`¹⁵⁷ returns parameters that are inconsistent with the scheduler type.

Returns

this

visitChildren(Consumer)*Signature*

```
public void  
visitChildren(java.util.function.Consumer<java.lang.ThreadGroup> visitor)
```

Description

Perform some operation on all the children of the current group.

¹⁵⁷Section 6.3.3.13.2

Parameters

visitor the function to be called on each child thread group.

Open issue 6.3.2

Should this return boolean so that the visit can be stopped before all children have been seen.

End of issue 6.3.2**6.3.3.14 SchedulingParameters**

Inheritance

java.lang.Object

[javafx.realtime.SchedulingParameters](#)

Interfaces

Cloneable

Serializable

Description

Subclasses of SchedulingParameters ([PriorityParameters](#)¹⁵⁸, [ImportanceParameters](#)¹⁵⁹, and any others defined for particular schedulers) provide the parameters to be used by the [Scheduler](#)¹⁶⁰. Changes to the values in a parameters object affects the scheduling behavior of all the [Schedulable](#)¹⁶¹ 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.

6.3.3.14.1 Constructors

SchedulingParameters*Signature*

¹⁵⁸Section [6.3.3.7](#)

¹⁵⁹Section [6.3.3.5](#)

¹⁶⁰Section [6.3.3.12](#)

¹⁶¹Section [6.3.1.3](#)

```
protected  
SchedulingParameters()
```

Description

Create a new instance of SchedulingParameters.

Available since RTSJ 1.0.1

6.3.3.14.2 Methods

clone

Signature

```
public java.lang.Object  
clone()
```

Description

Return a clone of this. This method should behave effectively as if it constructed a new object with clones of the high-resolution time values of this.

- The new object is in the current allocation context.
- clone does not copy any associations from this and it does not implicitly bind the new object to a SO.
- The new object has clones of all high-resolution time values (deep copy).
- References to event handlers are copied (shallow copy.)

Available since RTSJ 1.0.1

isCompatible(Class)

Signature

```
public boolean  
isCompatible(java.lang.Class<javax.realtime.Scheduler> type)
```

Description

Determine whether this scheduling parameters can be used by tasks scheduled by instances of type.

Parameters

type of scheduler to check against

Returns

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

Available since RTSJ 2.0

6.3.3.15 SporadicParameters

Inheritance

java.lang.Object

 javafx.realtime.ReleaseParameters

 javafx.realtime.AperiodicParameters

 javafx.realtime.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](#)¹⁶² parameter value. The value of each time object should be treated as when it were copied at the time it is passed to the parameter object, but the object reference must also be retained. Only changes to a SporadicParameters object caused by methods on that object cause the change to propagate to all schedulables using the parameter object. For instance, calling setCost on a SporadicParameters object will make the change, then notify that the scheduler that the parameter object has changed. At that point the object is reconsidered for every SO that uses it. Invoking a method on the RelativeTime object that is the cost for this object may change the cost but it does not pass the change to the scheduler at that time. That change must not change the behavior of the SOs that use the parameter object until a setter method on the SporadicParameters

¹⁶²Section [9.3.1.2](#)

object is invoked, or the parameter object is used in `setReleaseParameters()` or a constructor for an SO.

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

Table 6.9: SporadicParameters Default Values

Attribute	Value
minInterarrival time	No default. A value must be supplied
cost	<code>new RelativeTime(0,0)</code>
deadline	<code>new RelativeTime(mit)</code>
overrunHandler	None
missHandler	None
rounable	false
MIT violation policy	SAVE
Arrival queue overflow policy	SAVE
Initial arrival queue length	0

This class enables the application to specify one of four possible behaviors that indicate 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, what to do when, for any reason, the queue overflows, and the initial size of the queue.

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 Fields

6.3.3.15.2 Constructors

SporadicParameters(RelativeTime, RelativeTime, RelativeTime, AsyncEventHandler, AsyncEventHandler, boolean)

Signature

```
public  
SporadicParameters(RelativeTime minInterarrival,  
                    RelativeTime cost,  
                    RelativeTime deadline,  
                    AsyncEventHandler overrunHandler,  
                    AsyncEventHandler missHandler,  
                    boolean rousable)
```

Description

Create 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 effect the SporadicParameters object. There is no default value. When minInterarrival is null an illegal argument exception is thrown.

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

deadline The latest permissible completion time measured from the release time of the associated invocation of the schedulable. When null, the default value is a new instance of 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 handler will be used.

rounable determines whether or not an instance of Schedulable can be prematurely released by a thread interrupt.

Throws

IllegalArgumentException when minInterarrival is null or its time value is not greater than zero, or the time value of cost is less than zero, or the time value of deadline is not greater than zero, or when the chronograph associated with deadline and minInterarrival parameters are not identical or not an instance

of `Clock`¹⁶³.

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

SporadicParameters(RelativeTime, RelativeTime, RelativeTime, AsyncEventHandler, AsyncEventHandler)

Signature

```
public
SporadicParameters(RelativeTime minInterarrival,
                   RelativeTime cost,
                   RelativeTime deadline,
                   AsyncEventHandler overrunHandler,
                   AsyncEventHandler missHandler)
```

Description

Equivalent to `SporadicParameters(RelativeTime, RelativeTime, RelativeTime, AsyncEventHandler, AsyncEventHandler, boolean)`¹⁶⁴ with an argument list of (`minInterarrival`, `cost`, `deadline`, `overrunHandler`, `missHandler`, `false`).

SporadicParameters(RelativeTime, RelativeTime, AsyncEventHandler, boolean)

Signature

```
public
SporadicParameters(RelativeTime minInterarrival,
                   RelativeTime deadline,
                   AsyncEventHandler missHandler,
                   boolean rousable)
```

Description

Equivalent to `SporadicParameters(RelativeTime, RelativeTime, RelativeTime, AsyncEventHandler, AsyncEventHandler, boolean)`¹⁶⁵ with an argument list of (`minInterarrival`, `null`, `deadline`, `null`, `missHandler`, `rousable`).

¹⁶³Section 10.3.2.1

¹⁶⁴Section 6.3.3.15.2

¹⁶⁵Section 6.3.3.15.2

Available since RTSJ 2.0

SporadicParameters(RelativeTime)

Signature

```
public  
SporadicParameters(RelativeTime minInterarrival)
```

Description

Equivalent to `SporadicParameters(RelativeTime, RelativeTime, RelativeTime, AsyncEventHandler, AsyncEventHandler, boolean)`¹⁶⁶ with an argument list of (minInterarrival, null, null, null, null, false).

Available since RTSJ 1.0.1

6.3.3.15.3 Methods

getMinimalInterarrival

Signature

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

Description

Determine 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

¹⁶⁶Section 6.3.3.15.2

```
public javax.realtime.RelativeTime  
getMinimumInterarrival(RelativeTime value)
```

Description

Determine 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

Set the minimum interarrival time.

Parameters

minimum The release times of the schedulable will occur no closer than this interval.

Throws

`IllegalArgumentException` when minimum is null or its time value is not greater than zero.

`IllegalAssignmentError` when minimum cannot be stored in this.

Returns

this

setMinimumInterarrivalPolicy(MinimumInterarrivalPolicy)

Signature

```
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 is 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 underflow.

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 IllegalArgumentException
```

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.

Throws

IllegalArgumentException when policy is [QueueOverflowPolicy.DISABLE](#)¹⁶⁷.

Returns

this

¹⁶⁷Section [6.3.2.2.1](#)

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`. In POSIX terms, `SCHED_FIFO` meets the RTSJ requirements for the base scheduler, but `SCHED_RR` does not meet those requirements.

Open issue 6.4.1 (all)

Does this last sentence need changing? –Andy No, the default scheduler is still the FIFO scheduler. –jjh

End of issue 6.4.1

The specification supplies a second scheduler, `RoundRobinScheduler` for cases where timeslicing behavior is desired.

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 SchedulingGroup and ProcessingGroup

The SchedulingGroup and ProcessingGroup classes were added in RTSJ 2.0 to both support the notion of a subsystem constrained by the greater system configuration and 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 SchedulingGroup hierarchy may be used to constrain the maximum priority directly configurable by an entire subsystem. To achieve this, a SchedulingGroup with an appropriate maximum priority must be created, the security manager must be configured to disallow threads in that SchedulingGroup from accessing their parent SchedulingGroup, and all threads for the subsystem must be created in that SchedulingGroup. 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 SchedulingGroup 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 SchedulingGroup. 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 SchedulingGroup. 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 SchedulingGroup, may allow it to obtain realtime priorities. This can be avoided by placing the Java threads in a Java ThreadGroup which is in turn the child of an appropriately-configured ProcessingGroup and applying security manager restrictions.

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. The

following have had the most impact on the level of support that has been specified.

1. The notion of processor *affinity* is common across operating systems and has become the accepted way to specify the constraints on which processor a thread can execute. In some sense, processor affinities can be viewed as additional release or scheduling parameters. However, to add them to the parameter classes requires the support to be distributed throughout the specification with a proliferation of new constructor methods. To avoid this, support is grouped together within the Affinity class. The class also provides the addition of processor affinity support to Java threads without modifying the thread object's visible API.
2. The range of processors on which global scheduling is possible is dictated by the operating system. For SMP architectures, global scheduling across all the processors in the system is typically supported. However, an application and an operator can constrain threads and processes to execute only within a subset of the processors. As the number of processors increase, the scalability of global scheduling is called into question. Hence, for NUMA architectures some partitioning of the processors is likely to be performed by the OS. Hence, global scheduling across all processors will not be possible in these systems. For these reasons, the RTSJ supports an array of *predefined* affinities. These are implementation-defined. They can be used either to reflect the scheduling arrangement of the underlying OS or they can be used by the system designer to impose defaults for, say, Java threads, extraheap realtime schedulables etc. A program is only allowed to dynamically create new affinities with cardinality of one. This restriction reflects the concern that not all operating systems will support multiprocessor affinities.
3. Many OSs give system operators command-level dynamic control over the set of processors allocated to a processes. Consequently, the realtime JVM has no control over whether processors are dynamically added or removed from its OS process. Predictability is a prime concern of the RTSJ. Clearly, dynamic changes to the allocated processors will have a dramatic, and possibly catastrophic, effect on the ability of the program to meet timing requirements. Hence, the RTSJ assumes that the processor set allocated to the RTSJ process does not change during its execution. A system that is capable of such manipulations should not exercise it on RTSJ processes.
4. The reason the expert group decided not to add affinities to scheduling parameters is that ASEH do not have a single server thread, hence forcing a particular affinity would complicate the implementation.

6.4.3 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 released at time $T + \delta$. δ 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 δ is the value returned from calling the `getResolution` method of the associated clock. It is for this reason that the implementation of release times for periodic activities must use absolute rather than relative time values, in order to avoid the drift accumulating.

The second contribution to release jitter is also related to the clock/timer. It is the duration of interval between 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 δ 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 associated 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 signalled 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 signalled 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 requirement 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 instances of schedulable objects without heap access and schedulable objects 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 α is said to be *governed by* α .

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 or schedulable's scheduler; otherwise an IllegalThreadStateException is thrown. An implementation that defines a new MonitorControl subclass must document which (if any) schedulers do not support this policy.

7.2.2 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 scheduling groups' maximum scheduling parameters.
3. The *active priority* for a task is independent of its scheduling 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)`, when 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)`, when `sg` is in t 's `SchedulingGroup` 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 associated with all of these sources. The priority sources resulting from the monitor control policies defined in this section, and their associated priorities for a schedulable t , are as follows:
 - (a)

- | | | |
|-----|----------------------------|--|
| | <i>Source</i> | t itself |
| | <i>Associated Priority</i> | The base priority for t |
| | <i>Note</i> | This may have been changed (either synchronously or asynchronously) while t has been holding its lock(s). |
| (b) | <i>Source</i> | Each object locked by t and governed by a PriorityCeilingEmulation policy |
| | <i>Associated Priority</i> | The maximum value ceil, where ceil is the ceiling of a PriorityCeilingEmulation policy governing an object locked by t. |
| | <i>Note</i> | This value is also referred to as the <i>ceiling priority</i> for t. |
| (c) | <i>Source</i> | Each task attempting to synchronize on an object locked by t and governed by a PriorityInheritance policy |
| | <i>Associated Priority</i> | The maximum active priority over all such threads and schedulables |
| | <i>Note</i> | This rule accounts for recursive priority inheritance. |
| (d) | <i>Source</i> | Each task attempting to synchronize on an object locked by t and governed by a PriorityCeilingEmulation policy. |
| | <i>Associated Priority</i> | The maximum active priority over all such threads and schedulables |
| | <i>Note</i> | 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 `obj.wait()`, it releases the lock on `obj` and hence relinquishes the priority source(s) based on `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 `PriorityParameters` object may be associated with multiple schedulables), a given base priority may be the active priority for some but not all of its associated schedulables. It is a consequence of other rules that, when a thread or schedulable t attempts to synchronize on an object `obj` governed by a `PriorityCeilingEmulation` policy with ceiling `ceil`, then t 's active priority may exceed `ceil` but t 's base priority must not. In contrast, once t has successfully synchronized on `obj`, then t 's base priority may also exceed `obj`'s monitor control policy's ceiling. Note that either or both of t 's base priority and `obj`'s monitor control policy may have been dynamically modified.

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.

7.3 javax.realtime

7.3.1 Classes

7.3.1.1 MonitorControl

Inheritance

java.lang.Object

javax.realtime.MonitorControl

Description

Abstract superclass for all monitor control policy objects.

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)

Description

Gets the monitor control policy of the given instance of Object.

Parameters

obj The object being queried.

Throws

IllegalArgumentException 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)
```

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 subsequently constructed, 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\(Object, MonitorControl\)](#)¹, the setting of the default monitor control policy occurs immediately.

Parameters

¹Section [7.3.1.1.2](#)

policy The new monitor control policy. When null, the default MonitorControl policy is not changed.

Throws

SecurityException when the caller is not permitted to alter the default monitor control policy.

IllegalArgumentException when policy is not in immortal memory.

UnsupportedOperationException when policy is not a supported monitor control policy.

Returns

The default MonitorControl policy 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.

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

IllegalArgumentException Thrown when obj is null or policy is not in immortal memory.

UnsupportedOperationException when policy is not a supported monitor control policy.

IllegalMonitorStateException when the caller does not hold a lock on obj.

Returns

The current MonitorControl policy for obj, which will be replaced.

Available since RTSJ 1.0.1 The return type is changed from void to MonitorControl.

7.3.1.2 PriorityCeilingEmulation

Inheritance

```
java.lang.Object
  javax.realtime.MonitorControl
    javax.realtime.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 PriorityCeilingEmulation policy with ceiling *ceil*:

- *t*'s base priority does not exceed *ceil*
- *t*'s ceiling priority (when *t* is holding any other PriorityCeilingEmulation locks) does not exceed *ceil*.

Thus for any object *targetObj* that will be governed by priority ceiling emulation, the programmer needs to provide (via `MonitorControl.setMonitorControl(Object, MonitorControl)`²) a PriorityCeilingEmulation policy whose ceiling is at least as high as the maximum of the following values:

- the highest base priority of any thread or schedulable that could synchronize on *targetObj*

²Section 7.3.1.1.2

- the maximum ceiling priority value that any thread or schedulable object could have when it attempts to synchronize on targetObj.

More formally,

- when a thread or schedulable *t* whose base priority is *p1* attempts to synchronize on an object governed by a PriorityCeilingEmulation policy with ceiling *p2*, where $p1 > p2$, then a CeilingViolationException is thrown in *t*; likewise, a CeilingViolationException is thrown in *t* when *t* is holding a PriorityCeilingEmulation lock and has a ceiling priority exceeding *p2*.

The values of *p1* and *p2* are passed to the constructor for the exception and may be queried by an exception handler.

A consequence of the above rule is that a thread or schedulable may nest synchronizations on PriorityCeilingEmulation-governed objects as long as the ceiling for the inner lock is not less than the ceiling for the outer lock.

The possibility of nested synchronizations on objects governed by a mix of PriorityInheritance and PriorityCeilingEmulation policies requires one other piece of behavior in order to avoid unbounded priority inversions. When a thread or schedulable holds a PriorityInheritance lock, then any PriorityCeilingEmulation lock that it either holds or attempts to acquire will exhibit priority inheritance characteristics. This rule is captured above in the definition of priority sources (4.d).

When a thread or schedulable *t* attempts to synchronize on a PriorityCeilingEmulation-governed object with ceiling *ceil*, then *ceil* must be within the priority range allowed by *t*'s scheduler; otherwise, an IllegalStateException is thrown. Note that this does not prevent a regular Java thread from synchronizing on an object governed by a PriorityCeilingEmulation policy with a ceiling higher than 10.

The priority ceiling for an object *obj* can be modified by invoking MonitorControl.setMonitorControl(*obj*, newPCE) where newPCE's ceiling has the desired value.

See also [MonitorControl](#)³, [PriorityInheritance](#)⁴, and [CeilingViolationException](#)⁵.

7.3.1.2.1 Methods

instance(int)

³Section [7.3.1.1](#)

⁴Section [7.3.1.3](#)

⁵Section [15.2.2.2](#)

Signature

```
public static javax.realtime.PriorityCeilingEmulation  
instance(int ceiling)
```

Description

Return a PriorityCeilingEmulation object with the specified ceiling. This object is in ImmortalMemory. All invocations with the same ceiling value return a reference to the same object.

Parameters

ceiling Priority ceiling value.

Throws

IllegalArgumentException when ceiling is outside of the range of permitted priority values (e.g., less than PriorityScheduler.instance().getMinPriority() or greater than PriorityScheduler.instance().getMaxPriority() for the base scheduler).

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

Inheritance

java.lang.Object
 javax.realtime.MonitorControl
 javax.realtime.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 t2 when either t2 releases the lock on obj, or t1 ceases attempting to synchronize on obj (e.g., when t1 incurs an ATC).
2. Otherwise (i.e., t1's active priority exceeds the maximum priority allowed by t2's scheduler), an IllegalThreadStateException is thrown in t1.

Note on the 2nd rule, throwing the exception in t1, rather than in t2, ensures that the exception is synchronous.

See also [MonitorControl](#)⁶ and [PriorityCeilingEmulation](#)⁷

7.3.1.3.1 Methods

⁶Section [7.3.1.1](#)

⁷Section [7.3.1.2](#)

instance

Signature

```
public static javax.realtime.PriorityInheritance  
instance()
```

Description

Return a reference to the singleton PriorityInheritance.

This is the default MonitorControl policy in effect at system startup.

The PriorityInheritance instance shall be allocated in ImmortalMemory.

7.3.1.4 WaitFreeReadQueue

Inheritance

```
java.lang.Object  
  javax.realtime.WaitFreeReadQueue
```

Description

A queue that can be non-blocking for consumers. The WaitFreeReadQueue class is intended for single-reader multiple-writer communication, although it may also be used (with care) for multiple readers. A *reader* is generally a instance of Schedulable with may not use the heap, and the *writers* are generally regular Java threads or heap-using instances of 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.IllegalAccessException,
- java.lang.ClassNotFoundException, and
- java.lang.InstantiationException.

These exceptions were in error. Their deletion may cause compile-time errors in code using the previous constructor. The repair is to remove the exceptions from the catch clause around the constructor invocation.

7.3.1.4.1 Constructors

WaitFreeReadQueue(Runnable, Runnable, int, MemoryArea, boolean)

Signature

```
public
WaitFreeReadQueue(Runnable writer,
                  Runnable reader,
                  int maximum,
                  MemoryArea memory,
                  boolean notify)
throws IllegalArgumentException,
    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, 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`⁸ in which internal elements are allocated.

notify A flag that establishes whether a reader is notified when the queue becomes non-empty.

Throws

`IllegalArgumentException` when an argument holds an invalid value. The writer argument must be null, a reference to a `Thread`, or a reference to a schedulable (a `RealtimeThread`, or an `AsyncEventHandler`.) The reader argument must be null, a reference to a `Thread`, or a reference to a schedulable. The maximum argument must be greater than zero.

`InaccessibleAreaException` when memory is a scoped memory that is not on the caller's scope stack.

`MemoryScopeException` when either reader or writer is non-null and the memory argument is not compatible with reader and writer with respect to the assignment and access rules for memory areas.

WaitFreeReadQueue(Runnable, Runnable, int, MemoryArea)

Signature

```
public
WaitFreeReadQueue(Runnable writer,
                  Runnable reader,
                  int maximum,
                  MemoryArea memory)
```

⁸Section 11.3.2.3

throws `IllegalArgumentException`,
`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

```
public
WaitFreeReadQueue(int maximum,
                   MemoryArea memory,
                   boolean notify)
throws IllegalArgumentException,
    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(null, null, maximum, memory, notify)`

Available since RTSJ 1.0.1

WaitFreeReadQueue(int, boolean)

Signature

```
public
WaitFreeReadQueue(int maximum,
                   boolean notify)
throws IllegalArgumentException
```

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.

isEmpty

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

```
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

```
public void  
waitForData()  
throws UnsupportedOperationException,  
        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

`UnsupportedOperationException` when this has not been constructed with `notify` set to `true`.

`InterruptedException` when the thread is interrupted by `interrupt()` or [AsynchronouslyInterruptedException](#) `fire()`⁹ during the time between calling this method and returning from it.

Available since RTSJ 1.0.1 `InterruptedException` was added to the `throws` clause.

write(T)

Signature

```
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.

⁹Section [8.3.2.1.2](#)

Throws

InterruptedException when the thread is interrupted by `interrupt()` or `AsynchronouslyInterruptedException.fire()`¹⁰ 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

Inheritance

java.lang.Object
 javafx.realtime.WaitFreeWriteQueue

Description

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

¹⁰Section 8.3.2.1.2

- 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 IllegalArgumentException,
    MemoryScopeException,
    InaccessibleAreaException
```

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 Runnables are non-null, the constructor will select the nearest common scoped parent memory area, or when there is no such scope it will use immortal memory. When all three parameters are null, the queue will be allocated in immortal memory.

reader and writer are not necessarily the only threads or schedulables that will access the queues; moreover, there is no check that they actually access the queue at all.

Note, 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 is 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`¹¹ in which this and internal elements are allocated.

Throws

`IllegalArgumentException` when an argument holds an invalid value. The writer argument must be null, a reference to a `Thread`, or a reference to a schedulable (a `RealtimeThread`, or an `AsyncEventHandler`.) The reader argument must be null, a reference to a `Thread`, or a reference to a schedulable. The maximum argument must be greater than zero.

`MemoryScopeException` when either reader or writer is non-null and the memory argument is not compatible with reader and writer with respect to the assignment and access rules for memory areas.

`InaccessibleAreaException` when memory is a scoped memory that is not on the caller's scope stack.

WaitFreeWriteQueue(int, MemoryArea)

Signature

```
public
WaitFreeWriteQueue(int maximum,
                    MemoryArea memory)
throws IllegalArgumentException,
    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

¹¹Section 11.3.2.3

WaitFreeWriteQueue(int)

Signature

```
public  
WaitFreeWriteQueue(int maximum)  
throws IllegalArgumentException
```

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,mximum, null)

Available since RTSJ 1.0.1

7.3.1.5.2 Methods

clear

Signature

```
public void  
clear()
```

Description

Sets this to empty.

isEmpty

Signature

```
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*

```
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 synchronized T  
read()  
throws InterruptedException
```

Description

A synchronized and possibly blocking operation on the queue.

Throws

InterruptedException when the thread is interrupted by `interrupt()` or `AsynchronouslyInterruptedException.fire()`¹² 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

¹²Section [8.3.2.1.2](#)

size

Signature

```
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

```
public boolean  
force(T value)  
throws MemoryScopeException,  
        IllegalArgumentException
```

Description

Unconditionally insert 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.

`IllegalArgumentException` 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

```
public boolean  
write(T value)  
throws MemoryScopeException,  
        IllegalArgumentException
```

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.

IllegalArgumentException when value is null.

Returns

true when the queue was non-full; false otherwise.

7.4 Rationale

Java's rules for synchronized code provide a means for mutual exclusion but do not prevent unbounded priority inversions and thus are insufficient for realtime applications. This specification strengthens the semantics for synchronized code by mandating priority inversion control, in particular by furnishing classes for priority inheritance and priority ceiling emulation. Priority inheritance is more widely implemented in realtime operating systems and thus is 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.

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 the support for asynchronous control flow. Mechanisms are provided for both starting a task asynchronously and interrupting the execution of a thread or other task. This specifications provides mechanisms that

- bind the execution of program logic to the occurrence of internal and external events;
- enable asynchronous transfer of control; and
- facilitate the asynchronous termination of realtime threads.

The first of 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. The second is Asynchronous Transfer of Control (ATC), which provides a means of stopping some calculation prematurely. ATC may also be used to terminate a realtime thread safely.

Events and Event Handling

Asynchronous event handling is captured by the classes `AsyncBaseEvent` (AE), `AsyncBaseEventHandler` (AEH) and `AbstractBoundAsyncEventHandler`, 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 8.1: Event to Handler Matrix

Types	<code>AsyncEvent</code>	<code>AsyncLongEvent</code>	<code>AsyncObjectEvent</code>
<code>AsyncEventHandler</code>	Nothing	Nothing	Nothing
<code>AsyncLongEventHandler</code>	Event Id	Payload	Event Id
<code>AsyncObjectEventHandler</code>	Event Object	Event Object	Payload

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 manner in which the implementation selects a realtime thread to release a given AEH at a given release is implementation-defined. The interface `BoundAsyncBaseEventHandler` 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.

Asynchronous Transfer of Control

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 new 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.

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 one of the subclasses of the `BoundAsyncBaseEventHandler` class.

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 synchronized statement, a static initializer or any method or constructor without `AsynchronouslyInterruptedException` in its throws clause. As specified in the introduction to Chapter 8 in *Java Language Specification*, a synchronized method is equivalent to a non-synchronized method with the body of the method contained in a synchronized statement. Thus,

a synchronized AI method behaves like an AI method containing only an ATC-deferred statement.

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 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.

Interruptible Blocking Methods — The RTSJ and standard Java methods that are explicitly interruptible by an AIE. The interruptible blocking methods comprise

- `HighResolutionTime.waitForObject()`,
- `Object.wait()`,
- `Thread.sleep()`,
- `RealtimeThread.sleep()`,
- `Thread.join()`,
- `ScopedMemory.join()`,
- `ScopedMemory.joinAndEnter()`,
- `RealtimeThread.waitForNextRelease()`,
- `WaitFreeWriteQueue.read()`,
- `WaitFreeReadQueue.waitForData()`,
- `WaitFreeReadQueue.write()`,
- `WaitFreeDequeue.blockingRead()`,
- `WaitFreeDequeue.blockingWrite()`

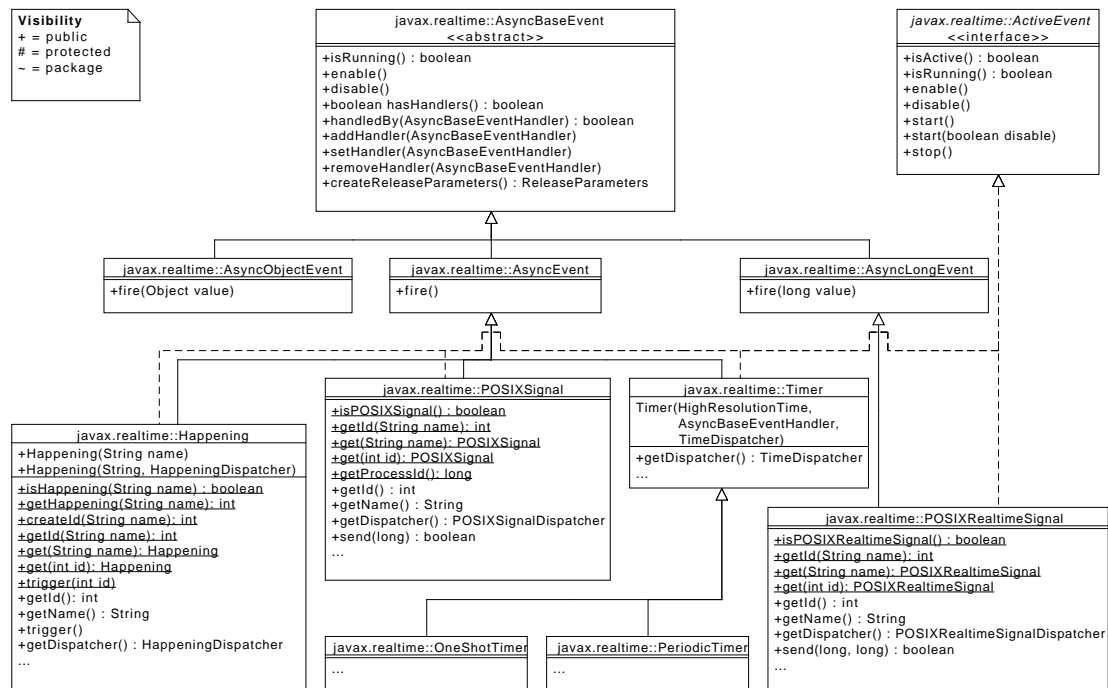
and their overloaded forms.

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 12), requires an execution context. The `ActiveEvent` interface is provided to mark these and provide additional execution semantics. Figure 8.1 illustrates the event hierarchy.

Figure 8.1: The Event Class Hierarchy



8.2.1 Asynchronous Events and their Handlers

The following twelve 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 (by either program logic or by the triggering of a happening) and the event is enabled, its attached handlers (that is, AEHs that have been added to the AE by the execution of `addHandler()`) are released for execution. Every occurrence of an event increments the `fireCount` in each

attached handler. Handlers may elect to execute logic for each occurrence of the event or not.

2. The release of attached handlers occurs in execution eligibility order (priority order, from highest to lowest, with the default `PriorityScheduler`) and at the active priority of the schedulable that invoked the `fire` method. The release of handlers resulting from a happening or a timer must begin within a bounded time (ignoring time consumed by unrelated activities in the system). This worst-case response interval must be documented for some reference architecture.
3. The release of attached handlers is an atomic operation with respect to adding and removing handlers.
4. The logical release of an attached handler may occur before the previous release has completed.
5. Each handler has an application configurable, handler type dependent queue for holding events that have been released before a previous release has completed.
6. The overflow policy of a handlers queue is also application configurable.
7. A deadline may be associated with each logical release of an attached handler. The deadline is relative to the occurrence of the associated event.
8. AEs and AEHs may be created and used by any program logic within the constraints of the memory assignment rules.
9. 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.
10. The same AEH may be added to more than one AE.
11. By default all AEHs are considered to be daemons (the daemon status being set by their constructors). An AEH can be set to have a non daemon status after it has been created and before it has been attached to an AE.
12. 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.
13. 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.
14. 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.

An RTSJ program terminates when and only when

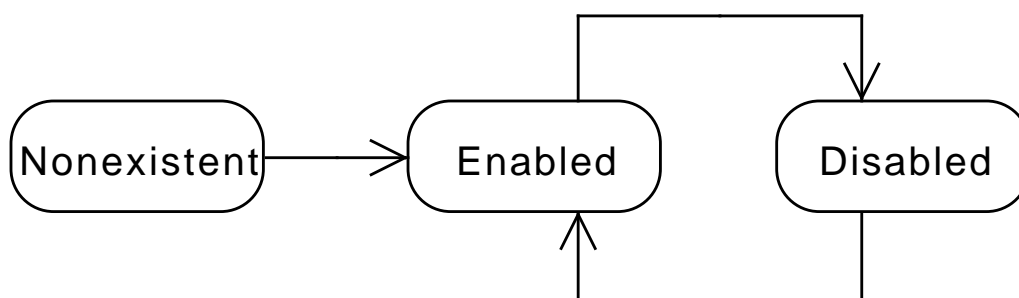
1. all nondaemon threads (either regular Java threads or realtime threads) are terminated,
2. the `fireCounts` of all nondaemon bound AEHs or nondaemon AEHs are zero and all releases are completed, and
3. there are no nondaemon Bound AEHs or AEHs attached to timers or async

events associated with happenings.

Though dispatchers have a thread, this thread is a daemon thread and does not itself hinder termination.

`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 8.2 illustrates this state space.

Figure 8.2: States of a Simple `AsyncBaseEvent`



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, or at least that context is of necessity limited and often needs to have a very short duration; dispatching an unlimited number of handlers is not acceptable. They require an additional execution context for releasing handlers.

In order to be able to distinguish between events that are caused to be fired by an outside mechanism from those that are fired from another thread, the former extend the `ActiveEvent` interface. 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 this other 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 does determine the priority and the affinity of a dispatcher, as well as determine the mapping between dispatchers and events.

Each event has a single dispatcher, but a dispatcher may serve many events. As with fire, the dispatcher releases handlers in reverse priority order, i.e., from highest to lowest. This enables the programmer to control the number of these execution contexts and still optimize how handlers are released.

The state space of an `ActiveEvent` is an extension of the state space for an `AsyncBaseEvent` depicted in Figure 8.2. `ActiveEvent` adds the notion of active and inactive on top of enabled and disabled, as depicted in Figure 8.3. Note that the enabled-disabled distinction only splits the active state. The inactive state is by definition disabled.

8.2.3 Asynchronous Transfer of Control

Asynchronously interrupting a schedulable consists of the following activities.

1. **Generation** of an asynchronous interrupt exception — this is the event in the underlying system that makes the AIE available to the program.
2. **Delivery** of the asynchronous interrupt exception to the target schedulable— this is the action that invokes the search for and execution of an appropriate handler.

Between the generation of an AIE and its delivery, the exception is held *pending*. The AIE remains pending (even after delivery) until it is **cleared** by the program logic using `clear()` or `doInterruptible()`.

The following eight points define the semantics of ATC. Semantics that apply to particular classes, constructors, methods, and fields will be found in their detail sections, respectively.

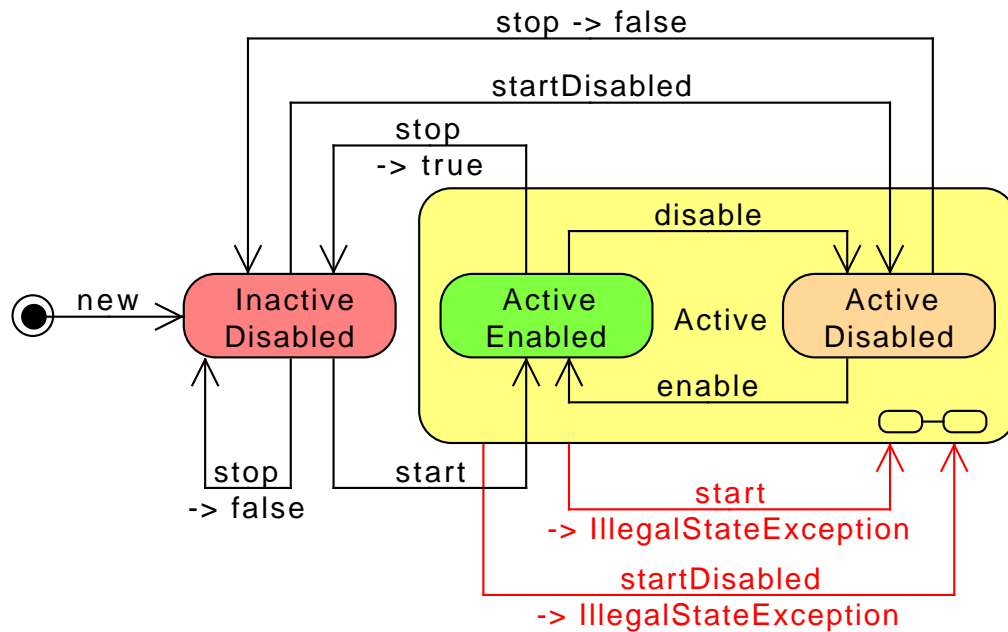
Please review for elb: `InterruptedIOException` and ATC

Verify that 3(b) below captures what we discussed on call 2016-03-21.

End review request

1. An AIE is generated for a given schedulable when the `fire()` method is called on an AIE for which the schedulable object is executing within the `doInterruptible()` method or the `Schedulable.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. An AIE

Figure 8.3: States of an ActiveEvent



becomes pending upon generation and remains pending until explicitly cleared or replaced by another AIE.

2. The `Schedulable.interrupt()` method causes the target task to throw a generic AIE and has the behaviors defined for `Thread.interrupt()`. This is the only interaction between the ATC mechanism and the conventional `interrupt()` mechanism.
3. An AIE is delivered to a schedulable when it is executing in 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 set 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 then 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.InterruptedIOException` (for example, blocking methods in `java.io.*`) must be prevented from blocking indefinitely when invoked from a method with `AsynchronouslyInterruptedException` in its throws clause. When an AIE is generated and the target schedulable's control is blocked inside one of these methods with an AI-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 AI-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. 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.
- 5. Native methods that include `AsynchronouslyInterruptedException` in their throws clause have implementation-specific behavior.
- 6. 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.
- 7. 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.

8. 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 affect on the pending state of any AIE, and no affect on the firing of the AIE concerned.

8.2.3.1 Summary of ATC Operation

The RTSJ's approach to ATC is designed to follow the above principles. It is based on exceptions and is an extension of the current Java language rules for `java.lang.Thread.interrupt()`. In summary, ATC works as follows.

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 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 which is in an ATC-deferred section. Intervening handlers and finally clauses that are not in ATC-deferred sections are not executed, but objects locks are released. See section 11.3 of *The Java Language Specification* second edition [] for an explanation of the terms *dynamically enclosing* and *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.

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 ($i = 1, \dots, n$, 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) 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.

8.3 javax.realtime

8.3.1 Interfaces

8.3.1.1 ActiveEvent

Interfaces

[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](#)¹ 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 a 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.

8.3.1.1.1 Methods

isActive

Signature

```
public boolean  
isActive()
```

Description

Determine the activation state of this event, i.e., it has been started but not yet stopped again.

Returns

¹Section [8.3.1.1.1](#)

true when active, false otherwise.

isRunning

Signature

```
public boolean  
isRunning()
```

Description

Determine the running state of this event, i.e., it is both active and enabled.

Returns

true when active and enabled, false otherwise.

start

Signature

```
public void  
start()  
throws IllegalStateException
```

Description

Start this active event.

Throws

IllegalStateException when this event has already been started.

start(boolean)

Signature

```
public void  
start(boolean disabled)  
throws IllegalStateException
```

Description

Start this active event.

Parameters

disabled true for starting in a disabled state.

Throws

IllegalStateException when this event has already been started.

stop

Signature

```
public boolean  
stop()  
throws IllegalStateException
```

Description

Stop this active event.

Throws

IllegalStateException when this event is not running.

Returns

the previous enabled state.

enable

Signature

```
public void  
enable()
```

Description

Change the state of the event so that associated handlers are release on fire. Each subclass provides a means of dispatching its handlers when requested. This method enables that request mechanism.

disable

Signature

```
public void  
disable()
```

Description

Change the state of the event so that associated handlers are skipped on fire. Each subclass provides a fire method as means of dispatching its handlers when requested. This method disables that request mechanism.

8.3.1.2 BoundAsyncBaseEventHandler

Interfaces

[javax.realtime.BoundSchedulable](#)

Description

An marker interface for all schedulables that are bound to a single thread of control. It is required to enable references to all bound handlers. A thread is bound to a handler of this type when it is first attached to an event. Thus security checks for thread use can be done when [AsyncBaseEvent.addHandler²](#) and [AsyncBaseEvent.setHandler³](#) are called.

8.3.1.3 Interruptible

Description

Interruptible is an interface implemented by classes that will be used as arguments on the methods `doInterruptible()` of [AsynchronouslyInterruptedException⁴](#) and its subclasses. `doInterruptible()` invokes the implementations of the methods in this interface.

8.3.1.3.1 Methods

`run(AsynchronouslyInterruptedException)`

Signature

```
public void  
run(AsynchronouslyInterruptedException exception)  
throws AsynchronouslyInterruptedException
```

Description

²Section [8.3.3.2.1](#)

³Section [8.3.3.2.1](#)

⁴Section [8.3.2.1](#)

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](#)⁵ 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. 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](#)⁶ from within the `interruptAction()` method.

8.3.1.4 Releasable

Description

A base interface for everything that has a dispatcher.

8.3.1.4.1 Methods

⁵Section [8.3.2.1](#)

⁶Section [8.3.2.1](#)

getDispatcher

Signature

```
public D extends javax.realtime.ActiveEventDispatcher<D, T>  
getDispatcher()
```

Description

Obtain the dispatcher for this.

Returns

that dispatcher.

8.3.2 Exceptions

8.3.2.1 AsynchronouslyInterruptedException

Inheritance

```
java.lang.Object  
  java.lang.Throwable  
    java.lang.Exception  
      java.lang.InterruptedException  
        javax.realtime.AsynchronouslyInterruptedException
```

Description

A special exception that is thrown in response to an attempt to asynchronously transfer the locus of control of a schedulable.

A schedulable that is executing a method or constructor, which is declared with an **AsynchronouslyInterruptedException**⁷ 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**⁸ method is called of an AIE for which *s* has a *doInterruptible* method call in progress.

⁷Section 8.3.2.1

⁸Section 8.3.2.1.2

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 `Schedulable.interrupt`⁹, 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-schedulable object.

Other sources (e.g., `AsynchronouslyInterruptedException.fire()` and `Timed`¹⁰) will generate specific instances of `AsynchronouslyInterruptedException` which applications can identify and thus limit propagation.

8.3.2.1.1 Constructors

AsynchronouslyInterruptedException

Signature

⁹Section 6.3.1.3.1

¹⁰Section 8.3.2.2

```
public  
AsynchronouslyInterruptedException()
```

Description

Create an instance of `AsynchronouslyInterruptedException`.

8.3.2.1.2 Methods

getGeneric*Signature*

```
public static javax.realtime.AsynchronouslyInterruptedException  
getGeneric()
```

Description

Gets the singleton system generic `AsynchronouslyInterruptedException` that is generated when `Schedulable.interrupt()`¹¹ is invoked.

Throws

`IllegalThreadStateException` when the current thread is a Java thread.

Returns

The generic `AsynchronouslyInterruptedException`.

enable*Signature*

```
public boolean  
enable()
```

Description

Enable the throwing of this exception. This method is valid only when the caller has a call to `doInterruptible()` in progress. When invoked when no call to `doInterruptible()` is in progress, `enable` returns false and does nothing.

Returns

¹¹Section 6.3.1.3.1

true, when this was disabled before the method was called and the call was invoked whilst the associated `doInterruptible()` is in progress, and false otherwise.

disable

Signature

```
public synchronized boolean  
disable()
```

Description

Disable the throwing of this exception. When the `fire`¹² method is called on this AIE whilst it is disabled, the fire is held pending and delivered as soon as the AIE is enabled and the interruptible code is within an AI-method. When an AIE is pending when the associated disable method is called, the AIE remains pending, and is delivered as soon as the AIE is enabled and the interruptible code is within an AI-method.

This method is valid only when the caller has a call to `doInterruptible()` in progress. If invoked when no call to `doInterruptible()` is in progress, `disable` returns false and does nothing.

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

Query the enabled status of this exception.

This method is valid only when the caller has a call to `doInterruptible()` in progress. If invoked when no call to `doInterruptible()` is in progress, `enable` returns false and does nothing.

Returns

true, when this is enabled and the method call was invoked in the context of the associated `doInterruptible()`, and false otherwise.

¹²Section 8.3.2.1.2

fire*Signature*

```
public boolean  
fire()
```

Description

Generate this exception when its `doInterruptible()` has been invoked and not completed. When this is the only outstanding AIE on the schedulable object that invoked this AIE's `doInterruptible(Interruptible)`¹³ 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.

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`¹⁴. This method may be on the stack in exactly one `Schedulable`¹⁵ object. An attempt to invoke this method in a schedulable while it is on the stack of another or the same schedulable will cause an immediate return with a value of false.

The run method of given `Interruptible` is always entered with the exception in the enabled state, but that state can be modified with `enable()`¹⁶ and `disable()`¹⁷ and the state can be observed with `isEnabled()`¹⁸.

This AIE is cleared on return from `doInterruptible()`.

Parameters

¹³Section 8.3.2.1.2

¹⁴Section 8.3.1.3

¹⁵Section 6.3.1.3

¹⁶Section 8.3.2.1.2

¹⁷Section 8.3.2.1.2

¹⁸Section 8.3.2.1.2

logic An instance of an [Interruptible](#)¹⁹ whose run() method will be called.

Throws

IllegalThreadStateException when called on the generic AsynchronouslyInterruptedException.

IllegalArgumentException 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 IllegalThreadStateException when called from a Java thread.

clear

Signature

```
public boolean  
clear()
```

Description

Atomically see if this is pending on the currently executing schedulable, and when so, make it non-pending.

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 IllegalThreadStateException when called from a Java thread.

8.3.2.2 Timed

Inheritance

```
java.lang.Object  
  java.lang.Throwable  
    java.lang.Exception  
      java.lang.InterruptedIOException
```

¹⁹Section [8.3.1.3](#)

`javax.realtime.AsynchronouslyInterruptedException`
`javax.realtime.Timed`

Description

Create a scope in a `Schedulable`²⁰ object which will be asynchronously interrupted at the expiration of a timer. This timer will begin measuring time at some point between the time `doInterruptible()` is invoked and the time the `run()` method of the `Interruptible` object is invoked. Each call of `doInterruptible()` on an instance of `Timed` will restart the timer for the amount of time given in the constructor or the most recent invocation of `resetTime()`. The timer is cancelled when it has not expired before the `doInterruptible()` method has finished.

All memory use of an instance of `Timed` occurs during construction or the first invocation of `doInterruptible()`. Subsequent invocations of `doInterruptible()` do not allocate memory.

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);`

8.3.2.2.1 Constructors

Timed(HighResolutionTime)

Signature

```
public  
Timed(javax.realtime.HighResolutionTime<?> time)  
throws IllegalArgumentException,  
    UnsupportedOperationException
```

Description

Create an instance of `Timed` with a timer set to `time`. When the time is in the past the `AsynchronouslyInterruptedException`²¹ mechanism is activated immediately after or when the `doInterruptible()` method is called.

Parameters

²⁰Section 6.3.1.3

²¹Section 8.3.2.1

time When time is a [RelativeTime](#)²² value, it is the interval of time between the invocation of `doInterruptible()` and when the schedulable is asynchronously interrupted. When time is an [AbsoluteTime](#)²³ value, the timer asynchronously interrupts at this time (assuming the timer has not been cancelled).

Throws

`IllegalArgumentException` when time is null.

`UnsupportedOperationException` when time is not based on a [Clock](#)²⁴.

8.3.2.2.2 Methods

doInterruptible(Interruptible)

Signature

```
public boolean  
doInterruptible(Interruptible logic)
```

Description

Execute a time-out method. Starts the timer and executes the `run()` method of the given [Interruptible](#)²⁵ object.

Parameters

logic logic An instance of an [Interruptible](#)²⁶ whose `run()` method will be called.

Throws

`IllegalArgumentException` `IllegalArgumentException` when logic is null.

`IllegalThreadStateException` `IllegalThreadStateException` when called on the generic `AsynchronouslyInterruptedException`.

Returns

true, when the method call completed normally, and false, when another call to `doInterruptible` has not completed.

²²Section [9.3.1.3](#)

²³Section [9.3.1.1](#)

²⁴Section [10.3.2.1](#)

²⁵Section [8.3.1.3](#)

²⁶Section [8.3.1.3](#)

resetTime(HighResolutionTime)

Signature

```
public void  
resetTime(javax.realtime.HighResolutionTime<?> time)
```

Description

To set the time-out 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](#)²⁷, the time-out is not changed.

restart(HighResolutionTime)

Signature

```
public void  
restart(javax.realtime.HighResolutionTime<?> time)
```

Description

Reset the timeout. When this [Timed](#)²⁸ instance is executing, adjust the timeout to time and restart the timer. When the instance is not executing, adjust the timeout for the next invocation.

Parameters

time The new timeout.

Throws

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

UnsupportedOperationException when time is not based on a [Clock](#)²⁹

Available since RTSJ 2.0

8.3.3 Classes

8.3.3.1 ActiveEventDispatcher

Inheritance

²⁷Section [10.3.2.1](#)

²⁸Section [8.3.2.2](#)

²⁹Section [10.3.2.1](#)

java.lang.Object

javax.realtime.ActiveEventDispatcher

Interfaces

javax.realtime.RealtimeExecutionContext

Description

Provides a means of dispatching a set of [ActiveEvent](#)³⁰s. It acts as if it contains a RealtimeThread to perform this task. The priority of this thread can be specified when a dispatcher object is created. The default dispatcher runs at the highest realtime priority on the base scheduler. Dispatchers do not maintain a queue of pending event.

Application code cannot extend this class.

8.3.3.1.1 Constructors

ActiveEventDispatcher(SchedulingParameters, SchedulingGroup)

Signature

protected

ActiveEventDispatcher(SchedulingParameters schedule,
SchedulingGroup group)

Description

Create a new dispatcher.

Parameters

schedule provide scheduling information to the new object.

group the SchedulingGroup of the thread of this dispatcher.

ActiveEventDispatcher(SchedulingParameters)

Signature

protected

ActiveEventDispatcher(SchedulingParameters schedule)

³⁰Section [8.3.1.1](#)

Description

Create a new dispatcher.

Parameters

schedule provide scheduling information to the new object.

8.3.3.1.2 Methods

getSchedulingParameters*Signature*

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

Description

Determine how the thread associated with this dispatcher is scheduled.

Returns

the scheduling parameters of the dispatcher thread.

getSchedulingGroup*Signature*

```
public javax.realtime.SchedulingGroup  
getSchedulingGroup()
```

Description

Determine in which group the thread associated with this dispatcher is.

Returns

the scheduling group of the dispatcher thread.

register(T)*Signature*

```
public abstract void  
register(T event)
```

throws `RegistrationException`,
`IllegalStateException`,
`IllegalArgumentException`

Description

Register an active event with this dispatcher.

Parameters

event to register

Throws

`RegistrationException` when event is already registered.
`IllegalStateException` when this object has been destroyed.
`IllegalArgumentException` when event is not stopped.

deregister(T)

Signature

```
public abstract void  
deregister(T event)  
throws DeregistrationException,  
       IllegalStateException,  
       IllegalArgumentException
```

Description

Deregister an active event from this dispatcher.

Parameters

event to deregister

Throws

`DeregistrationException` when event is already registered.
`IllegalStateException` when this object has been destroyed.
`IllegalArgumentException` when event is not stopped.

destroy

Signature

```
public abstract void  
destroy()
```

throws `IllegalStateException`

Description

Makes the dispatcher unusable.

Throws

`IllegalStateException` when called on a dispatcher that has one or more registered objects.

8.3.3.2 AsyncBaseEvent

Inheritance

`java.lang.Object`
`javax.realtime.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`³¹ with `AsyncLongEvent`³² and `AsyncObjectEvent`³³.

Note that when this class is collected, all its handlers are automatically removed as if `setHandler`³⁴ was called with a null parameter.

Available since RTSJ 2.0

8.3.3.2.1 Methods

isRunning

Signature

`public boolean`
`isRunning()`

Description

³¹Section 8.3.3.4

³²Section 8.3.3.6

³³Section 8.3.3.8

³⁴Section 8.3.3.2.1

Determine the firing state (releasing or skipping) of this event, i.e., whether it is enabled or disabled.

Returns

true when releasing, false when skipping.

handledBy(AsyncBaseEventHandler)

Signature

```
public boolean  
handledBy(javax.realtime.AsyncBaseEventHandler<?> handler)
```

Description

Test to see if 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.

enable

Signature

```
public void  
enable()
```

Description

Change the state of the event so that associated handlers are release on fire. Each subclass provides a means of dispatching its handlers when requested. This method enables that request mechanism.

disable

Signature

```
public void  
disable()
```

Description

Change the state of the event so that associated handlers are skipped on fire. Each subclass provides a fire method as means of dispatching its handlers when requested. This method disables that request mechanism.

addHandler(AsyncBaseEventHandler)

Signature

```
public void  
addHandler(javax.realtime.AsyncBaseEventHandler<?> handler)
```

Description

Add 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, 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

IllegalArgumentException when handler is null or the handler has [PeriodicParameters](#)³⁵. Only the subclass [PeriodicTimer](#)³⁶ is allowed to have handlers with [PeriodicParameters](#)³⁷.

[IllegalAssignmentError](#) when this AsyncBaseEvent cannot hold a reference to handler.

IllegalStateException when the configured Scheduler and SchedulingParameters 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.

³⁵Section [6.3.3.6](#)

³⁶Section [10.3.2.3](#)

³⁷Section [6.3.3.6](#)

setHandler(AsyncBaseEventHandler)

Signature

```
public void  
setHandler(javax.realtime.AsyncBaseEventHandler<?> handler)
```

Description

Associate a new handler with this event and remove all existing handlers. The execution of this method is atomic with respect to the execution of the fire() method.

Parameters

handler The instance of [AsyncBaseEventHandler](#)³⁸ to be associated with this. When handler is null then no handler will be associated with this, i.e., behave effectively as if setHandler(null) invokes [removeHandler\(AsyncBaseEventHandler\)](#)³⁹ for each associated handler.

Throws

[IllegalArgumentException](#) when handler has [PeriodicParameters](#)⁴⁰. Only the subclass [PeriodicTimer](#)⁴¹ is allowed to have handlers with [PeriodicParameters](#)⁴².
[IllegalAssignmentError](#) when this AsyncBaseEvent cannot hold a reference to handler.

removeHandler(AsyncBaseEventHandler)

Signature

```
public void  
removeHandler(javax.realtime.AsyncBaseEventHandler<?> handler)
```

Description

Remove a handler from the set associated with this event. The execution of this method is atomic with respect to the execution of the fire() method.

A removed handler continues to execute until its fireCount becomes zero and it completes.

When handler has a scoped non-default initial memory area and execution of this method causes handler to become unfirable, this method shall not return until all related finalization has completed.

³⁸Section [8.3.3.3](#)

³⁹Section [8.3.3.2.1](#)

⁴⁰Section [6.3.3.6](#)

⁴¹Section [10.3.2.3](#)

⁴²Section [6.3.3.6](#)

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.

hasHandlers*Signature*

```
public boolean  
hasHandlers()
```

Description

Determine whether or not this event has any handlers.

Returns

true when and only when at least one handler is associated with this event.

createReleaseParameters*Signature*

```
public javax.realtime.ReleaseParameters<?>  
createReleaseParameters()
```

Description

Create a [ReleaseParameters](#)⁴³ object appropriate to the release characteristics of this event. The default is the most pessimistic: [AperiodicParameters](#)⁴⁴. 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](#)⁴⁵ 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](#)⁴⁶, there is no requirement for an implementation to check that the handler is released periodically.

Returns

A new [ReleaseParameters](#)⁴⁷ object.

⁴³Section [6.3.3.10](#)

⁴⁴Section [6.3.3.2](#)

⁴⁵Section [6.3.3.10](#)

⁴⁶Section [6.3.3.6](#)

⁴⁷Section [6.3.3.10](#)

8.3.3.3 AsyncBaseEventHandler

Inheritance

java.lang.Object
javax.realtime.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](#)⁴⁸ with [AsyncLongEventHandler](#)⁴⁹ and [AsyncObjectEventHandler](#)⁵⁰.

Available since RTSJ 2.0

8.3.3.3.1 Methods

getCurrentConsumption(RelativeTime)

Signature

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

Description

Determine the CPU consumption for this release. When dest is null, return the CPU consumption in an otherwise unused [RelativeTime](#)⁵¹ instance in the current execution context. Otherwise, when dest is not null, return the CPU consumption in dest

Parameters

dest when not null is the object in which to return the result.

⁴⁸Section [8.3.3.5](#)

⁴⁹Section [8.3.3.7](#)

⁵⁰Section [8.3.3.9](#)

⁵¹Section [9.3.1.3](#)

Throws

IllegalStateException when the caller is not a [RealtimeThread](#)⁵².

Returns

the time consumed in the current release.

getCurrentConsumption

Signature

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

Description

Equivalent to `getCurrentConsumption(null)`.

Returns

the time consumed in the current release.

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](#)⁵³ 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`.

⁵²Section [5.3.2.2](#)

⁵³Section [8.3.3.4](#)

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

```
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 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 the case where a handler wishes to avoid the setup costs since the framework guarantees that `handleAsyncEvent()` will be invoked whenever the `fireCount` is greater than zero. The effect of a call to `getAndDecrementPendingFireCount` on the scheduling of this AEH depends on the semantics of the scheduler controlling this AEH.

Returns

The value held by `fireCount` prior to decrementing it by one.

getMemoryArea

Signature

```

public javax.realtime.MemoryArea
getMemoryArea()

```

Description

This is an accessor method for the initial instance of [MemoryArea](#)⁵⁴ associated with this.

To determine the current status of the memory area stack associated with this, use the static methods defined in the [RealtimeThread](#)⁵⁵ class. That is [RealtimeThread.getCurrentMemoryArea](#)⁵⁶, [RealtimeThread.getInitialMemoryAreaIndex](#)⁵⁷, [RealtimeThread.getMemoryAreaStackDepth](#)⁵⁸.

Returns

The instance of [MemoryArea](#)⁵⁹ which was passed as the area parameter when this was created (or the default value when area was allowed to default).

⁵⁴Section [11.3.2.3](#)

⁵⁵Section [5.3.2.2](#)

⁵⁶Section [5.3.2.2.2](#)

⁵⁷Section [5.3.2.2.2](#)

⁵⁸Section [5.3.2.2.2](#)

⁵⁹Section [11.3.2.3](#)

getMemoryParameters

Signature

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

Description

Gets a reference to the [MemoryParameters](#)⁶⁰ object for this schedulable.

Returns

A reference to the current [MemoryParameters](#)⁶¹ object.

getReleaseParameters

Signature

```
public javax.realtime.ReleaseParameters<?>  
getReleaseParameters()
```

Description

Gets a reference to the [ReleaseParameters](#)⁶² object for this schedulable.

Returns

A reference to the current [ReleaseParameters](#)⁶³ object.

getScheduler

Signature

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

Description

Gets a reference to the [Scheduler](#)⁶⁴ object for this schedulable.

Returns

A reference to the associated [Scheduler](#)⁶⁵ object.

⁶⁰Section [11.3.2.4](#)

⁶¹Section [11.3.2.4](#)

⁶²Section [6.3.3.10](#)

⁶³Section [6.3.3.10](#)

⁶⁴Section [6.3.3.12](#)

⁶⁵Section [6.3.3.12](#)

getSchedulingParameters

Signature

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

Description

Gets a reference to the [SchedulingParameters](#)⁶⁶ object for this schedulable.

Returns

A reference to the current [SchedulingParameters](#)⁶⁷ object.

getSchedulingGroup

Signature

```
public javax.realtime.SchedulingGroup  
getSchedulingGroup()
```

Description

Gets a reference to the [SchedulingGroup](#)⁶⁸ instance of this schedulable.

Returns

A reference to the current [SchedulingGroup](#)⁶⁹ object.

Available since since RTSJ 2.0

getConfigurationParameters

Signature

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

Description

Gets a reference to the [ConfigurationParameters](#)⁷⁰ object for this schedulable.

⁶⁶Section [6.3.3.14](#)

⁶⁷Section [6.3.3.14](#)

⁶⁸Section [6.3.3.13](#)

⁶⁹Section [6.3.3.13](#)

⁷⁰Section [5.3.2.1](#)

Returns

A reference to the associated [ConfigurationParameters](#)⁷¹ object.

Available since RTSJ 2.0

setMemoryParameters(MemoryParameters)

Signature

```
public T extends javax.realtime.AsyncBaseEventHandler<T>  
    setMemoryParameters(MemoryParameters memory)
```

Description

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

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

Parameters

memory A [MemoryParameters](#)⁷² 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](#)⁷³.)

Throws

`IllegalArgumentException` 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

setReleaseParameters(ReleaseParameters)

Signature

⁷¹Section [5.3.2.1](#)

⁷²Section [11.3.2.4](#)

⁷³Section [6.3.3.8](#)

```
public T extends javax.realtime.AsyncBaseEventHandler<T>
    setReleaseParameters(javax.realtime.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](#)⁷⁴ 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](#)⁷⁵.)

Throws

`IllegalArgumentException` Thrown 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.

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

Returns

this

setScheduler(Scheduler)*Signature*

```
public T extends javax.realtime.AsyncBaseEventHandler<T>
    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

⁷⁴Section [6.3.3.10](#)

⁷⁵Section [6.3.3.8](#)

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 if it is attached to an AsyncEvent (even if `AsyncBaseEvent.isRunning()`⁷⁶ would return false for that event).

Parameters

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

Throws

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

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

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

Returns

this

setScheduler(Scheduler, SchedulingParameters, ReleaseParameters, MemoryParameters)

Signature

```
public T extends javafx.realtime.AsyncBaseEventHandler<T>
    setScheduler(Scheduler scheduler,
                 SchedulingParameters scheduling,
                 javafx.realtime.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.

⁷⁶Section 8.3.3.2.1

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](#)⁷⁷ 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](#)⁷⁸.)

release A reference to the [ReleaseParameters](#)⁷⁹ 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](#)⁸⁰.)

memoryParameters A reference to the [MemoryParameters](#)⁸¹ 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](#)⁸².)

Throws

`IllegalArgumentException` Thrown 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.

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

Returns

this

setSchedulingParameters(SchedulingParameters)*Signature*

```
public T extends javax.realtime.AsyncBaseEventHandler<T>
    setSchedulingParameters(SchedulingParameters scheduling)
```

Description

⁷⁷Section [6.3.3.14](#)

⁷⁸Section [6.3.3.8](#)

⁷⁹Section [6.3.3.10](#)

⁸⁰Section [6.3.3.8](#)

⁸¹Section [11.3.2.4](#)

⁸²Section [6.3.3.8](#)

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`⁸³ 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`⁸⁴.)

Throws

`IllegalArgumentException` Thrown 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.

`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

this

setDaemon(boolean)

Signature

```
public final 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 daemon. 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

`IllegalSchedulableStateException` when this schedulable is active.

⁸³Section 6.3.3.14

⁸⁴Section 6.3.3.8

SecurityException 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

Get the dispatcher associated with this Timable.

See [Section Timable.getDispatcher\(\)](#)

getQueueLength

Signature

```
public int  
getQueueLength()
```

Description

Find 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.

getMinConsumption(RelativeTime)*Signature*

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

Description

Determine the minimum CPU consumption of all completed releases. When dest is null, return the CPU consumption in an otherwise unused [RelativeTime](#)⁸⁵ instance in the current execution context. Otherwise, when dest is not null, return the CPU consumption in dest

Parameters

dest when not null is the object in which to return the result.

Returns

the minimum time consumed in any release.

getMinConsumption*Signature*

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

Description

Same as [getMinConsumption\(RelativeTime\)](#)⁸⁶ with a null argument.

Returns

the minimum time consumed in any release.

getMaxConsumption(RelativeTime)*Signature*

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

⁸⁵Section [9.3.1.3](#)

⁸⁶Section [8.3.3.3.1](#)

Description

Determine the maximum CPU consumption of all completed releases. When `dest` is null, return the CPU consumption in an otherwise unused [RelativeTime](#)⁸⁷ instance in the current execution context. Otherwise, when `dest` is not null, return the CPU consumption in `dest`.

Parameters

`dest` when not null is the object in which to return the result.

Returns

the maximum time consumed in any release.

getMaxConsumption*Signature*

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

Description

Same as [getMaxConsumption\(RelativeTime\)](#)⁸⁸ with a null argument.

Returns

the maximum time consumed in any release.

mayUseHeap*Signature*

```
public boolean  
mayUseHeap()
```

Description

Determine 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.

⁸⁷Section [9.3.1.3](#)

⁸⁸Section [8.3.3.1](#)

isInterrupted

Signature

```
public boolean  
isInterrupted()
```

Description

Determines whether or not the generic [AsynchronouslyInterruptedException](#)⁸⁹ is pending.

Returns

true when and only when the generic [AsynchronouslyInterruptedException](#) is pending.

Available since RTSJ 2.0

interrupt

Signature

```
public void  
interrupt()
```

Description

Make the generic [AsynchronouslyInterruptedException](#)⁹⁰ 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 cause an early release. In any case, [AsynchronouslyInterruptedException](#) is thrown once a method is entered that implements [AsynchronouslyInterruptedException](#).

Available since RTSJ 2.0

isRousable

Signature

```
public boolean  
isRousable()
```

⁸⁹Section [8.3.2.1](#)

⁹⁰Section [8.3.2.1](#)

Description

Determine if it is possible for an interruptible to prematurely release the handler.

Returns

true when it is possible, otherwise it is not.

setRousable(boolean)*Signature*

```
public T extends javax.realtime.AsyncBaseEventHandler<T>
    setRousable(boolean value)
```

Description

Set the state for whether a interrupt can prematurely release this handler or not.

Parameters

value is the new value of the wake by interrupt state.

Returns

this

awaken*Signature*

```
public final void
    awaken()
```

Description

Indicate that a sleep has ended.

[See Section `Schedulable.awaken\(\)`](#)

run*Signature*

```
public 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.

8.3.3.4 AsyncEvent

Inheritance

java.lang.Object
 [javax.realtime.AsyncBaseEvent](#)
 [javax.realtime.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](#)⁹¹).

8.3.3.4.1 Constructors

AsyncEvent

Signature

```
public  
AsyncEvent()
```

Description

Create a new `AsyncEvent` object.

8.3.3.4.2 Methods

⁹¹Section [8.3.3.5](#)

fire*Signature*

```
public void
fire()
```

Description

When enabled, release the asynchronous events associated with this instance of AsyncEvent. When no handlers are attached or this object is disabled the method does nothing, i.e., it skips the release.

- When the instance of AsyncEvent has more than one instance of AsyncEventHandler with release parameters object of type AperiodicParameters attached and the execution of AsyncEvent.fire() introduces the requirement to throw at least one type of exception, then all instances of AsyncEventHandler not affected by the exception are handled normally
- When the instance of AsyncEvent has more than one instance of AsyncEventHandler with release parameters object of type SporadicParameters attached and the execution of AsyncEvent.fire() introduces the simultaneous requirement to throw more than one type of exception or error then [MITViolationException](#)⁹² has precedence over [ArrivalTimeQueueOverflowException](#)⁹³.

Throws

[MITViolationException](#) Thrown under the base priority scheduler's semantics when there is a handler associated with this event that has its MIT violated by the call to fire (and it has set the minimum inter-arrival time violation behavior to MITViolationExcept). Only the handlers which do not have their MITs violated are released in this situation.

[ArrivalTimeQueueOverflowException](#) when the queue of 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](#)⁹⁴, this exception is ignored.

8.3.3.5 AsyncEventHandler**Inheritance**

⁹²Section [15.2.2.7](#)

⁹³Section [15.2.2.1](#)

⁹⁴Section [8.3.1.1](#)

```
java.lang.Object
  javax.realtime.AsyncBaseEventHandler
    javax.realtime.AsyncEventHandler
```

Description

An asynchronous event handler encapsulates code that is released after an instance of `AsyncEvent`⁹⁵ 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()`⁹⁶ method. Control over the number of calls to `AsyncEventHandler.handleAsyncEvent()`⁹⁷ is given by methods which manipulate a fireCount. These may be called by the application via sub-classing and overriding `AsyncEventHandler.handleAsyncEvent()`⁹⁸.

Instances of `AsyncEventHandler` with a release parameter of type `SporadicParameters`⁹⁹ or `AperiodicParameters`¹⁰⁰ have a list of release times which correspond to the occurrence times of instances of `AsyncEvent`¹⁰¹ to which they are attached. The minimum interarrival time specified in `SporadicParameters`¹⁰² 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()`¹⁰³ 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`¹⁰⁴ method.

Normally, handlers are bound to an execution context dynamically when the instances of `AsyncEvent`¹⁰⁵s to which they are bound occur. This can introduce

⁹⁵Section 8.3.3.4

⁹⁶Section 8.3.3.5.2

⁹⁷Section 8.3.3.5.2

⁹⁸Section 8.3.3.5.2

⁹⁹Section 6.3.3.15

¹⁰⁰Section 6.3.3.2

¹⁰¹Section 8.3.3.4

¹⁰²Section 6.3.3.15

¹⁰³Section 8.3.3.5.2

¹⁰⁴Section 5.3.2.2.2

¹⁰⁵Section 8.3.3.4

a (small) time penalty. For critical handlers that cannot afford the expense, and where this penalty is a problem, `BoundAsyncEventHandler`¹⁰⁶s can be used.

The scheduler for an asynchronous event handler is inherited from the task that created it. When it was created from a Java thread, the scheduler is the current default scheduler.

The semantics for memory areas that were defined for realtime threads apply in the same way to instances of `AsyncEventHandler`. They may inherit a scope stack when they are created, and the single parent rule applies to the use of memory scopes for instances of `AsyncEventHandler` just as it does in realtime threads.

8.3.3.5.1 Constructors

AsyncEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, SchedulingGroup, ConfigurationParameters, Runnable)

Signature

```
public
AsyncEventHandler(SchedulingParameters scheduling,
                 javax.realtime.ReleaseParameters<?> release,
                 MemoryParameters memory,
                 MemoryArea area,
                 SchedulingGroup group,
                 ConfigurationParameters config,
                 Runnable logic)
```

Description

Create a handler with the given scheduling, release, memory, group, and configuration parameters to run the given logic.

Available since RTSJ 2.0

Parameters

¹⁰⁶Section 8.3.3.10

- scheduling parameters for scheduling the new handler (and possibly other instances of [Schedulable](#)¹⁰⁷). When scheduling is null and the creator is an instance of [Schedulable](#)¹⁰⁸, [SchedulingParameters](#)¹⁰⁹ is a clone of the creator's value created in the same memory area as this. When scheduling is null and the creator is a Java thread, the contents and type of the new [SchedulingParameters](#) object is governed by the associated scheduler.
- release parameters for scheduling the new handler (and possibly other instances of [Schedulable](#)¹¹⁰). When release is null the new [AsyncEventHandler](#) will use a clone of the default [ReleaseParameters](#)¹¹¹ for the associated scheduler created in the memory area that contains the [AsyncEventHandler](#) object.
- memory parameters for scheduling the new handler (and possibly other instances of [Schedulable](#)¹¹²). 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.
- group A [SchedulingGroup](#)¹¹³ object which will be associated with the constructed instance. When null, this will not be associated with any scheduling group.
- config parameters for reserving space for preallocated exceptions and change implementation specific per [Schedulable](#)¹¹⁴ memory reservations, such as Java stack size, for the new handler (and possibly other instances of [Schedulable](#)¹¹⁵. When initial is null, this [AsyncEventHandler](#) will reserve no space for pre-allocated exceptions and implementation-specific values will be set to their implementation-defined defaults.
- logic The [Runnable](#) object whose [run\(\)](#) method will serve as the logic for the new [AsyncEventHandler](#). When logic is null, the [handleAsyncEvent\(\)](#) method in the new object will serve as its logic.

AsyncEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, ConfigurationParameters, Runnable)

¹⁰⁷Section [6.3.1.3](#)

¹⁰⁸Section [6.3.1.3](#)

¹⁰⁹Section [6.3.3.14](#)

¹¹⁰Section [6.3.1.3](#)

¹¹¹Section [6.3.3.10](#)

¹¹²Section [6.3.1.3](#)

¹¹³Section [6.3.3.13](#)

¹¹⁴Section [6.3.1.3](#)

¹¹⁵Section [6.3.1.3](#)

Signature

```
public
AsyncEventHandler(SchedulingParameters scheduling,
                 javax.realtime.ReleaseParameters<?> release,
                 MemoryParameters memory,
                 MemoryArea area,
                 ConfigurationParameters config,
                 Runnable logic)
```

*Description***AsyncEventHandler(SchedulingParameters, ReleaseParameters, Runnable)***Signature*

```
public
AsyncEventHandler(SchedulingParameters scheduling,
                 javax.realtime.ReleaseParameters<?> release,
                 Runnable logic)
```

Description

Calling this constructor is equivalent to calling `AsyncEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, SchedulingGroup, ConfigurationParameters, Runnable)`¹¹⁶ with arguments (scheduling, release, null, null, null, null, logic).

Available since RTSJ 2.0

AsyncEventHandler(SchedulingParameters, ReleaseParameters)*Signature*

¹¹⁶Section 8.3.3.5.1


```
public
AsyncEventHandler(SchedulingParameters scheduling,
                 javax.realtime.ReleaseParameters<?> release)
```

Description

Calling this constructor is equivalent to calling `AsyncEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, SchedulingGroup, ConfigurationParameters, Runnable)`¹¹⁷ with arguments (scheduling, release, null, 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, SchedulingGroup, ConfigurationParameters, Runnable)`¹¹⁸ with arguments (null, null, null, null, null, null, logic).

AsyncEventHandler

Signature

```
public
AsyncEventHandler()
```

Description

Create an instance of AsyncEventHandler with default values for all parameters.

See Section `AsyncEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, SchedulingGroup, ConfigurationParameters, Runnable)`

¹¹⁷Section 8.3.3.5.1

¹¹⁸Section 8.3.3.5.1

8.3.3.5.2 Methods

handleAsyncEvent

Signature

```
public void  
handleAsyncEvent()
```

Description

This method holds the logic which is to be executed when any [AsyncEvent](#)¹¹⁹ 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.

¹¹⁹Section [8.3.3.4](#)

8.3.3.6 AsyncLongEvent

Inheritance

java.lang.Object
 javafx.runtime.AsyncBaseEvent
 javafx.runtime.AsyncLongEvent

Description

A new type of event that carries a long as a payload.

See [Section AsyncEvent](#)

Available since RTSJ 2.0

8.3.3.6.1 Constructors

AsyncLongEvent

Signature

```
public  
    AsyncLongEvent()
```

Description

Create a new AsyncLongEvent object.

8.3.3.6.2 Methods

fire(long)

Signature

```
public void  
    fire(long value)
```

throws `MITViolationException`,
`EventQueueOverflowException`

Description

When enabled, release the handlers associated with this instance of `AsyncLongEvent` with the long passed by `fire(long)`¹²⁰. When no handlers are attached or this object is disabled the method does nothing, i.e., it skips the release.

- When the instance of `AsyncLongEvent` is associated with more than one instance of `AsyncLongEventHandler`¹²¹ with release parameters object of type `AperiodicParameters`¹²² and the execution of `fire(long)`¹²³ introduces the requirement to throw at least one type of exception, then all instances of `AsyncLongEventHandler`¹²⁴ not affected by the exception are handled normally.
- When this instance of `AsyncLongEvent` is associated with more than one instance of `AsyncLongEventHandler`¹²⁵ with release parameters object of type `SporadicParameters`¹²⁶ and the execution of `fire(long)`¹²⁷ introduces the simultaneous requirement to throw more than one type of exception or error, then `MITViolationException`¹²⁸ has precedence over `ArrivalTimeQueueOverflowException`¹²⁹.

Parameters

value is the payload passed to the event.

Throws

`MITViolationException` Thrown under the base priority scheduler's semantics, when there is a handler associated with this event that has its MIT violated by the call to `fire` (and it has set the minimum inter-arrival time violation behavior to `MITViolationExcept`). Only the handlers which do not have their MITs violated are released in this situation.

`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

¹²⁰Section 8.3.3.6.2

¹²¹Section 8.3.3.7

¹²²Section 6.3.3.2

¹²³Section 8.3.3.6.2

¹²⁴Section 8.3.3.7

¹²⁵Section 8.3.3.7

¹²⁶Section 6.3.3.15

¹²⁷Section 8.3.3.6.2

¹²⁸Section 15.2.2.7

¹²⁹Section 15.2.2.1

infrastructure, such as for an [ActiveEvent](#)¹³⁰, this exception is ignored.

8.3.3.7 AsyncLongEventHandler

Inheritance

```
java.lang.Object
  javafx.runtime.AsyncBaseEventHandler
    javafx.runtime.AsyncLongEventHandler
```

Description

A version of [AsyncBaseEventHandler](#)¹³¹ that carries a long value as payload.

Available since RTSJ 2.0

8.3.3.7.1 Constructors

AsyncLongEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, SchedulingGroup, ConfigurationParameters, LongConsumer)

Signature

```
public
  AsyncLongEventHandler(SchedulingParameters scheduling,
                       javax.realtime.ReleaseParameters<?> release,
                       MemoryParameters memory,
                       MemoryArea area,
                       SchedulingGroup group,
                       ConfigurationParameters config,
                       LongConsumer logic)
  throws IllegalArgumentException
```

Description

Create an asynchronous event handler that receives a Long payload with each fire.

¹³⁰Section [8.3.1.1](#)

¹³¹Section [8.3.3.3](#)

Parameters

scheduling parameters for scheduling the new handler (and possibly other instances of [Schedulable](#)¹³²). When scheduling is null and the creator is an instance of [Schedulable](#)¹³³, [SchedulingParameters](#)¹³⁴ is a clone of the creator's value created in the same memory area as this. When scheduling is null and the creator is a Java thread, the contents and type of the new [SchedulingParameters](#) object is governed by the associated scheduler.

release parameters for scheduling the new handler (and possibly other instances of [Schedulable](#)¹³⁵). When release is null the new [AsyncEventHandler](#) will use a clone of the default [ReleaseParameters](#)¹³⁶ for the associated scheduler created in the memory area that contains the [AsyncEventHandler](#) object.

memory parameters for scheduling the new handler (and possibly other instances of [Schedulable](#)¹³⁷). 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.

group parameters for providing CPU cost management on a set of [Schedulable](#)¹³⁸s. When null, this will not be associated with any processing group.

config parameters for reserving space for preallocated exceptions and change implementation specific per [Schedulable](#)¹³⁹ memory reservations, such as Java stack size, for the new handler (and possibly other instances of [Schedulable](#)¹⁴⁰. When initial is null, this [AsyncEventHandler](#) will reserve no space for preallocated exceptions and implementation-specific values will be set to their implementation-defined defaults.

logic is the logic to run for each fire. When logic is null, the `handleAsyncEvent()` method in the new object will serve as its logic.

Throws

`IllegalArgumentException` when the event queue overflow policy is [QueueOverflowPolicy.DISABLE](#)¹⁴¹.

¹³²Section [6.3.1.3](#)

¹³³Section [6.3.1.3](#)

¹³⁴Section [6.3.3.14](#)

¹³⁵Section [6.3.1.3](#)

¹³⁶Section [6.3.3.10](#)

¹³⁷Section [6.3.1.3](#)

¹³⁸Section [6.3.1.3](#)

¹³⁹Section [6.3.1.3](#)

¹⁴⁰Section [6.3.1.3](#)

¹⁴¹Section [6.3.2.2.1](#)

AsyncLongEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, ConfigurationParameters, LongConsumer)

Signature

```
public
AsyncLongEventHandler(SchedulingParameters scheduling,
                      javax.realtime.ReleaseParameters<?> release,
                      MemoryParameters memory,
                      MemoryArea area,
                      ConfigurationParameters config,
                      LongConsumer logic)
throws IllegalArgumentException
```

Description

Calling this constructor is equivalent to calling [AsyncLongEventHandler\(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, SchedulingGroup, ConfigurationParameters, LongConsumer\)](#)¹⁴² with arguments (scheduling, release, memory, area, null, config, logic). This constructor is needed for SCJ.

AsyncLongEventHandler(SchedulingParameters, ReleaseParameters, LongConsumer)

Signature

```
public
AsyncLongEventHandler(SchedulingParameters scheduling,
                      javax.realtime.ReleaseParameters<?> release,
                      LongConsumer logic)
throws IllegalArgumentException
```

Description

Calling this constructor is equivalent to calling [AsyncLongEventHandler\(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, SchedulingGroup, ConfigurationParameters, LongConsumer\)](#)¹⁴³ with arguments (scheduling, release, null, null, null, null, logic).

¹⁴²Section 8.3.3.7.1

¹⁴³Section 8.3.3.7.1

AsyncLongEventHandler(SchedulingParameters, ReleaseParameters)

Signature

```
public  
AsyncLongEventHandler(SchedulingParameters scheduling,  
                     javax.realtime.ReleaseParameters<?> release)  
throws IllegalArgumentException
```

Description

Calling this constructor is equivalent to calling `AsyncLongEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, SchedulingGroup, ConfigurationParameters, LongConsumer)`¹⁴⁴ with arguments (scheduling, release, null, null, null, null, null)

AsyncLongEventHandler(LongConsumer)

Signature

```
public  
AsyncLongEventHandler(LongConsumer logic)
```

Description

Calling this constructor is equivalent to calling `AsyncLongEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, SchedulingGroup, ConfigurationParameters, LongConsumer)`¹⁴⁵ with arguments (null, null, null, null, null, null, logic).

AsyncLongEventHandler

Signature

```
public  
AsyncLongEventHandler()
```

¹⁴⁴Section 8.3.3.7.1

¹⁴⁵Section 8.3.3.7.1

Description

Create an instance of AsyncLongEventHandler with default values for all parameters.

See [Section AsyncLongEventHandler\(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, SchedulingGroup, ConfigurationParameters, LongConsumer\)](#)

8.3.3.7.2 Methods

handleAsyncEvent(long)

Signature

```
public void  
handleAsyncEvent(long payload)
```

Description

This method holds the logic which is to be executed when any [AsyncEvent¹⁴⁶](#) 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 is the long value associated with a fire.

peekPending

Signature

```
public long  
peekPending()  
throws IllegalStateException
```

¹⁴⁶Section [8.3.3.4](#)

Description

Determine the next value queued for handling.

Throws

IllegalStateException when the fire count is zero.

Returns

The long value at the head of the queue of longs to be passed to `handleAsyncEvent(long)`¹⁴⁷.

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.

8.3.3.8 AsyncObjectEvent

Inheritance

```
java.lang.Object  
  javax.realtime.AsyncBaseEvent  
    javax.realtime.AsyncObjectEvent
```

Description

A new type of event that carries an object as a payload.

See Section [AsyncEvent](#)

Available since RTSJ 2.0

¹⁴⁷Section [8.3.3.7.2](#)

8.3.3.8.1 Constructors

AsyncObjectEvent

Signature

```
public  
AsyncObjectEvent()
```

Description

Create a new AsyncObjectEvent instance.

8.3.3.8.2 Methods

fire(P)

Signature

```
public void  
fire(P value)  
throws MITViolationException,  
       EventQueueOverflowException,  
       IllegalAssignmentError
```

Description

When enabled, fire this instance of AsyncObjectEvent. The asynchronous event handlers associated with this event will be released with the object passed by [fire](#)¹⁴⁸. When no handlers are attached or this object is disabled the method does nothing, i.e., it skips the release.

- When the instance of AsyncObjectEvent is associated with more than one instance of [AsyncObjectEventHandler](#)¹⁴⁹ with release parameters object of type [AperiodicParameters](#)¹⁵⁰ and the execution of [fire](#)¹⁵¹ introduces the

¹⁴⁸Section [8.3.3.8.2](#)

¹⁴⁹Section [8.3.3.9](#)

¹⁵⁰Section [6.3.3.2](#)

¹⁵¹Section [8.3.3.8.2](#)

requirement to throw at least one type of exception, then all instances of `AsyncObjectEventHandler`¹⁵² not affected by the exception are handled normally.

- When this instance of `AsyncObjectEvent` is associated with more than one instance of `AsyncObjectEventHandler`¹⁵³ with release parameters object of type `SporadicParameters`¹⁵⁴ and the execution of `fire`¹⁵⁵ introduces the simultaneous requirement to throw more than one type of exception or error, then `MITViolationException`¹⁵⁶ has precedence over `ArrivalTimeQueueOverflowException`¹⁵⁷.

Parameters

value is the payload passed to the event.

Throws

`MITViolationException` Thrown under the base priority scheduler's semantics when there is a handler associated with this event that has its MIT violated by the call to `fire` (and it has set the minimum inter-arrival time violation behavior to `MITViolationExcept`). Only the handlers which do not have their MITs violated are released in this situation.

`ArrivalTimeQueueOverflowException` when the queue of 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`¹⁵⁸, this exception is ignored.

`IllegalAssignmentError` when P is not assignable the event queue of one of the associated handlers.

8.3.3.9 AsyncObjectEventHandler

Inheritance

```
java.lang.Object
  javax.realtime.AsyncBaseEventHandler
    javax.realtime.AsyncObjectEventHandler
```

Description

¹⁵²Section 8.3.3.9

¹⁵³Section 8.3.3.9

¹⁵⁴Section 6.3.3.15

¹⁵⁵Section 8.3.3.8.2

¹⁵⁶Section 15.2.2.7

¹⁵⁷Section 15.2.2.1

¹⁵⁸Section 8.3.1.1

A version of [AsyncBaseEventHandler](#)¹⁵⁹ that carries an Object value as payload.

Available since RTSJ 2.0

8.3.3.9.1 Constructors

AsyncObjectEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, SchedulingGroup, ConfigurationParameters, Consumer)

Signature

```
public
AsyncObjectEventHandler(SchedulingParameters scheduling,
                        javax.realtime.ReleaseParameters<?> release,
                        MemoryParameters memory,
                        MemoryArea area,
                        SchedulingGroup group,
                        ConfigurationParameters config,
                        java.util.function.Consumer<P> logic)
throws IllegalArgumentException
```

Description

Create an asynchronous event handler that receives a Long payload with each fire.

Parameters

scheduling parameters for scheduling the new handler (and possibly other instances of [Schedulable](#)¹⁶⁰). When scheduling is null and the creator is an instance of [Schedulable](#)¹⁶¹, [SchedulingParameters](#)¹⁶² 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.

¹⁵⁹Section [8.3.3.3](#)

¹⁶⁰Section [6.3.1.3](#)

¹⁶¹Section [6.3.1.3](#)

¹⁶²Section [6.3.3.14](#)

release parameters for scheduling the new handler (and possibly other instances of [Schedulable](#)¹⁶³). When release is null the new AsyncEventHandler will use a clone of the default [ReleaseParameters](#)¹⁶⁴ for the associated scheduler created in the memory area that contains the AsyncEventHandler object.

memory parameters for scheduling the new handler (and possibly other instances of [Schedulable](#)¹⁶⁵). 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.

group parameters for providing CPU cost management on a set of [Schedulable](#)¹⁶⁶s. When null, this will not be associated with any processing group.

config parameters for reserving space for preallocated exceptions and change implementation specific per [Schedulable](#)¹⁶⁷ memory reservations, such as Java stack size, for the new handler (and possibly other instances of [Schedulable](#)¹⁶⁸. When initial is null, this AsyncEventHandler will reserve no space for pre-allocated exceptions and implementation-specific values will be set to their implementation-defined defaults.

logic is the logic to run for each fire. When logic is null, the handleAsyncEvent method in the new object will serve as its logic.

Throws

IllegalArgumentException when the event queue overflow policy is [QueueOverflowPolicy.DISABLE](#)¹⁶⁹.

AsyncObjectEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, ConfigurationParameters, Consumer)

Signature

```
public
AsyncObjectEventHandler(SchedulingParameters scheduling,
                       javax.realtime.ReleaseParameters<?> release,
                       MemoryParameters memory,
```

¹⁶³Section [6.3.1.3](#)

¹⁶⁴Section [6.3.3.10](#)

¹⁶⁵Section [6.3.1.3](#)

¹⁶⁶Section [6.3.1.3](#)

¹⁶⁷Section [6.3.1.3](#)

¹⁶⁸Section [6.3.1.3](#)

¹⁶⁹Section [6.3.2.2.1](#)

MemoryArea area,
ConfigurationParameters config,
java.util.function.Consumer<P> logic)
throws IllegalArgumentException

Description

Calling this constructor is equivalent to calling `AsyncObjectEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, SchedulingGroup, ConfigurationParameters, Consumer)`¹⁷⁰ with arguments (scheduling, release, memory, area, null, config, logic). This constructor is needed for SCJ.

AsyncObjectEventHandler(SchedulingParameters, ReleaseParameters, Consumer)

Signature

public
AsyncObjectEventHandler(SchedulingParameters scheduling,
 javax.realtime.ReleaseParameters<?> release,
 java.util.function.Consumer<P> logic)
throws IllegalArgumentException

Description

Calling this constructor is equivalent to calling `AsyncObjectEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, SchedulingGroup, ConfigurationParameters, Consumer)`¹⁷¹ with arguments (scheduling, release, null, null, null, null, logic).

AsyncObjectEventHandler(SchedulingParameters, ReleaseParameters)

Signature

public
AsyncObjectEventHandler(SchedulingParameters scheduling,
 javax.realtime.ReleaseParameters<?> release)

¹⁷⁰Section 8.3.3.9.1

¹⁷¹Section 8.3.3.9.1

throws `IllegalArgumentException`

Description

Calling this constructor is equivalent to calling `AsyncObjectEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, SchedulingGroup, ConfigurationParameters, Consumer)`¹⁷² with arguments (scheduling, release, null, null, null, null, null)

AsyncObjectEventHandler(Consumer)

Signature

```
public  
AsyncObjectEventHandler(java.util.function.Consumer<P> logic)
```

Description

Calling this constructor is equivalent to calling `AsyncObjectEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, SchedulingGroup, ConfigurationParameters, Consumer)`¹⁷³ with arguments (null, null, null, null, null, null, logic).

Parameters

logic is the function to call on the object received.

AsyncObjectEventHandler

Signature

```
public  
AsyncObjectEventHandler()
```

Description

Create an instance of `AsyncObjectEventHandler` with default values for all parameters.

See Section `AsyncObjectEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, SchedulingGroup, ConfigurationParameters, Consumer)`

¹⁷²Section 8.3.3.9.1

¹⁷³Section 8.3.3.9.1

8.3.3.9.2 Methods

handleAsyncEvent(P)

Signature

```
public void  
handleAsyncEvent(P value)
```

Description

This method holds the logic which is to be executed when any [AsyncEvent](#)¹⁷⁴ 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 IllegalStateException
```

Description

Determine the next value queued for handling.

Throws

IllegalStateException when the fire count is zero.

Returns

The object reference at the head of the queue of object references to be passed to [handleAsyncEvent](#)¹⁷⁵.

¹⁷⁴Section [8.3.3.4](#)

¹⁷⁵Section [8.3.3.9.2](#)

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.

8.3.3.10 BoundAsyncEventHandler

Inheritance

```
java.lang.Object  
  javax.realtime.AsyncBaseEventHandler  
    javax.realtime.AsyncEventHandler  
      javax.realtime.BoundAsyncEventHandler
```

Interfaces

```
  javax.realtime.BoundAsyncBaseEventHandler
```

Description

A bound asynchronous event handler is an instance of [AsyncEventHandler](#)¹⁷⁶ 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.

8.3.3.10.1 Constructors

¹⁷⁶Section [8.3.3.5](#)

BoundAsyncEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, SchedulingGroup, ConfigurationParameters, Runnable)

Signature

```
public
BoundAsyncEventHandler(SchedulingParameters scheduling,
                       javax.realtime.ReleaseParameters<?> release,
                       MemoryParameters memory,
                       MemoryArea area,
                       SchedulingGroup group,
                       ConfigurationParameters config,
                       Runnable logic)
```

Description

Create an instance of BoundAsyncEventHandler with the specified parameters. The newly-created handler inherits the affinity of its creator.

Parameters

scheduling A [SchedulingParameters](#)¹⁷⁷ object which will be associated with the constructed instance. When null, and the creator is a Java thread, a SchedulingParameters object is created which has the default SchedulingParameters for the scheduler associated with the current thread. When null, and the creator is a schedulable object, the SchedulingParameters are inherited from the current schedulable (a new SchedulingParameters object is cloned).

release A [ReleaseParameters](#)¹⁷⁸ object which will be associated with the constructed instance. When null, this will have default ReleaseParameters for the BAEH's scheduler.

memory A [MemoryParameters](#)¹⁷⁹ 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](#)¹⁸⁰ for this. When null, the memory area will be that of the current thread/schedulable.

group A [SchedulingGroup](#)¹⁸¹ object which will be associated with the constructed instance. When null, this will not be associated with any scheduling group.

¹⁷⁷Section [6.3.3.14](#)

¹⁷⁸Section [6.3.3.10](#)

¹⁷⁹Section [11.3.2.4](#)

¹⁸⁰Section [11.3.2.3](#)

¹⁸¹Section [6.3.3.13](#)

config The [ConfigurationParameters](#)¹⁸² associated with this (and possibly other instances of [Schedulable](#)¹⁸³. 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\(\)](#)¹⁸⁴. When null, the default handleAsyncEvent() method invokes nothing.

Throws

[IllegalArgumentException](#) when mayUseHeap in memory is true and logic, any parameter object, or this is in heap memory. Also when noheap 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.

BoundAsyncEventHandler(SchedulingParameters, ReleaseParameters, Runnable)

Signature

```
public
BoundAsyncEventHandler(SchedulingParameters scheduling,
                      javax.realtime.ReleaseParameters<?> release,
                      Runnable logic)
```

Description

Create 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, config, logic)

BoundAsyncEventHandler(SchedulingParameters, ReleaseParameters)

¹⁸²Section [5.3.2.1](#)

¹⁸³Section [6.3.1.3](#)

¹⁸⁴Section [8.3.3.5.2](#)

Signature

```
public  
BoundAsyncEventHandler(SchedulingParameters scheduling,  
                       javax.realtime.ReleaseParameters<?> release)
```

Description

Create 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, logic)

BoundAsyncEventHandler(Runnable)

Signature

```
public  
BoundAsyncEventHandler(Runnable logic)
```

Description

Create 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)

BoundAsyncEventHandler

Signature

```
public  
BoundAsyncEventHandler()
```

Description

Create an instance of BoundAsyncEventHandler. The newly-created handler inherits the affinity of its creator.

Equivalent to BoundAsyncEventHandler(null, null, null, null, null, null, null)

8.3.3.11 BoundAsyncLongEventHandler

Inheritance

```

java.lang.Object
  javax.realtime.AsyncBaseEventHandler
    javax.realtime.AsyncLongEventHandler
      javax.realtime.BoundAsyncLongEventHandler

```

Interfaces

```

javax.realtime.BoundAsyncBaseEventHandler

```

Description

A bound asynchronous event handler is an instance of [AsyncLongEventHandler](#)¹⁸⁵ 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

8.3.3.11.1 Constructors

BoundAsyncLongEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, SchedulingGroup, ConfigurationParameters, LongConsumer)

Signature

```

public
BoundAsyncLongEventHandler(SchedulingParameters scheduling,
                           javax.realtime.ReleaseParameters<?> release,
                           MemoryParameters memory,
                           MemoryArea area,
                           SchedulingGroup group,
                           ConfigurationParameters config,
                           LongConsumer logic)

```

¹⁸⁵Section [8.3.3.7](#)

Description

Create an instance of BoundAsyncLongEventHandler which specifies all possible parameters. The newly-created handler inherits the affinity of its creator.

Parameters

- scheduling A [SchedulingParameters](#)¹⁸⁶ object which will be associated with the constructed instance. When null, and the creator is a Java thread, a SchedulingParameters object is created which has the default SchedulingParameters for the scheduler associated with the current thread. When null, and the creator is a schedulable object, the SchedulingParameters are inherited from the current schedulable (a new SchedulingParameters object is cloned).
- release A [ReleaseParameters](#)¹⁸⁷ object which will be associated with the constructed instance. When null, this will have default ReleaseParameters for the BAEH's scheduler.
- memory A [MemoryParameters](#)¹⁸⁸ 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](#)¹⁸⁹ for this. When null, the memory area will be that of the current thread/schedulable.
- group A [SchedulingGroup](#)¹⁹⁰ object which will be associated with the constructed instance. When null, this will not be associated with any scheduling group.
- config The [ConfigurationParameters](#)¹⁹¹ associated with this (and possibly other instances of [Schedulable](#)¹⁹². 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 LongConsumer object whose accept() method is executed by [AsyncLongEventHandler.handleAsyncEvent\(long\)](#)¹⁹³. When null, the default handleAsyncEvent(long) method invokes nothing.

Throws

- IllegalArgumentException when mayUseHeap in memory is true and logic, any parameter object, or this is in heap memory. Also when noheap is true and area is heap memory.

¹⁸⁶Section [6.3.3.14](#)

¹⁸⁷Section [6.3.3.10](#)

¹⁸⁸Section [11.3.2.4](#)

¹⁸⁹Section [11.3.2.3](#)

¹⁹⁰Section [6.3.3.13](#)

¹⁹¹Section [5.3.2.1](#)

¹⁹²Section [6.3.1.3](#)

¹⁹³Section [8.3.3.7.2](#)

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.

BoundAsyncLongEventHandler(SchedulingParameters, ReleaseParameters, LongConsumer)

Signature

```
public
BoundAsyncLongEventHandler(SchedulingParameters scheduling,
                           javax.realtime.ReleaseParameters<?> release,
                           LongConsumer logic)
```

Description

BoundAsyncLongEventHandler(SchedulingParameters, ReleaseParameters)

Signature

```
public
BoundAsyncLongEventHandler(SchedulingParameters scheduling,
                           javax.realtime.ReleaseParameters<?> release)
```

Description

BoundAsyncLongEventHandler(LongConsumer)

Signature

```
public
BoundAsyncLongEventHandler(LongConsumer logic)
```


Description

BoundAsyncLongEventHandler

Signature

```
public  
BoundAsyncLongEventHandler()
```

Description

Create an instance of BoundAsyncLongEventHandler using default values. This constructor is equivalent to BoundAsyncLongEventHandler(null, null, null, null, null, false, null)

8.3.3.12 BoundAsyncObjectEventHandler

Inheritance

```
java.lang.Object  
  javax.realtime.AsyncBaseEventHandler  
    javax.realtime.AsyncObjectEventHandler  
      javax.realtime.BoundAsyncObjectEventHandler
```

Interfaces

```
javax.realtime.BoundAsyncBaseEventHandler
```

Description

A bound asynchronous event handler is an instance of [AsyncObjectEventHandler](#)¹⁹⁴ 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

¹⁹⁴Section [8.3.3.9](#)

8.3.3.12.1 Constructors

BoundAsyncObjectEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, ProcessingGroup, ConfigurationParameters, Consumer)*Signature*

```

public
BoundAsyncObjectEventHandler(SchedulingParameters scheduling,
                             javax.realtime.ReleaseParameters<?> release,
                             MemoryParameters memory,
                             MemoryArea area,
                             ProcessingGroup group,
                             ConfigurationParameters config,
                             java.util.function.Consumer<P> logic)

```

Description

Create an instance of `BoundAsyncObjectEventHandler` which specifies all possible parameters. The newly-created handler inherits the affinity of its creator.

Parameters

`scheduling` A [SchedulingParameters](#)¹⁹⁵ object which will be associated with the constructed instance. When null, and the creator is a Java thread, a `SchedulingParameters` object is created which has the default `SchedulingParameters` for the scheduler associated with the current thread. When null, and the creator is a schedulable object, the `SchedulingParameters` are inherited from the current schedulable (a new `SchedulingParameters` object is cloned).

`release` A [ReleaseParameters](#)¹⁹⁶ object which will be associated with the constructed instance. When null, this will have default `ReleaseParameters` for the BAEH's scheduler.

`memory` A [MemoryParameters](#)¹⁹⁷ 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](#)¹⁹⁸ for this. When null, the memory area will be that of the

¹⁹⁵Section [6.3.3.14](#)

¹⁹⁶Section [6.3.3.10](#)

¹⁹⁷Section [11.3.2.4](#)

¹⁹⁸Section [11.3.2.3](#)

current thread/schedulable.

group A [SchedulingGroup](#)¹⁹⁹ object which will be associated with the constructed instance. When null, this will not be associated with any scheduling group.

config The [ConfigurationParameters](#)²⁰⁰ associated with this (and possibly other instances of [Schedulable](#)²⁰¹. 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](#)²⁰². When null, the default handleAsyncEvent method invokes nothing.

Throws

[IllegalArgumentException](#) when mayUseHeap in memory is true and logic, any parameter object, or this is in heap memory. Also when noheap 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.

BoundAsyncObjectEventHandler(SchedulingParameters, ReleaseParameters, Consumer)

Signature

```
public
BoundAsyncObjectEventHandler(SchedulingParameters scheduling,
                             javax.realtime.ReleaseParameters<?> release,
                             java.util.function.Consumer<P> logic)
```

Description

BoundAsyncObjectEventHandler(SchedulingParameters, ReleaseParameters)

¹⁹⁹Section [6.3.3.13](#)

²⁰⁰Section [5.3.2.1](#)

²⁰¹Section [6.3.1.3](#)

²⁰²Section [8.3.3.9.2](#)

Signature

```
public  
BoundAsyncObjectEventHandler(SchedulingParameters scheduling,  
                             javax.realtime.ReleaseParameters<?> release)
```

Description

BoundAsyncObjectEventHandler(Consumer)

Signature

```
public  
BoundAsyncObjectEventHandler(java.util.function.Consumer<P> logic)
```

Description

BoundAsyncObjectEventHandler

Signature

```
public  
BoundAsyncObjectEventHandler()
```

Description

Create an instance of `BoundAsyncObjectEventHandler` using default values. This constructor is equivalent to `BoundAsyncObjectEventHandler(null, null, null, null, null, null)`

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 (the interface `BoundAsyncBaseEventHandler`). The RTSJ does not define at what point a nonbound 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.

The ability to trigger an ATC in a schedulable is necessary in many kinds of realtime applications but must be designed carefully in order to minimize the risks of problems such as data structure corruption and deadlock. There is, invariably, a tension between the desire to cause an ATC to be immediate, and the desire to ensure that certain sections of code are executed to completion.

One basic decision was to allow ATC in a method only if the method explicitly permits this. The default of no ATC is reasonable, since legacy code might be written expecting no ATC, and asynchronously aborting the execution of such a method could lead to unpredictable results. Since the natural way to model ATC is with an exception (`AsynchronouslyInterruptedException`), the way that a method indicates its susceptibility to ATC is by including `AsynchronouslyInterruptedException` in its throws clause. Causing this exception to be thrown in a 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

```
    catch (Throwable e) return;
```

the AIE will remain pending and will be thrown next time control enters or returns to an AI method.

This specification does not provide a special mechanism for terminating a realtime thread; ATC can be used to achieve this effect. This means that, by default, a realtime thread cannot be asynchronously terminated; to support asynchronous termination it needs to enter methods that are AI enabled at frequent intervals. Allowing termination as the default would have been questionable, bringing the same insecurities that are found in `Thread.stop()` and `Thread.destroy()`.

Chapter 9

Time

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. Neither provides any type safety. This specification addresses this by requiring three time classes: `HighResolutionTime`, `AbsoluteTime`, and `RelativeTime`, where `HighResolutionTime` is the parent class of the other two.

Instances of `HighResolutionTime` 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.

Realtime Epoch — The time at which the realtime clock began ticking, defined by fiat as January 1, 1970 00:00:00 UTC.

Epoch — The date and time relative to which times on an RTSJ Chronograph, *c* are determined. The epoch for a chronograph is defined in terms of the *Realtime Epoch*, and is represented as the time elapsed on the realtime clock since the realtime Epoch at the time that *c* would have returned a timestamp 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, (`MAX_LONG` milliseconds, `MAX_INT` 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

time in ns	millis	nanos
2000000	2	0
1999999	1	999999
1000001	1	1
1	0	1
0	0	0
-1	0	-1
-999999	0	-999999
-1000000	-1	0
-1000001	-1	-1

3. Time objects maintain and report time values in normalized form, but the normalized form is not required for input parameter values. This allows computations individually with time constituent parts using the full *signed* range and restrictions of the underlying type.
 - (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 9.2: Semantics of Time Conversion

Chronograph association & conversion this has chronograph_a & ms,ns	returned/updated object
an_absolute.absolute(chronograph_a)	chronograph_a ms,ns
an_absolute.absolute(chronograph_b)	chronograph_b ms,ns
an_absolute.absolute(null)	realtime_clock ms,ns
an_absolute.relative(chronograph_a)	chronograph_a chronograph_a.getTime().subtract(ms,ns)
an_absolute.relative(chronograph_b)	chronograph_b chronograph_b.getTime().subtract(ms,ns)
an_absolute.relative(null)	realtime_clock realtime_clock.getTime().subtract(ms,ns)
a_relative.relative(chronograph_a)	chronograph_a ms,ns
a_relative.relative(chronograph_b)	chronograph_b ms,ns
a_relative.relative(null)	realtime_clock ms,ns
a_relative.absolute(chronograph_a)	chronograph_a chronograph_a.getTime().add(ms,ns)
a_relative.absolute(chronograph_b)	chronograph_b chronograph_b.getTime().add(ms,ns)
a_relative.absolute(null)	realtime_clock realtime_clock.getTime().add(ms,ns)

7. Time objects must implement the Comparable interface.

9.3 javafx.runtime

9.3.1 Classes

9.3.1.1 AbsoluteTime

Inheritance

```
java.lang.Object
  javafx.runtime.HighResolutionTime
    javafx.runtime.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 default realtime clock, the fixed point is the Epoch (January 1, 1970, 00:00:00 GMT). The correctness of the Epoch as a time base depends on the realtime clock synchronization with an external world time reference. This representation was designed to be compatible with the standard Java representation of an absolute time in the `java.util.Date` class.

A time object in normalized form represents negative time when both components are nonzero and negative, or one is nonzero and negative and the other is zero. For add and subtract negative values behave as they do in arithmetic.

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

9.3.1.1.1 Constructors

AbsoluteTime(long, int, Chronograph)

Signature

```
public
AbsoluteTime(long millis,
              int nanos,
              Chronograph chronograph)
throws IllegalArgumentException
```

Description

Construct an `AbsoluteTime` object with time millisecond and nanosecond components past the epoch for `Chronograph`.

The value of the `AbsoluteTime` instance is based on the parameter `millis` plus the parameter `nanos`. The construction is subject to `millis` and `nanos` parameters normalization. When, after normalization, the time object is negative, the time represented by this is time before this chronograph's epoch. The chronograph association is made with the `Chronograph` parameter. When `Chronograph` is null the association is made with the default realtime clock.

Note, the start of a chronograph's epoch is an attribute of the chronograph. It is defined as the Epoch (00:00:00 GMT on Jan 1, 1970) for the default realtime clock, but other classes of chronograph may define other epochs.

Available since RTSJ 2.0

Parameters

`millis` The desired value for the millisecond component of this. The actual value is the result of parameter normalization.

`nanos` The desired value for the nanosecond component of this. The actual value is the result of parameter normalization.

`chronograph` The chronograph providing the association for the newly constructed object. The realtime clock is used when this argument is null.

Throws

`IllegalArgumentException` when there is an overflow in the millisecond component when normalizing.

`AbsoluteTime(long, int)`

Signature

```
public
    AbsoluteTime(long millis,
                  int nanos)
    throws IllegalArgumentException
```

Description

Equivalent to `AbsoluteTime(long, int, Chronograph)`¹ with the argument list (long, int, null)

Parameters

¹Section [9.3.1.1.1](#)

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

nanos is the desired value for the nanosecond component of this. The actual value is the result of parameter normalization.

Throws

IllegalArgumentException when there is an overflow in the millisecond component when normalizing.

AbsoluteTime(Date, Chronograph)

Signature

```
public  
AbsoluteTime(Date date,  
              Chronograph chronograph)  
throws IllegalArgumentException
```

Description

Equivalent to [AbsoluteTime\(long, int, Chronograph\)](#)² 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 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

date The java.util.Date representation of the time past the Epoch.

chronograph The chronograph providing the association for the newly constructed object.

Throws

IllegalArgumentException when the date parameter is null.

²Section [9.3.1.1.1](#)

AbsoluteTime(Date)

Signature

```
public  
AbsoluteTime(Date date)  
throws IllegalArgumentException
```

Description

Equivalent to `AbsoluteTime(long, int, Chronograph)`³ with the argument list `(date.getTime(), 0, null)`.

Parameters

date the java.util.Date representation of the time past the Epoch.

Throws

IllegalArgumentException when the date parameter is null.

AbsoluteTime(AbsoluteTime)

Signature

```
public  
AbsoluteTime(AbsoluteTime time)  
throws IllegalArgumentException
```

Description

Equivalent to `AbsoluteTime(long, int, Chronograph)`⁴ with the argument list `(time.getMilliseconds(), time.getNanoseconds(), time.getChronograph())`.

Parameters

time The AbsoluteTime object which is the source for the copy.

Throws

IllegalArgumentException when the time parameter is null.

AbsoluteTime(Chronograph)

Signature

³Section 9.3.1.1.1

⁴Section 9.3.1.1.1

```
public  
AbsoluteTime(Chronograph chronograph)
```

Description

Equivalent to `AbsoluteTime(long, int, Chronograph)`⁵ with the argument list (0, 0, chronograph).

Available since RTSJ 2.0

Parameters

chronograph The chronograph providing the association for the newly constructed object.

AbsoluteTime

Signature

```
public  
AbsoluteTime()
```

Description

Equivalent to `AbsoluteTime(long, int, Chronograph)`⁶ with the argument list (0, 0, null).

9.3.1.1.2 Methods

absolute(Chronograph)

Signature

```
public javax.realtime.AbsoluteTime  
absolute(Chronograph chronograph)
```

Description

⁵Section 9.3.1.1.1

⁶Section 9.3.1.1.1

Return a copy of this modified when necessary to have the specified chronograph association. A new object is allocated for the result. This method is the implementation of the abstract method of the `HighResolutionTime` base class. No conversion into `AbsoluteTime` is needed in this case. The 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` The `Chronograph` parameter is used only as the new chronograph association with the result, since no conversion is needed.

Returns

The copy of this in a newly allocated `AbsoluteTime` object, associated with the `Chronograph` parameter.

absolute(Chronograph, AbsoluteTime)

Signature

```
public javax.realtime.AbsoluteTime  
absolute(Chronograph chronograph,  
         AbsoluteTime dest)
```

Description

Convert the time of this to an absolute time, using the given instance of [Chronograph](#)⁷ to determine the current time when necessary. When `Chronograph` is null the realtime chronograph is assumed. When `dest` is not null, the result is placed there and returned. Otherwise, a new object is allocated for the result. The chronograph association of the result is the `Chronograph` passed as a parameter. See the subclass comments for more specific information.

Parameters

`chronograph` The instance of [Chronograph](#)⁸ used to convert the time of this into absolute time, and the new chronograph association for the result.

`dest` When `dest` is not null, the result is placed there and returned. Otherwise, a new object is allocated for the result.

Returns

The `AbsoluteTime` conversion in `dest` when `dest` is not null, otherwise the result is returned in a newly allocated object. It is associated with the `Chronograph` parameter.

⁷Section [10.3.1.2](#)

⁸Section [10.3.1.2](#)

relative(Chronograph)

Signature

```
public javafx.runtime.RelativeTime  
relative(Chronograph chronograph)
```

Description

Convert the time of this to a relative time, using the given instance of [Chronograph](#)⁹ to determine the current time. The calculation is the current time indicated by the given instance of [Chronograph](#)¹⁰ 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 chronograph association of the result is with the Chronograph passed as a parameter.

Parameters

chronograph The instance of [Chronograph](#)¹¹ 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 javafx.runtime.RelativeTime  
relative(Chronograph chronograph,  
         RelativeTime dest)
```

Description

Convert the time of this to a relative time, using the given instance of [Chronograph](#)¹² to determine the current time. The calculation is the current time indicated by the given instance of [Chronograph](#)¹³ subtracted from the time given

⁹Section [10.3.1.2](#)

¹⁰Section [10.3.1.2](#)

¹¹Section [10.3.1.2](#)

¹²Section [10.3.1.2](#)

¹³Section [10.3.1.2](#)

by this. When Chronograph is null the default realtime clock is assumed. When dest is not null, the result is placed there and returned. Otherwise, a new object is allocated for the result. The chronograph association of the result is with the Chronograph passed as a parameter.

Parameters

chronograph The instance of [Chronograph](#)¹⁴ used to convert the time of this into relative time, and the new chronograph association for the result.

dest When dest is not null, the result is placed there and returned. Otherwise, a new object is allocated for the result.

Throws

ArithmeticException when the result does not fit in the normalized format.

Returns

The RelativeTime conversion in dest when dest is not null, otherwise the result is returned in a newly allocated object. It is associated with the Chronograph parameter.

add(long, int)

Signature

```
public javax.realtime.AbsoluteTime  
add(long millis,  
    int nanos)  
throws ArithmeticException
```

Description

Create a new object representing the result of adding millis and nanos to the values from this and normalizing the result. The result will have the same 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.

¹⁴Section [10.3.1.2](#)

add(long, int, AbsoluteTime)

Signature

```
public javafx.runtime.AbsoluteTime  
add(long millis,  
    int nanos,  
    AbsoluteTime dest)  
throws ArithmeticException
```

Description

Return an object containing the value resulting from adding `millis` and `nanos` to the values from this and normalizing the result. When `dest` is not null, the result is placed there and returned. Otherwise, a new object is allocated for the result. The result will have the same 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. Otherwise, a new object is allocated for the result.

Throws

ArithmeticException when the result does not fit in the normalized format.

Returns

the result of the normalization of this plus `millis` and `nanos` in `dest` when `dest` is not null, otherwise the result is returned in a newly allocated object.

add(RelativeTime)

Signature

```
public javafx.runtime.AbsoluteTime  
add(RelativeTime time)  
throws ArithmeticException,  
    IllegalArgumentException
```

Description

Create 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

IllegalArgumentException 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*

```
public javax.realtime.AbsoluteTime  
add(RelativeTime time,  
    AbsoluteTime dest)  
throws ArithmeticException,  
        IllegalArgumentException
```

Description

Return an object containing the value resulting from adding time to the value of this and normalizing the result. When dest is not null, the result is placed there and returned. Otherwise, a new object is allocated for the result. The 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 there and returned. Otherwise, a new object is allocated for the result.

Throws

IllegalArgumentException 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.

getDate

Signature

```
public java.util.Date  
    getDate()  
    throws UnsupportedOperationException
```

Description

Convert the time given by this to a Date format. Note that Date represents time as milliseconds so the nanoseconds of this will be lost.

Throws

UnsupportedOperationException 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

```
public javax.realtime.AbsoluteTime  
    set(Date date)  
    throws IllegalArgumentException
```

Description

Change the time represented by this to that given by the parameter. Note that Date represents time as milliseconds so the nanoseconds of this will be set to 0. The 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

IllegalArgumentException when the parameter date is null.

Returns

this

subtract(AbsoluteTime)*Signature*

```
public javax.realtime.RelativeTime  
subtract(AbsoluteTime time)  
throws IllegalArgumentException,  
        ArithmeticException
```

Description

Create 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

`IllegalArgumentException` 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*

```
public javax.realtime.RelativeTime  
subtract(AbsoluteTime time,  
        RelativeTime dest)  
throws IllegalArgumentException,  
        ArithmeticException
```

Description

Return an object containing the value resulting from subtracting time from the value of this and normalizing the result. When dest is not null, the result is placed there and returned. Otherwise, a new object is allocated for the result. The 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. Otherwise, a new object is allocated for the result.

Throws

IllegalArgumentException 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 IllegalArgumentException,  
           ArithmeticException
```

Description

Create 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

IllegalArgumentException 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)  
throws IllegalArgumentException,  
        ArithmeticException
```

Description

Return an object containing the value resulting from subtracting time from the value of this and normalizing the result. When dest is not null, the result is placed there and returned. Otherwise, a new object is allocated for the result. The 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. Otherwise, a new object is allocated for the result.

Throws

IllegalArgumentException 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


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

Description

Create a printable string of the time given by this.

The string shall be a decimal representation of the milliseconds and nanosecond values; formatted as follows "(2251 ms, 750000 ns)"

Returns

String object converted from the time given by this.

9.3.1.2 HighResolutionTime

Inheritance

java.lang.Object
 [javafx.realtime.HighResolutionTime](#)

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](#)¹⁵ or [RelativeTime](#)¹⁶ 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

¹⁵Section [9.3.1.1](#)

¹⁶Section [9.3.1.3](#)

waitForObject(Object, HighResolutionTime)*Signature*

```

public static boolean
waitForObject(Object target,
               javax.realtime.HighResolutionTime<?> time)
throws InterruptedException,
    IllegalMonitorStateException,
    IllegalArgumentException,
    UnsupportedOperationException

```

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 occurred. 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 on which to wait. The current thread must have a lock on the object.

time The time for which to wait. When it is `RelativeTime(0,0)` then wait indefinitely. When it is null then wait indefinitely.

Throws

`InterruptedException` when this schedulable is interrupted by `RealtimeThread.interrupt`¹⁷ or `AsynchronouslyInterruptedException.fire`¹⁸ while it is waiting.

`IllegalArgumentException` when time represents a relative time less than zero.

`IllegalMonitorStateException` when target is not locked by the caller.

`UnsupportedOperationException` when the wait operation is not supported using the clock associated with time.

Returns

true when the notify was received before the timeout; false otherwise.

¹⁷Section 5.3.2.2.2

¹⁸Section 8.3.2.1.2

Available since RTSJ 2.0 updated to add a return value.

equals(T)

Signature

```
public boolean  
equals(T time)
```

Description

Returns true when the argument time has the same type and values as this.
Equality includes Chronograph association.

Parameters

time Value compared to this.

Returns

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

getClock

Signature

```
public final javafx.realtime.Clock  
getClock()  
throws UnsupportedOperationException
```

Description

Returns a reference to the clock associated with this.

Throws

UnsupportedOperationException when the time is based on a [Chronograph](#)¹⁹ that is not a [Clock](#)²⁰.

Returns

A reference to the clock associated with this.

Available since RTSJ 1.0.1

¹⁹Section [10.3.1.2](#)

²⁰Section [10.3.2.1](#)

getChronograph

Signature

```
public final javax.realtime.Chronograph  
getChronograph()
```

Description

Get a reference to the [Chronograph²¹](#) associated with this.

Returns

A reference to the [Chronograph²²](#) associated with this.

Available since RTSJ 2.0

getMilliseconds

Signature

```
public final long  
getMilliseconds()
```

Description

Get the milliseconds component of this.

Returns

The milliseconds component of the time represented by this.

getNanoseconds

Signature

```
public final int  
getNanoseconds()
```

Description

Get the nanoseconds component of this.

Returns

The nanoseconds component of the time represented by this.

²¹Section [10.3.1.2](#)

²²Section [10.3.1.2](#)

set(T)*Signature*

```
public T extends javax.realtime.HighResolutionTime<T>  
    set(T time)
```

Description

Change 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

IllegalArgumentException 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.

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

millis This value shall be the value of the millisecond component of this at the completion of the call.

Returns

this

set(long, int)*Signature*

```
public T extends javax.realtime.HighResolutionTime<T>  
    set(long millis,  
        int nanos)  
    throws IllegalArgumentException
```

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

`IllegalArgumentException` when there is an overflow in the millisecond component while normalizing.

Returns

this

hashCode*Signature*

```
public int  
    hashCode()
```

Description

Returns a hash code for this object in accordance with the general contract of `Object.hashCode`. Time objects that are `equals`²³ equal have the same hash code.

Returns

The hashcode value for this instance.

²³Section 9.3.1.2.1

clone

Signature

```
public java.lang.Object  
clone()
```

Description

Return a clone of this. This method should behave effectively as when it constructed a new object with the visible values of this. The new object is created in the current allocation context.

Available since RTSJ 1.0.1

compareTo(T)

Signature

```
public int  
compareTo(T time)
```

Description

Compares this HighResolutionTime with the specified HighResolutionTime time.

Parameters

time Compares with the time of this.

Throws

ClassCastException when the time parameter is not of the same class as this.

IllegalArgumentException when the time parameter is not associated with the same 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.

equals(Object)

Signature

```
public boolean  
equals(Object object)
```

Description

Returns true when the argument object has the same type and values as this.
Equality includes Chronograph association.

Parameters

object Value compared to this.

Returns

true when the parameter object is of the same type and has the same values as this.

absolute(Chronograph, AbsoluteTime)

Signature

```
public abstract javax.realtime.AbsoluteTime
absolute(Chronograph chronograph,
         AbsoluteTime dest)
```

Description

Convert the time of this to an absolute time, using the given instance of [Chronograph](#)²⁴ to determine the current time when necessary. When Chronograph is null the realtime chronograph is assumed. When dest is not null, the result is placed there and returned. Otherwise, a new object is allocated for the result. The chronograph association of the result is the Chronograph passed as a parameter. See the subclass comments for more specific information.

Parameters

chronograph The instance of [Chronograph](#)²⁵ used to convert the time of this into absolute time, and the new chronograph association for the result.

dest When dest is not null, the result is placed there and returned. Otherwise, a new object is allocated for the result.

Returns

The AbsoluteTime conversion in dest when dest is not null, otherwise the result is returned in a newly allocated object. It is associated with the Chronograph parameter.

absolute(Chronograph)

Signature

²⁴Section [10.3.1.2](#)

²⁵Section [10.3.1.2](#)


```
public abstract javax.realtime.AbsoluteTime  
absolute(Chronograph chronograph)
```

Description

Convert the time of this to an absolute time, using the given instance of [Chronograph](#)²⁶ 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 is the instance of [Chronograph](#)²⁷ 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

Convert the time of this to a relative time, using the given instance of [Chronograph](#)²⁸ to determine the current time when necessary. When Chronograph is null the realtime chronograph is assumed. When dest is not null, the result is placed there and returned. Otherwise, a new object is allocated for the result. The chronograph association of the result is the Chronograph passed as a parameter. See the subclass comments for more specific information.

Parameters

chronograph The instance of [Chronograph](#)²⁹ used to convert the time of this into relative time, and the new chronograph association for the result.

²⁶Section [10.3.1.2](#)

²⁷Section [10.3.1.2](#)

²⁸Section [10.3.1.2](#)

²⁹Section [10.3.1.2](#)

dest When dest is not null, the result is placed there and returned. Otherwise, a new object is allocated for the result.

Returns

The [RelativeTime](#)³⁰ conversion in dest when dest is not null, otherwise the result is returned in a newly allocated object.

relative(Chronograph)

Signature

```
public abstract javax.realtime.RelativeTime
relative(Chronograph chronograph)
```

Description

Convert the time of this to a relative time, using the given instance of [Chronograph](#)³¹ to determine the current time when necessary. When Chronograph is null the realtime chronograph 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](#)³² 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.

9.3.1.3 RelativeTime

Inheritance

```
java.lang.Object
  javax.realtime.HighResolutionTime
    javax.realtime.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.

³⁰Section [9.3.1.3](#)

³¹Section [10.3.1.2](#)

³²Section [10.3.1.2](#)

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

```
public
RelativeTime(long millis,
              int nanos,
              Chronograph chronograph)
throws IllegalArgumentException
```

Description

Construct a RelativeTime object representing an interval based on the parameter millis plus the parameter nanos. The construction is subject to millis and nanos parameters normalization. When there is an overflow in the millisecond component when normalizing then an IllegalArgumentException will be thrown.

The chronograph association is made with the chronograph parameter. When chronograph is null the association is made with the default realtime clock.

Available since RTSJ 2.0

Parameters

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

nanos The desired value for the nanosecond component of this. The actual value is the result of parameter normalization.

chronograph The chronograph providing the association for the newly constructed object. Defaults to the realtime clock when null.

Throws

IllegalArgumentException when there is an overflow in the millisecond component when normalizing.

RelativeTime(long, int)

Signature

```
public  
RelativeTime(long millis,  
              int nanos)  
throws IllegalArgumentException
```

Description

Equivalent to `RelativeTime(long, int, Chronograph)`³³ 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.

nanos The desired value for the nanosecond component of this. The actual value is the result of parameter normalization.

Throws

IllegalArgumentException when there is an overflow in the millisecond component when normalizing.

RelativeTime(RelativeTime)

Signature

```
public  
RelativeTime(RelativeTime time)
```

Description

Equivalent to `RelativeTime(long, int, Chronograph)`³⁴ with argument list (time.getMilliseconds(), time.getNanoseconds(), time.getChronograph()).

Parameters

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

³³Section 9.3.1.3.1

³⁴Section 9.3.1.3.1

RelativeTime(Chronograph)

Signature

```
public  
RelativeTime(Chronograph chronograph)
```

Description

Equivalent to `RelativeTime(long, int, Chronograph)`³⁵ with argument list (0, 0, chronograph).

Available since RTSJ 2.0

Parameters

chronograph The chronograph providing the association for the newly constructed object.

RelativeTime

Signature

```
public  
RelativeTime()
```

Description

Equivalent to `RelativeTime(long, int, Chronograph)`³⁶ with argument list (0, 0, null).

9.3.1.3.2 Methods

absolute(Chronograph)

Signature

```
public javax.realtime.AbsoluteTime  
absolute(Chronograph chronograph)
```

³⁵Section 9.3.1.3.1

³⁶Section 9.3.1.3.1

Description

Convert the time of this to an absolute time, using the given instance of [Chronograph](#)³⁷ 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

Convert the time of this to an absolute time, using the given instance of [Chronograph](#)³⁸ to determine the current time when necessary. When Chronograph is null the realtime chronograph is assumed. When dest is not null, the result is placed there and returned. Otherwise, a new object is allocated for the result. The chronograph association of the result is the Chronograph passed as a parameter. See the subclass comments for more specific information.

Available since RTSJ 2.0

See [Section HighResolutionTime.absolute\(Chronograph, AbsoluteTime\)](#)

relative(Chronograph)*Signature*

```
public javax.realtime.RelativeTime  
relative(Chronograph chronograph)
```

³⁷Section [10.3.1.2](#)

³⁸Section [10.3.1.2](#)

Description

Convert the time of this to a relative time, using the given instance of [Chronograph](#)³⁹ to determine the current time when necessary. When Chronograph is null the realtime chronograph 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

```
public javax.realtime.RelativeTime  
relative(Chronograph chronograph,  
         RelativeTime dest)
```

Description

Convert the time of this to a relative time, using the given instance of [Chronograph](#)⁴⁰ to determine the current time when necessary. When Chronograph is null the realtime chronograph is assumed. When dest is not null, the result is placed there and returned. Otherwise, a new object is allocated for the result. The chronograph association of the result is the Chronograph passed as a parameter. See the subclass comments for more specific information.

Available since RTSJ 2.0

See [Section HighResolutionTime.relative\(Chronograph, RelativeTime\)](#)

add(long, int)

Signature

```
public javax.realtime.RelativeTime  
add(long millis,  
     int nanos)
```

³⁹Section [10.3.1.2](#)

⁴⁰Section [10.3.1.2](#)

throws `ArithmeticException`

Description

Create a new object representing the result of adding `millis` and `nanos` to the values from this and normalizing the result. The result will have the same 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

Return an object containing the value resulting from adding `millis` and `nanos` to the values from this and normalizing the result. When `dest` is not null, the result is placed there and returned. Otherwise, a new object is allocated for the result. The result will have the same 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. Otherwise, a new object is allocated for the result.

Throws

`ArithmeticException` when the result does not fit in the normalized format.

Returns

the result of the normalization of this plus millis and nanos in dest when dest is not null, otherwise the result is returned in a newly allocated object.

add(RelativeTime)*Signature*

```
public javafx.realtime.RelativeTime  
add(RelativeTime time)  
throws IllegalArgumentException,  
    ArithmeticException
```

Description

Create 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

IllegalArgumentException 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 plus the parameter time.

add(RelativeTime, RelativeTime)*Signature*

```
public javafx.realtime.RelativeTime  
add(RelativeTime time,  
    RelativeTime dest)  
throws IllegalArgumentException,  
    ArithmeticException
```

Description

Return an object containing the value resulting from adding time to the value of this and normalizing the result. When dest is not null, the result is placed there and returned. Otherwise, a new object is allocated for the result.

The 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. Otherwise, a new object is allocated for the result.

Throws

IllegalArgumentException 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.

subtract(RelativeTime)*Signature*

```
public javax.realtime.RelativeTime  
subtract(RelativeTime time)  
throws IllegalArgumentException,  
        ArithmeticException
```

Description

Create 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

IllegalArgumentException 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 parameter time.

subtract(RelativeTime, RelativeTime)*Signature*

```
public javax.realtime.RelativeTime  
subtract(RelativeTime time,  
         RelativeTime dest)  
throws IllegalArgumentException,  
         ArithmeticException
```

Description

Return an object containing the value resulting from subtracting the value of time from the value of this and normalizing the result. When dest is not null, the result is placed there and returned. Otherwise, a new object is allocated for the result.

The 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

IllegalArgumentException when the 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

```
public javax.realtime.RelativeTime  
scale(int factor)
```

Description

Change the length of this relative time by multiplying it by factor.

Parameters

factor 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

```
public javax.realtime.RelativeTime  
scale(int factor,  
      RelativeTime time)
```

Description

Set time to the value of this time by multiplied by factor.

Parameters

factor by which to increase the time interval.

time in which to store the results.

Returns

time with the value of this scaled by factor

Available since RTSJ 2.0

compareToZero

Signature

```
public int  
compareToZero()
```

Description

Compare 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

```
public int  
compareTo(RelativeTime time)
```

Description

Compares this `HighResolutionTime` with the specified `HighResolutionTime` time.

Parameters

time Compares with the time of this.

Throws

`ClassCastException` when the time parameter is not of the same class as this.

`IllegalArgumentException` when the time parameter is not associated with the same 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.

toString

Signature

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

Description

Create a printable string of the time given by this.

The string shall be a decimal representation of the milliseconds and nanosecond values; formatted as follows "(2251 ms, 750000 ns)"

Returns

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.

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`.

10.1 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 visa verse.

Monotonic Timing Mechanism — A timing mechanism whose time values always progress in one direction.

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.

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.
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 realtime clock Epoch. However, the results of any such comparison must be treated with caution as the accuracy of the two Chronograph objects may be different.

10.2.2 Clocks and Timables

A Clock is the basic mechanism of measuring time and triggering events based on the passage of time. A Timer can request a signal from the clock when a given time is reached. That signal should come as closed to the actual time requested as possible. A schedulable also uses a clock to implement the realtime sleep methods. Each clock instance shall be capable of reporting the achievable resolution of timers based on that clock. Each implementation shall have a default clock that is used whenever no other clock is specified. An application can also defined additional clocks.

A Timer 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 that is performed by a TimeDispatcher that informs the Timer that it is time to *fire* or *skip*, where skip causes the normal action of fire not to be carried out.

A Timer is *active* when it has been started and not stopped since last started and it has a time in the future at which it is expected to fire or skip, else it is *not active*.

In the context of a Timer, *enabling* cause the Timer to fire when it is triggered, while *disabling* causes the Timer to skip when it is triggered. Enabling and disabling act as a mask over firing.

The behavior of a OneShotTimer is that of a Timer that does not automatically reschedule its triggering after an initial triggering, regardless of whether it fires or skips (when *disabled* and *active* when triggered). It is specified using an initial firing time.

The behavior of a PeriodicTimer is that of a Timer that automatically reschedules after each triggering, regardless of whether the triggering results in a fire or a skip due to being disabled when triggered. It is specified using an initial firing time and an interval or period used for the self-rescheduling.

A Clock can also be used to regulate pauses in execution of any Schedulable through a realtime sleep method, hence timers and schedulables are classified as timables under the [Timable](#) interface.

Both OneShotTimer and PeriodTimer are given an initial firing time. A PeriodicTimer receives two clock references, within two HighResolutionTimer objects, which must be to the same clock. Thus the specification of the initial firing time and the interval or period must refer to the same clock.

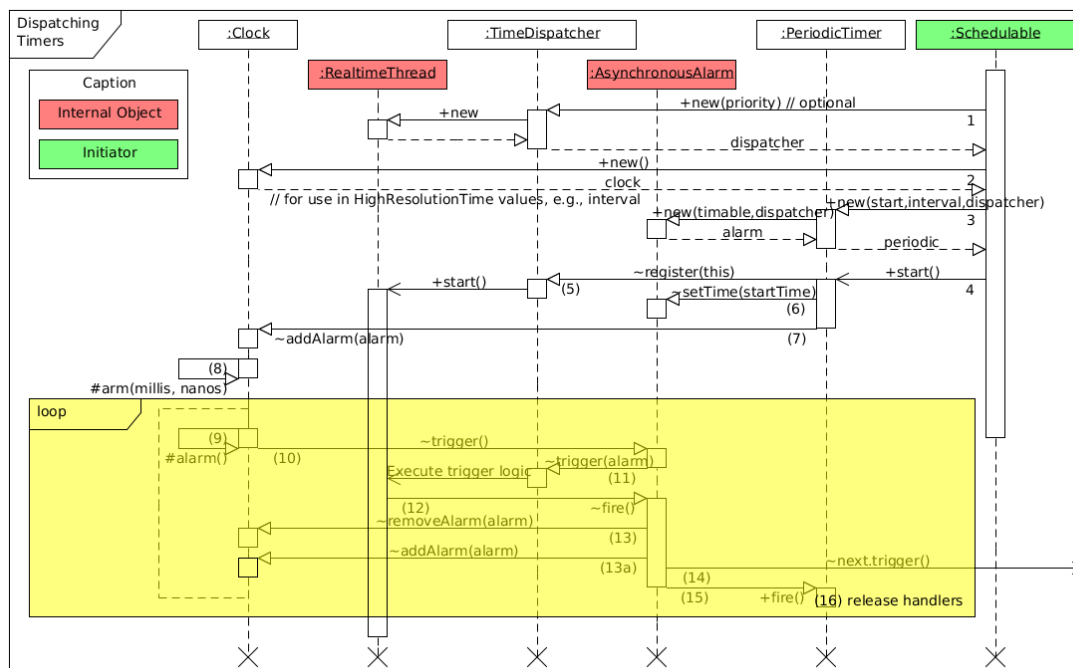
A Timer is an ActiveEvent. This means that is has an associated dispatcher called TimeDispatcher. As with other active events, the application can either use

the default dispatcher or create a new one with its own priority and affinity. A schedulable can also have a TimeDispatcher to manage sleeping.

At any given time, a timable, Timer 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.

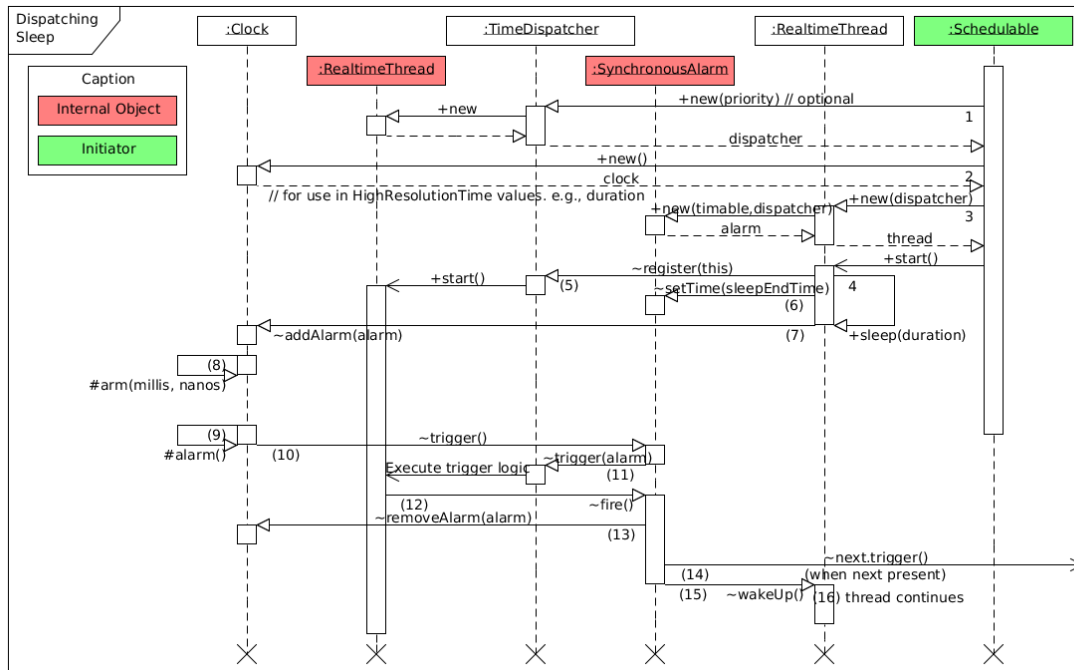
In the case of a timer, the dispatcher triggers the timer thereby indicating it should fire or skip. In the case of a schedulable, the dispatcher triggers the schedulable to wake up from its sleep. Figure 10.1 illustrates how a timer interacts with a application-defined clock and Figure 10.2 depicts 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.

Figure 10.2: Sequence Diagram for Realtime Sleep



Each timable acts as if it had an internal object, depicted as an instance of Alarm, to manage the relationship between a timable and its dispatcher and clock. Alarm is shown simply to illustrate this relationship. It is created, step three in both diagrams, when the timable is created and it represents the next alarm that the timable should receive: a fire for a time or a wake up call for a realtime sleep on a schedulable.

At step four, the two sequences diverge. The application start a timer with the start method, but a thread must call a realtime sleep method. In both cases, step four sets the timing in motion.

Steps (5) through (8) set up the time interval. For initiating the trigger for the first time, step (5) registers the timable with its dispatcher. Later starts or sleeps skip this step. Then the time is set in the alarm and the alarm is added to the clock.

When the new alarm is the next alarm to be triggered, the clock arranges to signal that time as in step (8). When the alarm is added anywhere else in the clock queue, step (8) is delayed until the removal of an alarm causes the added alarm to reach the top of the queue.

When the alarm time is reached, step (9), the clock triggers the alarm by calling trigger on the alarm event, step (10). This in turn triggers the dispatcher, step (11). This is an asynchronous call that causes the dispatcher's thread to take over control

from the clocks interrupt handler.

In step (12), the dispatcher thread removes the alarm from the clock queue, possibly causing a new alarm to become active. In the periodic thread case, the alarm is rescheduled by incrementing the time in the alarm by the interval and adding it back into the queue. In all other cases, no new alarm is set.

In step (13) any subsequent alarms that were scheduled are also kicked off. The Clock queue is a two dimensional queue that is organized by the time of the alarm and, within any given time, the priority order, highest to lowest, of the dispatchers associated with the alarms. The trigger in step (10) always goes to the alarm with the highest priority dispatcher.

Finally in step (14), the dispatcher fires the alarm which results its timable being fired or woken-up. In the case of a timer, this causes all its handlers to be released or, in the case of a schedulable, a sleep being woken up; this is marked as (15) in the diagrams.

Clocks and TimeDispatchers may be shared among many as timables as the needs of the application dictate. Different dispatchers can be used with a given clock and a dispatcher can service different clocks. The dispatcher should be chosen based on its priority and affinity, whereas a clock should be chosen based on the temporal reference, where the temporal reference may or may not be associated with clock time. For instance, one could use a clock to represent the rotation of a shaft.

10.2.3 Timers

A timer must be associated with a clock. That clock acts as if it provides an interrupt to each of its timers at the next instance of time at which the timer should do something. In other words, a clock fires its timer at a requested time. Timers can be modeled as counters, or as comparators.

10.2.3.1 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 $n = 0, 1, 2, \dots$ where S is as just specified, and T is the interval of the periodic timer.

For all timers, when the start or reschedule time is given as a relative time, R , S equals the time at which the *counting time*, started when the timer was made *active*, equals R . The transition to *not-active* by this timer causes the *counting time* to reset, effectively preventing this kind of timer from firing immediately, unless given a time value of 0.

When in a *not-active* state a Timer retains the parameters given at construction time or the parameters it had at de-activation time. Those are the parameters that will be used upon invocation of start while in that state, unless the parameters are explicitly changed before that, using `reschedule` and `setInterval` as appropriate.

When a Timer object is allocated in a scoped memory area, then it will increment the reference count associated with that area. Such a reference count will only be decremented when the Timer object is destroyed. (See semantics in the *Memory* chapter for details.) A Timer object will not fire before its due time.

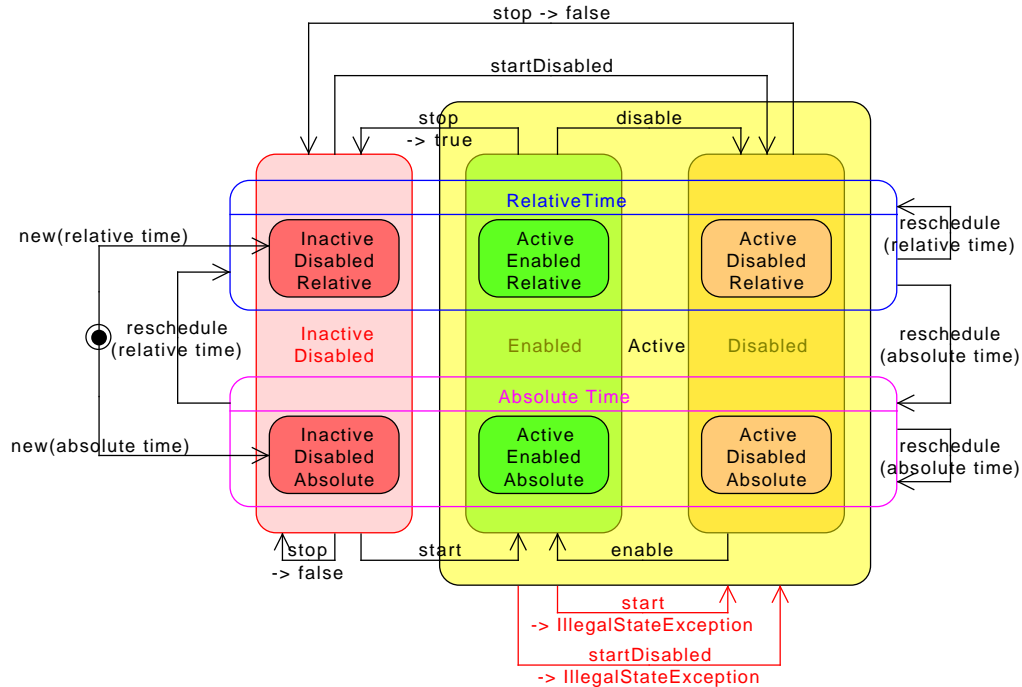
The states of a Timer are essentially the same as for an `ActiveEvent` as depicted in Figure 8.3. The main difference is that the time used for the next fire may be either an absolute time or a relative time. Figure 10.3 reflects this difference in a UML state diagram.

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 is thrown when phasing is `STRICT_PHASING`.

The default phasing is `ADJUST_TO_START`.

Figure 10.3: States of a Timer¹

10.3 javax.realtime

10.3.1 Interfaces

10.3.1.1 AsyncTimable

Interfaces

`javax.realtime.Timable`

Description

A common type for `Timer`² and `RealtimeThread`³ to indicate that they can be

¹Note that the semantics of the fire transition differ among the subclasses of `Timer`.

²Section 10.3.2.6

³Section 5.3.2.2

associated with a [Clock](#)⁴ and be suspended waiting for time events based 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

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

⁴Section [10.3.2.1](#)

getEpochOffset

Signature

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

Description

Determines the time on the realtime clock when this chronograph was zero.

Throws

UnsupportedOperationException when the chronograph does not have the concept of date.

Returns

A newly allocated [RelativeTime](#)⁵ object in the current execution context with the realtime clock as its chronograph and containing time when this chronograph was zero.

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](#)⁶ in the current allocation context, representing the current time. The returned object has this its chronograph.

getTime(AbsoluteTime)

Signature

⁵Section [9.3.1.3](#)

⁶Section [9.3.1.1](#)

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

Description

Obtain the current time. The time represented by the given [AbsoluteTime](#)⁷ 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 chronograph'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](#)⁸ object which will be updated in place.

Returns

The instance of [AbsoluteTime](#)⁹ 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

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

Description

Obtain the precision with which time can be read, i.e., the nominal interval between ticks. It is the same as calling [getQueryPrecision\(RelativeTime\)](#)¹⁰ with null as an argument.

Returns

a newly allocated time value holding the read precision.

getQueryPrecision(RelativeTime)

Signature

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

⁷Section [9.3.1.1](#)

⁸Section [9.3.1.1](#)

⁹Section [9.3.1.1](#)

¹⁰Section [10.3.1.2.1](#)

Description

Obtain the precision with which time can be read, i.e., the nominal interval between ticks.

Parameters

`dest` is a 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

Interfaces

[javax.realtime.Releasable](#)

Description

A type for all classes that can use a [Clock](#)¹¹ for timing, either for sleeping or for being released at a given time.

10.3.1.3.1 Methods

getDispatcher

Signature

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

Description

Get the dispatcher associated with this Timable.

Available since RTSJ 2.0

¹¹Section [10.3.2.1](#)

10.3.2 Classes

10.3.2.1 Clock

Inheritance

java.lang.Object
 javax.realtime.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.

10.3.2.1.2 Methods

getRealtimeClock

Signature

```
public static javax.realtime.Clock  
getRealtimeClock()
```

Description

There is always at least one clock object available: the system realtime clock. This is the default Clock.

Returns

The singleton instance of the default Clock

getEpochOffset

Signature

```
public final javax.realtime.RelativeTime  
getEpochOffset()  
throws UnsupportedOperationException
```

Description

Determines the time on the realtime clock when this chronograph was zero.

Throws

UnsupportedOperationException UnsupportedOperationException when the chronograph does not have the concept of date.

Available since RTSJ 1.0.1

getTime

Signature

```
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](#)¹² in the current allocation context, representing the current time. The returned object has this its chronograph.

getTime(AbsoluteTime)*Signature*

```
public abstract javax.realtime.AbsoluteTime  
getTime(AbsoluteTime dest)
```

Description

Obtain the current time. The time represented by the given [AbsoluteTime](#)¹³ 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 dest The instance of [AbsoluteTime](#)¹⁴ object which will be updated in place.

Returns

The instance of [AbsoluteTime](#)¹⁵ 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*

```
public abstract javax.realtime.RelativeTime  
getQueryPrecision()
```

Description

¹²Section [9.3.1.1](#)

¹³Section [9.3.1.1](#)

¹⁴Section [9.3.1.1](#)

¹⁵Section [9.3.1.1](#)

Obtain the precision with which time can be read, i.e., the nominal interval between ticks. It is the same as calling `getQueryPrecision(RelativeTime)`¹⁶ with null as an argument.

Returns

a newly allocated time value holding the read precision.

Available since RTSJ 2.0

getQueryPrecision(RelativeTime)

Signature

```
public abstract javax.realtime.RelativeTime  
getQueryPrecision(RelativeTime dest)
```

Description

Obtain the precision with which time can be read, i.e., the nominal interval between ticks.

Parameters

`dest` `dest` is a 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)`¹⁷ with null as its argument.

¹⁶Section 10.3.2.1.2

¹⁷Section 10.3.2.1.2

Returns

a value representing the drive precision.

Available since RTSJ 2.0

getDrivePrecision(RelativeTime)

Signature

```
public abstract javafx.runtime.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\(RelativeTime\)](#)¹⁸.

Parameters

dest return the relative time value in dest. When dest is null, it allocates a new [RelativeTime](#)¹⁹ instance to hold the returned value.

Returns

dest set to values representing the drive precision.

Available since RTSJ 2.0

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](#)²⁰, which in turn will cause a fire on an associated [AsyncTimable](#)²¹

¹⁸Section [10.3.2.1.2](#)

¹⁹Section [9.3.1.3](#)

²⁰Section [10.3.2.4](#)

²¹Section [10.3.1.1](#)

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 of the next alarm.
nanoseconds of the next alarm.

Available since RTSJ 2.0

clearAlarm

Signature

```
protected abstract void  
clearAlarm()
```

Description

Implemented by subclasses to cancel the current outstanding alarm.

Available since RTSJ 2.0

10.3.2.2 OneShotTimer

Inheritance

```

java.lang.Object
  javafx.runtime.AsyncBaseEvent
    javafx.runtime.AsyncEvent
      javafx.runtime.Timer
        javafx.runtime.OneShotTimer

```

Description

A timed [AsyncEvent](#)²² that is driven by a [Clock](#)²³. 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](#)²⁴ 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*

```

public
OneShotTimer(javafx.runtime.HighResolutionTime<?> time,
              TimeDispatcher dispatcher)
throws IllegalArgumentException,
       UnsupportedOperationException,
       IllegalAssignmentError

```

Description

Create an instance of [OneShotTimer](#)²⁵, based on the given clock, that will execute its fire method according to the given time. The [Clock](#)²⁶ association of the parameter time is ignored.

²²Section [8.3.3.4](#)

²³Section [10.3.2.1](#)

²⁴Section [10.3.2.6](#)

²⁵Section [10.3.2.2](#)

²⁶Section [10.3.2.1](#)

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

`IllegalArgumentException` when `time` is a `RelativeTime` instance less than zero.

`UnsupportedOperationException` when the [Chronograph](#)²⁷ associated with `time` is not a `Clock`²⁸.

`IllegalAssignmentError` when this `OneShotTimer` cannot hold references to `time`, `handler`, or `clock`.

OneShotTimer(HighResolutionTime, AsyncEventHandler)

Signature

```
public
    OneShotTimer(javax.realtime.HighResolutionTime<?> time,
                  AsyncEventHandler handler)
```

Description

The equivalent of calling [OneShotTimer\(HighResolutionTime, TimeDispatcher\)](#)²⁹ with arguments `time`, null followed by a call to `setHandler(handler)`.

Parameters

`time` is the time to release its handlers.

`handler` is the hanndler to release.

10.3.2.2.2 Methods

²⁷Section [10.3.1.2](#)

²⁸Section [10.3.2.1](#)

²⁹Section [10.3.2.2.1](#)

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 calls `Timer.stop()`³⁰. 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

Inheritance

```
java.lang.Object  
  javafx.realtime.AsyncBaseEvent  
    javafx.realtime.AsyncEvent  
      javafx.realtime.Timer  
        javafx.realtime.PeriodicTimer
```

Description

An `AsyncEvent`³¹ whose fire method is executed periodically according to the given parameters. The clock associated with the `Timer` start time must be identical to the the clock associated with the `Timer` interval

The first firing is at the beginning of the first interval.

When an interval greater than 0 is given, the timer will fire periodically. When an interval of 0 is given, the `PeriodicTimer` will only fire once, unless restarted after expiration, behaving like a `OneShotTimer`. In all cases, when the timer is *disabled* when the firing time is reached, that particular firing is lost (*skipped*). When *enabled* at a later time, it will fire at its next scheduled time.

When the clock time has already passed the beginning of the first period, the `PeriodicTimer` will first fire according to the `PhasingPolicy`³².

³⁰Section 10.3.2.6.2

³¹Section 8.3.3.4

³²Section 5.3.1.1

Semantics details are described in the [Timer](#)³³ 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(javax.realtime.HighResolutionTime<?> start,
               RelativeTime interval,
               TimeDispatcher dispatcher)
throws IllegalArgumentException,
       IllegalAssignmentError,
       UnsupportedOperationException
```

Description

Create 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 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](#)³⁴.

dispatcher is the dispatcher to use for triggering this event.

Throws

³³Section [10.3.2.6](#)

³⁴Section [10.3.2.2](#)

`IllegalArgumentException` when start or interval is a `RelativeTime` instance with a value less than zero; or the clocks associated with start and interval are not the identical.

`IllegalAssignmentError` when this `PeriodicTimer` cannot hold references to handler, clock and interval.

`UnsupportedOperationException` when the `Chronograph`³⁵ associated with time is not a `Clock`³⁶.

PeriodicTimer(HighResolutionTime, RelativeTime, AsyncEventHandler)

Signature

```
public
PeriodicTimer(javafx.realtime.HighResolutionTime<?> start,
               RelativeTime interval,
               AsyncEventHandler handler)
throws IllegalArgumentException,
    IllegalAssignmentError
```

Description

Create 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(javafx.realtime.AsyncBaseEventHandler<?> handler)
throws IllegalArgumentException,
    IllegalAssignmentError
```

³⁵Section 10.3.1.2

³⁶Section 10.3.2.1

Description

Add a handler to the set of handlers associated with this event. It overrides the method in [AsyncBaseEvent](#)³⁷ to allow the use of handlers with [PeriodicParameters](#)³⁸, but these parameters must match the period of this timer, otherwise `IllegalArgumentException` 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

`IllegalArgumentException` when handler is null or the handler has [PeriodicParameters](#)³⁹ 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*

```
public void
setHandler(javax.realtime.AsyncBaseEventHandler<?> handler)
throws IllegalArgumentException,
       IllegalAssignmentError
```

Description

Associate a new handler with this event and remove all existing handlers. It overrides the method in [AsyncBaseEvent](#)⁴⁰ to allow the use of handlers with [PeriodicParameters](#)⁴¹, but these parameters must match the period of this timer, otherwise `IllegalArgumentException` is thrown.

Parameters

handler The instance of [AsyncBaseEventHandler](#)⁴² to be associated with this. When handler is null, no handler will be associated with this, i.e., behave effectively as when `setHandler(null)` invokes `removeHandler(AsyncBaseEventHandler)` for each associated handler.

³⁷Section 8.3.3.2

³⁸Section 6.3.3.6

³⁹Section 6.3.3.6

⁴⁰Section 8.3.3.2

⁴¹Section 6.3.3.6

⁴²Section 8.3.3.3

Throws

`IllegalArgumentException` when handler has `PeriodicParameters`⁴³ 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

start(PhasingPolicy)

Signature

```
public void  
start(PhasingPolicy phasingPolicy)  
throws LateStartException,  
       IllegalArgumentException
```

Description

Start the timer with the specified `PhasingPolicy`⁴⁴.

Parameters

`phasingPolicy` determines what happens when the start is too late.

Throws

`LateStartException` when this method is called after its absolute start time and the `phasingPolicy` is `PhasingPolicy.STRICT_PHASING`⁴⁵.

`IllegalArgumentException` 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

```
public void  
start(boolean disabled,  
       PhasingPolicy phasingPolicy)
```

⁴³Section 6.3.3.6

⁴⁴Section 5.3.1.1

⁴⁵Section 5.3.1.1.1

throws `LateStartException`,
`IllegalArgumentException`

Description

Start the timer with the specified `PhasingPolicy`⁴⁶ and the specified disabled state.

Parameters

`disabled` determine the mode of start: `true` for enabled and `false` for disabled for consistency with `Timer.start(boolean)`⁴⁷.

`phasingPolicy` determines what happens when the start is too late.

Throws

`LateStartException` when this method is called after its absolute start time and the `phasingPolicy` is `PhasingPolicy.STRICT_PHASING`⁴⁸.

`IllegalArgumentException` when the start time of this timer is not an absolute time, or `phasingPolicy` is null.

Available since RTSJ 2.0

getClock

Signature

```
public javax.realtime.Clock  
getClock()  
throws IllegalStateException
```

Description

Each instance can only be associated with a single clock, which this method can obtain.

Throws

`IllegalStateException` when this has been destroyed.

Returns

the instance of `Clock`⁴⁹ that is associated with this.

Available since RTSJ 1.0.1

⁴⁶Section 5.3.1.1

⁴⁷Section 10.3.2.6.2

⁴⁸Section 5.3.1.1.1

⁴⁹Section 10.3.2.1

createReleaseParameters

Signature

```
public javafx.runtime.ReleaseParameters<?>  
createReleaseParameters()
```

Description

Create a release parameters object with new objects containing copies of the values corresponding to this timer. When the `PeriodicTimer` interval is greater than 0, create a [PeriodicParameters](#)⁵⁰ 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, create an [AperiodicParameters](#)⁵¹ object, since in this case the timer behaves like a [OneShotTimer](#)⁵².

When this timer is active, then the start time is the next firing (or skipping) time returned as an [AbsoluteTime](#)⁵³. Otherwise, the start time is the initial firing (or skipping) time, as set by the last call to [Timer.reschedule](#)⁵⁴, or when there was no such call, by the constructor of this timer.

Throws

`IllegalStateException` when this Timer has been *destroyed*.

Returns

A new release parameters object with new objects containing copies of the values corresponding to this timer. When the interval is greater than zero, return a new instance of [PeriodicParameters](#)⁵⁵. When the interval is zero return a new instance of [AperiodicParameters](#)⁵⁶.

getFireTime

Signature

```
public javafx.runtime.AbsoluteTime  
getFireTime()  
throws ArithmeticException,  
IllegalStateException
```

⁵⁰Section [6.3.3.6](#)

⁵¹Section [6.3.3.2](#)

⁵²Section [10.3.2.2](#)

⁵³Section [9.3.1.1](#)

⁵⁴Section [10.3.2.6.2](#)

⁵⁵Section [6.3.3.6](#)

⁵⁶Section [6.3.3.2](#)

Description

Get the time at which this `PeriodicTimer` is next expected to fire or to skip. When the `PeriodicTimer` is *disabled*, the returned time is that of the skipping of the firing. When the `PeriodicTimer` is *not-active* it throws `IllegalStateException`.

Throws

`ArithmeticException` when the result does not fit in the normalized format.
`IllegalStateException` when this Timer has been *destroyed*, or when it is *not-active*.

Returns

The absolute time at which this is next expected to fire or to skip, in a newly allocated `AbsoluteTime`⁵⁷ object. When the timer has been created or re-scheduled (see `Timer.reschedule(HighResolutionTime)`⁵⁸) using an instance of `RelativeTime` for its time parameter then it will return the sum of the current time and the `RelativeTime` remaining time before the timer is expected to fire/skip. Within a periodic timer activation, the returned time is associated with the start clock before the first fire (or skip) time, and associated with the interval clock otherwise.

getFireTime(AbsoluteTime)*Signature*

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

Description

Get the time at which this `PeriodicTimer` is next expected to fire or to skip. When the `PeriodicTimer` is *disabled*, the returned time is that of the skipping of the firing. When the `PeriodicTimer` is *not-active* it throws `IllegalStateException`.

Parameters

`dest` The instance of `AbsoluteTime`⁵⁹ which will be updated in place and returned. The clock association of the `dest` parameter is ignored. When `dest` is null a new object is allocated for the result.

Throws

`ArithmeticException` when the result does not fit in the normalized format.
`IllegalStateException` when this Timer has been *destroyed*, or when it is *not-active*.

⁵⁷Section 9.3.1.1

⁵⁸Section 10.3.2.6.2

⁵⁹Section 9.3.1.1

Returns

The instance of [AbsoluteTime](#)⁶⁰ passed as parameter, with time values representing the absolute time at which this is expected to fire or to skip. When the dest parameter is null the result is returned in a newly allocated object. When the timer has been created or re-scheduled (see [Timer.reschedule\(HighResolutionTime\)](#)⁶¹) using an instance of [RelativeTime](#) for its time parameter then it will return the sum of the current time and the [RelativeTime](#) remaining time before the timer is expected to fire/skip. Within a periodic timer activation, the returned time is associated with the start clock before the first fire (or skip) time, and associated with the interval clock otherwise.

Available since RTSJ 1.0.1

getInterval

Signature

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

Description

Gets the interval of this Timer.

Throws

[IllegalStateException](#) when this Timer has been *destroyed*.

Returns

The [RelativeTime](#) instance assigned as this periodic timer's interval by the constructor or [setInterval\(RelativeTime\)](#)⁶².

setInterval(RelativeTime)

Signature

```
public javax.realtime.PeriodicTimer  
setInterval(RelativeTime interval)
```

Description

Reset the interval value of this.

⁶⁰Section [9.3.1.1](#)

⁶¹Section [10.3.2.6.2](#)

⁶²Section [10.3.2.3.2](#)

Parameters

interval A [RelativeTime](#)⁶³ object which is the interval used to reset this Timer. A null interval is interpreted as `RelativeTime(0,0)`.

The interval does not affect the first firing (or skipping) of a timer's activation. At each firing (or skipping), the next fire (or skip) time of an *active* periodic timer is established based on the interval currently in use. Resetting the interval of an *active* periodic timer only effects future fire (or skip) times after the next.

Throws

`IllegalArgumentException` when interval is a `RelativeTime` instance with a value less than zero, or the clock associated with interval is different to the clock associated with this.

`IllegalAssignmentError` when this `PeriodicTimer` cannot hold a reference to interval.

`IllegalStateException` when this Timer has been *destroyed*.

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 distabled, but active, it simply rescheduled 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**Inheritance**

⁶³Section [9.3.1.3](#)

java.lang.Object
 javax.realtime.ActiveEventDispatcher
 javax.realtime.TimeDispatcher

Description

A dispatcher for time events: [Timer](#)⁶⁴ and [RealtimeThread.sleep](#)⁶⁵.

Available since RTSJ 2.0

10.3.2.4.1 Constructors

TimeDispatcher(SchedulingParameters, SchedulingGroup)

Signature

```
public  
TimeDispatcher(SchedulingParameters schedule,  
               SchedulingGroup group)
```

Description

Create a new dispatcher, whose dispatching thread runs with the given scheduling parameters.

Parameters

schedule give the parameters for scheduling this dispatcher

TimeDispatcher(SchedulingParameters)

Signature

```
public  
TimeDispatcher(SchedulingParameters schedule)
```

Description

Create a new dispatcher, whose dispatching thread runs with the given scheduling parameters.

⁶⁴Section [10.3.2.6](#)

⁶⁵Section [5.3.2.2.2](#)

Parameters

schedule give the parameters for scheduling this dispatcher

10.3.2.4.2 Methods

register(Timable)*Signature*

```
public void  
register(Timable target)  
throws RegistrationException,  
        IllegalStateException,  
        IllegalArgumentException
```

Description

Register a [AsyncTimable](#)⁶⁶ with this dispatcher.

Parameters

target to register

Throws

[RegistrationException](#) when target is already registered.
[IllegalStateException](#) when this object has been destroyed.
[IllegalArgumentException](#) when target is not stopped.

deregister(Timable)*Signature*

```
public void  
deregister(Timable target)  
throws DeregistrationException,  
        IllegalStateException,  
        IllegalArgumentException
```

Description

Deregister a [AsyncTimable](#)⁶⁷ from this dispatcher.

⁶⁶Section [10.3.1.1](#)

⁶⁷Section [10.3.1.1](#)

Parameters

target to deregister

Throws

[DeregistrationException](#) when target is not already registered.

[IllegalStateException](#) when this object has been destroyed.

[IllegalArgumentException](#) when target is not stopped.

destroy

Signature

public void

destroy()

throws [IllegalStateException](#)

Description

Release all resources thereby making the dispatcher unusable.

Throws

[IllegalStateException](#) when called on a dispatcher that has one or more registered [AsyncTimable](#)⁶⁸ objects.

10.3.2.5 TimeDispatcher.Runner

Inheritance

[java.lang.Object](#)

[java.lang.Thread](#)

[javax.realtime.RealtimeThread](#)

[javax.realtime.TimeDispatcher.Runner](#)

Description

10.3.2.5.1 Methods

⁶⁸Section [10.3.1.1](#)

run*Signature*

```
public void
run()
```

*Description***10.3.2.6 Timer****Inheritance**

```
java.lang.Object
  javax.realtime.AsyncBaseEvent
    javax.realtime.AsyncEvent
      javax.realtime.Timer
```

Interfaces

```
javax.realtime.AsyncTimable
javax.realtime.ActiveEvent
```

Description

A *timer* is a timed event that measures time according to a given [Clock](#)⁶⁹. This class defines basic functionality available to all timers. Applications will generally use either [PeriodicTimer](#)⁷⁰ to create an event that is fired repeatedly at regular intervals, or [OneShotTimer](#)⁷¹ for an event that just fires once at a specific time. A timer is always associated with at least one [Clock](#)⁷², 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.

10.3.2.6.1 Constructors

⁶⁹Section [10.3.2.1](#)

⁷⁰Section [10.3.2.3](#)

⁷¹Section [10.3.2.2](#)

⁷²Section [10.3.2.1](#)

Timer(HighResolutionTime, TimeDispatcher)

Signature

```
protected  
Timer(javafx.runtime.HighResolutionTime<?> time,  
      TimeDispatcher dispatcher)  
throws IllegalArgumentException,  
      UnsupportedOperationException,  
      IllegalAssignmentError
```

Description

Create a timer that fires according to the given time based on the [Clock](#)⁷³ associated with time and is dispatched by the specified dispatcher.

Available since version 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

`IllegalArgumentException` when time is a negative `RelativeTime` value.

`UnsupportedOperationException` when time has a [Chronograph](#)⁷⁴ is not a clock.

[IllegalAssignmentError](#) when this `Timer` cannot hold references to handler and clock.

Timer(HighResolutionTime)

Signature

```
protected  
Timer(javafx.runtime.HighResolutionTime<?> time)
```

⁷³Section [10.3.2.1](#)

⁷⁴Section [10.3.1.2](#)

throws `IllegalArgumentException`,
`UnsupportedOperationException`,
`IllegalAssignmentError`

Description

Create a timer that fires according to the given time based on the [Clock](#)⁷⁵ associated with time and is dispatched by the system default dispatcher.

This is equivalent to `Timer(time, null)`.

Available since version 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()`.

Throws

`IllegalArgumentException` when time is a negative `RelativeTime` value.

`UnsupportedOperationException` when time has a [Chronograph](#)⁷⁶ is not a clock.

[IllegalAssignmentError](#) when this `Timer` cannot hold references to handler and clock.

Timer(HighResolutionTime, Clock, AsyncEventHandler)

Signature

```
protected
Timer(javax.realtime.HighResolutionTime<?> time,
      Clock clock,
      AsyncEventHandler handler)
throws IllegalArgumentException,
      UnsupportedOperationException,
      IllegalAssignmentError
```

Description

Create a timer that fires according to the given time, which must be based on the supplied [Clock](#)⁷⁷ clock (if any), and is handled by the specified [AsyncEvent-](#)

⁷⁵Section [10.3.2.1](#)

⁷⁶Section [10.3.1.2](#)

⁷⁷Section [10.3.2.1](#)

Handler⁷⁸ 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 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()`.

clock The clock on which to base this timer. When null, the clock associated with **time** 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

`IllegalArgumentException` when **time** is a negative `RelativeTime` value or the supplied clock is not the `Clock` associated with **time**.

`UnsupportedOperationException` when **time** has a `Chronograph`⁷⁹ that is not an instance of `Clock`.

`IllegalAssignmentError` when this `Timer` cannot hold references to **handler** and **clock**.

10.3.2.6.2 Methods

getClock

Signature

```
public javax.realtime.Clock  
getClock()  
throws IllegalStateException
```

Description

Return the instance of `Clock`⁸⁰ on which this timer is based.

Throws

⁷⁸Section 8.3.3.5

⁷⁹Section 10.3.1.2

⁸⁰Section 10.3.2.1

IllegalStateException when this Timer has been *destroyed*.

Returns

The instance of [Clock](#)⁸¹ associated with this Timer.

getStart

Signature

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

Description

Get the start time of this Timer. Note that the start time uses copy semantics, so changes made to the value returned by this method do not effect the start time of this Timer.

Returns

a reference to the time (or start) parameter used when constructing this Timer, ensuring the content has the original values. Since RTSJ 2.0

getEffectiveStartTime

Signature

```
public javax.realtime.AbsoluteTime
    getEffectiveStartTime()
    throws IllegalStateException,
           ArithmeticException
```

Description

Return a newly-created time representing the time the timer actually started, or when the timer has been rescheduled, the effective start time after the reschedule.

Throws

IllegalStateException when the timer is not active or has been destroyed.

ArithmeticException when the result does not fit in the normalized format.

Returns

the time this actually started.

Available since RTSJ 2.0

⁸¹Section [10.3.2.1](#)

getEffectiveStartTime(AbsoluteTime)

Signature

```
public javax.realtime.AbsoluteTime  
getEffectiveStartTime(AbsoluteTime dest)  
throws IllegalStateException,  
    ArithmeticException
```

Description

Update dest to represent the time the timer actually started, or when the timer has been rescheduled, the effective start time after the reschedule. When dest is null, behave as if [getEffectiveStartTime\(\)](#)⁸² had been called.

Parameters

dest a place to store the time this actually started.

Throws

IllegalStateException when the timer is not active or has been destroyed.

ArithmeticException when the result does not fit in the normalized format.

Returns

The time the timer actually started, or when it has been rescheduled, the effective start time after the reschedule.

Available since RTSJ 2.0

getLastReleaseTime

Signature

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

Description

Get the last release time of this timer.

Throws

IllegalStateException when this timer has not been released since it was last started.

Returns

⁸²Section [10.3.2.6.2](#)

a reference to a newly-created [AbsoluteTime](#)⁸³ object representing this timer's last release time. When the timer has not been released since it was last started, throw an exception.

Available since RTSJ 2.0

getLastReleaseTime(AbsoluteTime)

Signature

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

Description

Returns

When `dest` is null, return a reference to a newly-created [AbsoluteTime](#)⁸⁴ object representing this timer's last release time. When `dest` is non-null, set `dest` to this timer's last release time. When the timer has not been released, return null. Since RTSJ 2.0

getFireTime

Signature

```
public javax.realtime.AbsoluteTime  
getFireTime()  
throws IllegalStateException,  
ArithmeticException
```

Description

Get the time at which this Timer is expected to fire. When the Timer is *disabled*, the returned time is that of the skipping of the firing. When the Timer is *not-active* it throws `IllegalStateException`.

Throws

`ArithmeticException` when the result does not fit in the normalized format.

`IllegalStateException` when this Timer has been *destroyed*, or when it is *not-active*.

⁸³Section [9.3.1.1](#)

⁸⁴Section [9.3.1.1](#)

Returns

The absolute time at which this is expected to fire (release handlers or skip), in a newly allocated [AbsoluteTime](#)⁸⁵ object. When the timer has been created or re-scheduled (see [Timer.reschedule](#)⁸⁶) 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

```
public javax.realtime.AbsoluteTime  
getFireTime(AbsoluteTime dest)  
throws IllegalStateException,  
       ArithmeticException
```

Description

Get the time at which this `Timer` is expected to fire. When the `Timer` is *disabled*, the returned time is that of the skipping of the firing. When the `Timer` is *not-active* it throws `IllegalStateException`.

Parameters

`dest` The instance of [AbsoluteTime](#)⁸⁷ which will be updated in place and returned. The clock association of the `dest` parameter is ignored. When `dest` is null a new object is allocated for the result.

Throws

`ArithmeticException` when the result does not fit in the normalized format.

`IllegalStateException` when this `Timer` has been *destroyed*, or when it is *not-active*.

Returns

The instance of [AbsoluteTime](#)⁸⁸ 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](#)⁸⁹) using an instance of `RelativeTime` for its time parameter then it will return the sum of the current time and the `RelativeTime` remaining time

⁸⁵Section 9.3.1.1

⁸⁶Section 10.3.2.6.2

⁸⁷Section 9.3.1.1

⁸⁸Section 9.3.1.1

⁸⁹Section 10.3.2.6.2

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

Get the dispatcher associated with this Timable.

Available since RTSJ 2.0

isActive

Signature

```
public boolean  
isActive()
```

Description

Determine the activation state of this happening, i.e., it has been started.

Returns

true when active, false otherwise.

isRunning

Signature

```
public boolean  
isRunning()  
throws IllegalStateException
```

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 answer the question "*Is firing expected?*".

Throws

IllegalStateException 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 IllegalStateException
```

Description

Replaced by [AsyncBaseEvent.handledBy\(AsyncBaseEventHandler\)](#)⁹⁰

Parameters

handler to add to the Timer

Throws

IllegalStateException when this Timer has been *destroyed*.

Returns

true when handler is associated with this, otherwise false.

Available since RTSJ 1.0.1

createReleaseParameters

Signature

```
public javax.realtime.ReleaseParameters<?>  
createReleaseParameters()  
throws IllegalStateException
```

Description

Create a [ReleaseParameters](#)⁹¹ object appropriate to the timing characteristics of this event. The default is the most pessimistic: [AperiodicParameters](#)⁹². This is

⁹⁰Section [8.3.3.2.1](#)

⁹¹Section [6.3.3.10](#)

⁹²Section [6.3.3.2](#)

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

`IllegalStateException` when this `Timer` has been *destroyed*.

Returns

A newly created `ReleaseParameters`⁹³ object.

enable

Signature

```
public void  
enable()  
throws IllegalStateException
```

Description

Re-enable this timer after it has been *disabled*. (See `Timer.disable()`⁹⁴.) When the `Timer` is already *enabled*, this method does nothing. When the `Timer` is *not-active*, this method does nothing.

Throws

`IllegalStateException` when this `Timer` has been *destroyed*.

disable

Signature

```
public void  
disable()  
throws IllegalStateException
```

Description

Disable this timer, preventing it from firing. It may subsequently be re-*enabled*. When the timer is *disabled* when its fire time occurs then it will not release its handlers. However, a *disabled* timer created using an instance of `RelativeTime` for its time parameter continues to count while it is *disabled*, and no changes take place in a *disabled* timer created using an instance of `AbsoluteTime`, in both cases the potential firing is simply masked, or skipped. When the timer is

⁹³Section 6.3.3.10

⁹⁴Section 10.3.2.6.2

subsequently re-*enabled* before its fire time and it is *enabled* when its fire time occurs, then it will fire. It is important to note that this method does not delay the time before a possible firing. For example, when the timer is set to fire at time 42 and the `disable()` is called at time 30 and `enable()` is called at time 40 the firing will occur at time 42 (not time 52). These semantics imply also that firings are not queued. Using the above example, when `enable` was called at time 43 no firing will occur, since at time 42 this was *disabled*. When the Timer is already *disabled*, whether it is *active* or *inactive*, this method does nothing.

Throws

`IllegalStateException` when this Timer has been *destroyed*.

start

Signature

```
public void  
start()  
throws IllegalStateException
```

Description

Start this timer. A timer starts measuring time from when it is started; this method makes the timer *active* and *enabled*.

Throws

`IllegalStateException` when this Timer has been *destroyed*, or when this timer is already *active*.

start(boolean)

Signature

```
public void  
start(boolean disabled)  
throws IllegalStateException
```

Description

Start this timer. A timer starts measuring time from when it is started. When `disabled` is true start the timer making it *active* in a *disabled* state. When `disabled` is false this method behaves like the `start()` method.

Parameters

disabled When true, the timer will be *active* but *disabled* after it is started. When false this method behaves like the `start()` method.

Throws

`IllegalStateException` when this `Timer` has been *destroyed*, or when this timer is *active*.

Available since RTSJ 1.0.1

stop

Signature

```
public boolean  
stop()  
throws IllegalStateException
```

Description

Stops a timer when it is *active* and changes its state to *inactive* and *disabled*.

Throws

`IllegalStateException` when this `Timer` has been *destroyed*.

Returns

true when this was *enabled* and false otherwise.

reschedule(HighResolutionTime)

Signature

```
public void  
reschedule(javax.realtime.HighResolutionTime<?> time)  
throws IllegalStateException,  
    IllegalArgumentException
```

Description

Change the scheduled time for this event. This method can take either an `AbsoluteTime` or a `RelativeTime` for its argument, and the `Timer` will behave as if created using that type for its time parameter. The rescheduling will take place between the invocation and the return of the method.

Note 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

IllegalArgumentException when time is a negative RelativeTime value.

IllegalStateException when this Timer has been *destroyed*.

10.4 Rationale

Clocks differ because of monotonicity, synchronization, jitter, stability, accuracy, precision, and resolution. There are many possible subclasses of clocks: realtime clocks, user time clocks, simulation time clocks, wall clocks.

The idea of using multiple clocks may at first seem unusual, but it enables the accommodation of difference kinds of clocks and as a possible resource allocation strategy. Consider a realtime system where the natural events of the system have different tolerances for jitter. Assume the system functions properly if event *A* is triggered within plus or minus 100 seconds of the actual time it should occur but event *B* must be triggered within 100 microseconds of its actual time. Further assume, without loss of generality, that events *A* and *B* are periodic. An application could then create two instances of PeriodicTimer based on two clocks. The timer for event *B* should be based on a Clock which checks its queue at least every 100 microseconds but the timer for event *A* could be based on a Clock that checked its queue only every 100 seconds. This use of two clocks reduces the queue size of the accurate clock and thus queue management overhead is reduced.

The importance of the use of one-shot timers for time-out behavior and the vagaries in the execution of code prior to starting the timer for short time-outs dictate that the triggering of the timer should be guaranteed. The problem is exacerbated for periodic timers where the importance of the periodic triggering outweighs the precision of the start time. In such cases, it is also convenient to allow, for example, a relative time of zero to be used as the start time.

Clock resolution is a complicated topic, and clock implementations may have differing 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()` and `getQueryPrecision` 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. For most realtime systems, a single heap with a deterministic garbage collector is usually also sufficient. For other situations, this specification defines classes directly related to memory and memory management. These classes provide a more generalized means of memory management than 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 is managing the allocation area that will be used when the thread/schedulable requests memory allocation using the Java new operator.

There are three types of memory area, distinguished by object lifetime semantics, defined by the RTSJ.

- Heap memory—the Java heap. 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.

Stacked Memory

RTSJ 2.0, 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 `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 `StackedMemory` areas, as well as the manner in which they may be arranged on the scope stack. These constraints are enforced by the implementation.

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.

11.1 Definitions

Allocation Context — An abstraction representing memory from which a new object can be allocated. In conventional Java, this is the Java heap. The `MemoryArea` class is the base class representing all allocation contexts in the RTSJ, of which the heap (represented by `HeapMemory`) is just one type.

Current Allocation Context — The memory area which will be used when object allocation is requested in the currently active thread of control.

Allocation Area — The area of memory that is managed by a `MemoryArea` from which objects are allocated. The allocation area for a extraheap memory area 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.

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` instances that is active,
3. it contains a firable asynchronous event handlers¹,
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

¹Defined in Section 8.1

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.

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 @code HeapMemory and ImmortalMemory are equivalent.

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.

Scope Stack — A sequence of the memory areas the 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.

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 3.2.2.3 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 the class description and the constructor, method, and field detail sections further on.

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. The linear time attribute requires that, ignoring performance variations due to hardware caches or similar optimizations and ignoring execution time of any static initializers, the execution time of new must be bounded by a polynomial, $f(n)$, where n is the size of the object and for all $n > 0$, $f(n) \leq Cn$ for some constant C .
2. 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.

3. A memory area is represented by an instance of a subclass of the `MemoryArea` class. When a memory area, *m*, is entered by calling `m.enter` (or another method from the family of enter-like methods defined in `MemoryArea` or its subclasses), *m* becomes the *allocation context* of the current schedulable object. When control returns from the `enter` method, the allocation context is restored to the value it had immediately before `enter` was called.
4. When a memory area, *m*, is entered by calling *m*’s `executeInArea` method, *m* becomes the current allocation context of the current schedulable. When control returns from the `executeInArea` method, the allocation context is restored to the value it had before `executeInArea` was called.
5. The initial allocation context for a schedulable is the memory area that was designated the *initial memory area* when the schedulable was constructed. This initial allocation context becomes the current allocation context for that schedulable when the schedulable object first becomes eligible for execution. For instances of `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*.
6. All object allocation through the `new` keyword will use the current allocation context, but note that allocation can be performed in a specific memory area using the `newInstance` and `newArray` methods on `MemoryArea`.
7. Instances of schedulables behave as if they stored their memory area context in a structure called the *scope stack*. This structure is manipulated by the instantiation of a schedulables, and the following methods from `MemoryArea` and its subclasses: all the `enter` and `joinAndEnter` methods, `executeInArea`, and both `newInstance` methods. See the semantics in *Maintaining the Scope Stack* for details.
8. The scope stack is accessible through a set of static methods on `RealtimeThread`. These methods allow outer allocation contexts to be accessed by their index number. Memory areas on a scope stack may be referred to as *inner* or *outer* relative to other entries in that scope stack. An “outer scope” is further from the current allocation context on the current scope stack and has a lower index.
9. The `executeInArea`, `newInstance` and `newArray` methods, when invoked on

an instance of `ScopedMemory` require that instance to be an outer allocation context on the current schedulable object's current scope stack.

10. An instance of `ScopedMemory` is said to be *in use* if it has a positive reference count as defined by semantic 17 below.

11.2.3 The Parent Scope

The following requirements apply to a scope's parent.

11. 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.
12. Instances of `ScopedMemory` must satisfy the *single parent rule*, which requires that each scoped memory has a unique parent as defined in semantic 11.

11.2.4 Memory Areas and Schedulables

The following requirements govern the relationship between memory and execution.

13. Pushing a scoped memory onto a scope stack is always subject to the single parent rule.
14. 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.
15. A Java thread cannot have a scope stack; consequently it can only be created and execute within heap or immortal memory. The thread starts execution with its allocation context set to the memory area containing the `Thread` object. An attempt to create a Java thread in a scoped memory area throws `IllegalAssignmentError`.
16. A Java thread may use `executeInArea`, and the `newInstance` and `newArray` methods from the `ImmortalMemory` and `HeapMemory` classes. These methods

enable it to execute with an immortal current allocation context, but semantic 15 applies even during execution of these methods.

11.2.5 Scoped Memory Reference Counting

The following requirements apply to references to scoped memory.

17. 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 contexts* or more exactly, the reference count is the number of causes for memory area to be an execution context.
18. Each instance of the `PinnableMemory` class must support a pinned count. This count is incremented for each call of the `pin` method and decremented for each call of the `unpin` method. The count is always greater than or equal to zero (that is, calling the `unpin` method has no effect if the count equals zero).
19. When the reference count for an instance of the class `ScopedMemory` is ready to be decremented from one to zero and the pinned count (if present) is equal to zero, all unfinalized objects within that area are considered ready for 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.
20. When the pinned count is ready to go to zero and the reference count is zero, all unfinalized objects within that area are considered ready for finalization,

and the same semantics as 19 above applies.

21. 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.
22. 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.
23. 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.
24. 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.

11.2.6 Immortal Memory

The following requirements apply to immortal memory.

25. 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.
26. 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.
27. 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.

28. Static initializers are executed effectively as if the current thread performed `ImmutableMemory.instance().executeInArea(r)` where `r` is a `Runnable` that executes the `<clinit>` method of the class being initialized.

11.2.7 Maintaining Referential Integrity

The following rules apply to references to objects in scoped memory.

29. 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. For this table,

Table 11.1: Memory Area Referencing Restrictions

Stored in Area	Reference to Object in Heap	Reference to Object in Immortal	Reference to Object in Scoped	null
Heap	Permit	Permit	Forbid	Permit
Immortal	Permit	Permit	Forbid	Permit
Scoped	Permit	Permit	Permit from same or less deeply nested scope	Permit
Local Variable	Permit	Permit	Permit	Permit

`ImmutableMemory` and `ImmutablePhysicalMemory` are equivalent, and all subclasses of `ScopedMemory` are equivalent.

30. 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.

11.2.8 Object Initialization

The following requirements apply to object initialization.

31. 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*.

11.2.9 Maintaining the Scope Stack

This section describes maintenance of a data structure that is called the *scope stack*. Implementations are not required to use a stack or implement the algorithms given here. It is only required that an implementation behave with respect to the ordering and accessibility of memory scopes effectively as if it implemented these algorithms. The scope stack is implicitly visible through the memory assignment rules, and the stack is explicitly visible through the static method `getOuterMemoryArea(int)` on `RealtimeThread`.

Four operations affect the scope stack: the `enter` methods defined in `MemoryArea` and its subclasses, instantiation of a new `Schedulable`, the `executeInArea` method in `MemoryArea`, and the `newInstance` 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_t , 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 allocation 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 `Schedulable` instance s with initial memory area D which is not currently in use, s gets the scope stack $A \rightarrow B \rightarrow C \rightarrow D$.

Note that in the last case, when the initial memory area is a scope already in a scoped stack created by adding the initial memory area to the current stack of the handler at the time the release happens, a `ScopedCycleException` will be thrown at release time and the code in the handler will not be executed for that release.

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 object 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, m , is entered by calling m 's `executeInArea` method or one of the `m.newInstance` methods, the scope stack before the method call is preserved and replaced with a scope stack constructed as follows:
 - (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.

11.2.10 The `enter` Method

For `ma.enter(logic)`:

```

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

```

11.2.11 The `executeInArea` or `newInstance` Methods

For `ma.executeInArea(logic)`, `ma.newInstance()`, or `ma.newArray()`:

```

1  when ma is an instance of heap immortal or ImmortalPhysicalMemory,
2    start a new scope stack containing only ma.
3    make the new scope stack the scope stack for the current
4    schedulable.
5  else if ma is in the scope stack for the current schedulable,
6    start a new scope stack containing ma and all
7    scopes below ma on the scope stack.
8    make the new scope stack the scope stack for the current
9    schedulable.
10 else
11   throw InaccessibleAreaException, execute logic.run,
12   or construct the object.
13   restore the previous scope stack for the current schedulable.
14   discard the new scope stack.
15 end

```

11.2.12 Constructor Methods for Schedulables

For construction of a schedulable in memory area `cma` with initial memory area of `ima`:

```
1  if cma is heap, immortal or ImmortalPhysicalMemory,  
2      create a new scope stack containing cma.  
3  else  
4      start a new scope stack containing the entire  
5          current scope stack.  
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.

11.2.13 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., was entered from `ImmortalMemory` or `Heap`, 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.

Except for the primordial scope, which represents heap, immortal and immortal physical memory, only scoped memory areas are visible to the single parent rule.

The operational effect of the single parent rule is that when a scoped memory area has a parent, the only legal change to that value is to “no parent.” Thus an ordering imposed by the first assignments of parents of a series of nested scoped memory areas is the only nesting order allowed until control leaves the scopes; then a new nesting order is possible. Thus, a schedulable attempting to enter a scope can only do so by entering in the established nesting order.

11.2.14 Scope Tree Maintenance

The single parent rule is enforced effectively as if there were a tree with the primordial scope (representing heap, immortal, and immortal physical memory) at its root, and other nodes corresponding to every scoped memory area that is currently on any schedulable's scope stack.

Each scoped memory has a reference to its parent memory area, `ma.parent`. The parent reference may indicate a specific scoped memory area, no parent, or the primordial parent.

When a scoped memory area is the explicit initial memory area of a realtime thread 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 is in a scope stack. 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.

11.2.14.1 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 scope stack do not violate the single parent rule.

```

1 preconditions
2
3   ma.parent is set to the correct parent (either a scoped
4   memory area or the primordial scope) or to null (no parent).
5
6   t.scopeStack is the scope stack of the current schedulable
7
8 Action
9
10  if ma is scoped,
11     parent = findFirstScope(t.scopeStack)
12  if ma.parent == null
13     ma.parent = parent.
14  else if ma.parent != parent
15     throw ScopedCycleException.
16  else
17     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`.

```

1 findFirstScope(scopeStack)
2 {
3   for s = top of scope stack to bottom of scope stack
```

```
4  if s is an instance of scopedMemory
5  return s return primordial scope
6  }
```

11.2.14.2 Popping a MemoryArea off the Scope Stack

```
1  ma = t.scopeStack.pop.
2  if ma is scoped
3      if !(ma.in_use || (ma.reserve_count > 0))
4      ma.parent = noParent
```

11.2.14.3 Reservation Management

Reservation management is separate from managing the scope stack for a task. When a realtime thread with an explicit initial scoped memory area (EISMA) is created or an ASEH with an EISMA is added to an ASE, the following happens atomically with respect to other tasks in the VM:

```
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 or an ASEH is removed from an ASE, the following happens atomically with respect to other tasks in the VM:

```
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.
```

11.2.15 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 their 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 their 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 12.2.1).

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 interface² or from some property file read at program initialization time.

²For 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/>

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 memory with certain characteristics and a virtual memory type (for example, SRAM that is permanently resident in memory).

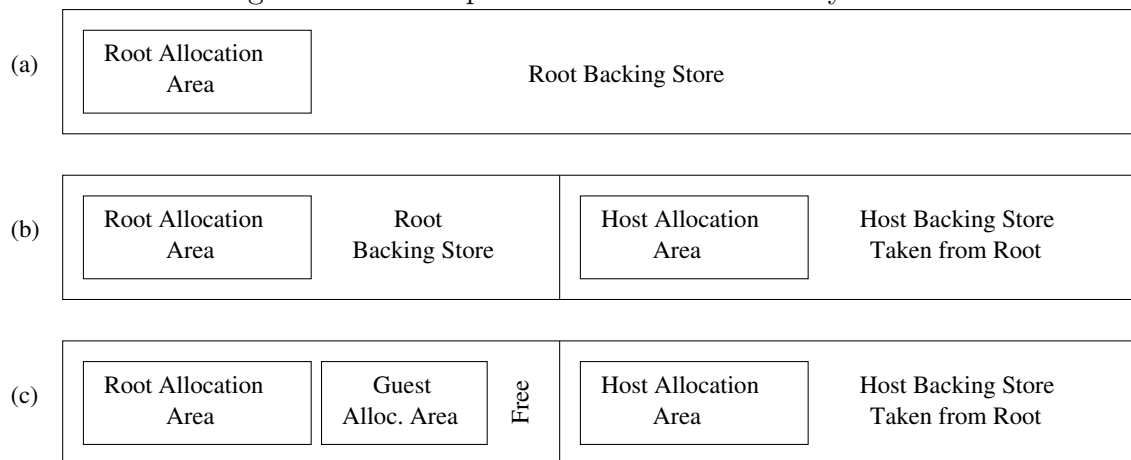
11.2.16 Stacked Memory

A StackedMemory area represents both an *allocation area* providing ScopedMemory semantics and an explicit *backing store* from which the 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.

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 a notional global 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 if 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

Figure 11.1: Manipulation of StackedMemory Areas



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.

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. (Implementations may require a constant amount of

overhead, drawn from the backing store, for each StackedMemory area created in the store.)

```
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 {
6     public void run()
7     {
8         // Create a host area with a 6 kB backing store and
9         // 2 kB allocation area
10        hostArea = new StackedMemory(2048, 6144); // (b)
11        // Create a guest area with a 2 kB allocation area
12        guestArea = new StackedMemory(1536); // (c)
13    }
14 });
```

Commented points (a), (b), and (c) correspond to their respective subfigures in Figure 11.1. At point (a), a root StackedMemory has been created with its 10 kB backing store drawn from the notional 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 StackedMemory is created at (b), reserving 6 kB of the root StackedMemory's backing store for its own use and creating a second 2 kB allocation area within that reservation. A new guest StackedMemory is then created at (c) in the root area (without entering the host child), occupying 1.5 kB of the remaining free 2 kB of the 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 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 StackedMemory areas beneath it in the stack.

11.3 javax.realtime

11.3.1 Interfaces

11.3.1.1 MemoryAreaVisitor

Description

This interface is used to visit memory areas. For example, the method [MemoryArea.visitNestedMemory](#)³ uses this visitor to process all memory areas nested the area upon which it is called.

11.3.1.1.1 Methods

visit(MemoryArea)

Signature

```
public R  
visit(MemoryArea memory)
```

Description

Visit the members of a collection of memory areas. It provides a means of accessing all live scopes contained in a memory area, even those to which no reference exists, such a [javax.realtime.memory.PinnableMemory](#)⁴ that is pinned or another [ScopedMemory](#) that contains a [Schedulable](#). The set may be concurrently modified by other tasks, but the view seen by the visitor may not be updated to reflect those changes.

Parameters

memory The memory area being visited.

Returns

Any object declared by the application or null. When visit returns an object, no more memory areas are visited and the [MemoryArea.visitNestedMemory](#)⁵ method returns the object returned by [visit\(MemoryArea\)](#)⁶.

³Section [11.3.2.3.2](#)

⁴Section [11.4.3.5](#)

⁵Section [11.3.2.3.2](#)

⁶Section [11.3.1.1.1](#)

11.3.2 Classes

11.3.2.1 HeapMemory

Inheritance

java.lang.Object
 javafx.runtime.MemoryArea
 javafx.runtime.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

Associate this memory area with the current schedulable for the duration of the execution of the `run()` method of the instance of `Runnable` given in the constructor. During this period of execution, this memory area becomes the default allocation context until another default allocation context is selected (using `enter`, or `executeInArea`⁷) or the `enter` method exits.

Throws

`IllegalThreadStateException` when the caller is a Java thread.

`IllegalArgumentException` `IllegalArgumentException` when the caller is a schedulable and a null value for logic was supplied when the memory area was constructed.

`MemoryAccessError` when caller is a schedulable which may not use the heap.

⁷Section 11.3.2.1.1

enter(Runnable)

Signature

```
public void  
enter(Runnable logic)
```

Description

Associate this memory area with the current schedulable for the duration of the execution of the `run()` method of the given `Runnable`. During this period of execution, this memory area becomes the default allocation context until another default allocation context is selected (using `enter`, or `executeInArea`⁸) 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.

`IllegalThreadStateException` `IllegalThreadStateException` when the caller is a Java thread.

`IllegalArgumentException` `IllegalArgumentException` 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`⁹ representing the Java heap. The singleton instance of this class shall be allocated in the `ImmortalMemory`¹⁰ area.

Returns

The singleton `HeapMemory`¹¹ object.

⁸Section 11.3.2.1.1

⁹Section 11.3.2.1

¹⁰Section 11.3.2.2

¹¹Section 11.3.2.1

executeInArea(Runnable)

Signature

```
public void  
executeInArea(Runnable logic)
```

Description

Execute the run method from the logic parameter using heap as the current allocation context. For a schedulable, this saves the current scope stack and replaces it with one consisting only of the HeapMemory instance; restoring the original scope stack upon completion.

Parameters

logic The runnable object whose run() method should be executed.

Throws

IllegalArgumentException 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

Allocate an array of the given type in this memory area. This method may be concurrently used by multiple threads.

Parameters

type 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 number The number of elements in the new array.

Throws

MemoryAccessError when caller is a schedulable which may not use the heap.

IllegalArgumentException IllegalArgumentException when number is less than zero, type is null, or type is java.lang.Void.TYPE.

OutOfMemoryError OutOfMemoryError 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

Allocate an object in this memory area. This method may be concurrently used by multiple threads.

Parameters

type type The class of which to create a new instance.

Throws

[MemoryAccessError](#) when caller is a schedulable which may not use the heap.

[IllegalAccessException](#) [IllegalAccessException](#) The class or initializer is inaccessible.

[IllegalArgumentException](#) [IllegalArgumentException](#) when type is null.

[ExceptionInInitializerError](#) [ExceptionInInitializerError](#) when an unexpected exception has occurred in a static initializer.

[OutOfMemoryError](#) [OutOfMemoryError](#) when space in the memory area is exhausted.

[InstantiationException](#) [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*

```
public T  
newInstance(java.lang.reflect.Constructor<T> c,  
        java.lang.Object[] args)
```

throws `IllegalAccessException`,
 `InstantiationException`,
 `InvocationTargetException`

Description

Allocate an object in this memory area. This method may be concurrently used by multiple threads.

Parameters

`c` `Tc` The constructor for the new instance.

`args` `args` An array of arguments to pass to the constructor.

Throws

`MemoryAccessError` when caller is a schedulable which may not use the heap.

`IllegalAccessException` `IllegalAccessException` when the class or initializer is inaccessible under Java access control.

`InstantiationException` `InstantiationException` when the specified class object could not be instantiated. Possible causes are it is an interface, it is abstract, it is an array.

`OutOfMemoryError` `OutOfMemoryError` when space in the memory area is exhausted.

`IllegalArgumentException` `IllegalArgumentException` when `c` is null, or the `args` array does not contain the number of arguments required by `c`. A null value of `args` is treated like an array of length 0.

`InvocationTargetException` `InvocationTargetException` when the underlying constructor throws an exception.

Returns

A new instance of the object constructed by `c`.

visitNestedMemory(MemoryAreaVisitor)

Signature

```
public R  
visitNestedMemory(javax.realtime.MemoryAreaVisitor<R> visitor)
```

Description

Visit each scoped memory area who's parent is the primordial scope and was created in heap memory.

Parameters

visitor invoke the `MemoryAreaVisitor.visit(MemoryArea)`¹² method for each member of the set of scoped memory areas that was created in heap memory and has the primordial scope as its parent.

Throws

`IllegalArgumentException` `IllegalArgumentException` when visitor is null.

Returns

null when all elements where visited and some object of type R when the visit is forced to terminate at the end of visiting that element.

11.3.2.2 ImmortalMemory

Inheritance

java.lang.Object
 javax.realtime.MemoryArea
 javax.realtime.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`¹³, `javax.realtime.memory.StackedMemory`¹⁴, or `javax.realtime.memory.PinnableMemory`¹⁵.

Objects in immortal have the same states with respect to finalization as objects in the standard Java heap, but there is no assurance that immortal objects will be finalized even when the JVM is terminated.

Methods from ImmortalMemory should be overridden only by methods that use super.

11.3.2.2.1 Methods

¹²Section 11.3.1.1.1

¹³Section 11.4.3.1

¹⁴Section 11.4.3.7

¹⁵Section 11.4.3.5

instance

Signature

```
public static javax.realtime.ImmortalMemory  
instance()
```

Description

Returns a pointer to the singleton [ImmortalMemory](#)¹⁶ object.

Returns

The singleton [ImmortalMemory](#)¹⁷ object.

executeInArea(Runnable)

Signature

```
public void  
executeInArea(Runnable logic)
```

Description

Execute the run method from the logic parameter using this memory area as the current allocation context. For a schedulable, this saves the current scope stack and replaces it with one consisting only of the [ImmortalMemory](#) instance; restoring the original scope stack upon completion.

Parameters

logic The runnable object whose run() method should be executed.

Throws

[IllegalArgumentException](#) when logic is null.

visitNestedMemory(MemoryAreaVisitor)

Signature

```
public R  
visitNestedMemory(javax.realtime.MemoryAreaVisitor<R> visitor)
```

Description

Visit each scoped memory area who's parent is the primordial scope and was created in this memory area.

¹⁶Section [11.3.2.2](#)

¹⁷Section [11.3.2.2](#)

Parameters

visitor invoke the `MemoryAreaVisitor.visit(MemoryArea)`¹⁸ method for each member of the set of scoped memory areas that was created in this immortal memory area and has the primordial scope as its parent.

Throws

`IllegalArgumentException` `IllegalArgumentException` when visitor is null.

Returns

null when all elements where visited and some object of type R when the visit is forced to terminate at the end of visiting that element.

11.3.2.3 MemoryArea

Inheritance

java.lang.Object
javax.realtime.MemoryArea

Description

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)
```

¹⁸Section 11.3.1.1.1

throws `IllegalArgumentException`,
`OutOfMemoryError`,
`IllegalAssignmentError`

Description

Create an instance of `MemoryArea`.

Parameters

`size` The size of `MemoryArea` to allocate, in bytes.

`logic` The `run()` method of this object will be called whenever `enter()`¹⁹ is called.
 When `logic` is null, this constructor is equivalent to `MemoryArea(long size)`.

Throws

`IllegalArgumentException` when the `size` parameter is less than zero.

`OutOfMemoryError` when there is insufficient memory for the `MemoryArea` object
 or for the backing memory.

`IllegalAssignmentError` when storing `logic` in this would violate the assignment
 rules.

MemoryArea(SizeEstimator, Runnable)

Signature

protected
`MemoryArea(SizeEstimator size,
 Runnable logic)`
 throws `IllegalArgumentException`,
`OutOfMemoryError`,
`IllegalAssignmentError`

Description

Equivalent to `MemoryArea(long, Runnable)`²⁰ with the argument list (`size.getEstimate()`,
`logic`).

Parameters

`size` A `SizeEstimator` object which indicates the amount of memory required by this
`MemoryArea`.

¹⁹Section 11.3.2.3.2

²⁰Section 11.3.2.3.1

logic The run() method of this object will be called whenever [enter\(\)](#)²¹ is called. When logic is null, this constructor is equivalent to MemoryArea(SizeEstimator size).

Throws

IllegalArgumentException when size is null or size.getEstimate() is negative.

OutOfMemoryError when there is insufficient memory for the MemoryArea object or for the backing memory.

[IllegalAssignmentError](#) when storing logic in this would violate the assignment rules.

MemoryArea(long)

Signature

```
protected  
MemoryArea(long size)  
throws IllegalArgumentException,  
    OutOfMemoryError
```

Description

Equivalent to [MemoryArea\(long, Runnable\)](#)²² with the argument list (size, null).

Parameters

size The size of MemoryArea to allocate, in bytes.

Throws

IllegalArgumentException when size is less than zero.

OutOfMemoryError when there is insufficient memory for the MemoryArea object or for the backing memory.

MemoryArea(SizeEstimator)

Signature

```
protected  
MemoryArea(SizeEstimator size)
```

²¹Section [11.3.2.3.2](#)

²²Section [11.3.2.3.1](#)

throws `IllegalArgumentException`,
`OutOfMemoryError`

Description

Equivalent to `MemoryArea(long, Runnable)`²³ with the argument list (`size.getEstimate()`, `null`).

Parameters

size A `SizeEstimator`²⁴ object which indicates the amount of memory required by this `MemoryArea`.

Throws

`IllegalArgumentException` when the size parameter is null, or `size.getEstimate()` is negative.

`OutOfMemoryError` when there is insufficient memory for the `MemoryArea` object or for the backing memory.

11.3.2.3.2 Methods

enter

Signature

```
public void  
enter()  
throws IllegalArgumentException,  
    OutOfMemoryError,  
    IllegalAssignmentError,  
    MemoryAccessError
```

Description

Associate this memory area with the current schedulable for the duration of the execution of the `run()` method of the instance of `Runnable` given in the constructor. During this period of execution, this memory area becomes the default allocation context until another default allocation context is selected (using `enter`, or `executeInArea`²⁵) or the `enter` method exits.

²³Section 11.3.2.3.1

²⁴Section 11.3.2.5

²⁵Section 11.3.2.3.2

Throws

`IllegalThreadStateException` when the caller is a Java thread.

`IllegalArgumentException` 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`²⁶, so the JVM cannot be permitted to deliver the exception. The `ThrowBoundaryError`²⁷ instance is preallocated by the VM to avoid cascading creation of `ThrowBoundaryError`²⁸.

`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

Associate this memory area with the current schedulable for the duration of the execution of the `run()` method of the given `Runnable`. During this period of execution, this memory area becomes the default allocation context until another default allocation context is selected (using `enter`, or `executeInArea`²⁹) or the `enter` method exits.

Parameters

logic The `Runnable` object whose `run()` method should be invoked.

Throws

`IllegalThreadStateException` when the caller is a Java thread.

`IllegalArgumentException` when the caller is a schedulable and logic is null.

`ThrowBoundaryError` Thrown when the JVM needs to propagate an exception allocated in this scope to (or through) the memory area of the caller. Storing a reference to that exception would cause an `IllegalAssignmentError`³⁰, so

²⁶Section 15.2.3.2

²⁷Section 15.2.3.8

²⁸Section 15.2.3.8

²⁹Section 11.3.2.3.2

³⁰Section 15.2.3.2

the JVM cannot be permitted to deliver the exception. The `ThrowBoundaryError`³¹ 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)`³² 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

```
public boolean  
enter(BooleanSupplier logic)
```

Description

Same as `enter(Runnable)`³³ 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.

³¹Section 15.2.3.8

³²Section 11.3.2.3.2

³³Section 11.3.2.3.2

enter(IntSupplier)

Signature

```
public int  
enter(IntSupplier logic)
```

Description

Same as `enter(Runnable)`³⁴ 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)
```

Description

Same as `enter(Runnable)`³⁵ 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)
```

Description

³⁴Section 11.3.2.3.2

³⁵Section 11.3.2.3.2

Same as `enter(Runnable)`³⁶ except that the executed method is called `get` and a `double` is returned.

Parameters

logic the object who's `get` method will be executed.

Returns

a result from the computation.

getMemoryArea(Object)

Signature

```
public static javax.realtime.MemoryArea  
getMemoryArea(Object object)
```

Description

Gets the `MemoryArea` in which the given object is located.

Throws

`IllegalArgumentException` when the value of object is null.

Returns

The instance of `MemoryArea` from which object was allocated.

memoryConsumed

Signature

```
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.

³⁶Section 11.3.2.3.2

memoryRemaining

Signature

```
public long  
memoryRemaining()
```

Description

An approximation to the total amount of memory currently available for future allocated objects, measured in bytes.

Returns

The amount of remaining memory in bytes.

newArray(Class, int)

Signature

```
public java.lang.Object  
newArray(java.lang.Class<?> type,  
         int number)  
throws IllegalArgumentException,  
        OutOfMemoryError,  
        SecurityException
```

Description

Allocate 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.

OutOfMemoryError when space in the memory area is exhausted.

SecurityException when the caller does not have permission to create a new instance.

Returns

A new array of class type, of number elements.

newInstance(Class)*Signature*

```
public T  
newInstance(java.lang.Class<T> type)  
throws IllegalAccessException,  
       IllegalArgumentException,  
       InstantiationException,  
       OutOfMemoryError,  
       ExceptionInInitializerError,  
       SecurityException
```

Description

Allocate 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 when type is null.

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.

ConstructorCheckedException a checked exception was thrown by the constructor.

OutOfMemoryError when space in the memory area is exhausted.

ExceptionInInitializerError when an unexpected exception has occurred in a static initializer.

SecurityException when the caller does not have permission to create a new instance.

Returns

A new instance of class type.

newInstance(Constructor, Object)*Signature*

```
public T  
newInstance(java.lang.reflect.Constructor<T> c,  
           java.lang.Object[] args)
```

throws `ExceptionInInitializerError`,
`IllegalAccessException`,
`IllegalArgumentException`,
`InstantiationException`,
`InvocationTargetException`,
`OutOfMemoryError`,
`SecurityException`

Description

Allocate 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.

`IllegalArgumentException` when c is null, or the args array does not contain the number of arguments required by c. A null value of args is treated like an array of length 0.

`InstantiationException` when the specified class object could not be instantiated. Possible causes are it is an interface, it is abstract, it is an array.

`InvocationTargetException` when the underlying constructor throws an exception.

`OutOfMemoryError` when space in the memory area is exhausted.

`SecurityException` 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

Query the size of the memory area. The returned value is the current size. Current size may be larger than initial size for those areas that are allowed to grow.

Returns

The size of the memory area in bytes.

executeInArea(Runnable)*Signature*

```
public void  
executeInArea(Runnable logic)  
throws IllegalArgumentException
```

Description

Execute the run method from the logic parameter using this memory area as the current allocation context. The effect of executeInArea on the scope stack is specified in the subclasses of MemoryArea.

Parameters

logic The runnable object whose run() method should be executed.

Throws

IllegalArgumentException when logic is null.

executeInArea(Supplier)*Signature*

```
public T  
executeInArea(java.util.function.Supplier<T> logic)
```

Description

Same as `executeInArea(Runnable)`³⁷ except that the executed method is called get and an object is returned.

Parameters

logic the object who's get method will be executed.

Returns

a result from the computation.

³⁷Section 11.3.2.3.2

executeInArea(BooleanSupplier)

Signature

```
public boolean  
executeInArea(BooleanSupplier logic)
```

Description

Same as `executeInArea(Runnable)`³⁸ except that the executed method is called `get` and a boolean is returned.

Parameters

`logic` the object who's `get` method will be executed.

Returns

a result from the computation.

executeInArea(IntSupplier)

Signature

```
public int  
executeInArea(IntSupplier logic)
```

Description

Same as `executeInArea(Runnable)`³⁹ except that the executed method is called `get` and an int is returned.

Parameters

`logic` the object who's `get` method will be executed.

Returns

a result from the computation.

executeInArea(LongSupplier)

Signature

```
public long  
executeInArea(LongSupplier logic)
```

Description

³⁸Section 11.3.2.3.2

³⁹Section 11.3.2.3.2

Same as `executeInArea(Runnable)`⁴⁰ 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)
```

Description

Same as `executeInArea(Runnable)`⁴¹ 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.

visitNestedMemory(MemoryAreaVisitor)

Signature

```
public R  
visitNestedMemory(javax.realtime.MemoryAreaVisitor<R> visitor)  
throws IllegalArgumentException
```

Description

A means of accessing all live nested memory areas contained in this memory area, even those to which no reference exists, such as a `javax.realtime.memory.PinnableMemory`⁴² that is pinned or another `javax.realtime.memory.ScopedMemory` that contains a `Schedulable`. The set may be concurrently modified by other tasks, but the view seen by the visitor may not be updated to reflect those changes. The following is guaranteed even when the set is disturbed by other tasks:

⁴⁰Section 11.3.2.3.2

⁴¹Section 11.3.2.3.2

⁴²Section 11.4.3.5

- 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.

Perform an action on all children scopes of this memory area, so long as the `MemoryAreaVisitor.visit(MemoryArea)`⁴³ method returns null. When that method returns an object, the visit is terminated and that object is returned by this method,

When execution of the visitor's visit method terminated abruptly by throwing an exception, then execution of `visitScopedChildren` also terminates abruptly by throwing the same exception.

Parameters

visitor determines the action to be performed on each of the children scopes.

Throws

`IllegalArgumentException` when visitor is null.

Returns

null when all elements were visited and some object of type R when the visit is forced to terminate at the end of visiting that element.

mayHoldReferenceTo

Signature

```
public boolean  
mayHoldReferenceTo()
```

Description

Determine whether an object A allocated in the memory area represented by this can hold a reference to an object B allocated in the current memory area.

Returns

true when B can be assigned to a field of A, otherwise false.

mayHoldReferenceTo(Object)

Signature

⁴³Section 11.3.1.1.1

```
public boolean  
mayHoldReferenceTo(Object value)
```

Description

Determine whether an object A allocated in the memory area represented by this can hold a reference to the object value.

Parameters

value is the object to test.

Returns

true when value can be assigned to a field of A, otherwise false.

11.3.2.4 MemoryParameters

Inheritance

```
java.lang.Object  
  javax.realtime.MemoryParameters
```

Interfaces

```
Cloneable  
Serializable
```

Description

Memory parameters can be given on the constructor of [RealtimeThread](#)⁴⁴ and [AsyncEventHandler](#)⁴⁵. These can be used both for the purposes of admission control by the scheduler and for the purposes of pacing the garbage collector (if any) to satisfy all of the schedulable memory allocation rates.

The limits in a MemoryParameters instance are enforced when a schedulable creates a new object, e.g., uses the new operation. When a schedulable exceeds its allocation or allocation rate limit, the error is handled as if the allocation failed because of insufficient memory. The object allocation throws an OutOfMemoryError.

When a reference to a MemoryParameters object is given as a parameter to a constructor, the MemoryParameters object becomes bound to the object being created. Changes to the values in the MemoryParameters object affect the constructed object. When given to more than one constructor, then changes to the values in the MemoryParameters object affect *all* of the associated objects. Note that this is a one-to-many relationship and *not* a many-to-many.

⁴⁴Section [5.3.2.2](#)

⁴⁵Section [8.3.3.5](#)

A MemoryParameters object may be shared, but that does not cause the memory budgets reflected by the parameter to be shared among the schedulables that are associated with the parameter object.

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

NO_MAX

```
public static final NO_MAX
```

Description

Specifies no maximum limit.

11.3.2.4.2 Constructors

MemoryParameters(long, long, boolean)

Signature

```
public  
MemoryParameters(long maxMemoryArea,  
                  long maxImmortal,  
                  boolean mayUseHeap)
```

Description

Create a MemoryParameters object with the given values.

Available since RTSJ 2.0

Parameters

maxMemoryArea A limit on the amount of memory the schedulable may allocate in its initial memory area. Units are in bytes. When zero, no allocation allowed in the memory area. To specify no limit, use NO_MAX.

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

mayUseHeap indicates whether or not the schedulable may use the heap. The default is true when an instance of this class is not provided.

Throws

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

MemoryParameters(long, long, long)

Signature

```
public
MemoryParameters(long maxMemoryArea,
                  long maxImmortal,
                  long allocationRate)
```

Description

Create a `MemoryParameters` object with the given values and `mayUseHeap`⁴⁶ returns true.

Parameters

maxMemoryArea A limit on the amount of memory the schedulable may allocate in its initial memory area. Units are in bytes. When zero, no allocation allowed in the memory area. To specify no limit, use `NO_MAX`.

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

allocationRate A limit on the rate of allocation in the heap. Units are in bytes per second of wall clock time. When `allocationRate` is zero, no allocation is allowed in the heap. To specify no limit, use `NO_MAX`. Measurement starts when the schedulable is first released for execution (not when it is constructed.) Enforcement of the allocation rate is an implementation option. When the implementation does not enforce allocation rate limits, it treats all non-zero allocation rate limits as `NO_MAX`.

Throws

`IllegalArgumentException` when any value other than positive. zero, or `NO_MAX` is passed as the value of `maxMemoryArea` or `maxImmortal`, or `allocationRate`.

⁴⁶Section [11.3.2.4.3](#)

MemoryParameters(long, long)

Signature

```
public  
MemoryParameters(long maxMemoryArea,  
                  long maxImmortal)
```

Description

Create a MemoryParameters object with the given values and `mayUseHeap`⁴⁷ returns false.

Parameters

`maxMemoryArea` A limit on the amount of memory the schedulable may allocate in its initial memory area. Units are in bytes. When zero, no allocation allowed in the memory area. To specify no limit, use `NO_MAX`.

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

Throws

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

MemoryParameters(boolean)

Signature

```
public  
MemoryParameters(boolean mayUseHeap)
```

Description

Create a MemoryParameters object with the given values.

Available since RTSJ 2.0

Parameters

`mayUseHeap` indicates whether or not the schedulable may use the heap. The default is true when an instance of this class is not provided.

⁴⁷Section 11.3.2.4.3

11.3.2.4.3 Methods

clone

Signature

```
public java.lang.Object  
clone()
```

Description

Return a clone of this. This method should behave effectively as if it constructed a new object with the visible values of this.

- The new object is in the current allocation context.
- clone does not copy any associations from this and it does not implicitly bind the new object to a SO.
-

Available since RTSJ 1.0.1

getAllocationRate

Signature

```
public long  
getAllocationRate()
```

Description

Gets 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 **NO_MAX**⁴⁸ then the allocation rate on the heap is uncontrolled.

getMaxImmortal

Signature

⁴⁸Section 11.3.2.4.1

```
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 `NO_MAX`⁴⁹ then there is no limit for allocation in immortal memory.

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`⁵⁰ then there is no limit for allocation in the initial memory area.

mayUseHeap

Signature

```
public boolean  
mayUseHeap()
```

Description

Determine whether or not this parameter object specifies that the heap may be used.

Returns

true when heap may be used and false otherwise.

⁴⁹Section 11.3.2.4.1

⁵⁰Section 11.3.2.4.1

setAllocationRate(long)

Signature

```
public javax.realtime.MemoryParameters  
setAllocationRate(long allocationRate)
```

Description

Sets the limit on the rate of allocation in the heap.

Parameters

allocationRate Units are in bytes per second of wall-clock time. When allocationRate is zero, no allocation is allowed in the heap. To specify no limit, use NO_MAX. Measurement starts when the schedulable starts (not when it is constructed.) Enforcement of the allocation rate is an implementation option. When the implementation does not enforce allocation rate limits, it treats all non-zero allocation rate limits as NO_MAX.

Throws

IllegalArgumentException when any value other than positive, zero, or NO_MAX is passed as the value of allocationRate.

Returns

this

11.3.2.5 SizeEstimator

Inheritance

```
java.lang.Object  
  javax.realtime.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\(SizeEstimator\)](#)

See Section [LTMemory.LTMemory\(SizeEstimator\)](#)

11.3.2.5.1 Constructors

SizeEstimator

Signature

```
public  
SizeEstimator()
```

Description

11.3.2.5.2 Methods

reserve(Class, int)

Signature

```
public void  
reserve(java.lang.Class<?> c,  
        int number)
```

Description

Take into account additional number instances of Class c when estimating the size of the [MemoryArea](#)⁵¹.

Parameters

⁵¹Section [11.3.2.3](#)

c The class to take into account.

number The number of instances of c to estimate.

Throws

IllegalArgumentException when c is null.

reserve(SizeEstimator, int)

Signature

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

Description

Take into account additional number instances of SizeEstimator size when estimating the size of the [MemoryArea](#)⁵².

Parameters

estimator The given instance of [SizeEstimator](#)⁵³.

number The number of times to reserve the size denoted by estimator.

Throws

IllegalArgumentException when estimator is null.

reserve(SizeEstimator)

Signature

```
public void  
reserve(SizeEstimator size)
```

Description

Take into account an additional instance of SizeEstimator size when estimating the size of the [MemoryArea](#)⁵⁴.

Parameters

size The given instance of SizeEstimator.

Throws

IllegalArgumentException when size is null.

⁵²Section [11.3.2.3](#)

⁵³Section [11.3.2.5](#)

⁵⁴Section [11.3.2.3](#)

reserveArray(int)

Signature

```
public void  
reserveArray(int length)
```

Description

Take into account an additional instance of an array of length reference values when estimating the size of the [MemoryArea](#)⁵⁵.

Parameters

length The number of entries in the array.

Throws

IllegalArgumentException when length is negative.

Available since RTSJ 1.0.1

reserveArray(int, Class)

Signature

```
public void  
reserveArray(int length,  
              java.lang.Class<?> type)
```

Description

Take into account an additional instance of an array of length primitive values when estimating the size of the [MemoryArea](#)⁵⁶.

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

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

Available since RTSJ 1.0.1

⁵⁵Section [11.3.2.3](#)

⁵⁶Section [11.3.2.3](#)

getEstimate*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.

11.4 javax.realtime.memory

11.4.1 Interfaces

11.4.1.1 PhysicalMemoryCharacteristic

Description

A tagging interface used to identify physical memory characteristics. Applications can give names to regions of memory that are described by [PhysicalMemoryRegion](#)⁵⁷. 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

Inheritance

java.lang.Object
 java.lang.Enum
 [javax.realtime.memory.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

DISABLED

public static final DISABLED

Description

⁵⁷Section [11.4.3.3](#)

WRITE_THROUGH

```
public static final WRITE_THROUGH
```

Description

WRITE_BACK

```
public static final WRITE_BACK
```

Description

11.4.2.1.2 Methods

values

Signature

```
public static javax.realtime.memory.PhysicalMemorySelector.CachingBehavior[]  
values()
```

Description

valueOf(String)

Signature

```
public static javax.realtime.memory.PhysicalMemorySelector.CachingBehavior  
valueOf(String name)
```

Description

11.4.2.2 PhysicalMemorySelector.PagingBehavior

Inheritance

java.lang.Object

java.lang.Enum

javafx.runtime.memory.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 DIRECT

Description

FIXED

public static final FIXED

Description

SWAPPABLE

public static final SWAPPABLE

Description

11.4.2.2.2 Methods

values

Signature

```
public static javax.realtime.memory.PhysicalMemorySelector.PagingBehavior[]  
values()
```

Description

valueOf(String)

Signature

```
public static javax.realtime.memory.PhysicalMemorySelector.PagingBehavior  
valueOf(String name)
```

Description

11.4.3 Classes

11.4.3.1 LTMemory

Inheritance

```
java.lang.Object  
  javax.realtime.MemoryArea  
    javax.realtime.memory.ScopedMemory  
      javax.realtime.memory.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 *initial* size. Execution time for allocation is allowed to vary when memory consumption is between the initial size and the maximum size for the area. Furthermore, the underlying system is not required to guarantee that memory between initial and maximum will always be available.

The memory area described by a LTMemory instance does not exist in the Java heap, and is not subject to garbage collection. Thus, it is safe to use a LTMemory object as the initial memory area for a [javax.realtime.Schedulable](#)⁵⁸

⁵⁸Section 6.3.1.3

instance which may not use the [javafx.runtime.HeapMemory](#)⁵⁹ or to enter the memory area using the [ScopedMemory.enter](#)⁶⁰ method within such an instance.

Enough memory must be committed by the completion of the constructor to satisfy the initial memory requirement. (Committed means that this memory must always be available for allocation). The initial memory allocation must behave, with respect to successful allocation, as if it were contiguous; i.e., a correct implementation must guarantee that any sequence of object allocations that could ever succeed without exceeding a specified initial memory size will always succeed without exceeding that initial memory size and succeed for any instance of LTMemory with that initial memory size.

Note, to ensure that all requested memory is available set initial and maximum to the same value.

Methods from LTMemory should be overridden only by methods that use `super`.

See [Section javafx.runtime.MemoryArea](#)

See [Section ScopedMemory](#)

See [Section javafx.runtime.Schedulable](#)

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

⁵⁹Section [11.3.2.1](#)

⁶⁰Section [11.4.3.6.1](#)

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 initial memory area. When logic is null, this constructor is equivalent to `LTMemory(long)`⁶¹.

Throws

`IllegalArgumentException` when size is less than zero.

`javax.realtime.StaticOutOfMemoryError` when there is insufficient memory for the LTMemory object or for the backing memory.

`javax.realtime.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, Runnable)`⁶² with argument list (size.getEstimate(), runnable).

Available since RTSJ 1.0.1

Parameters

size An instance of `javax.realtime.SizeEstimator`⁶³ used to give an estimate of the initial size. This memory must be committed before the completion of the constructor.

⁶¹Section 11.4.3.1.1

⁶²Section 11.4.3.1.1

⁶³Section 11.3.2.5

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)`⁶⁴.

Throws

`IllegalArgumentException` when size is null.

`javax.realtime.StaticOutOfMemoryError` when there is insufficient memory for the LTMemory object or for the backing memory.

`javax.realtime.IllegalAssignmentError` when storing logic in this would violate the assignment rules.

LTMemory(long)

Signature

```
public  
LTMemory(long size)
```

Description

Equivalent to `LTMemory(long, Runnable)`⁶⁵ with the argument list ((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.

`javax.realtime.StaticOutOfMemoryError` when there is insufficient memory for the LTMemory object or for the backing memory.

LTMemory(SizeEstimator)

Signature

```
public  
LTMemory(SizeEstimator size)
```

⁶⁴Section 11.4.3.1.1

⁶⁵Section 11.4.3.1.1

Description

Equivalent to `LTMemory(long, Runnable)`⁶⁶ with argument list `(size.getEstimate(), null)`.

Available since RTSJ 1.0.1

Parameters

size An instance of `javax.realtime.SizeEstimator`⁶⁷ 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.

`javax.realtime.StaticOutOfMemoryError` when there is insufficient memory for the `LTMemory` object or for the backing memory.

11.4.3.1.2 Methods

toString

Signature

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

Description

Create a string representation of this object. The string is of the form `(LTMemory) Scoped memory # num` where num uniquely identifies the `LTMemory` area.

Returns

A string representing the value of this.

11.4.3.2 PhysicalMemoryFactory

Inheritance

⁶⁶Section 11.4.3.1.1

⁶⁷Section 11.3.2.5

java.lang.Object

javax.realtime.memory.PhysicalMemoryFactory

Description

Both associate memory ranges, in the form of [PhysicalMemoryRegion](#)⁶⁸ instances with physical memory characteristics in the form of [PhysicalMemoryCharacteristic](#)⁶⁹ instances, and create memory areas in those modules.

Each physical memory module can have more than one 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 create.

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 are may not span more than one physical memory module. To find a memory range that supports PMC A and PMC B uses set intersection modules(A) \$ cap\$ modules(B)

Available since RTSJ 2.0

11.4.3.2.1 Constructors

PhysicalMemoryFactory

Signature

```
public  
PhysicalMemoryFactory()
```

Description

Create an empty factory, but when only one factor is required, use [getDefault](#)⁷⁰ instead.

⁶⁸Section [11.4.3.3](#)

⁶⁹Section [11.4.1.1](#)

⁷⁰Section [11.4.3.2.2](#)

11.4.3.2.2 Methods

getDefault

Signature

```
public static javax.realtime.memory.PhysicalMemoryFactory  
getDefault()
```

Description

associate(PhysicalMemoryCharacteristic, PhysicalMemoryRegion)

Signature

```
public void  
associate(PhysicalMemoryCharacteristic name,  
          PhysicalMemoryRegion module)  
throws IllegalArgumentException,  
        IllegalStateException
```

Description

Associates a programmer-defined name with a physical address range.

Parameters

name is the physical memory characteristic. e.g STATIC_RAM.

module is the object representing a range of contiguous physical addresses

Throws

IllegalArgumentException when either name or module is null

IllegalStateException when module overlaps a previously associated PhysicalMemoryRegion instance.

associate(PhysicalMemoryCharacteristic, PhysicalMemoryRegion)

Signature

```
public void  
associate(javax.realtime.memory.PhysicalMemoryCharacteristic[] names,  
          PhysicalMemoryRegion module)  
throws IllegalArgumentException,  
        IllegalStateException
```

Description

Associates an array of programmer-defined names with a physical address range.

Parameters

names is the array of physical memory characteristics. e.g { STATIC_RAM }.
module is the object representing a range of contiguous physical addresses

Throws

IllegalArgumentException when either names or module is null
IllegalStateException when module overlaps a previously associated PhysicalMemoryRegion instance.

associate(PhysicalMemoryCharacteristic, PhysicalMemoryRegion)

Signature

```
public static void  
associate(PhysicalMemoryCharacteristic name,  
          javax.realtime.memory.PhysicalMemoryRegion[] modules)  
throws IllegalArgumentException,  
        IllegalStateException
```

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

IllegalArgumentException when either name or modules is null
IllegalStateException when module overlaps a previously associated PhysicalMemoryRegion instance.

createImmortalMemory(PhysicalMemorySelector, long, Runnable)

Signature

```
public javax.realtime.ImmortalMemory  
createImmortalMemory(PhysicalMemorySelector selector,  
                      long size,  
                      Runnable logic)  
throws SecurityException,  
        SizeOutOfBoundsException,  
        UnsupportedPhysicalMemoryException,  
        MemoryTypeConflictException,  
        IllegalArgumentException
```

Description

Instantiate a [javax.realtime.ImmortalMemory⁷¹](#) object in a [PhysicalMemoryRegion⁷²](#) matching the [PhysicalMemoryCharacteristic⁷³](#) in selector and then with virtual memory parameters of selector applied.

Parameters

selector to use to choose the memory module and set the virtual mapping

size is the size of memory to be taken out of the selected module

logic the logic to execute on entry (may be null)

Throws

[SecurityException](#) 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 [PhysicalMemoryCharacteristic⁷⁴](#) 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.

[IllegalArgumentException](#) when size is less than zero.

Returns

⁷¹Section [11.3.2.2](#)

⁷²Section [11.4.3.3](#)

⁷³Section [11.4.1.1](#)

⁷⁴Section [11.4.1.1](#)

the new memory area

createLTMemory(PhysicalMemorySelector, long, Runnable)

Signature

```
public javax.realtime.memory.PinnableMemory  
createLTMemory(PhysicalMemorySelector selector,  
                long size,  
                Runnable logic)  
throws SecurityException,  
        SizeOutOfBoundsException,  
        UnsupportedPhysicalMemoryException,  
        MemoryTypeConflictException,  
        IllegalArgumentException
```

Description

Instantiate a [LTMemory](#)⁷⁵ object in a [PhysicalMemoryRegion](#)⁷⁶ matching the [PhysicalMemoryCharacteristic](#)⁷⁷ in selector and then with virtual memory parameters of selector applied.

Parameters

selector to use to choose the memory module and set the virtual mapping
size is the size of memory to be taken out of the selected module
logic the logic to execute on entry (may be null)

Throws

[SecurityException](#) 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](#)⁷⁸ 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.

⁷⁵Section [11.4.3.1](#)

⁷⁶Section [11.4.3.3](#)

⁷⁷Section [11.4.1.1](#)

⁷⁸Section [11.4.1.1](#)

IllegalArgumentException when size is less than zero.

Returns

the new memory area

createPinnableMemory(PhysicalMemorySelector, long, Runnable)

Signature

```
public javax.realtime.memory.PinnableMemory  
createPinnableMemory(PhysicalMemorySelector selector,  
                      long size,  
                      Runnable logic)  
  
throws SecurityException,  
        SizeOutOfBoundsException,  
        UnsupportedPhysicalMemoryException,  
        MemoryTypeConflictException,  
        IllegalArgumentException
```

Description

Instantiate a [PinnableMemory](#)⁷⁹ object in a [PhysicalMemoryRegion](#)⁸⁰ matching the [PhysicalMemoryCharacteristic](#)⁸¹ in selector and then with virtual memory parameters of selector applied.

Parameters

selector to use to choose the memory module and set the virtual mapping

size is the size of memory to be taken out of the selected module

logic the logic to execute on entry (may be null)

Throws

SecurityException 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](#)⁸² has been registered with this PhysicalMemoryFactory.

⁷⁹Section [11.4.3.5](#)

⁸⁰Section [11.4.3.3](#)

⁸¹Section [11.4.1.1](#)

⁸²Section [11.4.1.1](#)

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.

Returns

the new memory area

createStackedMemory(PhysicalMemorySelector, long, long, Runnable)

Signature

```
public javax.realtime.memory.StackedMemory  
createStackedMemory(PhysicalMemorySelector selector,  
                    long scopeSize,  
                    long backingSize,  
                    Runnable logic)  
  
throws SecurityException,  
        SizeOutOfBoundsException,  
        UnsupportedPhysicalMemoryException,  
        MemoryTypeConflictException,  
        IllegalArgumentException
```

Description

Instantiate a **StackedMemory**⁸³ object in a **PhysicalMemoryRegion**⁸⁴ matching the **PhysicalMemoryCharacteristic**⁸⁵ in selector and then with virtual memory parameters of selector applied.

Parameters

selector to use to choose the memory module and set the virtual mapping

scopeSize is the size of the scope to be created

backingSize is the size of the backing store to take out of the selected module

logic the logic to execute on entry (may be null)

Throws

SecurityException when the application does not have permissions to access physical memory or the given range of memory.

⁸³Section 11.4.3.7

⁸⁴Section 11.4.3.3

⁸⁵Section 11.4.1.1

`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`⁸⁶ 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.

`IllegalArgumentException` when `scopeSize` or `backingSize` is less than zero.

Returns

the new memory area

11.4.3.3 PhysicalMemoryRegion

Inheritance

`java.lang.Object`
`javax.realtime.memory.PhysicalMemoryRegion`

Description

Enable an application to define a range of physical memory addresses.

Available since RTSJ 2.0

11.4.3.3.1 Constructors

PhysicalMemoryRegion(long, long)

Signature

```
public
PhysicalMemoryRegion(long base,
                      long length)
```

Description

⁸⁶Section 11.4.1.1

Creates an instance representing a range of contiguous physical memory.

Parameters

base is a physical address

length is size of contiguous memory from that base

Throws

IllegalArgumentException when length is less than or equal to 0, or when base is less than 0 or when this module overlaps with another memory module.

javax.realtime.SizeOutOfBoundsException when base + length is greater than the physical address range of the processor

11.4.3.3.2 Methods

getBase

Signature

```
public long  
getBase()
```

Description

Gets the base address of the contiguous memory represented by this.

Returns

the base address

getLength

Signature

```
public long  
getLength()
```

Description

Gets the length of the contiguous memory represented by this.

Returns

the length

11.4.3.4 PhysicalMemorySelector

Inheritance

java.lang.Object
 javafx.runtime.memory.PhysicalMemorySelector

Description

Provides both 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.

Available since RTSJ 2.0

11.4.3.4.1 Constructors

PhysicalMemorySelector(PhysicalMemoryCharacteristic, PhysicalMemoryCharacteristic, CachingBehavior, PagingBehavior)

Signature

```
public  
PhysicalMemorySelector(javafx.runtime.memory.PhysicalMemoryCharacteristic[] request,  
                        javafx.runtime.memory.PhysicalMemoryCharacteristic[] reject,  
                        PhysicalMemorySelector.CachingBehavior caching,  
                        PhysicalMemorySelector.PagingBehavior paging)
```

Description

11.4.3.4.2 Methods

getSupportedCachingBehavior

Signature

```
public static javax.realtime.memory.PhysicalMemorySelector.CachingBehavior[]  
getSupportedCachingBehavior()
```

Description

Get the caching behaviors that are supported by this JVM

Returns

an array of the supported caching behaviors.

getSupportedPagingBehavior

Signature

```
public static javax.realtime.memory.PhysicalMemorySelector.PagingBehavior[]  
getSupportedPagingBehavior()
```

Description

Get the paging behaviors that are supported by this JVM

Returns

an array of the supported paging behaviors.

getRequestSet

Signature

```
public javax.realtime.memory.PhysicalMemoryCharacteristic[]  
getRequestSet()
```

Description

A getter for the PhysicalMemoryCharacteristic list to be requested

Returns

the PhysicalMemoryCharacteristic list

getRejectSet

Signature

```
public javax.realtime.memory.PhysicalMemoryCharacteristic[]  
getRejectSet()
```

Description

A getter for the PhysicalMemoryCharacteristic list to be excluded

Returns

the PhysicalMemoryCharacteristic list

getCachingBehavior

Signature

```
public javax.realtime.memory.PhysicalMemorySelector.CachingBehavior  
getCachingBehavior()
```

Description

A getter for the CachingBehavior to be requested

Returns

the CachingBehavior

getPagingPagingBehavior

Signature

```
public javax.realtime.memory.PhysicalMemorySelector.PagingBehavior  
getPagingPagingBehavior()
```

Description

A getter for the PagingBehavior to be requested

Returns

the PagingBehavior

11.4.3.5 PinnableMemory

Inheritance

java.lang.Object
 javafx.runtime.MemoryArea
 javafx.runtime.memory.ScopedMemory
 javafx.runtime.memory.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.

Open issue 11.4.1

more javadoc information needed for this class

End of issue 11.4.1

Available since RTSJ 2.0

11.4.3.5.1 Constructors

PinnableMemory(long)

Signature

```
public  
PinnableMemory(long size)  
throws IllegalArgumentException,  
      StaticOutOfMemoryError
```

Description

Create a scoped memory of fixed size that can be held open when no `javafx.runtime.Schedulable`⁸⁷ has it on its scoped memory stack.

Parameters

⁸⁷Section 6.3.1.3

size is the number of bytes in the memory area.

Throws

IllegalArgumentException when size is less than zero.

[javax.realtime.StaticOutOfMemoryError](#) when there is insufficient memory for the PinnableMemory object or for its backing memory.

PinnableMemory(SizeEstimator)

Signature

```
public  
PinnableMemory(SizeEstimator size)  
throws IllegalArgumentException,  
    StaticOutOfMemoryError
```

Description

Equivalent to [PinnableMemory\(long\)](#)⁸⁸ with size.getEstimate() as its argument.

Parameters

size is an estimator for determining the number of bytes in the memory area.

Throws

IllegalArgumentException when size is null.

[javax.realtime.StaticOutOfMemoryError](#) when there is insufficient memory for the PinnableMemory object or for its backing memory.

11.4.3.5.2 Methods

pin

Signature

```
public void  
pin()
```

Description

Prevent the contents from being freed.

⁸⁸Section [11.4.3.5.1](#)

unpin

Signature

```
public void  
unpin()
```

Description

Allow the contents to be freed the next time no [javafx.runtime.Schedulable](#)⁸⁹ or [javafx.runtime.AsyncBaseEvent](#)⁹⁰ is active withing the scope.

isPinned

Signature

```
public boolean  
isPinned()
```

Description

Determine whether the scope may be cleared on last exit.

Returns

true when yes, otherwise false.

getPinCount

Signature

```
public int  
getPinCount()
```

Description

Find out how many times the scope has been pinned, but not unpinned.

Returns

the number of outstanding pins.

⁸⁹Section [6.3.1.3](#)

⁹⁰Section [8.3.3.2](#)

joinPinned

Signature

```
public void  
joinPinned()  
throws InterruptedException
```

Description

Wait until the scope has been cleared and then pin it.

Throws

InterruptedException When this schedulable is interrupted by `javafx.realtime.RealtimeThread.interrupt()`⁹¹ or `javafx.realtime.AsynchronouslyInterruptedException.fire()`⁹² while waiting for the reference count to go to zero.

joinPinned(HighResolutionTime)

Signature

```
public void  
joinPinned(javafx.realtime.HighResolutionTime<T> limit)  
throws InterruptedException
```

Description

Wait until the scope has been cleared and then pin it, within a specified time frame.

Parameters

limit is the maximum time to wait

Throws

InterruptedException When this schedulable is interrupted by `javafx.realtime.RealtimeThread.interrupt()`⁹³ or `javafx.realtime.AsynchronouslyInterruptedException.fire()`⁹⁴ while waiting for the reference count to go to zero.

⁹¹Section 5.3.2.2.2

⁹²Section 8.3.2.1.2

⁹³Section 5.3.2.2.2

⁹⁴Section 8.3.2.1.2

joinPinnedAndEnter(Runnable)

Signature

```
public void  
joinPinnedAndEnter(Runnable logic)  
throws InterruptedException,  
    ScopedCycleException
```

Description

Wait until the scope has been cleared and then pin it and enter it.

Parameters

logic is the logic to execute upon entry

Throws

InterruptedException When this schedulable is interrupted by `javafx.realtime.RealtimeThread.interrupt()`⁹⁵ or `javafx.realtime.AsynchronouslyInterruptedException.fire()`⁹⁶ 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.

joinPinnedAndEnter(Runnable, HighResolutionTime)

Signature

```
public void  
joinPinnedAndEnter(Runnable logic,  
                    javafx.realtime.HighResolutionTime<T> limit)  
throws InterruptedException,  
    ScopedCycleException
```

Description

Wait until the scope has been cleared and then pin it and enter it, within a specified time frame.

Parameters

logic is the logic to execute upon entry

limit is the maximum time to wait.

Throws

⁹⁵Section 5.3.2.2.2

⁹⁶Section 8.3.2.1.2

InterruptedException When this schedulable is interrupted by `javafx.realtime.RealtimeThread.interrupt()`⁹⁷ or `javafx.realtime.AsynchronouslyInterruptedException.fire()`⁹⁸ 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.

joinPinnedAndEnter

Signature

```
public void
joinPinnedAndEnter()
throws InterruptedException,
    IllegalSchedulableStateException,
    ThrowBoundaryError,
    ScopedCycleException,
    MemoryAccessError
```

Description

Wait until the scope has been cleared and then pin it and enter it.

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 `javafx.realtime.IllegalAssignmentError`⁹⁹, so the JVM cannot be permitted to deliver the exception. The `javafx.realtime.ThrowBoundaryError`¹⁰⁰ 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.

InterruptedException When this schedulable is interrupted by `javafx.realtime.RealtimeThread.interrupt()`¹⁰¹ or `javafx.realtime.AsynchronouslyInterruptedException.fire()`¹⁰² while waiting for the reference count to go to zero.

IllegalSchedulableStateException when the caller is a Java thread, or when this method is invoked during finalization of objects in scoped memory and enter-

⁹⁷Section 5.3.2.2.2

⁹⁸Section 8.3.2.1.2

⁹⁹Section 15.2.3.2

¹⁰⁰Section 15.2.3.8

¹⁰¹Section 5.3.2.2.2

¹⁰²Section 8.3.2.1.2

ing 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.

joinPinnedAndEnter(HighResolutionTime)

Signature

```
public void
joinPinnedAndEnter(javax.realtime.HighResolutionTime<T> limit)
throws InterruptedException,
    IllegalSchedulableStateException,
    IllegalArgumentException,
    UnsupportedOperationException,
    ThrowBoundaryError,
    ScopedCycleException,
    MemoryAccessError
```

Description

Wait until the scope has been cleared and then pin it and enter it, within a specified time frame.

Parameters

limit is 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**¹⁰³, so the JVM cannot be permitted to deliver the exception. The **javax.realtime.ThrowBoundaryError**¹⁰⁴ 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.

InterruptedException When this schedulable is interrupted by **javax.realtime.RealtimeThread.interrupt()**¹⁰⁵ or **javax.realtime.AsynchronouslyInterruptedException.fire()**¹⁰⁶

¹⁰³Section 15.2.3.2

¹⁰⁴Section 15.2.3.8

¹⁰⁵Section 5.3.2.2.2

¹⁰⁶Section 8.3.2.1.2

while waiting for the reference count to go to zero.

IllegalSchedulableStateException 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 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.

UnsupportedOperationException when the wait operation is not supported using the clock associated with time.

11.4.3.6 ScopedMemory

Inheritance

```
java.lang.Object
  javax.realtime.MemoryArea
    javax.realtime.memory.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 object has access to the objects in the scoped memory.

A ScopedMemory area is a connection to a particular region of memory and reflects the current status of that memory. The object does not necessarily contain direct references to the region of memory. That is implementation dependent.

When a ScopedMemory area is instantiated, the object itself is allocated from the current memory allocation context, but the memory space that object represents (it's backing store) is allocated from memory that is not otherwise directly visible to Java code; e.g., it might be allocated with the C malloc function. This backing store behaves effectively as if it were allocated when the associated scoped memory object is constructed and freed at that scoped memory object's finalization.

The **ScopedMemory.enter**¹⁰⁷ method of ScopedMemory is one mechanism used to make a memory area the current allocation context. The other mechanism

¹⁰⁷Section 11.4.3.6.1

for activating a memory area is making it the initial memory area for a realtime thread or async event handler. Entry into the scope is accomplished, for example, by calling the method:

```
public void enter(Runnable logic)
```

where `logic` is a instance of `Runnable` whose `run()` method represents the entry point of the code that will run in the new scope. Exit from the scope occurs between the time the `runnable.run()` method completes and the time control returns from 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 `javafx.realtime`.

11.4.3.6.1 Methods

enter

Signature

```
public void  
enter()
```

Description

Associate this memory area with the current schedulable for the duration of the execution of the `run()` method of the instance of `Runnable` given in the constructor. During this period of execution, this memory area becomes the default allocation context until another default allocation context is selected (using `enter`, or `executeInArea`¹⁰⁸) or the `enter` method exits.

Throws

`javafx.realtime.ScopedCycleException` when this invocation would break the single parent rule.

¹⁰⁸Section 11.4.3.6.1

`javafx.runtime.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 `javafx.runtime.IllegalAssignmentError`¹⁰⁹, so the JVM cannot be permitted to deliver the exception. The `javafx.runtime.ThrowBoundaryError`¹¹⁰ 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` `IllegalArgumentException` when the caller is a schedulable and a null value for logic was supplied when the memory area was constructed.

`MemoryAccessError` `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

Associate this memory area with the current schedulable for the duration of the execution of the `run()` method of the given `Runnable`. During this period of execution, this memory area becomes the default allocation context until another default allocation context is selected (using `enter`, or `executeInArea`¹¹¹) or the `enter` method exits.

Parameters

logic logic The `Runnable` object whose `run()` method should be invoked.

Throws

`javafx.runtime.ScopedCycleException` when this invocation would break the single parent rule.

`javafx.runtime.ThrowBoundaryError` Thrown when the JVM needs to propagate an exception allocated in this scope to (or through) the memory area of the

¹⁰⁹Section 15.2.3.2

¹¹⁰Section 15.2.3.8

¹¹¹Section 11.4.3.6.1

caller. Storing a reference to that exception would cause an `javax.realtime.IllegalAssignmentError`¹¹², so the JVM cannot be permitted to deliver the exception. The `javax.realtime.ThrowBoundaryError`¹¹³ 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` `IllegalArgumentException` when the caller is a schedulable and logic is null.

enter(Supplier)

Signature

```
public T  
enter(java.util.function.Supplier<T> logic)
```

Description

Same as `enter(Runnable)`¹¹⁴ except that the executed method is called `get` and an object is returned.

Parameters

logic the object who's `get` method will be executed.

Returns

a result from the computation.

enter(BooleanSupplier)

Signature

```
public boolean  
enter(BooleanSupplier logic)
```

Description

Same as `enter(Runnable)`¹¹⁵ except that the executed method is called `get` and a boolean is returned.

¹¹²Section 15.2.3.2

¹¹³Section 15.2.3.8

¹¹⁴Section 11.4.3.6.1

¹¹⁵Section 11.4.3.6.1

Parameters

logic the object who's get method will be executed.

Returns

a result from the computation.

enter(IntSupplier)*Signature*

```
public int  
enter(IntSupplier logic)
```

Description

Same as `enter(Runnable)`¹¹⁶ except that the executed method is called get and an int is returned.

Parameters

logic the object who's get method will be executed.

Returns

a result from the computation.

enter(LongSupplier)*Signature*

```
public long  
enter(LongSupplier logic)
```

Description

Same as `enter(Runnable)`¹¹⁷ except that the executed method is called get and a long is returned.

Parameters

logic the object who's get method will be executed.

Returns

a result from the computation.

¹¹⁶Section 11.4.3.6.1

¹¹⁷Section 11.4.3.6.1

enter(DoubleSupplier)

Signature

```
public double  
enter(DoubleSupplier logic)
```

Description

Same as `enter(Runnable)`¹¹⁸ 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)
```

Description

Execute the `run` method from the `logic` parameter using this memory area as the current allocation context. This method behaves as if it moves the allocation context down the scope stack to the occurrence of this.

Parameters

`logic` The `Runnable` object whose `run()` method should be executed.

Throws

`IllegalThreadStateException` when the caller is a Java thread.

`InaccessibleAreaException` when the memory area is not in the schedulable's scope stack.

`IllegalArgumentException` when the caller is a schedulable and `logic` is null.

executeInArea(Supplier)

Signature

¹¹⁸Section 11.4.3.6.1

```
public T  
executeInArea(java.util.function.Supplier<T> logic)
```

Description

Same as `executeInArea(Runnable)`¹¹⁹ 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)
```

Description

Same as `executeInArea(Runnable)`¹²⁰ 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)
```

Description

Same as `executeInArea(Runnable)`¹²¹ except that the executed method is called `get` and an int is returned.

¹¹⁹Section 11.4.3.6.1

¹²⁰Section 11.4.3.6.1

¹²¹Section 11.4.3.6.1

Parameters

logic the object who's get method will be executed.

Returns

a result from the computation.

executeInArea(LongSupplier)*Signature*

```
public long  
executeInArea(LongSupplier logic)
```

Description

Same as `executeInArea(Runnable)`¹²² except that the executed method is called `get` and a long is returned.

Parameters

logic the object who's get method will be executed.

Returns

a result from the computation.

executeInArea(DoubleSupplier)*Signature*

```
public double  
executeInArea(DoubleSupplier logic)
```

Description

Same as `executeInArea(Runnable)`¹²³ except that the executed method is called `get` and a double is returned.

Parameters

logic the object who's get method will be executed.

Returns

a result from the computation.

¹²²Section 11.4.3.6.1

¹²³Section 11.4.3.6.1

getPortal

Signature

```
public java.lang.Object  
getPortal()
```

Description

Return a reference to the portal object in this instance of ScopedMemory.

Assignment rules are enforced on the value returned by getPortal as if the return value were first stored in an object allocated in the current allocation context, then moved to its final destination.

Throws

`javafx.realtime.IllegalAssignmentError` when a reference to the portal object cannot be stored in the caller's allocation context; that is, when this is "inner" relative to the current allocation context or not on the caller's scope stack.

`IllegalThreadStateException` when the caller is a Java thread.

Returns

A reference to the portal object or null when there is no portal object. The portal value is always set to null when the contents of the memory are deleted.

getReferenceCount

Signature

```
public int  
getReferenceCount()
```

Description

Returns the reference count of this ScopedMemory.

Note, a reference count of 0 reliably means that the scope is not referenced, but other reference counts are subject to artifacts of lazy/eager maintenance by the implementation.

Returns

The reference count of this ScopedMemory.

join

Signature


```
public void  
join()  
throws InterruptedException
```

Description

Wait until the reference count of this ScopedMemory goes down to zero. Return immediately when the memory is unreferenced.

Throws

InterruptedException When this schedulable is interrupted by `javax.realtime.RealtimeThread.interrupt()`¹²⁴ or `javax.realtime.AsynchronouslyInterruptedException.fire()`¹²⁵ while waiting for the reference count to go to zero.
IllegalThreadStateException when the caller is a Java thread.

join(HighResolutionTime)

Signature

```
public void  
join(javax.realtime.HighResolutionTime<?> time)  
throws InterruptedException
```

Description

Wait at most until the time designated by the time parameter for the reference count of this ScopedMemory to drop to zero. Return immediately when the memory area is unreferenced.

Since the time is expressed as a `javax.realtime.HighResolutionTime`¹²⁶, 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

¹²⁴Section 5.3.2.2.2

¹²⁵Section 8.3.2.1.2

¹²⁶Section 9.3.1.2

RelativeTime) the wait is bounded by a the specified interval from some time between the time join is called and the time it starts waiting for the reference count to reach zero.

Throws

InterruptedException When this schedulable is interrupted by `javafx.realtime.RealtimeThread.interrupt()`¹²⁷ or `javafx.realtime.AsynchronouslyInterruptedException.fire()`¹²⁸ while waiting for the reference count to go to zero.

IllegalThreadStateException when the caller is a Java thread.

IllegalArgumentException when the caller is a schedulable and time is null.

UnsupportedOperationException when the wait operation is not supported using the clock associated with time.

joinAndEnter

Signature

```
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 enter the `ScopedMemory` and execute the run method from logic passed in the constructor. When no instance of `Runnable` was passed to the memory area's constructor, the method throws `IllegalArgumentException` immediately.

When multiple threads are waiting in `joinAndEnter` family methods for a memory area, at most *one* of them will be released each time the reference count goes to zero.

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

¹²⁷Section 5.3.2.2.2

¹²⁸Section 8.3.2.1.2

InterruptedException When this schedulable is interrupted by `javax.realtime.RealtimeThread.interrupt()`¹²⁹ or `javax.realtime.AsynchronouslyInterruptedException.fire()`¹³⁰ 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.

javax.realtime.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`¹³¹, so the JVM cannot be permitted to deliver the exception. The `javax.realtime.ThrowBoundaryError`¹³² 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.

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

```
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

¹²⁹Section 5.3.2.2.2

¹³⁰Section 8.3.2.1.2

¹³¹Section 15.2.3.2

¹³²Section 15.2.3.8

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 `javax.realtime.HighResolutionTime`¹³³, 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`¹³⁴.

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` 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`¹³⁵, so the JVM cannot be permitted to deliver the exception. The `javax.realtime.ThrowBoundaryError`¹³⁶ is allocated in the current allocation context and contains information about the exception it replaces.

`InterruptedException` When this schedulable is interrupted by `javax.realtime.RealtimeThread.interrupt()`¹³⁷ or `javax.realtime.AsynchronouslyInterruptedException.fire()`¹³⁸ 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.

¹³³Section 9.3.1.2

¹³⁴Section 11.4.3.6.1

¹³⁵Section 15.2.3.2

¹³⁶Section 15.2.3.8

¹³⁷Section 5.3.2.2.2

¹³⁸Section 8.3.2.1.2

`javax.realtime.ScopedCycleException` when the caller is a schedulable and this invocation would break the single parent rule.

`IllegalArgumentException` when the caller is a schedulable, and time is null or no non-null logic value was supplied to the memory area's constructor.

`UnsupportedOperationException` when the wait operation is not supported using the clock associated with time.

`MemoryAccessError` when calling schedulable may not use the heap and this memory area's logic value is allocated in heap memory.

joinAndEnter(Runnable)

Signature

```
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 `javax.realtime.RealtimeThread.interrupt()`¹³⁹ or `javax.realtime.AsynchronouslyInterruptedException.fire()`¹⁴⁰ while waiting for the reference count to go to zero.

¹³⁹Section 5.3.2.2.2

¹⁴⁰Section 8.3.2.1.2

`IllegalThreadStateException` when the caller is a Java thread, or when this method is invoked during finalization of objects in scoped memory and entering this scoped memory area would force deletion of the SO that triggered finalization. This would include the scope containing the SO, and the scope (if any) containing the scope containing the SO.

`javafx.runtime.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 `javafx.runtime.IllegalAssignmentError`¹⁴¹, so the JVM cannot be permitted to deliver the exception. The `javafx.runtime.ThrowBoundaryError`¹⁴² is allocated in the current allocation context and contains information about the exception it replaces.

`javafx.runtime.ScopedCycleException` when this invocation would break the single parent rule.

`IllegalArgumentException` when the caller is a schedulable and logic is null.

joinAndEnter(Supplier)

Signature

```
public T  
joinAndEnter(java.util.function.Supplier<T> logic)
```

Description

Same as `joinAndEnter(Runnable)`¹⁴³ 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.

joinAndEnter(BooleanSupplier)

Signature

```
public boolean  
joinAndEnter(BooleanSupplier logic)
```

¹⁴¹Section 15.2.3.2

¹⁴²Section 15.2.3.8

¹⁴³Section 11.4.3.6.1

Description

Same as `joinAndEnter(Runnable)`¹⁴⁴ except that the executed method is called `get` and a boolean is returned.

Parameters

`logic` the object who's `get` method will be executed.

Returns

a result from the computation.

joinAndEnter(IntSupplier)

Signature

```
public int  
joinAndEnter(IntSupplier logic)
```

Description

Same as `joinAndEnter(Runnable)`¹⁴⁵ except that the executed method is called `get` and an int is returned.

Parameters

`logic` the object who's `get` method will be executed.

Returns

a result from the computation.

joinAndEnter(LongSupplier)

Signature

```
public long  
joinAndEnter(LongSupplier logic)
```

Description

Same as `joinAndEnter(Runnable)`¹⁴⁶ except that the executed method is called `get` and a long is returned.

Parameters

`logic` the object who's `get` method will be executed.

Returns

a result from the computation.

¹⁴⁴Section 11.4.3.6.1

¹⁴⁵Section 11.4.3.6.1

¹⁴⁶Section 11.4.3.6.1

joinAndEnter(DoubleSupplier)

Signature

```
public double  
joinAndEnter(DoubleSupplier logic)
```

Description

Same as [joinAndEnter\(Runnable\)](#)¹⁴⁷ 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.

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 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](#)¹⁴⁸, 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\(Runnable\)](#)¹⁴⁹.

¹⁴⁷Section [11.4.3.6.1](#)

¹⁴⁸Section [9.3.1.2](#)

¹⁴⁹Section [11.4.3.6.1](#)

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

`InterruptedException` When this schedulable is interrupted by `javax.realtime.RealtimeThread.interrupt()`¹⁵⁰ or `javax.realtime.AsynchronouslyInterruptedException.fire()`¹⁵¹ 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.

`javax.realtime.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 a `javax.realtime.IllegalAssignmentError`¹⁵², so the JVM cannot be permitted to deliver the exception. The `javax.realtime.ThrowBoundaryError`¹⁵³ is preallocated and saves information about the exception it replaces.

`javax.realtime.ScopedCycleException` when the caller is a schedulable and this invocation would break the single parent rule.

`IllegalArgumentException` when the caller is a schedulable and `time` or `logic` is null.

`UnsupportedOperationException` when the wait operation is not supported using the clock associated with `time`.

joinAndEnter(Supplier, HighResolutionTime)

Signature

public P

¹⁵⁰Section 5.3.2.2.2

¹⁵¹Section 8.3.2.1.2

¹⁵²Section 15.2.3.2

¹⁵³Section 15.2.3.8

```
joinAndEnter(java.util.function.Supplier<P> logic,  
             javax.realtime.HighResolutionTime<?> time)
```

Description

Same as `joinAndEnter(Runnable, HighResolutionTime)`¹⁵⁴ 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.

joinAndEnter(BooleanSupplier, HighResolutionTime)

Signature

```
public boolean  
joinAndEnter(BooleanSupplier logic,  
             javax.realtime.HighResolutionTime<?> time)
```

Description

Same as `joinAndEnter(Runnable, HighResolutionTime)`¹⁵⁵ 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.

joinAndEnter(IntSupplier, HighResolutionTime)

Signature

```
public int  
joinAndEnter(IntSupplier logic,  
             javax.realtime.HighResolutionTime<?> time)
```

Description

¹⁵⁴Section 11.4.3.6.1

¹⁵⁵Section 11.4.3.6.1

Same as `joinAndEnter(Runnable, HighResolutionTime)`¹⁵⁶ 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.

joinAndEnter(LongSupplier, HighResolutionTime)

Signature

```
public long  
joinAndEnter(LongSupplier logic,  
              javax.realtime.HighResolutionTime<?> time)
```

Description

Same as `joinAndEnter(Runnable, HighResolutionTime)`¹⁵⁷ 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.

joinAndEnter(DoubleSupplier, HighResolutionTime)

Signature

```
public double  
joinAndEnter(DoubleSupplier logic,  
              javax.realtime.HighResolutionTime<?> time)
```

Description

Same as `joinAndEnter(Runnable, HighResolutionTime)`¹⁵⁸ except that the executed method is called `get` and a `double` is returned.

Parameters

`logic` the object whose `get` method will be executed.

¹⁵⁶Section 11.4.3.6.1

¹⁵⁷Section 11.4.3.6.1

¹⁵⁸Section 11.4.3.6.1

Returns

a result from the computation.

getParent*Signature*

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

Description

Return a reference to this scopes parent scope (e.g., its parent in the single-parent-rule tree).

Returns

a reference to the next outer scoped memory region on the caller's scope stack.

- When there is no outer scoped memory and the primordial parent is heap memory, return a reference to this.
- When there is no outer scoped memory and the primordial parent is immortal, or when this is unreferenced and unpinned, return null

Problem. The single-parent tree is RTT-independent except for the primordial scope. The type of the primordial scope is RTT-dependent. What should we do about that? When called from a RTT that has entered this, the above rules make some sense, but what if the caller has not even entered the scope, should we throw an exception? Or just return null? I think the right solution is to return this whatever the type of the primordial scope. The app can then know that null means the scope is not pinned and not referenced, and this means the parent is either heap or immortal. At that point, the app can learn what it wants to know by just finding what memory area contains the scope object.

Available since RTSJ 2.0

visitNestedMemory(MemoryAreaVisitor)*Signature*

```
public R  
visitNestedMemory(javax.realtime.MemoryAreaVisitor<R> visitor)
```

Description

A means of accessing all live nested memory areas contained in this memory area, even those to which no reference exists, such a [javax.realtime.memory.PinnableMemory](#)¹⁵⁹ that is pinned or another `javax.realtime.memory.ScopedMemory` that contains a `Schedulable`. The set may be concurrently modified by other tasks, but the view seen by the visitor may not be updated to reflect those changes. The following is guarantees even when the set is disturbed by other tasks:

- the visitor shall visit no member more than once,
- it shall visit only scopes that were a member of the set at some time during the enumeration of the set, and
- it shall visit all the scopes that are not deleted during the execution of the visitor.

Perform an action on all children scopes of this memory area, so long as the [MemoryAreaVisitor.visit\(MemoryArea\)](#)¹⁶⁰ method returns null. When that method returns an object, the visit is terminated and that object is returned by this method,

When execution of the visitor's visit method terminated abruptly by throwing an exception, then execution of `visitScopedChildren` also terminates abruptly by throwing the same exception.

Throws

`IllegalArgumentException` when visitor is null.

newArray(Class, int)

Signature

```
public java.lang.Object  
newArray(java.lang.Class<?> type,  
          int number)
```

Description

Allocate an array of the given type in this memory area. This method may be concurrently used by multiple threads.

Parameters

type `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 `number` The number of elements in the new array.

¹⁵⁹Section [11.4.3.5](#)

¹⁶⁰Section [11.3.1.1.1](#)

Throws

IllegalArgumentException IllegalArgumentException when number is less than zero, type is null, or type is java.lang.Void.TYPE.

javax.realtime.StaticOutOfMemoryError null

IllegalThreadStateException when the caller is a Java thread.

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*

```
public T  
newInstance(java.lang.Class<T> type)  
throws IllegalAccessException,  
        InstantiationException
```

Description

Allocate an object in this memory area. This method may be concurrently used by multiple threads.

Parameters

type type The class of which to create a new instance.

Throws

IllegalAccessException IllegalAccessException The class or initializer is inaccessible.

IllegalArgumentException IllegalArgumentException when type is null.

ExceptionInInitializerError ExceptionInInitializerError when an unexpected exception has occurred in a static initializer.

javax.realtime.StaticOutOfMemoryError null

InstantiationException 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 is a Java thread.

InaccessibleAreaException when the memory area is not in the schedulable's scope stack.

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 IllegalAccessException,  
       InstantiationException,  
       InvocationTargetException
```

Description

Allocate an object in this memory area. This method may be concurrently used by multiple threads.

Parameters

`c` `T c` The constructor for the new instance.
`args` `args` An array of arguments to pass to the constructor.

Throws

`IllegalAccessException` `IllegalAccessException` when the class or initializer is inaccessible under Java access control.

`InstantiationException` `InstantiationException` when the specified class object could not be instantiated. Possible causes are it is an interface, it is abstract, it is an array.

`javax.realtime.StaticOutOfMemoryError` `null`

`IllegalArgumentException` `IllegalArgumentException` when `c` is null, or the `args` array does not contain the number of arguments required by `c`. A null value of `args` is treated like an array of length 0.

`IllegalThreadStateException` when the caller is a Java thread.

`InvocationTargetException` `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 javax.realtime.memory.ScopedMemory  
setPortal(Object object)
```

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

`IllegalThreadStateException` when the caller is a Java Thread.

`javax.realtime.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.

Returns

this

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

11.4.3.7 StackedMemory

Inheritance


```
java.lang.Object
  javax.realtime.MemoryArea
    javax.realtime.memory.ScopedMemory
      javax.realtime.memory.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 which case it is returned to its enclosing scope's backing store when the object is finalized. Implementations are not required to return the space occupied by backing stores taken from the global backing store when their associated StackedMemory object is finalized.

These backing store semantics divide instances of StackedMemory into two categories:

- *host* — this denotes a StackedMemory with an object allocation area created in a new backing store, allocated either from the global store or from a parent StackedMemory's backing store, 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 objects, *root*. A root StackedMemory is a host StackedMemory created with a backing store drawn directly from the global backing store, created in an allocation context of some type other than StackedMemory.

Allocations from a StackedMemory object allocation area are guaranteed to run in time linear in the size of the allocation. All memory for the backing store must be reserved at object construction time.

StackedMemory memory areas have two additional stacking constraints in addition to the single parent rule, designed to enable fragmentation-free manipu-

lation:

- 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.StaticOutOfMemoryError`. This overhead must be bounded by a constant.

Available since RTSJ 2.0

11.4.3.7.1 Constructors

StackedMemory(long, long, Runnable)

Signature

```
public
  StackedMemory(long scopeSize,
                long backingSize,
                Runnable logic)
```

Description

Create a host StackedMemory with an object allocation area and backing store of the specified sizes, bound to the specified Runnable. The backing store is

allocated from the currently active memory area when it is also a StackedMemory, and the global backing store otherwise. The object allocation area is allocated from the backing store.

Parameters

scopeSize Size of the allocation area

backingSize Size of the total backing store

logic Runnable to be entered using this as its current memory area when `enter()`¹⁶¹ is called.

Throws

IllegalArgumentException when either scopeSize or backingSize is less than zero, or when scopeSize is too large to be allocated from a backing store of size backingSize.

`javafx.runtime.StaticOutOfMemoryError` when there is insufficient memory available to reserve the requested backing store.

StackedMemory(SizeEstimator, SizeEstimator, Runnable)

Signature

```
public  
StackedMemory(SizeEstimator scopeSize,  
               SizeEstimator backingSize,  
               Runnable logic)
```

Description

Equivalent to `StackedMemory(long, long, Runnable)`¹⁶² with argument list (scopeSize.getEstimate(), backingSize.getEstimate(), runnable).

Parameters

scopeSize SizeEstimator indicating the size of the object allocation area

backingSize SizeEstimator indicating the size of the total backing store

logic Runnable to be entered using this as its current memory area when `enter()`¹⁶³ is called.

Throws

¹⁶¹Section 11.4.3.7.2

¹⁶²Section 11.4.3.7.1

¹⁶³Section 11.4.3.7.2

`IllegalArgumentException` 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()`.

`javax.realtime.StaticOutOfMemoryError` when there is insufficient memory available to reserve the requested backing store.

StackedMemory(long, long)

Signature

```
public  
StackedMemory(long scopeSize,  
               long backingSize)
```

Description

Equivalent to `StackedMemory(long, long, Runnable)`¹⁶⁴ with argument list (`scopeSize`, `backingSize`, null).

Parameters

`scopeSize` Size of the allocation area

`backingSize` Size of the total backing store

Throws

`IllegalArgumentException` when either `scopeSize` or `backingSize` is less than zero, or when `scopeSize` is too large to be allocated from a backing store of size `backingSize`.

`javax.realtime.StaticOutOfMemoryError` when there is insufficient memory available to reserve the requested backing store.

StackedMemory(SizeEstimator, SizeEstimator)

Signature

```
public  
StackedMemory(SizeEstimator scopeSize,  
               SizeEstimator backingSize)
```

Description

¹⁶⁴Section 11.4.3.7.1

Equivalent to `StackedMemory(long, long, Runnable)`¹⁶⁵ with argument list (scopeSize, getEstimate(), backingSize.getEstimate(), null).

Parameters

scopeSize SizeEstimator indicating the size of the object allocation area

backingSize SizeEstimator indicating the size of the total backing store

Throws

IllegalArgumentException when either scopeSize or backingSize is null, or when scopeSize.getEstimate() is too large to be allocated from a backing store of size backingSize.getEstimate().

`javafx.runtime.StaticOutOfMemoryError` when there is insufficient memory available to reserve the requested backing store.

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

logic Runnable to be entered using this as its current memory area when `enter()`¹⁶⁶ is called.

Throws

IllegalStateException when the parent memory area is not a StackedMemory, or when the parent StackedMemory already has a child that is also a guest StackedMemory.

IllegalArgumentException when scopeSize is less than zero.

¹⁶⁵Section 11.4.3.7.1

¹⁶⁶Section 11.4.3.7.2

`javax.realtime.StaticOutOfMemoryError` 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.

StackedMemory(SizeEstimator, Runnable)

Signature

```
public  
StackedMemory(SizeEstimator scopeSize,  
               Runnable logic)
```

Description

Equivalent to `StackedMemory(long, Runnable)`¹⁶⁷ with argument list (scopeSize, getEstimate(), runnable).

Parameters

scopeSize `SizeEstimator` indicating the size of the object allocation area
logic `Runnable` to be entered using this as its current memory area when `enter()`¹⁶⁸ is called.

Throws

`IllegalStateException` when the parent memory area is not a `StackedMemory`, or when the parent `StackedMemory` already has a child that is also a guest `StackedMemory`.

`IllegalArgumentException` when scopeSize is null.

`javax.realtime.StaticOutOfMemoryError` 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.

StackedMemory(long)

Signature

```
public  
StackedMemory(long scopeSize)
```

Description

¹⁶⁷Section 11.4.3.7.1

¹⁶⁸Section 11.4.3.7.2

Equivalent to `StackedMemory(long, Runnable)`¹⁶⁹ with argument list (scopeSize, null).

Parameters

scopeSize Size of the allocation area

Throws

`IllegalStateException` when the parent memory area is not a `StackedMemory`, or when the parent `StackedMemory` already has a child that is also a guest `StackedMemory`.

`IllegalArgumentException` when scopeSize is less than zero.

`javax.realtime.StaticOutOfMemoryError` 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.

StackedMemory(SizeEstimator)

Signature

```
public  
StackedMemory(SizeEstimator scopeSize)
```

Description

Equivalent to `StackedMemory(long, Runnable)`¹⁷⁰ with argument list (scopeSize.getEstimate(), null).

Parameters

scopeSize `SizeEstimator` indicating the size of the object allocation area

Throws

`IllegalStateException` when the parent memory area is not a `StackedMemory`, or when the parent `StackedMemory` already has a child that is also a guest `StackedMemory`.

`IllegalArgumentException` when scopeSize is null.

`javax.realtime.StaticOutOfMemoryError` 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.

¹⁶⁹Section 11.4.3.7.1

¹⁷⁰Section 11.4.3.7.1

11.4.3.7.2 Methods

resize(long)

Signature

```
public void  
resize(long scopeSize)
```

Description

Change the size of the object allocation area for this scope. This method may be used to either grow or shrink the allocation area when there are no objects allocated in the scope and no `Schedulable` object has this area as its current allocation context. It may be used to shrink the allocation area down to the size of its current usage when the calling `Schedulable` object is the only object that has this area on its scope stack and there are no guest `StackedMemory` object allocation areas created after this area in the same backing store but not yet finalized.

Parameters

scopeSize The new allocation area size for this scope

Throws

`IllegalStateException` when the caller is not permitted to perform the requested adjustment or there are additional guest `StackedMemory` allocation areas after this one in the backing store.

`javax.realtime.StaticOutOfMemoryError` when the remaining backing store is insufficient for the requested adjustment.

getMaximumSize

Signature

```
public long  
getMaximumSize()
```

Description

Get the maximum size this memory area can attain. The value returned by this function is the maximum size that can currently be passed to `resize(long)`¹⁷¹ without triggering an `OutOfMemoryException`.

¹⁷¹Section 11.4.3.7.2

Returns

The maximum size attainable.

enter*Signature*

```
public void  
enter()
```

Description

Associate this memory area with the current `Schedulable` object for the duration of the `run()` method of the instance of `Runnable` given in this object's constructor. During this period of execution, this memory area becomes the default allocation context until another default allocation context is selected.

This method may only be called from the memory area in which this scope was created.

Throws

`IllegalStateException` when the currently active memory area is a `StackedMemory` and is not the area in which this scope was created, or the current memory area is not a `StackedMemory` and this `StackedMemory` is not a root area.

`ThrowBoundaryError` null

`IllegalThreadStateException` `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.

`MemoryAccessError` `MemoryAccessError` `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*

```
public void  
enter(Runnable logic)
```

Description

Associate this memory area with the current `Schedulable` object for the duration of the `run()` method of the given `Runnable`. During this period of execution, this memory area becomes the default allocation context until another default allocation context is selected.

This method may only be called from the memory area in which this scope was created.

Throws

`IllegalStateException` when the currently active memory area is a `StackedMemory` and is not the area in which this scope was created, or the current memory area is not a `StackedMemory` and this `StackedMemory` is not a root area.

`ThrowBoundaryError` null

`IllegalThreadStateException` `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.

`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 enter the `ScopedMemory` and execute the `run` method from logic passed in the constructor. When no instance of `Runnable` was passed to the memory area's constructor, the method throws `IllegalArgumentException` immediately.

When multiple threads are waiting in `joinAndEnter` family methods for a memory area, at most *one* of them will be released each time the reference count goes to zero.

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 `javax.realtime.RealtimeThread.interrupt()`¹⁷² or `javax.realtime.AsynchronouslyInterruptedException.fire()`¹⁷³ 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.

`javax.realtime.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`¹⁷⁴, so the JVM cannot be permitted to deliver the exception. The `javax.realtime.ThrowBoundaryError`¹⁷⁵ 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.

`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

```
public void  
joinAndEnter(javax.realtime.HighResolutionTime<?> time)  
throws InterruptedException
```

Description

¹⁷²Section 5.3.2.2.2

¹⁷³Section 8.3.2.1.2

¹⁷⁴Section 15.2.3.2

¹⁷⁵Section 15.2.3.8

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 `javax.realtime.HighResolutionTime`¹⁷⁶, 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`¹⁷⁷.

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` 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`¹⁷⁸, so the JVM cannot be permitted to deliver the exception. The `javax.realtime.ThrowBoundaryError`¹⁷⁹ is allocated in the current allocation context and contains information about the exception it replaces.

`InterruptedException` When this schedulable is interrupted by `javax.realtime.RealtimeThread.interrupt()`¹⁸⁰ or `javax.realtime.AsynchronouslyInterruptedException.fire()`¹⁸¹ while waiting for the reference count to go to zero.

`IllegalThreadStateException` when the caller is a Java thread, or when this method is

¹⁷⁶Section 9.3.1.2

¹⁷⁷Section 11.4.3.7.2

¹⁷⁸Section 15.2.3.2

¹⁷⁹Section 15.2.3.8

¹⁸⁰Section 5.3.2.2.2

¹⁸¹Section 8.3.2.1.2

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.ScopedCycleException when the caller is a schedulable and this invocation would break the single parent rule.

IllegalArgumentException when the caller is a schedulable, and time is null or no non-null logic value was supplied to the memory area's constructor.

UnsupportedOperationException when the wait operation is not supported using the clock associated with time.

MemoryAccessError when calling schedulable may not use the heap and this memory area's logic value is allocated in heap memory.

joinAndEnter(Runnable)

Signature

```
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 `javafx.realtime.RealtimeThread.interrupt()`¹⁸² or `javafx.realtime.AsynchronouslyInterruptedException.fire()`¹⁸³ 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.

`javafx.realtime.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 `javafx.realtime.IllegalAssignmentError`¹⁸⁴, so the JVM cannot be permitted to deliver the exception. The `javafx.realtime.ThrowBoundaryError`¹⁸⁵ is allocated in the current allocation context and contains information about the exception it replaces.

`javafx.realtime.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

```
public void
joinAndEnter(Runnable logic,
              javafx.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 `javafx.realtime.HighResolutionTime`¹⁸⁶, this method is an accurate timer with nanosecond granularity. The actual resolution

¹⁸²Section 5.3.2.2.2

¹⁸³Section 8.3.2.1.2

¹⁸⁴Section 15.2.3.2

¹⁸⁵Section 15.2.3.8

¹⁸⁶Section 9.3.1.2

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)`¹⁸⁷.

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

`InterruptedException` When this schedulable is interrupted by `javax.realtime.RealtimeThread.interrupt()`¹⁸⁸ or `javax.realtime.AsynchronouslyInterruptedException.fire()`¹⁸⁹ 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.

`javax.realtime.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 a `javax.realtime.IllegalAssignmentError`¹⁹⁰, so the JVM cannot be permitted to deliver the exception. The `javax.realtime.ThrowBoundaryError`¹⁹¹ is preallocated and saves information about the exception it replaces.

`javax.realtime.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.

¹⁸⁷Section 11.4.3.7.2

¹⁸⁸Section 5.3.2.2.2

¹⁸⁹Section 8.3.2.1.2

¹⁹⁰Section 15.2.3.2

¹⁹¹Section 15.2.3.8

UnsupportedOperationException when the wait operation is not supported using the clock associated with time.

11.5 The Rationale

11.5.1 The Scoped Memory Model

Languages that employ automatic reclamation of blocks of memory allocated in what is conventionally called the heap by program logic also typically use an algorithm called a garbage collector. Garbage collection algorithms and implementations vary in the amount of nondeterminacy 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 fireable, but its reference count is not incremented until the thread is started or the async 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 affect on EISMA finalization. Only another execution context can prevent finalization by the handler at release end.

11.5.2 The Physical Memory Model

Embedded systems may have many different types of directly addressable memory available to them. Each type has its own characteristics [2] that determine whether it is

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 costs per byte. SRAM has faster access times and is more expensive. Both DRAM and SRAM are example of asynchronous memory, SDRAM and SSRAM are their synchronized counterparts. Another important difference is that DRAM requires periodic refresh operations, which may interfere with execution time determinism.
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 times the same location can be overwritten. EEPROMs are expensive to manufacture, fast to read but slow to write.
 - (b) FLASH memory – this is nonvolatile that is writable at the sector level. Like EEPROM there are limits on how many times the same location can be overwritten and they are fast to read but slow to write. Flash memory is cheaper to manufacture than EEPROM.

Some embedded systems may have multiple types of random-access memory, and multiple ways of accessing memory. For instance, there may be a small amount of very fast RAM on the processor chip, memory that is on the same board as the processor, memory that may be added and removed from the system dynamically, memory that is accessed across a bus, access to memory that is mediated by a cache, access where the cache is partially disabled so all stores are “write through”, memory that is demand paged, and other types of memory and memory-access attributes only limited by physics and the imagination of electrical engineers. Some of these memory types will have no impact on the programmer, others will.

Individual computers are often targeted at a particular application domain. This domain will often dictate the cost and performance requirements, and therefore, the memory type used. Some embedded systems are highly optimized and need to explore different options in memory to meet their performance requirements. Here are five example scenarios.

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 lookaside buffer (TLB). A few very frequently accessed objects are to be locked in the cache and a larger number of objects that have jitter requirements can be TLB-locked to avoid TLB faults.
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 100 ms or so. To exploit this, the objects used by each phase of our program can be collected in a separate bank of this special memory.

To be clear, few embedded systems are this aggressive in their hardware optimization. The majority of embedded systems have only ROM, RAM, and maybe flash memory. Configuration-controlled memory attributes (such as page locking, and TLB behavior) are more common.

As well as having different types of memory, many computers map input and output devices so that their registers can be accessed as if they were resident within the computer memory (see Section 12.2.1). 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 platform¹⁹². 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.

11.5.2.1 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

¹⁹²This is reflected by the OS support provided. For example, most POSIX systems only offer programs a choice of demand paged or page-locked memory.

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 can interact in order 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 PMM. However, the programming of filters is difficult for the following reasons.

1. Physical memory type filters included a memory allocation function that must respond to allocation requests with whether a requested range of physical memory is free and when it was not, the physical address of the next free physical memory of the requested type. This is complex because requests for compound types of physical memory must find a free segment that satisfies all attributes of the compound type.
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 were 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.

11.5.2.2 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. It, therefore, cannot have detailed knowledge of the underlying memory architecture. It is only concerned with the standard RAM provided to it by the host operating system.

The design of 2.0 model is based on two constraints.

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 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 can be used any objects store therein may contain references to objects in volatile memory. Although these memory types are prohibited from being used as backing stores, they contain objects of primitive Java types and be accessed via the RTSJ Raw Memory facilities (see Section 12.2.1).
2. Any API must delegates detailed knowledge of the memory architecture to the programmer of the specific embedded system to be implemented. There is less requirement for portability here, as embedded systems are usually optimized for their host environment. The model assumes that the programmer is aware of the memory map, either through some native operating system interface¹⁹³ 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,

¹⁹³For 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/>

```

1 PhysicalMemoryCharacteristic sram = new PhysicalMemoryCharacteristic({});
2 PhysicalMemoryCharacteristic[] characteristics =
3   new PhysicalMemoryCharacteristic[]{ sram };
4 PhysicalMemorySelector selector =
5   new PhysicalMemorySelector(null, null, WRITE_THROUGH, FIXED);
6 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.

11.5.2.3 An example

Consider an example of a system that has a SRAM physical memory module configured at a physical base address of 0x10000000 and of length 0x20000000. Another module (base address of 0xA0000000 and of length 0x10000000) also supports SRAM, but this module has been configured so that it saves power by sleeping when not in use. The following subsections illustrate how the embedded programmer informs the PMM about the structure during the program's initialization phase, and how the memory may be subsequently used after this. The example assumes that the PMM supports the virtual memory characteristics defined above.

11.5.2.3.1 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.

```

1 public class SRAMType implements PhysicalMemoryCharacteristic {}

```

The initialization method must now create instances of the `PhysicalMemoryRegion` class to represent the physical memory module memory modules to represent

```

1 PhysicalMemoryRegion staticRam =
2   new PhysicalMemoryRegion(0x10000000L, 0x100000000L);
3 PhysicalMemoryRegion staticSleepableRam =

```

```
4  new PhysicalMemoryRegion(0xA0000000L, 0x100000000L);
```

It then creates names for the characteristics that the program wants to associate with each memory module.

```
1  PhysicalMemoryCharacteristic STATIC_RAM = new MyMemoryType();
2  PhysicalMemoryCharacteristic AUTO_SLEEPABLE = new MyMemoryType();
```

It then informs the PMM of the appropriate associations:

```
1  PhysicalMemoryFactory factory = PhysicalMemoryFactory.getDefault();
2  factory.associate(STATIC_RAM, staticRam);
3  factory.associate(STATIC_RAM, staticSleepableRam);
4  factory.associate(AUTO_SLEEPABLE, staticSleepableRam);
```

Once this is done, the program can now create a selector with the required properties. In this case, it is for some SRAM that must be auto sleepable.

```
1  PhysicalMemoryCharacteristic [] PMC =
2  new PhysicalMemoryCharacteristic[2];
3  PMC[0] = STATIC_RAM;
4  PMC[1] = AUTO_SLEEPABLE;
5
6  PhysicalMemorySelector selector =
7  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.

11.5.2.3.2 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 straight forward. Here is an example.

```
1  ImmortalMemory IM = factory.createImmortalMemory(selector, 0x1000);
2  IM.enter(new Runnable()
3  {
4      public void run()
5      {
6          // The code executing here is running with its allocation
7          // context set to a physical immortal memory area that is
```

```
8      // mapped to RAM which is auto sleepable.  
9      // Any objects created will be placed in that  
10     // part of physical memory.  
11     }  
12  });
```

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

Devices and Triggering

Interacting with the external environment in a timely manner is an important requirement for realtime, embedded systems. From an embedded systems' perspective, all interactions with the physical world are performed by input and output devices. Hence, the problem is one of controlling and monitoring of devices. This is an area insufficiently addressed by other Java standards. A conventional Java Virtual Machine is not designed to support device access and interrupt handling. Programs that need this functionality must resort to code written in another language and called via the Java Native Interface (JNI). This specification addresses the problem by providing APIs for interrupt handling and direct memory access without resorting to JNI.

In contrast to earlier versions of this specification, 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.

In execution environment 1, interaction with the embedded environment is usually via operating system calls using Java's connection-oriented APIs. The Java program will typically have no direct access to the I/O devices. Although some limited access to physical memory may be provided, it is unlikely that interrupts can be directly handled. However, asynchronous interaction with the environment is still possible, for example, via POSIX signals.

In execution environments 2, 3, and 4, the Java program may be able to directly

access devices and handle interrupts.

A device can be anything from a simple set of registers wired to sensors and actuators to a full processor performing some fixed task. The interface to a device is usually through a set of device registers. Depending on the I/O architecture of the processor, the programmer can either access these registers via predetermined memory location (called *memory mapped I/O*) or via special assembler instructions (called *port-mapped I/O*).

A computer system with processing devices can be considered to be a collection of parallel threads. The device ‘thread’ can communicate and synchronize with the tasks executing inside the main processor either by having the main processor poll registers of the device or via a signal from the device. This signal is usually referred to as an interrupt. All high-level models of device programming must provide [3]

1. facilities for representing, addressing and manipulating device registers; and
2. a suitable representation of interrupts (if interrupts are to be handled).

Version 1.0 of the RTSJ went some way towards supporting this model through the notion of *happenings* and the *raw memory* access facilities. Unfortunately, happenings were under defined and the mechanisms for physical and raw memory were overly complex with no clear delineation of the separations of concerns between application developers and JVM implementers.

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](#).

12.1 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 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 bit of code that is executed when an interrupt happens. This code runs above the normal priorities 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 12.1.1 (elb)

Pull in some relevant definitions from the JMM.

End of issue 12.1.1

12.2 Semantics

The classes in this Chapter are part of the Device Module introduced in Section 3.2.2.3 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 provides 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.

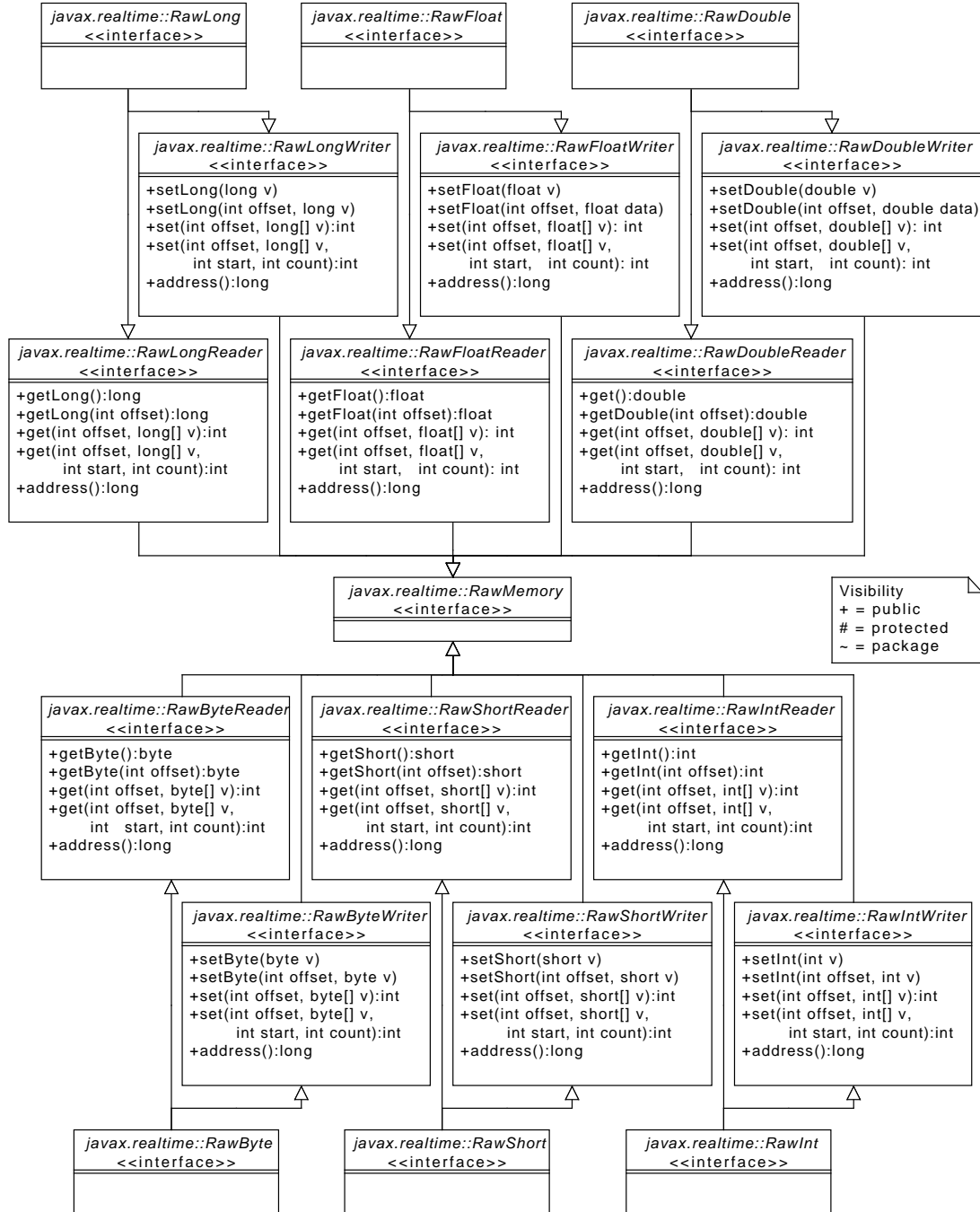
12.2.1 Raw Memory

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.

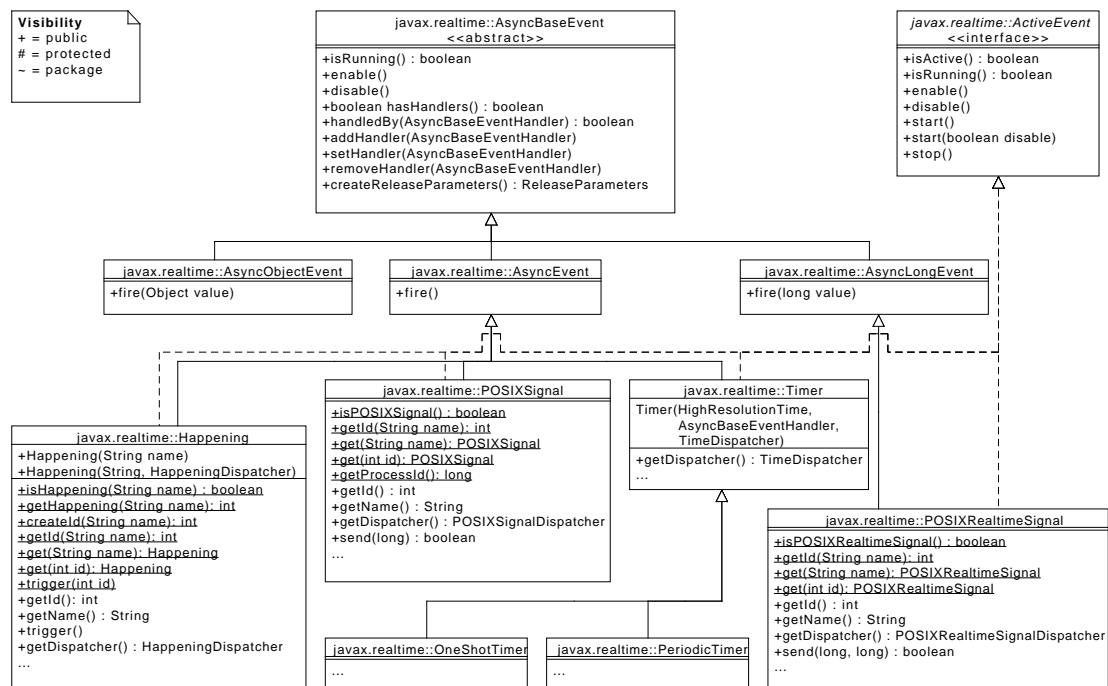
1. Each area of memory supporting raw memory access is identified by a subclass of `RawMemoryRegion`.
 - (a) The raw memory region `RawMemoryFactory.MEMORY_MAPPED_REGION` facilitates access to memory locations that are outside the main memory used by the JVM. It is used to access input and output device registers when such registers are memory mapped.

Figure 12.1: Raw Memory Interface



- (b) The raw memory region `RawMemoryFactory.IO_PORT_MAPPED_REGION` facilitates access to locations that are outside the main memory used by the JVM. It is used to access input and output device registers when such registers are port-based and can only be accessed by special hardware instructions.
- (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.
- Access to raw memory is controlled by implementation-defined objects, called *accessor objects*. These implement specification-defined interfaces (e.g., `RawByte`, `RawShort`, `RawInt`, etc.) and are created by implementation-defined factory objects. Each factory implements the `RawMemoryRegionFactory` interface, and is identified by its `RawMemoryRegion`.
 - The `RawMemoryFactory` class defines the applications programmers interface to the raw memory facilities.
 - The `RawMemoryRegionFactory` interface defines the interface that all factories must support for creating accessor objects.

Figure 12.2: Event Classes



12.2.1.1 Raw Memory Region

Raw memory is designed to support arbitrary I/O address spaces. The simplest is through the processor address space and is accessible via standard memory access instructions, such as load and store. This provide access to memory mapped I/O devices, but there are others address spaces as well. Each of these address spaces is referred to as a *Raw Memory Region*.

There are two raw memory regions that can be supported generically. Memory mapped I/O is one. The other is port mapped I/O. The most common instance is the I/O space provided by Intel x86 compatible processors through their **in** and **out** instructions. The memory mapped I/O raw memory region must be supported by all implementations, but the port mapped I/O raw memory region must only be supported on processors that support it.

All other raw memory regions are optional and may be provided by a system integrator or an application developer. The API provides an interface, `RawMemoryRegionFactory`, that can be implemented to provide a means of creating accessor objects for that region. These additional regions can be anything from an I/O space provided by a memory mapped device, using memory mapped I/O to implement it, to a purely synthetic I/O space to emulated hardware that has not yet been built.

Each raw memory region is identified by its raw memory region object. These “types” are defined by instances of `RawMemoryRegion: RawMemoryFactory.MEMORY_MAPPED_REGION` for memory mapped devices and `RawMemoryFactory.IO_PORT_MAPPED_REGION` for port mapped devices for processors that have instructions for reading and writing an I/O bus directly. The instances are used to get accessors of a region instead of using a `RawMemoryRegionFactory` directly.

12.2.1.2 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 I^2C raw memory region by implementing a factory for it using a memory mapped I^2C controller.

12.2.1.3 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 were contiguous.

Since stride is designed to support mapping devices that have a smaller word size than the host machine, the implementation is allowed to assume that the padding between values is “do not care” data, and can be overwritten arbitrarily.

12.2.2 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 is a bit different than for other `java.nio` mechanisms.

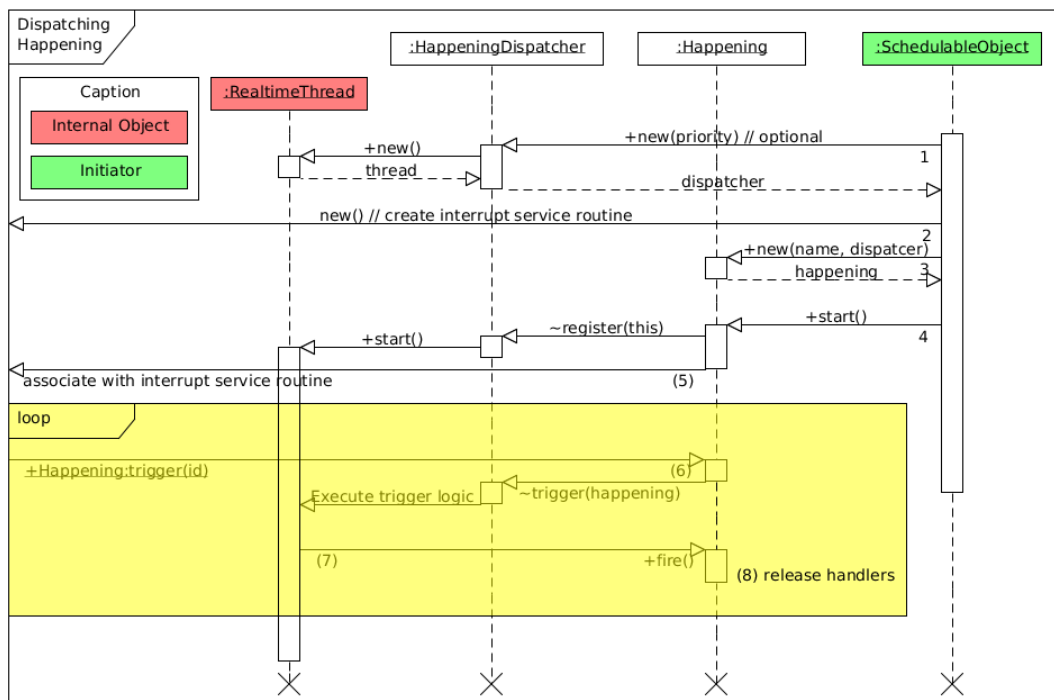
These differences are covered with a special byte buffer factory: `DMAByteBufferFactory`. An instance of this factory can produce direct byte buffers within a given memory range. This range can be chosen by the programmer to be within the range of a given DMA controller. The factory also provides methods 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 `DMAByteBufferFactory` class also provides static methods for ensuring that Java-generated changes to DMA-mapped memory buffers are visible to native code, and vice-versa.

12.2.3 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 handler is provided by the virtual machine and asynchronous events provide the second level event handling. For external events and additional clocks, where the programmer needs to be able to define new instances and provide for their triggering, additional classes are provided to manage both the first level, as well as the second level handling. In all cases, the user can control the priority and affinity of the dispatching between the first level and second level handling.

Figure 12.3: Happening State Transition Diagram



12.2.3.1 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 triggerable from an external event, such as an interrupt, the Happening class also implements [ActiveEvent](#). As with other active

events, `Happening` has its own dispatcher class: `HappeningDispatcher`. There is a default happening dispatcher that is used when none is provided at creation time, otherwise, the programmer can provide one to change the priority and affinity of dispatching.

Normally, happenings are triggered either from an `InterruptServiceRoutine` or from JNI code. For the later, the interface provides a means of linking a happening by name. This enables native code to get a handle for triggering a happening without have a direct reference. The given name must follow the Java naming conventions. A happening name defined outside of this specification should not begin with `java` or `javax`.

Figure 12.3 illustrates the sequence of actions necessary for defining and using a `Happening`. When using an application-defined dispatcher, it must be created first (1). When using an `InterruptServiceRoutine` to trigger the happening, it may be created before (2) or after the happening is create. After creating the happening (3), the happening must be started to be registered with it dispatcher to be triggered from native code. Of course, the JVM must have direct access to an interrupt, either by being directly bound in the kernel or by some other means, such as a system call, for setting up user-space device drivers. Only after both an `InterruptServiceRoutine` is registered and a `Happening` with the same name is started, can that happening be triggered (6–8).

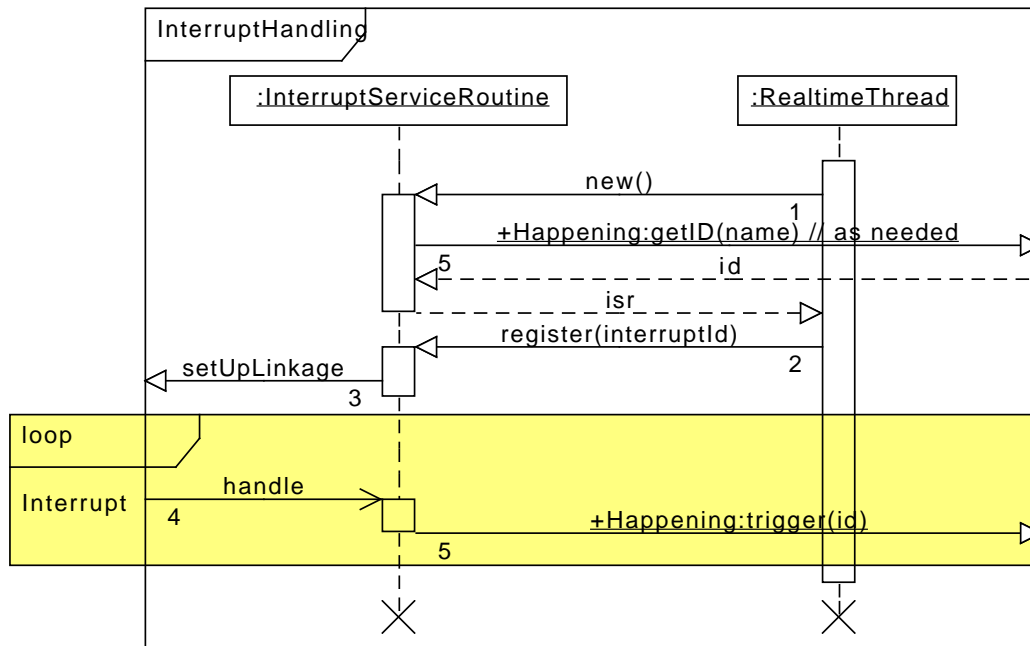
There are three main differences between this mechanism and the string based API.

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, a native code function that permits native application code to trigger the happening.
3. Finally, `Happening` is a subclass of `AsyncEvent` just as with `Timer` instead of having a happening attached to an `AsyncEvent`.

12.2.4 Interrupt Service Routines

In Java-based systems, JNI is typically used to transfer control between the assembler/C *interrupt service routine* (ISR) and the program. 2.0 of the RTSJ supports the possibility of the ISR containing Java code. This is clearly an area where it is difficult to maintain the portability goal of Java. Furthermore, not all RTSJ deployments can support `InterruptServiceRoutine`. A JVM that runs in user space does not generally have access to interrupts.

Figure 12.4: Interrupt servicing



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 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 imple-

mentation 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 method must be synchronized with a priority high enough to lock out the requisite interrupts. This synchronized uses priority ceiling emulation to inhibit the corresponding interrupt (and all lower priority interrupts). The default allocation context of the handle method is the memory area passed during construction.
Any exception propagated from the handle method shall be caught by the JVM and ignored.
9. Code running in the context of an ISR may only attempt to acquire a lock that has priority ceiling emulation as its monitor control policy. The behavior is undefined, when an ISR attempt to acquire a lock that has a monitor control policy other than priority ceiling emulation.

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 interrupt priority;
3. associated with the interrupt lines is a (logical) interrupt vector that contains the addresses of the ISRs;
4. the processor has instructions that allow interrupts from a particular line to be disabled/masked irrespective of whether (or the type of) device attached;
5. disabling interrupts from a specific line may disables the interrupts from lines of lower priority;
6. a device can be connected to an arbitrary interrupt line;
7. when an interrupt is signalled on an interrupt line by a device, the processor uses the identity of the interrupt line to index into the interrupt vector and jumps to the address of the ISR; the hardware automatically disables further interrupts (either of the same priority and lower or, possibly, all interrupts);

8. on return from the ISR, interrupts are automatically re-enabled.

For each of the interrupt, the RTSJ has an associated hardware priority that can be used to set the ceiling of an ISR object. The RTSJ virtual machine may use this to disable the interrupts from the associated interrupt line and lower priority interrupts, when it is executing a synchronized method of the interrupt-handling object. On a multicore system, the situation is more complex, since there may be other cores available to handle other interrupts, even at lower priorities, and some other locking mechanism may be necessary as well.

Though synchronization is not required in general, it is required to enforce visibility of changes made to any variables shared between some normal Schedulable and a handle method. For the handle method, this may be done automatically by the hardware interrupt handling mechanism or it may require added support from the realtime Java virtual machine. However, for clarity of the model, RTSJ recommends that the handle method should be defined as synchronized.

Support for interrupt handling is encapsulated in the `InterruptServiceRoutine` abstract class that has two main methods. The first is the `final register` method that will register an instance of the class with the system so that the appropriate interrupt vector can be initialized. The second is the abstract `handle` method that provides the code to be executed in response to the interrupt occurring. An individual real-time JVM may place restrictions of the code that can be written in this method. The process is illustrated in Figure 12.4, and is described below.

1. The ISR is created by some application real-time thread.
2. The created ISR is registered with the JVM, the interrupt id is passed as a parameter.
3. As part of the registration process, some internal interface is used to set up the code that will set the underlying interrupt vectors to some C/assembly code that will provide the necessary linkage to allow the callback to the Java handler.
4. When the interrupt occurs, the handler is called.

In order to integrate further the interrupt handling with the Java application, the `handle` method may trigger a happening or fire an event.

Typically an implementation of the RTSJ that supports first-level interrupt handling will document the following items.

1. For each interrupt, its identifying integer value, the priority at which the interrupt occurs and whether it can be inhibited or not, and the effects of registering ISRs to non-inhabitable 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.

12.3 javax.realtime.device

12.3.1 Interfaces

12.3.1.1 RawByte

Interfaces

[javax.realtime.device.RawByteReader](#)

[javax.realtime.device.RawByteWriter](#)

Description

A marker for an object that can be used to access to a single byte. Read and write access to that byte is checked by the factory that creates the instance; therefore, no access checking is provided by this interface, only bounds checking.

Available since RTSJ 2.0

12.3.1.2 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 transfered in a single atomic operation. Groups of bytes may be transfered together; however, this is not required.

Objects of this type are created with the method [RawMemoryFactory.createRawByteReader](#)¹ and [RawMemoryFactory.createRawByte](#)². Each object references a range of elements in the [RawMemoryRegion](#)³ starting at the *base address* provided to the factory method. The size provided to the factor method determines the number of elements accessable.

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

¹Section [12.3.2.6.3](#)

²Section [12.3.2.6.3](#)

³Section [12.3.2.7](#)

Available since RTSJ 2.0

12.3.1.2.1 Methods

getBytes

Signature

```
public byte  
getBytes()
```

Description

Get the value at the first position referenced by this instance, i.e., the value at its start address. This operation must be atomic with respect to all other raw memory accesses to the address.

Returns

the value at the *base address*.

getBytes(int)

Signature

```
public byte  
getBytes(int offset)  
throws OffsetOutOfBoundsException
```

Description

Get the value at the address: base address + offset x stride x 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
```

Description

Fill values with elements from this instance, where the *n*th element is at the address: *base address* + (*offset*+*n*) x *stride* x *element size* in bytes. Only the bytes in the intersection of the start and end of values and the *base address* and the end of the memory region are transferred. When an exception is thrown, no data is transferred.

Parameters

offset of the first byte in the memory region to transfere
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 actual 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

Fill values from index start with elements from this instance, where the nth element is at the address: base address + (offset+n) x stride x element size in bytes. The number of bytes transfered is the minimum of count, the *size* of the memory region minus offset, and length of values minus start. When an exception is thrown, no data is transfered.

Parameters

offset of the first byte in the memory region to transfere
values the array to receive the bytes
start the first index in array to fill
count the maximum number of bytes to copy

Throws

[OffsetOutOfBoundsException](#) when offset is negative or either offset or offset + count is greater than or equal to the size of this raw memory area.
[ArrayIndexOutOfBoundsException](#) when start is negative or either start or start + count is greater than or equal to the size of values.
[NullPointerException](#) when values is null or count is negative.

Returns

the number of bytes actually transfered.

12.3.1.3 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 transfered in a single atomic operation. Groups of bytes may be transfered together; however, this is not required.

Objects of this type are created with the method [RawMemoryFactory.createRawByteWriter](#)⁴ and [RawMemoryFactory.createRawByte](#)⁵. Each object references a range of elements in the [RawMemoryRegion](#)⁶ starting at the *base address* provided to the factory method. The size provided to the factor method determines the number of elements accessible.

⁴Section [12.3.2.6.3](#)

⁵Section [12.3.2.6.3](#)

⁶Section [12.3.2.7](#)

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

12.3.1.3.1 Methods

setByte(byte)

Signature

```
public void  
setByte(byte value)
```

Description

Set the value at the first position referenced by this instance, i.e., the value at its start address. This operation must be atomic with respect to all other raw memory accesses to the address.

Parameters

value is the new value for the element.

setByte(int, byte)

Signature

```
public void  
setByte(int offset,  
        byte value)  
throws OffsetOutOfBoundsException
```

Description

Set the value of the n^{th} element referenced by this instance, where n is offset and the address is *base address* + $offset * size\ of\ Byte$. 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 is the new value for the element.

Throws

OffsetOutOfBoundsException when offset is negative or greater than or equal to the number of elements in the raw memory region.

set(int, byte)

Signature

```
public int  
set(int offset,  
    byte[] values)  
throws OffsetOutOfBoundsException,  
        NullPointerException
```

Description

Copy from values to the memory region from index start, to elements where the nth element is at the address: base address + (offset+n) x stride x element size in bytes. Only the bytes in the intersection of values and the end of the memory region are transferred. When an exception is thrown, no data is transferred.

Parameters

offset of first byte in the memory region to be set.

values is 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

Copy values to the memory region, where offset is first byte in the memory region to write and start is the first index in values from which to read. The number of bytes transfered is the minimum of count, the *size* of the memory region minus offset, and length of values minus start. When an exception is thrown, no data is transfered.

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 transfered.

12.3.1.4 RawDouble

Interfaces

`javax.realtime.device.RawDoubleReader`
`javax.realtime.device.RawDoubleWriter`

Description

A marker for an object that can be used to access to a single double. Read and write access to that double is checked by the factory that creates the instance; therefore, no access checking is provided by this interface, only bounds checking.

Available since RTSJ 2.0

12.3.1.5 RawDoubleReader

Interfaces

[javafx.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](#)⁷ and [RawMemoryFactory.createRawDouble](#)⁸. Each object references a range of elements in the [RawMemoryRegion](#)⁹ starting at the *base address* provided to the factory method. The size provided to the factor method determines the number of elements 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

12.3.1.5.1 Methods

getDouble

Signature

```
public double  
getDouble()
```

Description

Get the value at the first position referenced by this instance, i.e., the value at its start address. This operation must be atomic with respect to all other raw memory accesses to the address.

⁷Section [12.3.2.6.3](#)

⁸Section [12.3.2.6.3](#)

⁹Section [12.3.2.7](#)

Returns

the value at the *base address*.

getDouble(int)*Signature*

```
public double  
getDouble(int offset)  
throws OffsetOutOfBoundsException
```

Description

Get the value at the address: base address + offset x stride x 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 OffsetOutOfBoundsException,  
        NullPointerException
```

Description

Fill values with elements from this instance, where the nth element is at the address: base address + (offset+n) x stride x element size in bytes. Only the doubles in the intersection of the start and end of values and the *base address* and the end of the memory region are 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

Fill values from index start with elements from this instance, where the nth element is at the address: base address + (offset+n) x stride x element size in bytes. The number of bytes 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 transfered.

12.3.1.6 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 transfered in a single atomic operation. Groups of doubles may be transfered together; however, this is not required.

Objects of this type are created with the method [RawMemoryFactory.createRawDoubleWriter](#)¹⁰ and [RawMemoryFactory.createRawDouble](#)¹¹. Each object references a range of elements in the [RawMemoryRegion](#)¹² starting at the *base address* provided to the factory method. The size provided to the factor method determines the number of elements 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

12.3.1.6.1 Methods

setDouble(double)

Signature

```
public void  
setDouble(double value)
```

¹⁰Section [12.3.2.6.3](#)

¹¹Section [12.3.2.6.3](#)

¹²Section [12.3.2.7](#)

Description

Set the value at the first position referenced by this instance, i.e., the value at its start address. This operation must be atomic with respect to all other raw memory accesses to the address.

Parameters

value is the new value for the element.

setDouble(int, double)*Signature*

```
public void  
setDouble(int offset,  
          double value)  
throws OffsetOutOfBoundsException
```

Description

Set the value of the n^{th} element referenced by this instance, where n is offset and the address is *base address* + *offset* * *size of Double*. This operation must be atomic with respect to all other raw memory accesses to the address. When an exception is thrown, no data is transferred.

Parameters

offset of double in the memory region.
value is the new value for the element.

Throws

OffsetOutOfBoundsException when offset is negative or greater than or equal to the number of elements in the raw memory region.

set(int, double)*Signature*

```
public int  
set(int offset,  
    double[] values)  
throws OffsetOutOfBoundsException,  
        NullPointerException
```

Description

Copy from values to the memory region from index start, to elements where the nth element is at the address: base address + (offset+n) x stride x element size in bytes. Only the doubles in the intersection of values and the end of the memory region are transferred. When an exception is thrown, no data is transferred.

Parameters

offset of first double in the memory region to be set.
values is 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, double, int, int)*Signature*

```
public int  
set(int offset,  
    double[] values,  
    int start,  
    int count)  
throws OffsetOutOfBoundsException,  
        ArrayIndexOutOfBoundsException,  
        NullPointerException
```

Description

Copy values to the memory region, where offset is first double in the memory region to write and start is the first index in values from which to read. The number of bytes 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

[OffsetOutOfBoundsException](#) when offset is negative or either offset or offset + count is greater than or equal to the size of this raw memory area.

[ArrayIndexOutOfBoundsException](#) when start is negative or either start or start + count is greater than or equal to the size of values.

[NullPointerException](#) when values is null.

Returns

the number of doubles actually transfered.

12.3.1.7 RawFloat

Interfaces

[javafx.realtime.device.RawFloatReader](#)

[javafx.realtime.device.RawFloatWriter](#)

Description

A marker for an object that can be used to access to a single float. Read and write access to that float is checked by the factory that creates the instance; therefore, no access checking is provided by this interface, only bounds checking.

Available since RTSJ 2.0

12.3.1.8 RawFloatReader

Interfaces

[javafx.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 transfered in a single atomic operation. Groups of floats may be transfered together; however, this is not required.

Objects of this type are created with the method [RawMemoryFactory.createRawFloatReader](#)¹³ and [RawMemoryFactory.createRawFloat](#)¹⁴. Each object references a range of

¹³Section [12.3.2.6.3](#)

¹⁴Section [12.3.2.6.3](#)

elements in the [RawMemoryRegion](#)¹⁵ starting at the *base address* provided to the factory method. The size provided to the factor method determines the number of elements 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

12.3.1.8.1 Methods

getFloat

Signature

```
public float  
getFloat()
```

Description

Get the value at the first position referenced by this instance, i.e., the value at its start address. This operation must be atomic with respect to all other raw memory accesses to the address.

Returns

the value at the *base address*.

getFloat(int)

Signature

```
public float  
getFloat(int offset)  
throws OffsetOutOfBoundsException
```

Description

Get the value at the address: $\text{base address} + \text{offset} \times \text{stride} \times \text{element size}$ in bytes. When an exception is thrown, no data is transferred.

¹⁵Section [12.3.2.7](#)

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*

```
public int  
get(int offset,  
    float[] values)  
throws OffsetOutOfBoundsException,  
        NullPointerException
```

Description

Fill values with elements from this instance, where the *n*th element is at the address: base address + (offset+n) x stride x element size in bytes. Only the floats in the intersection of the start and end of values and the *base address* and the end of the memory region are transferred. When an exception is thrown, no data is transferred.

Parameters

offset of the first float in the memory region to transfere
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 actuall transferred to values

get(int, float, int, int)*Signature*

```

public int
get(int offset,
    float[] values,
    int start,
    int count)
throws OffsetOutOfBoundsException,
        ArrayIndexOutOfBoundsException,
        NullPointerException

```

Description

Fill values from index start with elements from this instance, where the nth element is at the address: base address + (offset+n) x stride x element size in bytes. The number of bytes transfered is the minimum of count, the *size* of the memory region minus offset, and length of values minus start. When an exception is thrown, no data is transfered.

Parameters

offset of the first float in the memory region to transfere
 values the array to receive the floats
 start the first index in array to fill
 count the maximum number of floats to copy

Throws

[OffsetOutOfBoundsException](#) when offset is negative or either offset or offset + count is greater than or equal to the size of this raw memory area.
[ArrayIndexOutOfBoundsException](#) when start is negative or either start or start + count is greater than or equal to the size of values.
[NullPointerException](#) when values is null or count is negative.

Returns

the number of floats actually transfered.

12.3.1.9 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 transfered in a single atomic operation. Groups of floats may be transfered together; however, this is not required.

Objects of this type are created with the method [RawMemoryFactory.createRawFloatWriter](#)¹⁶ and [RawMemoryFactory.createRawFloat](#)¹⁷. Each object references a range of elements in the [RawMemoryRegion](#)¹⁸ starting at the *base address* provided to the factory method. The size provided to the factor method determines the number of elements accessable.

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

Available since RTSJ 2.0

12.3.1.9.1 Methods

setFloat(float)

Signature

```
public void  
setFloat(float value)
```

Description

Set the value at the first position referenced by this instance, i.e., the value at its start address. This operation must be atomic with respect to all other raw memory accesses to the address.

Parameters

value is the new value for the element.

setFloat(int, float)

Signature

¹⁶Section [12.3.2.6.3](#)

¹⁷Section [12.3.2.6.3](#)

¹⁸Section [12.3.2.7](#)

```
public void
setFloat(int offset,
         float value)
throws OffsetOutOfBoundsException
```

Description

Set the value of the n^{th} element referenced by this instance, where n is offset and the address is *base address* + *offset* * *size of Float*. This operation must be atomic with respect to all other raw memory accesses to the address. When an exception is thrown, no data is transferred.

Parameters

offset of float in the memory region.
value is the new value for the element.

Throws

[OffsetOutOfBoundsException](#) when offset is negative or greater than or equal to the number of elements in the raw memory region.

set(int, float)*Signature*

```
public int
set(int offset,
    float[] values)
throws OffsetOutOfBoundsException,
        NullPointerException
```

Description

Copy from values to the memory region from index start, to elements where the n^{th} element is at the address: *base address* + (*offset*+ n) x *stride* x *element size* in bytes. Only the floats in the intersection of values and the end of the memory region are transferred. When an exception is thrown, no data is transferred.

Parameters

offset of first float in the memory region to be set.
values is 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)  
throws OffsetOutOfBoundsException,  
        ArrayIndexOutOfBoundsException,  
        NullPointerException
```

Description

Copy values to the memory region, where offset is first float in the memory region to write and start is the first index in values from which to read. The number of bytes transfered is the minimum of count, the *size* of the memory region minus offset, and length of values minus start. When an exception is thrown, no data is 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 transfered.

12.3.1.10 RawInt

Interfaces

[javax.realtime.device.RawIntReader](#)
[javax.realtime.device.RawIntWriter](#)

Description

A marker for an object that can be used to access to a single int. Read and write access to that int is checked by the factory that creates the instance; therefore, no access checking is provided by this interface, only bounds checking.

Available since RTSJ 2.0

12.3.1.11 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 transfered in a single atomic operation. Groups of ints may be transfered together; however, this is not required.

Objects of this type are created with the method [RawMemoryFactory.createRawIntReader](#)¹⁹ and [RawMemoryFactory.createRawInt](#)²⁰. Each object references a range of elements in the [RawMemoryRegion](#)²¹ starting at the *base address* provided to the factory method. The size provided to the factor method determines the number of elements 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

¹⁹Section [12.3.2.6.3](#)

²⁰Section [12.3.2.6.3](#)

²¹Section [12.3.2.7](#)

12.3.1.11.1 Methods

getInt

Signature

```
public int  
getInt()
```

Description

Get the value at the first position referenced by this instance, i.e., the value at its start address. This operation must be atomic with respect to all other raw memory accesses to the address.

Returns

the value at the *base address*.

getInt(int)

Signature

```
public int  
getInt(int offset)  
throws OffsetOutOfBoundsException
```

Description

Get the value at the address: base address + offset x stride x 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

Fill values with elements from this instance, where the *n*th element is at the address: *base address* + (*offset*+*n*) x *stride* x *element size* in bytes. Only the ints in the intersection of the start and end of values and the *base address* and the end of the memory region are transferred. When an exception is thrown, no data is transferred.

Parameters

offset of the first int in the memory region to transfere
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 actual 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

Fill values from index start with elements from this instance, where the nth element is at the address: base address + (offset+n) x stride x element size in bytes. The number of bytes transfered is the minimum of count, the *size* of the memory region minus offset, and length of values minus start. When an exception is thrown, no data is transfered.

Parameters

offset of the first int in the memory region to transfere
values the array to receive the ints
start the first index in array to fill
count the maximum number of ints to copy

Throws

[OffsetOutOfBoundsException](#) when offset is negative or either offset or offset + count is greater than or equal to the size of this raw memory area.
[ArrayIndexOutOfBoundsException](#) when start is negative or either start or start + count is greater than or equal to the size of values.
[NullPointerException](#) when values is null or count is negative.

Returns

the number of ints actually transfered.

12.3.1.12 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 transfered in a single atomic operation. Groups of ints may be transfered together; however, this is not required.

Objects of this type are created with the method [RawMemoryFactory.createRawIntWriter](#)²² and [RawMemoryFactory.createRawInt](#)²³. Each object references a range of elements in the [RawMemoryRegion](#)²⁴ starting at the *base address* provided to the factory method. The size provided to the factor method determines the number of elements accessable.

²²Section [12.3.2.6.3](#)

²³Section [12.3.2.6.3](#)

²⁴Section [12.3.2.7](#)

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

12.3.1.12.1 Methods

setInt(int)

Signature

```
public void  
setInt(int value)
```

Description

Set the value at the first position referenced by this instance, i.e., the value at its start address. This operation must be atomic with respect to all other raw memory accesses to the address.

Parameters

value is the new value for the element.

setInt(int, int)

Signature

```
public void  
setInt(int offset,  
       int value)  
throws OffsetOutOfBoundsException
```

Description

Set the value of the n^{th} element referenced by this instance, where n is offset and the address is *base address* + offset * *size of 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 is the new value for the element.

Throws

OffsetOutOfBoundsException when offset is negative or greater than or equal to the number of elements in the raw memory region.

set(int, int)

Signature

```
public int  
set(int offset,  
    int[] values)  
throws OffsetOutOfBoundsException,  
        NullPointerException
```

Description

Copy from values to the memory region from index start, to elements where the nth element is at the address: base address + (offset+n) x stride x element size in bytes. Only the ints in the intersection of values and the end of the memory region are transferred. When an exception is thrown, no data is transferred.

Parameters

offset of first int in the memory region to be set.

values is 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

Copy values to the memory region, where offset is first int in the memory region to write and start is the first index in values from which to read. The number of bytes transfered is the minimum of count, the *size* of the memory region minus offset, and length of values minus start. When an exception is thrown, no data is transfered.

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 start is negative or either start or start + count is greater than or equal to the size of values.
`NullPointerException` when values is null.

Returns

the number of ints actually transfered.

12.3.1.13 RawLong

Interfaces

`javax.realtime.device.RawLongReader`
`javax.realtime.device.RawLongWriter`

Description

A marker for an object that can be used to access to a single long. Read and write access to that long is checked by the factory that creates the instance; therefore, no access checking is provided by this interface, only bounds checking.

Available since RTSJ 2.0

12.3.1.14 RawLongReader

Interfaces

[javafx.realtime.device.RawMemory](#)

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 [RawMemoryFactory.createRawLongReader](#)²⁵ and [RawMemoryFactory.createRawLong](#)²⁶. Each object references a range of elements in the [RawMemoryRegion](#)²⁷ starting at the *base address* provided to the factory method. The size provided to the factor method determines the number of elements 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

12.3.1.14.1 Methods

getLong

Signature

```
public long  
getLong()
```

Description

Get the value at the first position referenced by this instance, i.e., the value at its start address. This operation must be atomic with respect to all other raw memory accesses to the address.

²⁵Section [12.3.2.6.3](#)

²⁶Section [12.3.2.6.3](#)

²⁷Section [12.3.2.7](#)

Returns

the value at the *base address*.

getLong(int)*Signature*

```
public long  
getLong(int offset)  
throws OffsetOutOfBoundsException
```

Description

Get the value at the address: base address + offset x stride x 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

[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*

```
public int  
get(int offset,  
    long[] values)  
throws OffsetOutOfBoundsException,  
        NullPointerException
```

Description

Fill values with elements from this instance, where the nth element is at the address: base address + (offset+n) x stride x element size in bytes. Only the longs in the intersection of the start and end of values and the *base address* and the end of the memory region are transferred. When an exception is thrown, no data is transferred.

Parameters

offset of the first long in the memory region to transfere
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 actual transferred to values

get(int, long, int, int)*Signature*

```
public int  
get(int offset,  
    long[] values,  
    int start,  
    int count)  
throws OffsetOutOfBoundsException,  
        ArrayIndexOutOfBoundsException,  
        NullPointerException
```

Description

Fill values from index start with elements from this instance, where the nth
element is at the address: base address + (offset+n) x stride x element size in
bytes. The number of bytes transfered is the minimum of count, the *size* of
the memory region minus offset, and length of values minus start. When an
exception is thrown, no data is transfered.

Parameters

offset of the first long in the memory region to transfere

values the array to receive the longs

start the first index in array to fill

count the maximum number of longs to copy

Throws

OffsetOutOfBoundsException when offset is negative or either offset or offset +
count is greater than or equal to the size of this raw memory area.

ArrayIndexOutOfBoundsException when start is negative or either start or start
+ count is greater than or equal to the size of values.

NullPointerException when values is null or count is negative.

Returns

the number of longs actually transfered.

12.3.1.15 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 transfered in a single atomic operation. Groups of longs may be transfered together; however, this is not required.

Objects of this type are created with the method [RawMemoryFactory.createRawLongWriter](#)²⁸ and [RawMemoryFactory.createRawLong](#)²⁹. Each object references a range of elements in the [RawMemoryRegion](#)³⁰ starting at the *base address* provided to the factory method. The size provided to the factor method determines the number of elements 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

12.3.1.15.1 Methods

setLong(long)*Signature*

```
public void  
setLong(long value)
```

²⁸Section [12.3.2.6.3](#)

²⁹Section [12.3.2.6.3](#)

³⁰Section [12.3.2.7](#)

Description

Set the value at the first position referenced by this instance, i.e., the value at its start address. This operation must be atomic with respect to all other raw memory accesses to the address.

Parameters

value is the new value for the element.

setLong(int, long)*Signature*

```
public void  
setLong(int offset,  
        long value)  
throws OffsetOutOfBoundsException
```

Description

Set the value of the n^{th} element referenced by this instance, where n is offset and the address is *base address* + $offset * size\ of\ Long$. 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 is the new value for the element.

Throws

OffsetOutOfBoundsException when offset is negative or greater than or equal to the number of elements in the raw memory region.

set(int, long)*Signature*

```
public int  
set(int offset,  
    long[] values)  
throws OffsetOutOfBoundsException,  
        NullPointerException
```

Description

Copy from values to the memory region from index start, to elements where the nth element is at the address: base address + (offset+n) x stride x element size in bytes. Only the longs in the intersection of values and the end of the memory region are transferred. When an exception is thrown, no data is transferred.

Parameters

offset of first long in the memory region to be set.

values is 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*

```
public int
set(int offset,
    long[] values,
    int start,
    int count)
throws OffsetOutOfBoundsException,
        ArrayIndexOutOfBoundsException,
        NullPointerException
```

Description

Copy values to the memory region, where offset is first long in the memory region to write and start is the first index in values from which to read. The number of bytes 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 transfered.

12.3.1.16 RawMemory

Description

A marker for all raw memory accessor objects.

Available since RTSJ 2.0

12.3.1.16.1 Methods

getAddress

Signature

```
public long  
getAddress()
```

Description

Get the base physical address of this object.

Returns

the first physical address this raw memory object can access.

getSize

Signature

```
public int  
getSize()
```

Description

Get the number of bytes that this object spans.

Returns

the size of this raw memory

getStride*Signature*

```
public int  
getStride()
```

Description

Get the distance between elements in multiples of element size.

Returns

the span between elements of this raw memory

12.3.1.17 RawMemoryRegionFactory

Description

A class to give an application the ability to provide support for a [RawMemoryRegion](#)³¹ that is not already provided by the standard. An instance of this call can be registered with a [RawMemoryFactory](#)³² and provides the object that that factory should return for a given 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

12.3.1.17.1 Methods

³¹Section [12.3.2.7](#)

³²Section [12.3.2.6](#)

getRegion

Signature

```
public javax.realtime.device.RawMemoryRegion  
getRegion()
```

Description

Determine for what region this factory creates raw memory objects.

Returns

the region of this factory.

getName

Signature

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

Description

Determine 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,  
       OffsetOutOfBoundsException,  
       SizeOutOfBoundsException,  
       UnsupportedRawMemoryRegionException,  
       MemoryTypeConflictException
```

Description

Create an instance of a class that implements `RawByte`³³ and accesses memory of `getRegion`³⁴ in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is `stride * size of RawByte * count`. The object is allocated in the current memory area of the calling thread.

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 `RawByte`³⁵ 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,
```

³³Section 12.3.1.1

³⁴Section 12.3.1.17.1

³⁵Section 12.3.1.1

UnsupportedRawMemoryRegionException,
MemoryTypeConflictException

Description

Create an instance of a class that implements [RawByteReader](#)³⁶ and accesses memory of [getRegion](#)³⁷ in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride * *size of RawByteReader* * count. The object is allocated in the current memory area of the 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](#)³⁸ 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,
```

³⁶Section [12.3.1.2](#)

³⁷Section [12.3.1.17.1](#)

³⁸Section [12.3.1.2](#)

```

        int count,
        int stride)
throws SecurityException,
        OffsetOutOfBoundsException,
        SizeOutOfBoundsException,
        UnsupportedRawMemoryRegionException,
        MemoryTypeConflictException

```

Description

Create an instance of a class that implements [RawByteWriter](#)³⁹ and accesses memory of [getRegion](#)⁴⁰ in the address range described by base, stride, and count. 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](#)⁴¹ and supports access to the specified range in the memory region.

Available since RTSJ 2.0

³⁹Section [12.3.1.3](#)

⁴⁰Section [12.3.1.17.1](#)

⁴¹Section [12.3.1.3](#)

createRawShort(long, int, int)

Signature

```
public javafx.realtime.device.RawShort  
createRawShort(long base,  
                int count,  
                int stride)  
throws SecurityException,  
        OffsetOutOfBoundsException,  
        SizeOutOfBoundsException,  
        UnsupportedRawMemoryRegionException,  
        MemoryTypeConflictException
```

Description

Create an instance of a class that implements [RawShort](#)⁴² and accesses memory of [getRegion](#)⁴³ in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride * *size of RawShort* * count. The object is allocated in the current memory area of the calling thread.

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](#)⁴⁴ and supports access to the specified range in the memory region.

⁴²Section [12.3.1.18](#)

⁴³Section [12.3.1.17.1](#)

⁴⁴Section [12.3.1.18](#)

Available since RTSJ 2.0

createRawShortReader(long, int, int)

Signature

```
public javax.realtime.device.RawShortReader  
createRawShortReader(long base,  
                      int count,  
                      int stride)  
  
throws SecurityException,  
        OffsetOutOfBoundsException,  
        SizeOutOfBoundsException,  
        UnsupportedRawMemoryRegionException,  
        MemoryTypeConflictException
```

Description

Create an instance of a class that implements [RawShortReader](#)⁴⁵ and accesses memory of [getRegion](#)⁴⁶ in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is *stride * size of RawShortReader * count*. The object is allocated in the current memory area of the calling thread.

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.

⁴⁵Section [12.3.1.19](#)

⁴⁶Section [12.3.1.17.1](#)

Returns

an object that implements [RawShortReader](#)⁴⁷ and supports access to the specified range in the memory region.

Available since RTSJ 2.0

createRawShortWriter(long, int, int)*Signature*

```
public javafx.realtime.device.RawShortWriter  
createRawShortWriter(long base,  
                      int count,  
                      int stride)  
  
throws SecurityException,  
        OffsetOutOfBoundsException,  
        SizeOutOfBoundsException,  
        UnsupportedRawMemoryRegionException,  
        MemoryTypeConflictException
```

Description

Create an instance of a class that implements [RawShortWriter](#)⁴⁸ and accesses memory of [getRegion](#)⁴⁹ in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is $\text{stride} * \text{size of RawShortWriter} * \text{count}$. 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.

⁴⁷Section [12.3.1.19](#)

⁴⁸Section [12.3.1.20](#)

⁴⁹Section [12.3.1.17.1](#)

[OffsetOutOfBoundsException](#) when base is invalid.

[SizeOutOfBoundsException](#) when the memory addressed by the object would extend into an invalid range of memory.

[MemoryTypeConflictException](#) when base does not point to memory that matches the type served by this factory.

Returns

an object that implements [RawShortWriter](#)⁵⁰ and supports access to the specified range in the memory region.

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

Create an instance of a class that implements [RawInt](#)⁵¹ and accesses memory of [getRegion](#)⁵² in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride * *size of RawInt* * count. The object is allocated in the current memory area of the calling thread.

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

⁵⁰Section [12.3.1.20](#)

⁵¹Section [12.3.1.10](#)

⁵²Section [12.3.1.17.1](#)

`IllegalArgumentException` when base is negative, count is not greater than zero, or stride is not greater than zero.

`SecurityException` when the caller does not have permissions to access the given memory region or the specified range of addresses.

`OffsetOutOfBoundsException` when base is invalid.

`SizeOutOfBoundsException` when the memory addressed by the object would extend into an invalid range of memory.

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

Returns

an object that implements `RawInt`⁵³ and supports access to the specified range in the memory region.

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

Create an instance of a class that implements `RawIntReader`⁵⁴ and accesses memory of `getRegion`⁵⁵ in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is `stride * size of RawIntReader * count`. The object is allocated in the current memory area of the calling thread.

Parameters

⁵³Section 12.3.1.10

⁵⁴Section 12.3.1.11

⁵⁵Section 12.3.1.17.1

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 **RawIntReader**⁵⁶ and supports access to the specified range in the memory region.

Available since RTSJ 2.0

createRawIntWriter(long, int, int)

Signature

```
public javax.realtime.device.RawIntWriter
createRawIntWriter(long base,
                   int count,
                   int stride)
throws SecurityException,
       OffsetOutOfBoundsException,
       SizeOutOfBoundsException,
       UnsupportedRawMemoryRegionException,
       MemoryTypeConflictException
```

Description

Create an instance of a class that implements **RawIntWriter**⁵⁷ and accesses memory of **getRegion**⁵⁸ in the address range described by base, stride, and count.

⁵⁶Section 12.3.1.11

⁵⁷Section 12.3.1.12

⁵⁸Section 12.3.1.17.1

The actual extent of the memory addressed by the object is $\text{stride} * \text{size of RawIntWriter} * \text{count}$. 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 **RawIntWriter**⁵⁹ and supports access to the specified range in the memory region.

Available since RTSJ 2.0

createRawLong(long, int, int)

Signature

```
public javafx.realtime.device.RawLong  
createRawLong(long base,  
               int count,  
               int stride)  
throws SecurityException,  
       OffsetOutOfBoundsException,  
       SizeOutOfBoundsException,  
       UnsupportedRawMemoryRegionException,  
       MemoryTypeConflictException
```

⁵⁹Section 12.3.1.12

Description

Create an instance of a class that implements `RawLong`⁶⁰ and accesses memory of `getRegion`⁶¹ in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is `stride * size of RawLong * count`. The object is allocated in the current memory area of the calling thread.

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`⁶² 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)
```

⁶⁰Section 12.3.1.13

⁶¹Section 12.3.1.17.1

⁶²Section 12.3.1.13

throws `SecurityException`,
`OffsetOutOfBoundsException`,
`SizeOutOfBoundsException`,
`UnsupportedRawMemoryRegionException`,
`MemoryTypeConflictException`

Description

Create an instance of a class that implements `RawLongReader`⁶³ and accesses memory of `getRegion`⁶⁴ in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is $\text{stride} * \text{size of RawLongReader} * \text{count}$. 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 `RawLongReader`⁶⁵ and supports access to the specified range in the memory region.

Available since RTSJ 2.0

createRawLongWriter(long, int, int)

⁶³Section 12.3.1.14

⁶⁴Section 12.3.1.17.1

⁶⁵Section 12.3.1.14

Signature

```
public javax.realtime.device.RawLongWriter
createRawLongWriter(long base,
                    int count,
                    int stride)
throws SecurityException,
        OffsetOutOfBoundsException,
        SizeOutOfBoundsException,
        UnsupportedRawMemoryRegionException,
        MemoryTypeConflictException
```

Description

Create an instance of a class that implements [RawLongWriter](#)⁶⁶ and accesses memory of [getRegion](#)⁶⁷ in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is *stride * size of RawLongWriter * count*. The object is allocated in the current memory area of the calling thread.

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 [RawLongWriter](#)⁶⁸ and supports access to the specified range in the memory region.

⁶⁶Section [12.3.1.15](#)

⁶⁷Section [12.3.1.17.1](#)

⁶⁸Section [12.3.1.15](#)

Available since RTSJ 2.0

createRawFloat(long, int, int)

Signature

```
public javafx.realtime.device.RawFloat  
createRawFloat(long base,  
                int count,  
                int stride)  
throws SecurityException,  
        OffsetOutOfBoundsException,  
        SizeOutOfBoundsException,  
        UnsupportedRawMemoryRegionException,  
        MemoryTypeConflictException
```

Description

Create an instance of a class that implements [RawFloat](#)⁶⁹ and accesses memory of [getRegion](#)⁷⁰ in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride * *size of RawFloat* * count. The object is allocated in the current memory area of the calling thread.

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.

⁶⁹Section [12.3.1.7](#)

⁷⁰Section [12.3.1.17.1](#)

Returns

an object that implements [RawFloat](#)⁷¹ 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

Create an instance of a class that implements [RawFloatReader](#)⁷² and accesses memory of [getRegion](#)⁷³ in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride * *size of RawFloatReader* * count. The object is allocated in the current memory area of the calling thread.

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.

⁷¹Section [12.3.1.7](#)

⁷²Section [12.3.1.8](#)

⁷³Section [12.3.1.17.1](#)

[OffsetOutOfBoundsException](#) when base is invalid.

[SizeOutOfBoundsException](#) when the memory addressed by the object would extend into an invalid range of memory.

[MemoryTypeConflictException](#) when base does not point to memory that matches the type served by this factory.

Returns

an object that implements [RawFloatReader](#)⁷⁴ and supports access to the specified range in the memory region.

Available since RTSJ 2.0

createRawFloatWriter(long, int, int)

Signature

```
public javafx.realtime.device.RawFloatWriter  
createRawFloatWriter(long base,  
                      int count,  
                      int stride)  
  
throws SecurityException,  
       OffsetOutOfBoundsException,  
       SizeOutOfBoundsException,  
       UnsupportedRawMemoryRegionException,  
       MemoryTypeConflictException
```

Description

Create an instance of a class that implements [RawFloatWriter](#)⁷⁵ and accesses memory of [getRegion](#)⁷⁶ in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride * *size of RawFloatWriter* * count. The object is allocated in the current memory area of the calling thread.

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.

⁷⁴Section [12.3.1.8](#)

⁷⁵Section [12.3.1.9](#)

⁷⁶Section [12.3.1.17.1](#)

Throws

`IllegalArgumentException` when base is negative, count is not greater than zero, or stride is not greater than zero.

`SecurityException` when the caller does not have permissions to access the given memory region or the specified range of addresses.

`OffsetOutOfBoundsException` when base is invalid.

`SizeOutOfBoundsException` when the memory addressed by the object would extend into an invalid range of memory.

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

Returns

an object that implements `RawFloatWriter`⁷⁷ and supports access to the specified range in the memory region.

Available since RTSJ 2.0

createRawDouble(long, int, int)

Signature

```
public javax.realtime.device.RawDouble  
createRawDouble(long base,  
                 int count,  
                 int stride)  
throws SecurityException,  
        OffsetOutOfBoundsException,  
        SizeOutOfBoundsException,  
        UnsupportedRawMemoryRegionException,  
        MemoryTypeConflictException
```

Description

Create an instance of a class that implements `RawDouble`⁷⁸ and accesses memory of `getRegion`⁷⁹ in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is `stride * size of RawDouble * count`. The object is allocated in the current memory area of the calling thread.

Parameters

⁷⁷Section 12.3.1.9

⁷⁸Section 12.3.1.4

⁷⁹Section 12.3.1.17.1

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**⁸⁰ and supports access to the specified range in the memory region.

Available since RTSJ 2.0

createRawDoubleReader(long, int, int)

Signature

```
public javafx.realtime.device.RawDoubleReader  
createRawDoubleReader(long base,  
                        int count,  
                        int stride)  
  
throws SecurityException,  
        OffsetOutOfBoundsException,  
        SizeOutOfBoundsException,  
        UnsupportedRawMemoryRegionException,  
        MemoryTypeConflictException
```

Description

Create an instance of a class that implements **RawDoubleReader**⁸¹ and accesses memory of **getRegion**⁸² in the address range described by base, stride, and count.

⁸⁰Section 12.3.1.4

⁸¹Section 12.3.1.5

⁸²Section 12.3.1.17.1

The actual extent of the memory addressed by the object is $\text{stride} * \text{size of RawDoubleReader} * \text{count}$. 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 **RawDoubleReader**⁸³ 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
```

⁸³Section 12.3.1.5

Description

Create an instance of a class that implements [RawDoubleWriter](#)⁸⁴ and accesses memory of [getRegion](#)⁸⁵ in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is $\text{stride} * \text{size of RawDoubleWriter} * \text{count}$. 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](#)⁸⁶ and supports access to the specified range in the memory region.

Available since RTSJ 2.0

12.3.1.18 RawShort

Interfaces

[javafx.realtime.device.RawShortReader](#)
[javafx.realtime.device.RawShortWriter](#)

Description

⁸⁴Section [12.3.1.6](#)

⁸⁵Section [12.3.1.17.1](#)

⁸⁶Section [12.3.1.6](#)

A marker for an object that can be used to access to a single short. Read and write access to that short is checked by the factory that creates the instance; therefore, no access checking is provided by this interface, only bounds checking.

Available since RTSJ 2.0

12.3.1.19 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](#)⁸⁷ and [RawMemoryFactory.createRawShort](#)⁸⁸. Each object references a range of elements in the [RawMemoryRegion](#)⁸⁹ starting at the *base address* provided to the factory method. The size provided to the factor method determines the number of elements 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

12.3.1.19.1 Methods

getShort

Signature

⁸⁷Section [12.3.2.6.3](#)

⁸⁸Section [12.3.2.6.3](#)

⁸⁹Section [12.3.2.7](#)

```
public short  
getShort()
```

Description

Get the value at the first position referenced by this instance, i.e., the value at its start address. This operation must be atomic with respect to all other raw memory accesses to the address.

Returns

the value at the *base address*.

getShort(int)*Signature*

```
public short  
getShort(int offset)  
throws OffsetOutOfBoundsException
```

Description

Get the value at the address: base address + offset x stride x 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*

```
public int  
get(int offset,  
    short[] values)  
throws OffsetOutOfBoundsException,  
        NullPointerException
```

Description

Fill values with elements from this instance, where the *n*th element is at the address: base address + (offset+n) x stride x element size in bytes. Only the shorts in the intersection of the start and end of values and the *base address* and the end of the memory region are transferred. When an exception is thrown, no data is transferred.

Parameters

offset of the first short in the memory region to transfere
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 actual transferred to values

get(int, short, int, int)*Signature*

```
public int  
get(int offset,  
    short[] values,  
    int start,  
    int count)  
throws OffsetOutOfBoundsException,  
        ArrayIndexOutOfBoundsException,  
        NullPointerException
```

Description

Fill values from index start with elements from this instance, where the *n*th element is at the address: base address + (offset+n) x stride x 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 transfere
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 transfered.

12.3.1.20 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 transfered in a single atomic operation. Groups of shorts may be transfered together; however, this is not required.

Objects of this type are created with the method [RawMemoryFactory.createRawShortWriter](#)⁹⁰ and [RawMemoryFactory.createRawShort](#)⁹¹. Each object references a range of elements in the [RawMemoryRegion](#)⁹² starting at the *base address* provided to the factory method. The size provided to the factor method determines the number of elements 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

12.3.1.20.1 Methods

⁹⁰Section [12.3.2.6.3](#)

⁹¹Section [12.3.2.6.3](#)

⁹²Section [12.3.2.7](#)

setShort(short)

Signature

```
public void  
setShort(short value)
```

Description

Set the value at the first position referenced by this instance, i.e., the value at its start address. This operation must be atomic with respect to all other raw memory accesses to the address.

Parameters

value is the new value for the element.

setShort(int, short)

Signature

```
public void  
setShort(int offset,  
         short value)  
throws OffsetOutOfBoundsException
```

Description

Set the value of the n^{th} element referenced by this instance, where n is offset and the address is *base address* + *offset* * *size of 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 is the new value for the element.

Throws

OffsetOutOfBoundsException when offset is negative or greater than or equal to the number of elements in the raw memory region.

set(int, short)

Signature

```
public int
set(int offset,
    short[] values)
throws OffsetOutOfBoundsException,
    NullPointerException
```

Description

Copy from values to the memory region from index start, to elements where the nth element is at the address: base address + (offset+n) x stride x element size in bytes. Only the shorts in the intersection of values and the end of the memory region are transferred. When an exception is thrown, no data is transferred.

Parameters

offset of first short in the memory region to be set.

values is 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

Copy values to the memory region, where offset is first short in the memory region to write and start is the first index in values from which to read. The number of bytes 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 transfered.

12.3.2 Classes

12.3.2.1 DMABufferFactory

Inheritance

java.lang.Object
`javax.realtime.device.DMABufferFactory`

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 DMABufferFactory instance may overlap that of another DMABufferFactory instance.

12.3.2.1.1 Constructors

DMABufferFactory(DMARegion, long, long)

Signature

```
public
DMABufferFactory(DMARegion region,
                 long base,
                 long size)
throws MemoryInUseException
```

Description

Create a factory for allocating buffers in a particular address range. Whether the address is physical or virtual is system dependent.

Parameters

region is the area of memory a DMA controller can reference, from which this factory takes its memory.

base is the base address of a memory range in region for buffer allocation

size is the number of bytes in the memory range

Throws

[MemoryInUseException](#) when the memory area provide is already in use by or reserved for a [javafx.runtime.MemoryArea](#)⁹³, program code, or other sytem or VM structure.

12.3.2.1.2 Methods

allocateDirectByteBuffer(int)

Signature

```
public java.nio.ByteBuffer
allocateDirectByteBuffer(int capacity)
```

Description

Create a direct byte buffer with the given capacity within the range of this factory.

Parameters

capacity the number of bytes in the buffer.

Throws

[javafx.runtime.StaticOutOfMemoryError](#) when no memory is available

Returns

the new buffer.

⁹³Section [11.3.2.3](#)

free(ByteBuffer)

Signature

```
public void  
free(ByteBuffer buffer)
```

Description

Free the memory associated with the given ByteBuffer 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 is the ByteBuffer to free.

Throws

IllegalArgumentException when buffer was not allocated from this factory.

IllegalStateException when buffer has already been freed.

Open issue 12.3.1

Should we have our own buffer class with this methods defined there instead of here?

End of issue 12.3.1

inRange(ByteBuffer)

Signature

```
public boolean  
inRange(ByteBuffer buffer)
```

Description

Check to see if the buffer's data area is within the range of this factory.

Parameters

buffer to check

Returns

true when and only when buffer's data area is within the range of this factory;
otherwise false

addressOf(ByteBuffer)

Signature

```
public long  
addressOf(ByteBuffer buffer)
```

Description

Give the location of this buffers data in memory. The address shall be in the address space of the DMA controller.

Parameters

buffer of which to get the address

Returns

the start address of the data range of this buffer

writeFence(ByteBuffer)

Signature

```
public static void  
writeFence(ByteBuffer buffer)
```

Description

Ensures that all changes to the DirectByteBuffer buffer by the current thread have been flushed in a manner that makes them visible to other threads (including native threads), and behaves as a volatile store with respect to the Java Memory Model synchronization order.

This method 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(ByteBuffer)

Signature

```
public static void  
readFence(ByteBuffer buffer)
```

Description

Ensures that any previous changes to the memory represented by the given DirectByteBuffer by other threads (including native threads) will be visible when

it is next accessed by the current thread, and behaves as a volatile load with respect to the Java Memory Model synchronization order.

This method 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(ByteBuffer)`⁹⁴ or its native equivalent on the buffer's memory.

Parameters

buffer the byte buffer which will be updated

12.3.2.2 DMARegion

Inheritance

java.lang.Object
[javafx.runtime.device.DMARegion](#)

Description

Define the reachable memory for a given DMA controller in terms of the physical address space of the system.

12.3.2.2.1 Constructors

DMARegion(long, long)

Signature

```
public
DMARegion(long start,
           long size)
throws IllegalArgumentException
```

Description

Create a DMA memory definition.

Parameters

⁹⁴Section [12.3.1](#)

start address of the DMA address space in the physical address address space of the main processor.

size is the number of bytes in the DMA address space

Throws

IllegalArgumentException when start is less than zero or start + size is larger than the physical memory of the system.

12.3.2.2.2 Methods

regionAddressOf(long)

Signature

```
public long  
regionAddressOf(long address)  
throws IllegalArgumentException
```

Description

Translate a physical address into a DMA region address.

Parameters

address to translate

Throws

IllegalArgumentException when the results is outside the DMA space.

Returns

the equivalent address in DMA space.

physicalAddressOf(long)

Signature

```
public long  
physicalAddressOf(long address)  
throws IllegalArgumentException
```

Description

Translate a DMA space address into a physical address.

Parameters

address to translate

Throws

`IllegalArgumentException` when the input is outside the DMA space.

Returns

the corresponding physical address

12.3.2.3 Happening

Inheritance

`java.lang.Object`

`javafx.realtime.AsyncBaseEvent`

`javafx.realtime.AsyncEvent`

`javafx.realtime.device.Happening`

Interfaces

`javafx.realtime.ActiveEvent`

Description

This class provides second level handling for external events such as interrupts. A happening can be triggered by an `InterruptServiceRoutine`⁹⁵ or from native code. Application-defined Happenings can be identified by an application-provided name or a system-provided id, both of which must be unique. A system Happening has a name provide by the system which is a string beginning with @.

Available since RTSJ 2.0

12.3.2.3.1 Constructors

Happening(String, HappeningDispatcher)

Signature

public

Happening(String name,
HappeningDispatcher dispatcher)

⁹⁵Section 12.3.2.5

throws IllegalArgumentException

Description

Create a Happening with the given name.

Parameters

name of the happening.

dispatcher to use when being triggered.

Throws

IllegalArgumentException 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 IllegalArgumentException
```

Description

Create a Happening with the given name and the default dispatcher.

Parameters

name of the happening.

Throws

IllegalArgumentException 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.

12.3.2.3.2 Methods

getHappening(String)

Signature

```
public static javax.realtime.device.Happening  
getHappening(String name)
```

Description

Find an active happening by its name.

Parameters

name of the happening to get.

Throws

IllegalArgumentException when name is null.

Returns

a reference to the happening with name name, or null if no happening is found.

isHappening(String)

Signature

```
public static boolean  
isHappening(String name)
```

Description

Is there an active happening with name name?

Parameters

name A string that might name an active happening.

Throws

IllegalArgumentException when name is null.

Returns

True only when there is a registered happening with the name name.

createId(String)

Signature

```
public static int  
createId(String name)  
throws IllegalStateException
```

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](#)⁹⁶ 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 is a happening name string.

Throws

IllegalStateException when name is already registered.

IllegalArgumentException when name is null.

Returns

an ID assigned by the system

getId(String)

Signature

```
public static int  
getId(String name)
```

Description

Return the ID of name, when one exists or -1, when name is not registered.

This must take no more than linear time in the number of ID (n) registered, but should be $O(\log_2(n))$.

Parameters

name is a happening name string.

Throws

IllegalArgumentException when name is null.

Returns

The id, or -1 when no happening is found with that name.

get(int)

Signature

⁹⁶Section [12.3.2.5](#)

```
public static javax.realtime.device.Happening  
get(int id)
```

Description

Get the external event corresponding to a given id.

Parameters

id of a registered signal

Returns

the signal corresponding to id.

get(String)*Signature*

```
public static javax.realtime.device.Happening  
get(String name)
```

Description

Get the external event corresponding to a given name.

Parameters

name of a registered signal

Throws

IllegalArgumentException 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](#)⁹⁷ method.

⁹⁷Section [12.3.2.5.2](#)

trigger() and any native code analog to it interact with other `javax.realtime.ActiveEvent`⁹⁸ 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 if a happening with id happeningId was found, false otherwise.

getId

Signature

```
public final int  
getId()
```

Description

Get the number of this happening.

Returns

the happening number or -1, when not registered.

getName

Signature

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

Description

Get the name of this happening.

Returns

the name of this happening.

⁹⁸Section 8.3.1.1

start*Signature*

```
public void  
start()  
throws IllegalStateException
```

Description

Start this Happening, i.e., change to the active and enabled state. 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

IllegalStateException 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 IllegalStateException
```

Description

Start this Happening, but leave 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

IllegalStateException when this Happening has already been started.

See [Section stop\(\)](#)

stop*Signature*


```
public boolean  
stop()  
throws IllegalStateException
```

Description

Stop this happening from responding to the fire and trigger methods.

Throws

IllegalStateException when this Happening is not active.

Returns

true when this is in the *enabled* state false otherwise.

isActive*Signature*

```
public boolean  
isActive()
```

Description

Determine the activation state of this happening, i.e., it has been started.

Returns

true when active, false otherwise.

isRunning*Signature*

```
public boolean  
isRunning()
```

Description

Determine whether or not this Happening is both active an enabled.

Returns

true when this Happening is both active and enabled, false otherwise.

trigger

Signature

```
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](#)⁹⁹ method.

This method must execute in constant time.

getDispatcher

Signature

```
public javax.realtime.device.HappeningDispatcher  
getDispatcher()
```

Description

Obtain the dispatcher for this.

Returns

that dispatcher.

12.3.2.4 HappeningDispatcher

Inheritance

```
java.lang.Object  
  javax.realtime.ActiveEventDispatcher  
    javax.realtime.device.HappeningDispatcher
```

Description

This class provides a means of dispatching a set of [Happening](#)¹⁰⁰.

12.3.2.4.1 Constructors

⁹⁹Section [12.3.2.5.2](#)

¹⁰⁰Section [12.3.2.3](#)

HappeningDispatcher(SchedulingParameters, SchedulingGroup)

Signature

```
public  
HappeningDispatcher(SchedulingParameters schedule,  
                     SchedulingGroup group)
```

Description

Create a new dispatcher, whose dispatching thread runs with the given scheduling parameters.

Parameters

schedule give the parameters for scheduling this dispatcher

HappeningDispatcher(SchedulingParameters)

Signature

```
public  
HappeningDispatcher(SchedulingParameters schedule)
```

Description

Create a new dispatcher, whose dispatching thread runs with the given scheduling parameters.

Parameters

schedule give the parameters for scheduling this dispatcher

12.3.2.4.2 Methods

register(Happening)

Signature

```
public synchronized void  
register(Happening happening)
```

throws `RegistrationException`,
`IllegalStateException`,
`IllegalArgumentException`

Description

Register a `Happening`¹⁰¹ with this dispatcher.

Parameters

happening to register

Throws

`RegistrationException` when happening is already registered.
`IllegalStateException` when this object has been destroyed.
`IllegalArgumentException` when happening is not stopped.

deregister(`Happening`)

Signature

```
public synchronized void  
deregister(Happening happening)  
throws DeregistrationException,  
       IllegalStateException,  
       IllegalArgumentException
```

Description

Unregister a `Happening`¹⁰² from this dispatcher.

Parameters

happening to unregister

Throws

`DeregistrationException` when happening is not already registered.
`IllegalStateException` when this object has been destroyed.
`IllegalArgumentException` when happening is not stopped.

destroy

Signature

¹⁰¹Section 12.3.2.3

¹⁰²Section 12.3.2.3

public void
destroy()
throws IllegalStateException

Description

Release all resources thereby making the dispatcher unusable.

Throws

IllegalStateException when called on a dispatcher that has one or more registered [Happening¹⁰³](#) objects.

12.3.2.5 InterruptServiceRoutine

Inheritance

java.lang.Object
[javax.realtime.device.InterruptServiceRoutine](#)

Interfaces

[javax.realtime.RealtimeExecutionContext](#)

Description

A class for defining a first level interrupt handler. The implementation must override the [handle¹⁰⁴](#) 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.

12.3.2.5.1 Constructors

InterruptServiceRoutine(MemoryArea)

Signature

public
InterruptServiceRoutine(MemoryArea area)

¹⁰³Section [12.3.2.3](#)

¹⁰⁴Section [12.3.2.5.2](#)

throws `NullPointerException`,
`IllegalArgumentException`

Description

Create an interrupt service routine with a particular memory area.

Parameters

area the allocation context in which the `handle`¹⁰⁵ method runs.

Throws

`NullPointerException` when area is null.

`IllegalArgumentException` when area is a memory are that cannot be accessed from an ISR.

12.3.2.5.2 Methods

validInterruptIds

Signature

```
public static int[]  
validInterruptIds()
```

Description

Determine which interrupt identifiers are valid.

Returns

an ordered array of integers representing the valid interrupts in the system. On a machine that does not support any interrupts, a zero length array is returned.

getHandler(int)

Signature

```
public static javax.realtime.device.InterruptServiceRoutine  
getHandler(int interrupt)
```

Description

Find the `InterruptServiceRoutine` that is handling a given interrupt.

¹⁰⁵Section [12.3.2.5.2](#)

Parameters

interrupt for which to find the InterruptServiceRoutine

Returns

the InterruptServiceRoutine registered to the given interrupt. Null is returned when nothing is registered for that interrupt.

getMaximumInterruptPriority

Signature

```
public static int  
getMaximumInterruptPriority()
```

Description

Retrieve the maximum interrupt priority. It must be greater than or equal to the result of [getMinimumInterruptPriority](#)¹⁰⁶.

Returns

the maximum interrupt priority.

getMinimumInterruptPriority

Signature

```
public static int  
getMinimumInterruptPriority()
```

Description

Retrieve the minimum interrupt priority. It must be higher than all other priorities provided by the system.

Returns

the minimum interrupt priority.

getInterruptPriority(int)

Signature

```
public static int  
getInterruptPriority(int interruptId)
```

¹⁰⁶Section [12.3.2.5.2](#)

throws `IllegalArgumentException`

Description

Get the interrupt priority of a given interrupt.

Throws

`IllegalArgumentException` when there is no interrupt corresponding to `interruptId`

Returns

the priority at which the `handle`¹⁰⁷ method is invoked. The returned value is always greater than `javax.realtime.PriorityScheduler.getMaxPriority()`¹⁰⁸.

isRegistered

Signature

```
public final boolean  
isRegistered()
```

Description

A predicate for the registration state.

Returns

true when registered, otherwise false.

register(int)

Signature

```
public void  
register(int interrupt)  
throws RegistrationException
```

Description

Register this interrupt service routine with the system so that it can be triggered.

Parameters

interrupt a system dependent identifier for the interrupt.

Throws

`RegistrationException` when this is already registered or some other `InterruptServiceRoutine`¹⁰⁹ is registered for interrupt.

¹⁰⁷Section 12.3.2.5.2

¹⁰⁸Section 6.3.3.8.3

¹⁰⁹Section 12.3.2.5

unregister

Signature

```
public void  
unregister()  
throws DeregistrationException
```

Description

Deregister this interrupt service routine with the system so that it can no longer be triggered.

Throws

[DeregistrationException](#) when this interrupt service routine is not registered.

handle

Signature

```
protected abstract void  
handle()
```

Description

The code to execute for first level interrupt handling. A subclass defines this to give the required behavior. [RawMemory](#)¹¹⁰ classes may be used to access the associated device registers and a [Happening](#)¹¹¹ may be triggered for second level interrupt handling.

The code used to implement this method should not block itself for an unbounded amount of time or induce a context switch, e.g., sleeping. 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 `Object.wait()` should not be called.

Unless the overridden method is synchronized, the infrastructure shall provide no synchronization for the execution of this method. When the [javax.realtime.MemoryArea](#)¹¹² provided at creation is a [javax.realtime.memory.ScopedMemory](#)¹¹³, its count is incremented on entry to this method and decremented on exit.

Open issue 12.3.2

¹¹⁰Section [12.3.1.16](#)

¹¹¹Section [12.3.2.3](#)

¹¹²Section [11.3.2.3](#)

¹¹³Section [11.4.3.6](#)

What is the ISR handle method allowed to do and how are errors handled?
Should there be an error handler?

End of issue 12.3.2

12.3.2.6 RawMemoryFactory

Inheritance

java.lang.Object
[javax.realtime.device.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\(RawMemoryRegionFactory\)](#)¹¹⁴ 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\(RawMemoryRegion, long, int, int\)](#)¹¹⁵ method returns a reference to an object that implements the [RawByte](#)¹¹⁶ interface and supports access to the requested type of memory and address range. Each create method is permitted to optimize error checking and access based on the requested memory type and address range.

The usage pattern for raw memory, assuming the necessary factory has been registered, is illustrated by this example.

```
// Get an accessor object that can access memory starting at
// baseAddress, for size bytes.
RawInt memory =
    RawMemoryFactory.createRawInt(RawMemoryFactory.MEMORY_MAPPED_REGION,
                                  address, count, stride, false);
// Use the accessor to load from and store to raw memory.
int loadedData = memory.getInt(someOffset);
memory.setInt(otherOffset, intVal);
```

When an application needs to access a class of memory that is not already supported by a registered factory, the developer must implement and register a factory that implements the [RawMemoryRegionFactory](#)¹¹⁷) which can create

¹¹⁴Section [12.3.2.6.3](#)

¹¹⁵Section [12.3.2.6.3](#)

¹¹⁶Section [12.3.1.1](#)

¹¹⁷Section [12.3.1.17](#)

objects to access memory in that region.

A raw memory region factory is identified by a `RawMemoryRegion`¹¹⁸ that is used by each create method, e.g., `createRawByte(RawMemoryRegion, long, int, int)`¹¹⁹, to locate the appropriate factory. The name is provided to `register(RawMemoryRegionFactory)`¹²⁰ through the factory's `RawMemoryRegionFactory.getName`¹²¹ method.

The `register(RawMemoryRegionFactory)`¹²² method is only used when by application code when it needs to add support for a new type of raw memory.

Whether a give offset addresses a high-order or low-order byte of an aligned short in memory is determined by the value of the `javafx.realtime.RealtimeSystem.BYTE_ORDER`¹²³ static byte variable in class `javafx.realtime.RealtimeSystem`¹²⁴, the start address of the object, and the offset given the stride of the object. Regardless of the byte ordering, accessor methods continue to select bytes starting at offset from the base address and continuing toward greater addresses.

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

Atomic loads and stores on raw memory are defined in terms of physical memory. This memory may be accessible to threads outside the JVM and to non-programmed access (e.g., DMA). Consequently, atomic access must be supported by hardware. This specification is written with the assumption that all suitable hardware platforms support atomic loads from raw memory for aligned bytes, shorts, and ints. Atomic access beyond the specified minimum may be supported by the implementation.

Storing values into raw memory is more hardware-dependent than loading values. Many processor architectures do not support atomic stores of variables except for aligned stores of the processor's word size. For instance, storing a byte into memory might require reading a 32-bit quantity into a processor register, updating the register to reflect the new byte value, then restoring the whole 32-bit quantity. Changes to other bytes in the 32-bit quantity that take place between the load and the store are lost.

Some processors have mechanisms that can be used to implement an atomic store of a byte, but those mechanisms are often slow and not universally supported.

¹¹⁸Section 12.3.2.7

¹¹⁹Section 12.3.2.6.3

¹²⁰Section 12.3.2.6.3

¹²¹Section 12.3.1.17.1

¹²²Section 12.3.2.6.3

¹²³Section 14.2.2.3.1

¹²⁴Section 14.2.2.3

This class need not support unaligned access to data; but if it does, it is not require the implementation to make such access atomic. Accesses to data aligned on its natural boundary will be atomic if the processor implements atomic loads and stores of that data size.

Except where noted, accesses to raw memory are not atomic with respect to the memory or with respect to schedulable objects. A raw memory region could be updated by another schedulable object, or even unmapped in the middle of an access method, or even *removed* mid method.

The characteristics of raw-memory access are necessarily platform dependent. This specification provides a minimum requirement for the RTSJ platform, but it also supports optional system properties that identify a platform's level of support for atomic raw put and get. The properties represent a four-dimensional sparse array of access type, data type, alignment, and atomicity with boolean values indicating whether that combination of access attributes is atomic. The default value for array entries is false. The dimension are

Table 12.1: Properties Array

Attribute	Values	Comment
Access type	read, write	
Data type	byte, short, int, long, float, double	
Alignment	0	aligned
	1 to one less than data type size	the first byte of the data is <i>alignment</i> bytes away from natural alignment.
Atomicity	processor	means access is atomic with respect to other taska on processor.
	smp	means access is <i>processor</i> atomic, and atomic with respect to all processors in an SMP.
	memory	means that access is <i>smp</i> atomic, and atomic with respect to all access to the memory including DMA.

The true values in the table are represented by properties of the following form.
 javax.realtime.atomicaccess_<access>_<type>_<alignment>_atomicity=true
 for example,

javax.realtime.atomicaccess_read_byte_0_memory=true

Table entries with a value of false may be explicitly represented, but since false

is the default value, such properties are redundant.

All raw memory access is treated as volatile, and *serialized*. The 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

12.3.2.6.1 Fields

MEMORY__MAPPED__REGION

```
public static final 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 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.

12.3.2.6.2 Constructors

RawMemoryFactory

Signature

```
public  
RawMemoryFactory()
```

Description

Create an empty factory. For a factory with support for the platform defined regions, use `getDefaultFactory`¹²⁵ instead.

12.3.2.6.3 Methods

getDefaultFactory*Signature*

```
public static javax.realtime.device.RawMemoryFactory  
getDefaultFactory()
```

Description

Get 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

Add support for a new memory region

Parameters

factory is the `RawMemoryRegionFactory`¹²⁶ to use for creating `RawMemory`¹²⁷ objects for the `RawMemoryRegion`¹²⁸s it makes available.

Throws

`RegistrationException` when the factory already is already registered.

¹²⁵Section [12.3.2.6.3](#)

¹²⁶Section [12.3.1.17](#)

¹²⁷Section [12.3.1.16](#)

¹²⁸Section [12.3.2.7](#)

deregister(RawMemoryRegionFactory)

Signature

```
public void  
deregister(RawMemoryRegionFactory factory)  
throws DeregistrationException
```

Description

Remove support for a new memory region

Parameters

factory is the [RawMemoryRegionFactory](#)¹²⁹ to make unavailable.

Throws

[RegistrationException](#) when the factory is not registered.

createRawByte(RawMemoryRegion, long, int, int)

Signature

```
public javafx.realtime.device.RawByte  
createRawByte(RawMemoryRegion region,  
               long base,  
               int count,  
               int stride)  
throws SecurityException,  
       OffsetOutOfBoundsException,  
       SizeOutOfBoundsException,  
       MemoryTypeConflictException,  
       UnsupportedRawMemoryRegionException
```

Description

Create an instance of a class that implements [RawByte](#)¹³⁰ 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.

¹²⁹Section [12.3.1.17](#)

¹³⁰Section [12.3.1.1](#)

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 `RawByte131` and supports access to the specified range in the memory region.

createRawByteReader(RawMemoryRegion, long, int, int)

Signature

```
public javax.realtime.device.RawByteReader
createRawByteReader(RawMemoryRegion region,
                    long base,
                    int count,
                    int stride)
throws SecurityException,
       OffsetOutOfBoundsException,
       SizeOutOfBoundsException,
       MemoryTypeConflictException,
       UnsupportedRawMemoryRegionException
```

Description

Create an instance of a class that implements `RawByteReader132` 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.

¹³¹Section 12.3.1.1

¹³²Section 12.3.1.2

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 [RawByteReader](#)¹³³ and supports access to the specified range in the memory region.

createRawByteWriter(RawMemoryRegion, long, int, int)*Signature*

```
public javafx.realtime.device.RawByteWriter  
createRawByteWriter(RawMemoryRegion region,  
                    long base,  
                    int count,  
                    int stride)  
  
throws SecurityException,  
        OffsetOutOfBoundsException,  
        SizeOutOfBoundsException,  
        MemoryTypeConflictException,  
        UnsupportedRawMemoryRegionException
```

Description

Create an instance of a class that implements [RawByteWriter](#)¹³⁴ and accesses

¹³³Section [12.3.1.2](#)

¹³⁴Section [12.3.1.3](#)

memory of region in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is $\text{stride} * \text{size of RawByteWriter} * \text{count}$. 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 **RawByteWriter**¹³⁵ and supports access to the specified range in the memory region.

createRawShort(RawMemoryRegion, long, int, int)

Signature

```
public javax.realtime.device.RawShort
createRawShort(RawMemoryRegion region,
               long base,
               int count,
               int stride)
throws SecurityException,
       OffsetOutOfBoundsException,
       SizeOutOfBoundsException,
       MemoryTypeConflictException,
```

¹³⁵Section 12.3.1.3

UnsupportedRawMemoryRegionException

Description

Create an instance of a class that implements [RawShort](#)¹³⁶ 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 $\text{stride} * \text{size of RawShort} * \text{count}$. 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 [RawShort](#)¹³⁷ and supports access to the specified range in the memory region.

createRawShortReader(RawMemoryRegion, long, int, int)*Signature*

```
public javafx.realtime.device.RawShortReader  
createRawShortReader(RawMemoryRegion region,  
                     long base,  
                     int count,  
                     int stride)
```

¹³⁶Section [12.3.1.18](#)

¹³⁷Section [12.3.1.18](#)

throws `SecurityException`,
`OffsetOutOfBoundsException`,
`SizeOutOfBoundsException`,
`MemoryTypeConflictException`,
`UnsupportedRawMemoryRegionException`

Description

Create an instance of a class that implements `RawShortReader`¹³⁸ and accesses memory of region in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is `stride * size of RawShortReader * count`. The object is allocated in the current memory area of the calling thread.

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 `RawShortReader`¹³⁹ and supports access to the specified range in the memory region.

createRawShortWriter(RawMemoryRegion, long, int, int)

Signature

¹³⁸Section 12.3.1.19

¹³⁹Section 12.3.1.19

```
public javax.realtime.device.RawShortWriter
createRawShortWriter(RawMemoryRegion region,
                    long base,
                    int count,
                    int stride)
throws SecurityException,
       OffsetOutOfBoundsException,
       SizeOutOfBoundsException,
       MemoryTypeConflictException,
       UnsupportedRawMemoryRegionException
```

Description

Create an instance of a class that implements [RawShortWriter](#)¹⁴⁰ 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 $\text{stride} * \text{size of RawShortWriter} * \text{count}$. 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 [RawShortWriter](#)¹⁴¹ and supports access to the specified range in the memory region.

¹⁴⁰Section [12.3.1.20](#)

¹⁴¹Section [12.3.1.20](#)

createRawInt(RawMemoryRegion, long, int, int)*Signature*

```
public javax.realtime.device.RawInt
createRawInt(RawMemoryRegion region,
             long base,
             int count,
             int stride)
throws SecurityException,
       OffsetOutOfBoundsException,
       SizeOutOfBoundsException,
       MemoryTypeConflictException,
       UnsupportedRawMemoryRegionException
```

Description

Create an instance of a class that implements [RawInt](#)¹⁴² 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 $\text{stride} * \text{size of RawInt} * \text{count}$. 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

¹⁴²Section [12.3.1.10](#)

an object that implements [RawInt](#)¹⁴³ 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

Create an instance of a class that implements [RawIntReader](#)¹⁴⁴ 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 $\text{stride} * \text{size of RawIntReader} * \text{count}$. 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.

¹⁴³Section [12.3.1.10](#)

¹⁴⁴Section [12.3.1.11](#)

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**¹⁴⁵ and supports access to the specified range in the memory region.

createRawIntWriter(RawMemoryRegion, long, int, int)

Signature

```
public javax.realtime.device.RawIntWriter
createRawIntWriter(RawMemoryRegion region,
                  long base,
                  int count,
                  int stride)
throws SecurityException,
       OffsetOutOfBoundsException,
       SizeOutOfBoundsException,
       MemoryTypeConflictException,
       UnsupportedRawMemoryRegionException
```

Description

Create an instance of a class that implements **RawIntWriter**¹⁴⁶ 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

¹⁴⁵Section 12.3.1.11

¹⁴⁶Section 12.3.1.12

`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`¹⁴⁷ and supports access to the specified range in the memory region.

createRawLong(RawMemoryRegion, long, int, int)

Signature

```
public javafx.realtime.device.RawLong  
createRawLong(RawMemoryRegion region,  
              long base,  
              int count,  
              int stride)  
throws SecurityException,  
       OffsetOutOfBoundsException,  
       SizeOutOfBoundsException,  
       MemoryTypeConflictException,  
       UnsupportedRawMemoryRegionException
```

Description

Create an instance of a class that implements `RawLong`¹⁴⁸ 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.

¹⁴⁷Section 12.3.1.12

¹⁴⁸Section 12.3.1.13

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`¹⁴⁹ and supports access to the specified range in the memory region.

createRawLongReader(RawMemoryRegion, long, int, int)

Signature

```
public javax.realtime.device.RawLongReader
createRawLongReader(RawMemoryRegion region,
                    long base,
                    int count,
                    int stride)
throws SecurityException,
       OffsetOutOfBoundsException,
       SizeOutOfBoundsException,
       MemoryTypeConflictException,
       UnsupportedRawMemoryRegionException
```

Description

Create an instance of a class that implements `RawLongReader`¹⁵⁰ 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.

¹⁴⁹Section 12.3.1.13

¹⁵⁰Section 12.3.1.14

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](#)¹⁵¹ and supports access to the specified range in the memory region.

createRawLongWriter(RawMemoryRegion, long, int, int)*Signature*

```
public javafx.realtime.device.RawLongWriter  
createRawLongWriter(RawMemoryRegion region,  
                    long base,  
                    int count,  
                    int stride)  
  
throws SecurityException,  
        OffsetOutOfBoundsException,  
        SizeOutOfBoundsException,  
        MemoryTypeConflictException,  
        UnsupportedRawMemoryRegionException
```

Description

Create an instance of a class that implements [RawLongWriter](#)¹⁵² and accesses

¹⁵¹Section [12.3.1.14](#)

¹⁵²Section [12.3.1.15](#)

memory of region in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is $\text{stride} * \text{size of RawLongWriter} * \text{count}$. 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 **RawLongWriter**¹⁵³ 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,
```

¹⁵³Section 12.3.1.15

UnsupportedRawMemoryRegionException

Description

Create an instance of a class that implements [RawFloat](#)¹⁵⁴ 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 $\text{stride} * \text{size of RawFloat} * \text{count}$. 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](#)¹⁵⁵ and supports access to the specified range in the memory region.

createRawFloatReader(RawMemoryRegion, long, int, int)*Signature*

```
public javafx.realtime.device.RawFloatReader  
createRawFloatReader(RawMemoryRegion region,  
                     long base,  
                     int count,  
                     int stride)
```

¹⁵⁴Section [12.3.1.7](#)

¹⁵⁵Section [12.3.1.7](#)

throws `SecurityException`,
`OffsetOutOfBoundsException`,
`SizeOutOfBoundsException`,
`MemoryTypeConflictException`,
`UnsupportedRawMemoryRegionException`

Description

Create an instance of a class that implements `RawFloatReader`¹⁵⁶ 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.
`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 `RawFloatReader`¹⁵⁷ and supports access to the specified range in the memory region.

createRawFloatWriter(RawMemoryRegion, long, int, int)

Signature

¹⁵⁶Section 12.3.1.8

¹⁵⁷Section 12.3.1.8

```
public javax.realtime.device.RawFloatWriter
createRawFloatWriter(RawMemoryRegion region,
                    long base,
                    int count,
                    int stride)
throws SecurityException,
    OffsetOutOfBoundsException,
    SizeOutOfBoundsException,
    MemoryTypeConflictException,
    UnsupportedRawMemoryRegionException
```

Description

Create an instance of a class that implements [RawFloatWriter](#)¹⁵⁸ 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 $\text{stride} * \text{size of RawFloatWriter} * \text{count}$. 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 [RawFloatWriter](#)¹⁵⁹ and supports access to the specified range in the memory region.

¹⁵⁸Section [12.3.1.9](#)

¹⁵⁹Section [12.3.1.9](#)

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

Create an instance of a class that implements [RawDouble](#)¹⁶⁰ 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 $\text{stride} * \text{size of RawDouble} * \text{count}$. 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

¹⁶⁰Section [12.3.1.4](#)

an object that implements [RawDouble](#)¹⁶¹ 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

Create an instance of a class that implements [RawDoubleReader](#)¹⁶² 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 $\text{stride} * \text{size of RawDoubleReader} * \text{count}$. 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.

¹⁶¹Section [12.3.1.4](#)

¹⁶²Section [12.3.1.5](#)

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**¹⁶³ and supports access to the specified range in the memory region.

createRawDoubleWriter(RawMemoryRegion, long, int, int)

Signature

```
public javax.realtime.device.RawDoubleWriter
createRawDoubleWriter(RawMemoryRegion region,
                      long base,
                      int count,
                      int stride)
throws SecurityException,
       OffsetOutOfBoundsException,
       SizeOutOfBoundsException,
       MemoryTypeConflictException,
       UnsupportedRawMemoryRegionException
```

Description

Create an instance of a class that implements **RawDoubleWriter**¹⁶⁴ 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

¹⁶³Section 12.3.1.5

¹⁶⁴Section 12.3.1.6

`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`¹⁶⁵ and supports access to the specified range in the memory region.

12.3.2.7 RawMemoryRegion

Inheritance

`java.lang.Object`

`javafx.runtime.device.RawMemoryRegion`

Description

`RawMemoryRegion` is a class for typing raw memory regions. It is returned by the `RawMemoryRegionFactory.getRegion`¹⁶⁶ methods of the raw memory region factory classes, and it is used with methods such as `RawMemoryFactory.createRawByte(RawMemoryRegion, long, int, int)`¹⁶⁷ and `RawMemoryFactory.createRawDouble(RawMemoryRegion, long, int, int)`¹⁶⁸ methods to identify the region from which the application wants to get an accessor instance.

Available since RTSJ 2.0

12.3.2.7.1 Methods

¹⁶⁵Section 12.3.1.6

¹⁶⁶Section 12.3.1.17.1

¹⁶⁷Section 12.3.2.6.3

¹⁶⁸Section 12.3.2.6.3

getRegion(String)

Signature

```
public static javax.realtime.device.RawMemoryRegion  
getRegion(String name)
```

Description

Get a region type when it already exists or creates a new one.

Parameters

name of the region

Returns

the region type object.

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

Gets a printable representation for a Region.

Returns

the name of this memory region type.

12.4 Rationale

12.4.1 Raw Memory

Raw memory in the RTSJ refers to any memory in which only objects of primitive types can be stored; *Java objects or their references cannot be stored in raw memory*. 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.

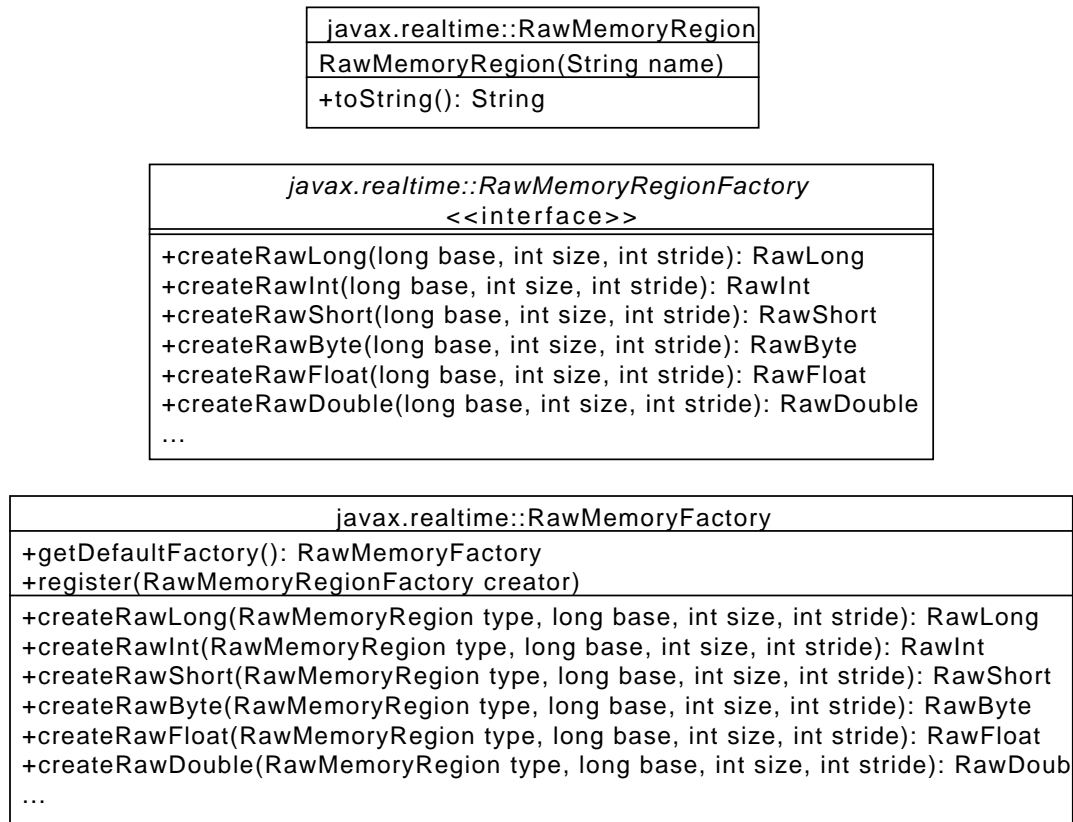
Each of these categories of memory is represented by an instance of `RawMemoryRegion`. In addition, the application can define other regions outside these two, either for accessing devices registers in some other address space or for other purposes, such as emulating device access.

Java's primitive types are partitioned into two groups: integral (short, int, long, byte) and real (float, double) types, including arrays of each type. For integral types, individual interfaces are also defined to facilitate greater type security during access. Objects that support these interfaces are created by factory methods, which again have predefined interfaces. Such objects are called *accessor* objects as they encapsulates the access protocol to the raw memory.

Control over all these objects is managed by the `RawMemoryFactory` class that provides a set of static methods, as shown in Figure 12.5. There are two groups of methods, those that

1. enable a factory to be registered, and
2. request the creation of *accessor* object for a particular memory type at a particular address.

Figure 12.5: Creating Raw Memory Accessors



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 a two device registers: a control/status register that is a 32 bits integer, and a data register that is a 64 bits long. The registers have been memory mapped to locations: `0x20` and `0x24` respectively. Assuming the realtime JVM has registered a factory for the `IO_MEMORY_MAPPED_REGION` raw memory name, then the following code will create the objects that facilitate the

memory access

```
1 RawMemoryFactory factor = RawMemoryFactory.getDefault();
2 RawInt controlReg =
3   factory.createRawInt(RawMemoryFactory.IO_MEMORY_MAPPED_REGION, 0x20);
4 RawLong dataReg =
5   factory.createRawLong(RawMemoryFactory.IO_MEMORY_MAPPED_REGION, 0x24);
```

The above definitions reflect the structure of the actual registers. The JVM will check that the memory locations are on the correct boundaries and that they can be accessed without any hardware exceptions being generated. If they cannot, the create methods will throw an appropriate exceptions. If successfully created, all future access to the controlReg and dataReg will be exception free. The registers can be manipulated by calling the appropriate methods, as in the following example.

```
1 dataReg.setLong(l);
2   // where l is of type long and is data to be sent to the device
3 controlReg.setInt(i);
4   // where i is of type int and is the command to the device
```

In the general case, programmers themselves may create their own memory categories and provide associated factories (that may use the implementation-defined factories). These factories are written in Java and are, therefore, constrained by what the language allows them to do. Typically, they will use the JVM-supplied raw memory types to facilitate access to a device's external memory.

The facilities provided by the RTSJ allow an application to supports the notion of removable memory. When this memory is inserted or removed, an asynchronous event can be set up to fire, thereby alerting the application that the device has become active. Of course, any removable memory has to be treated with extreme caution. Hence, the RTSJ facilities allows it only to be accessed as a raw memory device. An example of this will be given in Section [12.4.3](#).

12.4.1.1 Direct memory access

DMA requires access to memory out side of the heap. It is often crucial for performance in embedded systems; however, it does cause problems both from a realtime analysis perspective and from a JVM-implementation perspective. The latter is the primary concern here.

There are a few crucial points to note about DMA and the RTSJ.

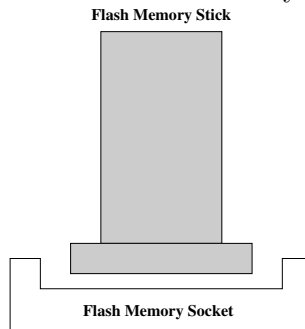
1. The RTSJ does not address issues of persistent objects; so the input and output of Java objects to devices (other than by using the Java serialization mechanism) is not supported.

2. The RTSJ requires that RTSJ programs can be compiled by regular Java compilers. Different bytecode compilers (and their supporting JVM) use different representation for objects. Java arrays (even of primitive types) are objects, and the data they contain might not be stored in contiguous memory.
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 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 12.6: Flash memory device



12.4.2 Interrupt Handling

Handling interrupts is a necessary part of many embedded systems. Interrupt handlers have traditionally been implemented in assembler code or C. With the growing popularity of high-level concurrent languages, there has been interest in

better integration between the interrupt handling code and the application. Ada, for example, allows a “protected” procedure to be called directly from an interrupt [3].

Regehr [7] defines the terms used for the core components of interrupts and their handlers as follows.

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 Controller¹⁶⁹. 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 usually be used to determine the worst-case stack usage of all interrupt handlers. If stack is borrowed from the executing thread then this amount must be added to the worst-case stack usage of all threads.
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]).

¹⁶⁹ See <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0375a/Cegbfjhf.html>

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.

12.4.3 An Illustrative Example

Consider an embedded system that has a simple flash memory device that supports a single type of removable flash memory stick, as illustrated in Figure 12.6.

When the memory stick is inserted or removed, an interrupt is generated. This interrupt is known to the realtime JVM. The interrupt is also generated when operations requested on the device are completed. For simplicity, it is assumed that the application has associated this interrupt to an happening called `FlashHappening` with a default happening dispatcher.

The example illustrates how

1. a programmer can use the RTSJ facilities to write a device handler,
2. a factory class can be constructed and how the accessor objects police the access,
3. removable memory can be handled.

The flash memory device is accessed via several associated registers, which are shown in Table 12.2. These have all been memory mapped to the indicated locations.

12.4.3.1 Software architecture

There are many ways in which the software architecture for the example could be constructed. Here, for simplicity of representation, an architecture is chosen with

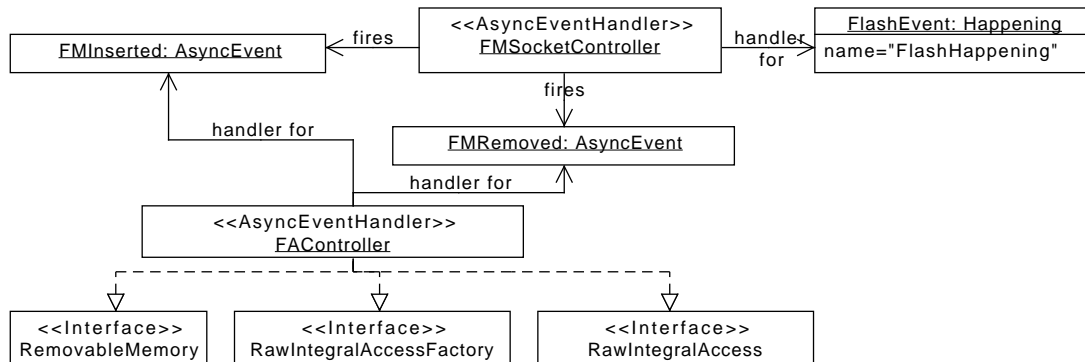
Table 12.2: Device registers

Register	Location	Bit Positions	Values
Command	0x20	0	0 = Disable device, 1 = Enable device
		4	0 = Disable interrupts, 1 = Enable interrupts
		5-8	1 = Read byte, 2 = Write byte 3 = Read short, 4 = Write short 5 = Read int, 6 = Write int 7 = Read long, 8 = Write long
		9	0 = DMA Read, 1 = DMA
		31-63	Offset into flash memory
Data	0x28	0-63	Simple data or memory address if DMA
Length	0x30	0-31	Length of data transfer
Status	0x38	0	1 = Device enabled
		3	1 = Interrupts enabled
		4	1 = Device in error
		5	1 = Transfer complete
		6	1 = Memory stick present 0 = Memory stick absent
		7	1 = Memory stick inserted
		8	0 = Memory stick removed

a minimal number of classes. It is illustrated in Figure 12.7. There are three key components.

1. FlashHappening—This is the happening that has been associated with the flash device’s interrupt. The RTSJ will provide a default dispatcher, which will release any associated handler when the interrupt occurs and the happening is triggered.
2. FMSocketController—This is the object that encapsulates the access to the flash memory device. In essence, it is the device driver; it is also the handler for the FlashHappening and is responsibly for firing the FMInserted and FMRemoved asynchronous events.
3. FAController—This is the object that controls access to the flash memory, it
 - (a) acts as the factory for the creating objects that will facilitate access to the flash memory itself (using the mechanisms provided by the FMSocketController),
 - (b) is the asynchronous event handler that responds to the firing of the FMInserted and FMRemoved asynchronous events, and
 - (c) also acts as the accessor object for the memory.

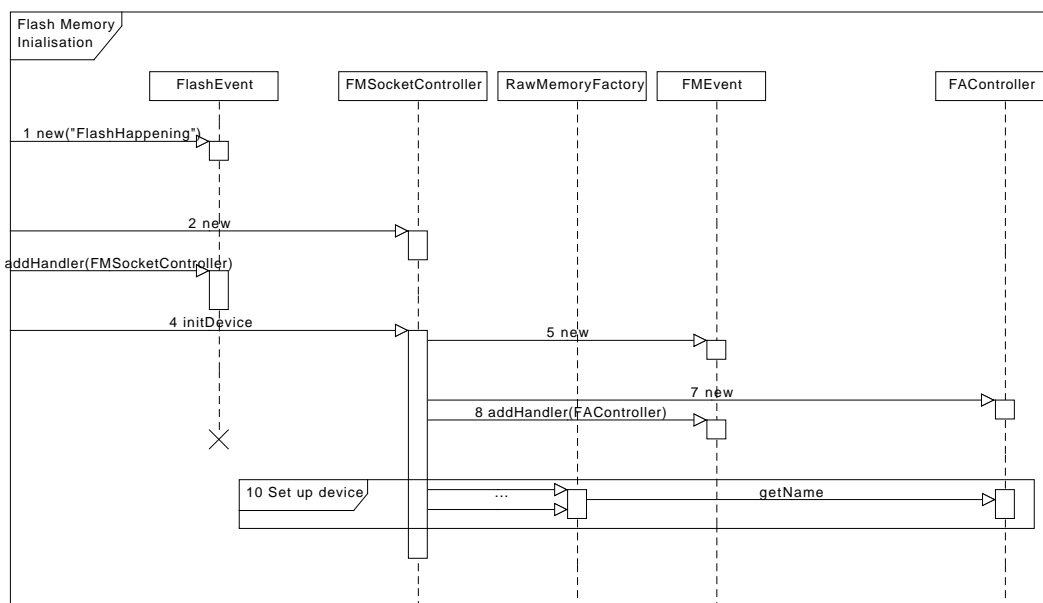
Figure 12.7: Flash memory classes



12.4.3.2 Device initialization

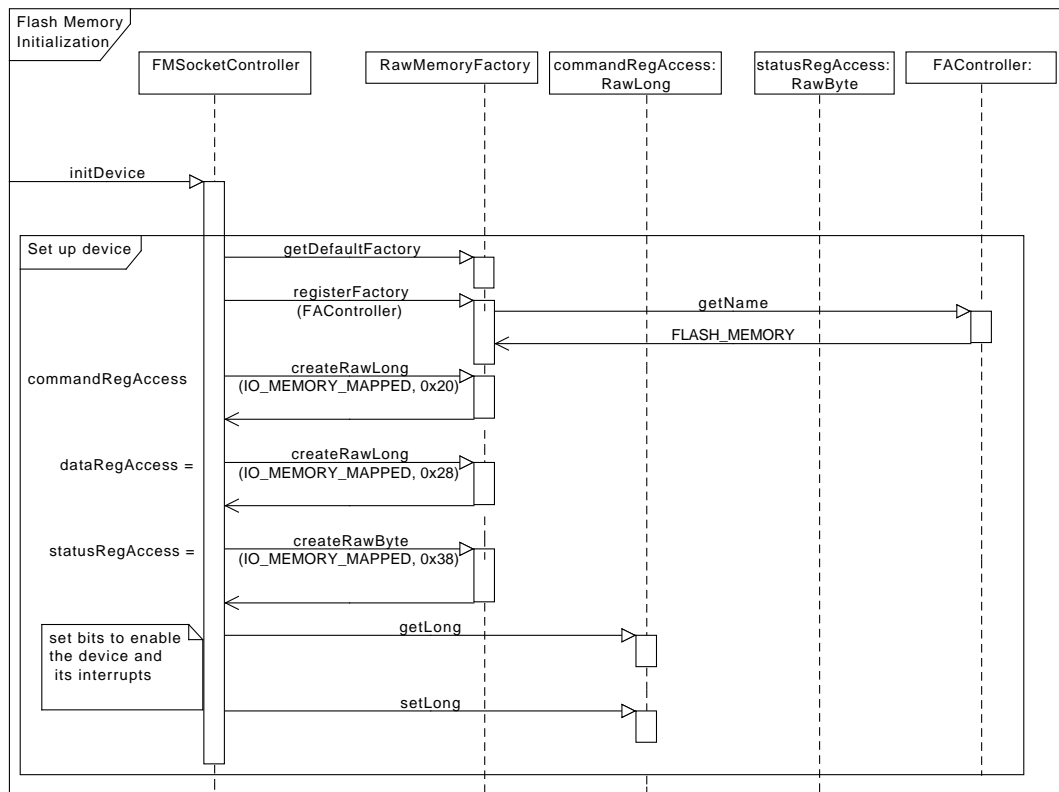
Figure 12.8 shows the sequence of operations that the program must perform to initialize the flash memory device. The main steps are as follows.

Figure 12.8: Sequence diagram showing initialization operations



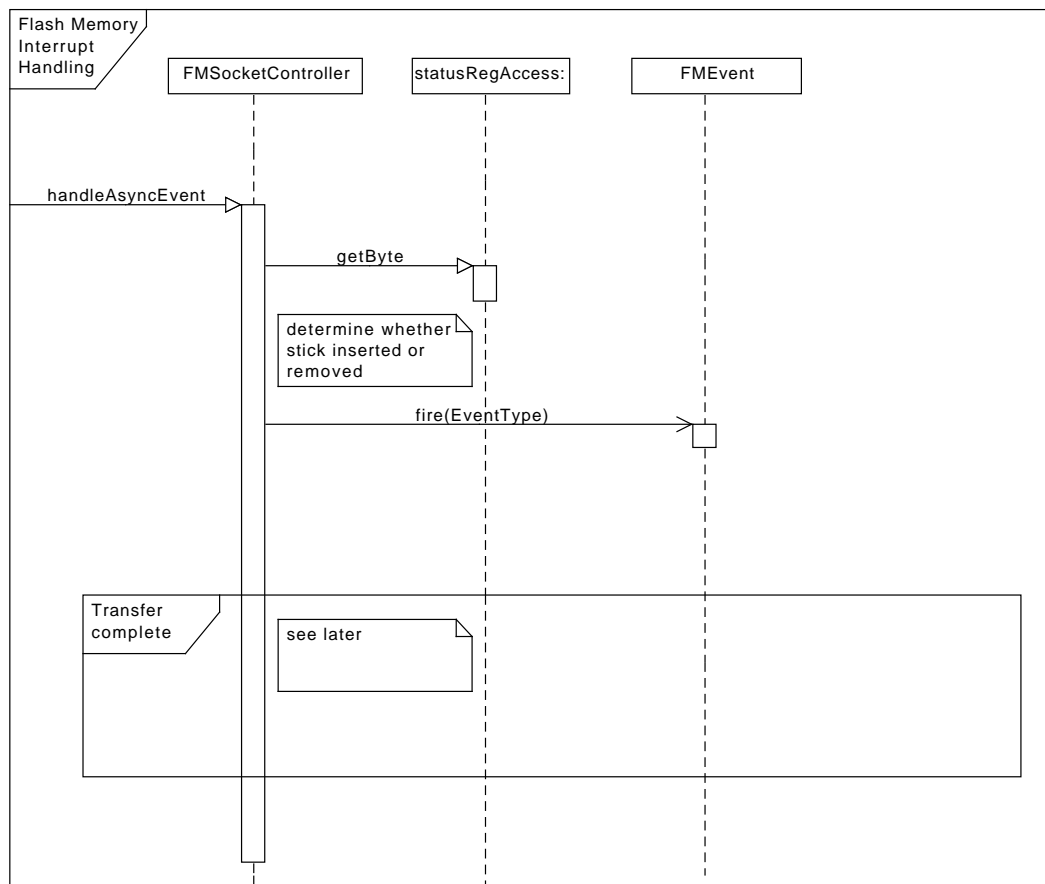
- 1 The happening (FlashEvent) associated with the flash happening must be created.

Figure 12.9: Sequence diagrams showing operations to initialize the hardware device



- 2-3 The (FMSocketController) object is created and added as a handler for FlashEvent.
- 4 An initialization method is called (`initDevice`) to perform all the operations necessary to configure the infrastructure and initialize the hardware device.
- 5-6 Two new asynchronous events are created to represent insertion and removal of the flash memory stick.
- 7-9 The FAController class is created. It is added as the handler for the two events created in steps 5 and 6.
- 10 Setting up the device and registering the factory is shown in detail in Figure 12.9. It involves registering the FAController object via the static methods in the RawMemoryFactory class and creating and using the JVM-supplied factory to access the memory-mapped I/O registers.

Figure 12.10: The FMSocketController.handleAsync method



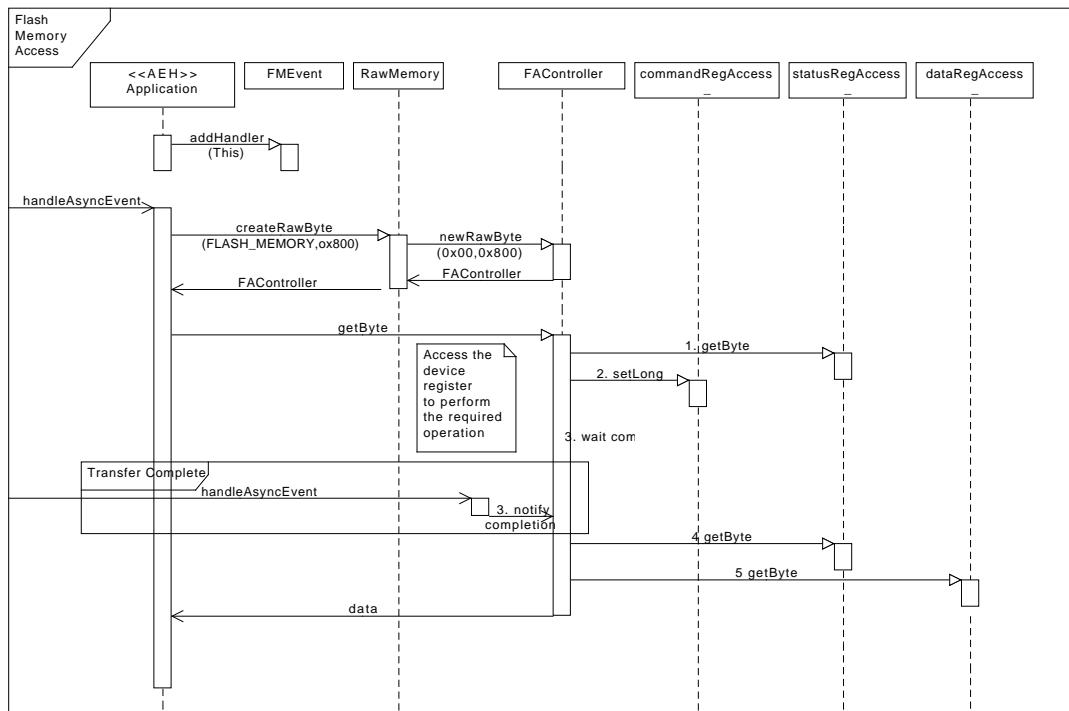
12.4.3.3 Responding to external happenings

In the example, interrupts are handled by the JVM, which turns them into an external happening. The application code that indirectly responds to the happening is provided in the `handleAsyncEvent` method in the `FMSocketController` object. Figure 12.10 illustrates the approach. In this example, the actions in response to the *memory stick inserted* and *memory stick removed* flash events is simply shown as the execution of the `FMInserted` and `FMRemoved` handlers. These will inform the application. The memory accessor classes themselves will ensure that the stick is present when performing the required application accesses.

12.4.3.4 Access to the flash controller's device registers

Figure 12.11 shows the sequence of events that the application follows. First it must register a handler with the `FMInserted` asynchronous event. Here, the application

Figure 12.11: Application usage



itself is an asynchronous event handler. When this is released, the memory has been inserted.

In this simple example, the application simply reads a byte from an offset within the memory stick. It, therefore, creates an accessor to access the data. When this has been returned (it is the *FAController* itself), the application can now call the *getByte* method (called *FA getByte*, in the following, for clarity). This method must implement the sequence of raw memory access on the device's registers to perform the operation. In Figure 12.11, they are as follows.

1. *FA getByte* calls the *getByte* method of the status register's accessor object. This can check to make sure that the flash memory is present (bit 6, as shown in Table 12.2). If it is not, an exception can be thrown.
2. Assuming the memory is present, it then sets the control register with the offset required (bits 31–63, as shown in Table 12.2) and sets the read byte request bit (bits 5–8, as shown in Table 12.2).
3. The *FA getByte* method must then wait for indication that the requested operation has been completed by the device. This is detected by the *handleAsyncEvent* method of the *FMController*, which performs the necessary notify.

4. Once notified of completion, the *FA* `getBytes` method, again reads the status register to make sure there were no errors on the device (bit 4 in Table 12.2) and that the memory is still present
5. The *FA* `getLong` then reads the data register to get the requested data, which it returns.

Chapter 13

Interprocess Signalling

On many operating systems, it is possible for one process to signal another. POSIX provides a well defined means of signalling other processes and receiving signals from them, therefore one would like to be able to use this facility when 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`.

13.1 Definitions

Signal — A notification between two system process, which may or may not contain a data packet.

Realtime Signal — A special type of signal that carries a bit of data with it.

13.2 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.

13.2.1 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.

13.2.2 POSIX Realtime Signals

The `RealtimeSignal` class represents POSIX realtime events. It also implements `ActiveEvent`, but is 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.

13.3 javax.realtime.posix

13.3.1 Classes

13.3.1.1 RealtimeSignal

Inheritance

java.lang.Object
 javax.realtime.AsyncBaseEvent
 javax.realtime.AsyncLongEvent
 javax.realtime.posix.RealtimeSignal

Interfaces

 javax.realtime.ActiveEvent

Description

A [javax.realtime.ActiveEvent¹](#) subclass for defining a POSIX realtime signal.

Available since RTSJ 2.0

13.3.1.1.1 Methods

isPOSIXRealtimeSignal(String)

Signature

```
public static boolean  
isPOSIXRealtimeSignal(String name)
```

Description

Determine if a signal with a given name is registered.

Parameters

name of the signal

Returns

true when a signal with the given name is registered

¹Section [8.3.1.1](#)

getId(String)

Signature

```
public static int  
getId(String name)
```

Description

Get the ID of a registered signal.

Parameters

name of the signal for which to search

Returns

the ID of the signal named by name

get(String)

Signature

```
public static javax.realtime.posix.RealtimeSignal  
get(String name)
```

Description

Get the registered realtime signal with the given name.

Parameters

name of the signal to get.

Returns

the registered signal with name or null.

get(int)

Signature

```
public static javax.realtime.posix.RealtimeSignal  
get(int id)
```

Description

Get the realtime signal corresponding to a given id.

Parameters

id of a registered signal

Returns

the signal corresponding to id.

getId

Signature

```
public int  
getId()
```

Description

Get the name of this realtime signal.

Returns

the ID of this signal.

getName

Signature

```
public final java.lang.String  
getName()
```

Description

Get the name of this signal.

Returns

the name of this signal.

getDispatcher

Signature

```
public javax.realtime.posix.RealtimeSignalDispatcher  
getDispatcher()
```

Description

Obtain the dispatcher for this.

Returns

that dispatcher.

isActive*Signature*

```
public boolean  
isActive()
```

Description

Determine the activation state of this signal, i.e., it has been started.

Returns

true when active, false otherwise.

isRunning*Signature*

```
public boolean  
isRunning()
```

Description

Determine the firing state (releasing or skipping) of this signal, i.e., it is active and enabled.

Returns

true when releasing, false when skipping.

start*Signature*

```
public final synchronized void  
start()  
throws IllegalStateException
```

Description

Start this RealtimeSignal, i.e., change to a running state. A running realtime signal is a source of activation when in a scoped memory and is a member of the root set when in the heap. A running realtime signal can be triggered.

Throws

IllegalStateException when this RealtimeSignal has already been started.

See Section [stop\(\)](#)

start(boolean)

Signature

```
public final synchronized void  
start(boolean disabled)  
throws IllegalStateException
```

Description

Start this RealtimeSignal, i.e., change to a running state. A running realtime signal is a source of activation when in a scoped memory and is a member of the root set when in the heap. A running realtime signal can be triggered.

Parameters

disabled true for starting in a disabled state.

Throws

IllegalStateException when this RealtimeSignal has already been started.

See [Section stop\(\)](#)

stop

Signature

```
public final boolean  
stop()  
throws IllegalStateException
```

Description

Stop this RealtimeSignal. A stopped realtime signal ceases to be a source of activation and no longer cause any AE attached to it to be a source of activation.

Throws

IllegalStateException when this RealtimeSignal is not running.

Returns

true when this was *enabled* and false otherwise.

send(long, long)

Signature

```
public native boolean  
send(long pid,  
      long payload)
```

Description

Send this signal to another process

Parameters

pid of the process to which to send the signal

Returns

true when signal can be sent, otherwise false.

13.3.1.2 RealtimeSignalDispatcher

Inheritance

```
java.lang.Object  
  javax.realtime.ActiveEventDispatcher  
    javax.realtime.posix.RealtimeSignalDispatcher
```

Description

Provides a means of dispatching a set of [RealtimeSignal²](#)s. An application can provide its own dispatcher, providing the priority for the internal dispatching thread. This dispatching thread calls `process()` each time the signal is triggered.

Available since RTSJ 2.0

13.3.1.2.1 Constructors

RealtimeSignalDispatcher(SchedulingParameters, Scheduling-Group)

Signature

²Section [13.3.1.1](#)


```
public  
RealtimeSignalDispatcher(SchedulingParameters schedule,  
                          SchedulingGroup group)
```

Description

Create a new dispatcher, whose dispatching thread runs with the given scheduling parameters.

Parameters

schedule give the parameters for scheduling this dispatcher

RealtimeSignalDispatcher(SchedulingParameters)

Signature

```
public  
RealtimeSignalDispatcher(SchedulingParameters schedule)
```

Description

Create a new dispatcher, whose dispatching thread runs with the given scheduling parameters.

Parameters

schedule give the parameters for scheduling this dispatcher

13.3.1.2.2 Methods

register(RealtimeSignal)

Signature

```
public void  
register(RealtimeSignal signal)  
throws RegistrationException,  
        IllegalStateException,  
        IllegalArgumentException
```

Description

Register signal with this dispatcher.

Parameters

signal to register

Throws

[RegistrationException](#) when signal is already registered.

[IllegalStateException](#) when this object has been destroyed.

[IllegalArgumentException](#) when signal is not stopped.

deregister(RealtimeSignal)

Signature

```
public void  
deregister(RealtimeSignal signal)  
throws DeregistrationException,  
        IllegalStateException,  
        IllegalArgumentException
```

Description

Deregister the signal form this dispatcher.

Parameters

signal to unregister

Throws

[DeregistrationException](#) when signal is not already registered.

[IllegalStateException](#) when this object has been destroyed.

[IllegalArgumentException](#) when signal is not stopped.

destroy

Signature

```
public void  
destroy()  
throws IllegalStateException
```

Description

Release all reasources thereby making the dispatcher unusable.

Throws

IllegalStateException when called on a dispatcher that has one or more registered [RealtimeSignal](#)³ objects.

13.3.1.3 Signal

Inheritance

java.lang.Object
 [javax.realtime.AsyncBaseEvent](#)
 [javax.realtime.AsyncEvent](#)
 [javax.realtime.posix.Signal](#)

Interfaces

[javax.realtime.ActiveEvent](#)

Description

A [javax.realtime.ActiveEvent](#)⁴ subclass for defining a POSIX signal.

Available since RTSJ 2.0

13.3.1.3.1 Fields

MAX_NUM_SIGNALS

public static final MAX_NUM_SIGNALS

Description

this number of signals can be processed.

13.3.1.3.2 Methods

³Section [13.3.1.1](#)

⁴Section [8.3.1.1](#)

isPOSIXSignal(String)

Signature

```
public static boolean  
isPOSIXSignal(String name)
```

Description

Determine if a signal with a given name is registered.

Parameters

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

Get the ID of a registered signal.

Parameters

name of the signal for which to search

Returns

the ID of the signal named by name

get(String)

Signature

```
public static javax.realtime.posix.Signal  
get(String name)
```

Description

Get the registered signal with the given name.

Parameters

name of the signal to get.

Returns

the registered signal with name or null.

get(int)

Signature

```
public static javax.realtime.posix.Signal  
get(int id)
```

Description

Get the signal corresponding to a given id.

Parameters

id of a registered signal

Returns

the signal corresponding to id or null.

getProcessId

Signature

```
public static long  
getProcessId()
```

Description

Obtain 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)`⁵.

Returns

the OS process id.

getId

Signature

```
public int  
getId()
```

Description

Get the number of this signal.

Returns

the signal number

⁵Section 13.3.1.3.2

getName*Signature*

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

Description

Get the name of this signal.

Returns

the name of this signal.

getDispatcher*Signature*

```
public javax.realtime.posix.SignalDispatcher  
getDispatcher()
```

Description

Obtain the dispatcher for this.

Returns

that dispatcher.

isActive*Signature*

```
public boolean  
isActive()
```

Description

Determine the activation state of this signal, i.e., it has been started.

Returns

true when active, false otherwise.

isRunning

Signature

```
public boolean  
isRunning()
```

Description

Determine the firing state (releasing or skipping) of this signal, i.e., it is active and enabled.

Returns

true when releasing, false when skipping.

start

Signature

```
public void  
start()  
throws IllegalStateException
```

Description

Start this Signal, i.e., change to a running state. A running signal is a source of activation when in a scoped memory and is a member of the root set when in the heap. A running signal can be triggered.

Throws

IllegalStateException when this Signal has already been started.

[See Section stop\(\)](#)

start(boolean)

Signature

```
public void  
start(boolean disabled)  
throws IllegalStateException
```

Description

Start this Signal, i.e., change to a running state. A running signal is a source of activation when in a scoped memory and is a member of the root set when in the heap. A running signal can be triggered.

Parameters

disabled true for starting in a disabled state.

Throws

IllegalStateException when this Signal has already been started.

See [Section stop\(\)](#)

stop

Signature

```
public boolean  
stop()  
throws IllegalStateException
```

Description

Stop this Signal. A stopped signal ceases to be a source of activation and no longer cause any AE attached to it to be a source of activation.

Throws

IllegalStateException when this Signal is not running.

Returns

true when this was *enabled* and false otherwise.

send(long)

Signature

```
public void  
send(long pid)  
throws POSIXInvalidSignalException,  
       POSIXSignalPermissionException,  
       POSIXInvalidTargetException
```

Description

Send 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.

13.3.1.4 SignalDispatcher

Inheritance

`java.lang.Object`

`javax.realtime.ActiveEventDispatcher`

`javax.realtime.posix.SignalDispatcher`

Description

Provides a means of dispatching a set of `Signal`⁶s. 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.

13.3.1.4.1 Constructors

⁶Section 13.3.1.3

SignalDispatcher(SchedulingParameters, SchedulingGroup)*Signature*

```
public  
SignalDispatcher(SchedulingParameters scheduling,  
                 SchedulingGroup group)
```

Description

Create a new dispatcher, whose dispatching thread runs with the given scheduling parameters.

Parameters

scheduling give the parameters for scheduling this dispatcher

SignalDispatcher(SchedulingParameters)*Signature*

```
public  
SignalDispatcher(SchedulingParameters scheduling)
```

Description

Create a new dispatcher, whose dispatching thread runs with the given scheduling parameters.

Parameters

scheduling give the parameters for scheduling this dispatcher

13.3.1.4.2 Methods

register(Signal)*Signature*

```
public synchronized void  
register(Signal signal)
```

throws `RegistrationException`,
`IllegalStateException`,
`IllegalArgumentException`

Description

Register a POSIX signal with this dispatcher.

Parameters

signal to register

Throws

`RegistrationException` when signal is already registered.

`IllegalStateException` when this object has been destroyed.

`IllegalArgumentException` when signal is not stopped.

deregister(Signal)

Signature

```
public synchronized void  
deregister(Signal signal)  
throws DeregistrationException,  
       IllegalStateException,  
       IllegalArgumentException
```

Description

Deregister a POSIX Signal form this dispatcher. (This is a really naive implementation.)

Parameters

signal to deregister

Throws

`DeregistrationException` when signal not is already registered.

`IllegalStateException` when this object has been destroyed.

`IllegalArgumentException` when signal is not stopped.

destroy

Signature

```
public void  
destroy()
```

throws `IllegalStateException`

Description

Release all resources thereby making the dispatcher unusable.

Throws

`IllegalStateException` when called on a dispatcher that has one or more registered `Signal`⁷ objects.

13.4 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.

⁷Section 13.3.1.3

Chapter 14

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. Even the ones that do vary in their degree of compliance. Also, the type of garbage collection provided may also vary from on implementation to another. The specification defines classes to help manage these differences providing the following:

- a class that contains operations and semantics that affect the entire system;
- the security semantics required by the additional features in the entirety of this specification, which are additional to those required by implementations of the Java Language Specification; and
- a class that provides some basic information about the garbage collector.

14.1 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.

14.1.1 `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.

14.1.2 RealtimeSecurity

The `RealtimeSecurity` class controls access to key realtime features. Particularly critical is access to memory outside the heap. Core RTSJ features also have security checks. This should enable an application to restrict the use of the RTSJ, particularly for dynamically loaded code. 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.

Open issue 14.1.1 (jjh)

Look at security closely. Try to make this a Java Security Manager if possible. Here is an initial list of methods that should have security checks.

End of issue 14.1.1

Open issue 14.1.2 (jjh)

Check Scheduler to see what security checks it needs.

End of issue 14.1.2

Open issue 14.1.3 (elb)

Check ThreadGroup/SchedulingGroup/ProcessingGroup to see what security checks they need.

End of issue 14.1.3

- Core Module
 - `ActiveEvent.start`
 - `ActiveEvent.stop`
 - `ActiveEventDispatcher.register`
 - `ActiveEventDispatcher.deregister`
 - `Affinity.applyTo`
 - `Affinity.set`
 - `AsyncBaseEvent.enable`
 - `AsyncBaseEvent.disable`
 - `AsyncBaseEvent.addHandler`
 - `AsyncBaseEvent.removeHandler`
 - `AsyncBaseEvent.setHandler`
 - `Schedulable.setDaemon` `RealtimeSecurity.checkAEHSetDaemon`
 - `Clock.Clock`
 - `MemoryArea.executeInArea`
 - `MemoryArea.enter`
 - `MonitorControl.setMonitorControl` `RealtimeSecurity.checkSetMonitorControl`
 - `RealtimeThread.RealtimeThread` `RealtimeSecurity.checkCreateRealtimeThread`
 - `Scheduler.setDefaultScheduler` `RealtimeSecurity.checkSetScheduler`
 - `Timer.Timer` `RealtimeSecurity.checkCreateTimer`

- POSIX Module
 - `RealtimeSignal.get`
 - `RealtimeSignal.getId`
 - `RealtimeSignal.send`
 - `RealtimeSignal.getDispatcher`
 - `RealtimeSignal.start`
 - `RealtimeSignal.stop`
 - `Signal.get`
 - `Signal.getId`
 - `Signal.send`
 - `Signal.getDispatcher`
 - `Signal.start`
 - `Signal.stop`
- Alternate Memory Module
 - `PhysicalMemoryFactory.associate` `RealtimeSecurity.checkAccessPhysical` `RealtimeSecurity.checkAccessPhysicalRange`
 - `PhysicalMemoryFactory.createImmortalMemory`
 - `PhysicalMemoryFactory.createLTMemory`
 - `PhysicalMemoryFactory.createPinnableMemory`
 - `PhysicalMemoryFactory.createStackedMemory`
 - `LTMemory.LTMemory`
 - `PinnableMemory.PinnableMemory`
 - `StackedMemory.StackedMemory`
- Device Module
 - `DMABufferFactory.DMABufferFactory`
 - `Happening.createId`
 - `Happening.getHappening`
 - `Happening.get`
 - `Happening.createId`
 - `Happening.Happening`
 - `RawMemoryFactory.register`
 - `RawMemoryFactory.deregister`
 - `RawMemoryFactory.RawMemoryFactory`
 - `RawMemoryFactory.createRawByte`
 - `RawMemoryFactory.createRawByteReader`
 - `RawMemoryFactory.createRawByteWriter`
 - ...
 - `RawMemoryFactory.createRawDouble`
 - `RawMemoryFactory.createRawDoubleReader`
 - `RawMemoryFactory.createRawDoubleWriter`

14.1.3 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.

14.1.4 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".

14.2 javax.realtime

14.2.1 Enumerations

14.2.1.1 RTSJModule

Inheritance

java.lang.Object
 java.lang.Enum
 javax.realtime.RTSJModule

Description

Modules an RTSJ implementation may provide.

14.2.1.1.1 Enumeration Constants

CORE

public static final CORE

Description

Indicates the presence of the core module.

DEVICE

public static final DEVICE

Description

Indicates the presence of the device access module.

MEMORY

public static final MEMORY

Description

Indicates the presence of the alternative memory areas module.

POSIX

public static final POSIX

Description

Indicates the presence of the POSIX module.

SCJ

public static final SCJ

Description

Indicates the presence of the Safety-Critical Java module.

14.2.1.1.2 Methods

values*Signature*

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

*Description***valueOf(String)***Signature*

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

Description

value

Signature

```
public int  
value()
```

Description

Determine 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, see 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.

14.2.2 Classes

14.2.2.1 GarbageCollector

Inheritance

java.lang.Object

javax.realtime.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 system. This information shall be made available to applications

via methods on subclasses of `GarbageCollector`. Implementations are allowed to provide any set of methods in subclasses as long as the temporal behavior and overhead are sufficiently categorized. The implementations are also required to fully document the subclasses.

A reference to the garbage collector responsible for heap memory is available from `RealtimeSystem.currentGC()`¹.

14.2.2.1.1 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 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.

¹Section 14.2.2.3.2

14.2.2.2 RealtimeSecurity

Inheritance

java.lang.Object
javax.realtime.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.setSecurityManager²](#). 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.

14.2.2.2.1 Constructors

RealtimeSecurity

Signature

```
public  
RealtimeSecurity()
```

Description

Create an RealtimeSecurity object.

14.2.2.2.2 Methods

²Section [14.2.2.3.2](#)

checkAccessPhysical

Signature

```
public void  
checkAccessPhysical()  
throws SecurityException
```

Description

Check whether the application is allowed to access physical memory.

Throws

SecurityException The application doesn't have permission to access physical memory.

checkAccessPhysicalRange(long, long)

Signature

```
public void  
checkAccessPhysicalRange(long base,  
                           long size)  
throws SecurityException
```

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.

Throws

SecurityException The application doesn't have permission to access the memory in the given range.

checkSetFilter

Signature

```
public void  
checkSetFilter()  
throws SecurityException
```

Description

Checks whether the application is allowed to register [PhysicalMemoryTypeFilter](#)³ objects with the [PhysicalMemoryManager](#)⁴.

Throws

SecurityException The application doesn't have permission to register filter objects.

checkSetMonitorControl(MonitorControl)

Signature

```
public void  
checkSetMonitorControl(MonitorControl policy)  
throws SecurityException
```

Description

Checks whether the application is allowed to set the default monitor control policy.

Parameters

policy The new policy

Throws

SecurityException when the application doesn't have permission to change the default monitor control policy to policy.

Available since RTSJ 1.0.1

checkAEHSetDaemon

Signature

```
public void  
checkAEHSetDaemon()  
throws SecurityException
```

Description

Checks whether the application is allowed to set the daemon status of an AEH.

Throws

SecurityException when the application is not permitted to alter the daemon status.

³Section [A.2.1.1](#)

⁴Section [A.2.3.19](#)

Available since RTSJ 1.0.1

checkSetScheduler

Signature

```
public void  
checkSetScheduler()  
throws SecurityException
```

Description

Checks whether the application is allowed to set the scheduler.

Throws

SecurityException The application doesn't have permission to set the scheduler.

checkCreateRealtimeThread

Signature

```
public void  
checkCreateRealtimeThread()  
throws SecurityException
```

Description

Check if an application may create a realtime thread.

Throws

SecurityException when not allowed

Available since RTSJ 2.0

checkCreateTimer

Signature

```
public void  
checkCreateTimer()  
throws SecurityException
```

Description

Check if an application may create a Timer.

Throws

SecurityException when not allowed.

Available since RTSJ 2.0

checkPOSIXSendSignal(Signal, long)

Signature

```
public void  
checkPOSIXSendSignal(Signal signal,  
                     long pid)  
throws SecurityException
```

Description

Check if the given signal can be sent to the given process id.

Parameters

signal is the signal being sent
pid is the id to which the signal is being set.

Throws

SecurityException when the operation is not allowed.

Available since RTSJ 2.0

14.2.2.3 RealtimeSystem

Inheritance

```
java.lang.Object  
  javax.realtime.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.

14.2.2.3.1 Fields

BIG_ENDIAN

public static final BIG_ENDIAN

Description

Value indicating that the highest order byte of a bit word is stored at the lowest byte address: the int 0x0A0B0C0D is stored in the byte sequence 0x0A, 0x0B, 0x0C, 0x0D. and the long 0x0102030405060708 is stored in the sequence 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08.

LITTLE_ENDIAN

public static final LITTLE_ENDIAN

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

public static final 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.

BYTE_ORDER

public static final BYTE_ORDER

Description

The byte ordering of the underlying hardware.

14.2.2.3.2 Methods

currentGC

Signature

```
public static javax.realtime.GarbageCollector  
currentGC()
```

Description

Return a reference to the currently active garbage collector for the heap.

Returns

A [GarbageCollector](#)⁵ 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, return -1. Note that when the number of concurrent locks is not tracked, the number of available concurrent locks is effectively unlimited.

getMaximumConcurrentLocks

Signature

⁵Section [14.2.2.1](#)

```
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 give 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.

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](#)⁶ object representing the default realtime security manager.

setMaximumConcurrentLocks(int)

Signature

```
public static void  
setMaximumConcurrentLocks(int numLocks)
```

Description

⁶Section [14.2.2.2](#)

Sets the anticipated maximum number of locks that may be held or waited on concurrently. Provide a hint to systems that use a monitor cache as to how much space to dedicate to the cache.

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\(\)](#)⁷.

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. Provide 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\(\)](#)⁸.

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](#)⁹ is thrown. When the system does not limit use of concurrent locks, this parameter is silently ignored.

setSecurityManager(RealtimeSecurity)

Signature

⁷Section [14.2.2.3.2](#)

⁸Section [14.2.2.3.2](#)

⁹Section [15.2.3.4](#)

```
public static void  
setSecurityManager(RealtimeSecurity manager)
```

Description

Sets a new realtime security manager.

Parameters

manager A [RealtimeSecurity](#)¹⁰ object which will become the new security manager.

Throws

SecurityException 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

Determine if a particular module is supported.

Parameters

module of interest.

Returns

true when module is supported; otherwise false.

¹⁰Section [14.2.2.2](#)

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

canEnforceCost

Signature

```
public static boolean  
canEnforceCost()
```

Description

Determine whether or not hard cost enforcement is supported.

Returns

true when cost enforcement is supported, otherwise false.

Available since RTSJ 2.0

canEnforceAllocationRate

Signature

```
public static boolean  
canEnforceAllocationRate()
```

Description

Determine 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

```
public static void  
setDefaultConfiguration(ConfigurationParameters parameters)
```

Description

Set 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

getDefaultConfiguration

Signature

```
public static javax.realtime.ConfigurationParameters  
getDefaultConfiguration()
```

Description

Determine the current configurations used by tasks that are not explicitly provided with one.

Returns

the current configurations.

Available since RTSJ 2.0

14.3 Rationale

This specification accommodates the variation in underlying systems in a number of ways. The `RealtimeSystem` class functions in similar capacity to `java.lang.System`. Similarly, the `RealtimeSecurity` class functions similarly to `java.lang.SecurityManager`.

The concept of optionally required classes provides additional flexibility. Such classes provide a commonality that can be relied upon by program logic that intends to execute on implementations that supports a given function, such as `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 15

Exceptions

As with other Java specifications, the RTSJ uses exceptions and errors to signal conditions that are abnormal, incorrect, or disallowed. In cases where these exceptional and error conditions are substantially the same as those defined in conventional Java, those 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 other cases, new exceptions are defined in the `javax.realtime` package. These 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 use too much memory to be useful.

15.1 Semantics

Except for how information associated with a `Throwable` is stored and managed, the semantics of the subclasses of `Error`, `Exception`, and `RuntimeException` are the same as for all other Java throwables. 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, as opposed to those that are standard exceptions used without change by the specification, are statically allocated (and implement the `StaticThrowable` interface). 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 `StaticThrowables` 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` class defines an exception, it provides additional functionality for supporting ATC. This functionality is more closely related to asynchronous operation than to exception handling. For this reason, it is not included in this chapter, but rather in Chapter 8 on asynchrony.

15.2 javax.realtime

15.2.1 Interfaces

15.2.1.1 StaticThrowable

Description

A marker interface to indicate that a Throwable is intended to be created once and reused. Throwables that implement this interface kept their state in a RealtimeThread local 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 [AsyncBaseEventHandler](#)¹ always have some instance of RealtimeThread 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 initialize the stack backtrace. The message and cause should be cleared.

An application which throws a static exception should use the following paradigm:

```
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 static throwables should extend one of [StaticError](#)², [StaticCheckedException](#)³, or [StaticRuntimeException](#)⁴

[See Section ConfigurationParameters](#)

Available since RTSJ 2.0

15.2.1.1.1 Methods

¹Section [8.3.3.3](#)

²Section [15.2.3.5](#)

³Section [15.2.2.21](#)

⁴Section [15.2.2.22](#)

initMessage(String)

Signature

```
public java.lang.Throwable  
initMessage(String message)
```

Description

Set the message in SO local storage. This is the only method not defined in `java.lang.Throwable`.

Parameters

message is the text to save.

getMessage

Signature

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

Description

get the message describing the problem from SO local memory.

Returns

the message given to the constructor or null when no message was set.

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

Initializes the cause to the given Throwable in SO local memory.

Parameters

causingThrowable the reason why this Throwable gets thrown.

Throws

IllegalArgumentException when the cause is this Throwable itself.

Returns

the reference to this Throwable.

getCause

Signature

```
public java.lang.Throwable  
getCause()
```

Description

getCause returns the cause of this exception or null when no cause was set. The cause is another exception that was caught before this exception was created.

Returns

The cause or null.

fillInStackTrace

Signature

```
public java.lang.Throwable  
fillInStackTrace()
```

Description

Calls into the virtual machine to capture the current stack trace in SO local memory.

Returns

a reference to this Throwable.

setStackTrace(StackTraceElement)

Signature

```
public void  
setStackTrace(java.lang.StackTraceElement[] new_stackTrace)  
throws NullPointerException
```

Description

This method allows overriding the stack trace that was filled during construction of this object. It is intended to be used in a serialization context when the stack trace of a remote exception should be treated like a local.

Parameters

`new_stackTrace` the stack trace to replace be used.

Throws

`NullPointerException` when `new_stackTrace` or any element of `new_stackTrace` is `null`.

getStackTrace

Signature

```
public java.lang.StackTraceElement[]  
getStackTrace()
```

Description

Get the stack trace created by `fillInStackTrace` for this `Throwable` as an array of `StackTraceElements`.

The stack trace does not need to contain entries for all methods that are actually on the call stack, the virtual machine may decide to skip some stack trace entries. Even an empty array is a valid result of this function.

Repeated calls of this function without intervening calls to `fillInStackTrace` will return the same result.

When memory areas of the RTSJ are used (see [MemoryArea⁵](#)), 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.

⁵Section [11.3.2.3](#)

Returns

array representing the stack trace, never null.

printStackTrace*Signature*

```
public void  
printStackTrace()
```

Description

Print stack trace of this Throwable to System.err.

The printed stack trace contains the result of toString() as the first line followed by one line for each stack trace element that contains the name of the method or constructor, optionally followed by the source file name and source file line number when available.

printStackTrace(PrintStream)*Signature*

```
public void  
printStackTrace(PrintStream stream)
```

Description

Print the stack trace of this Throwable to the given stream.

The printed stack trace contains the result of toString() as the first line followed by one line for each stack trace element that contains the name of the method or constructor, optionally followed by the source file name and source file line number when available.

Parameters

stream the stream to print to.

printStackTrace(PrintWriter)*Signature*

```
public void  
printStackTrace(PrintWriter s)
```

Description

Print the stack trace of this Throwable to the given PrintWriter.

The printed stack trace contains the result of `toString()` as the first line followed by one line for each stack trace element that contains the name of the method or constructor, optionally followed by the source file name and source file line number when available.

Parameters

s the PrintWriter to write to.

15.2.2 Exceptions

15.2.2.1 ArrivalTimeQueueOverflowException

Inheritance

```
java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      java.lang.RuntimeException
        javax.realtime.StaticRuntimeException
          javax.realtime.EventQueueOverflowException
            javax.realtime.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`

Deprecated RTSJ 2.0 replaced by `EventQueueOverflowException`⁶

15.2.2.1.1 Constructors

⁶Section ??

ArrivalTimeQueueOverflowException

Signature

```
public  
ArrivalTimeQueueOverflowException()
```

Description

The default constructor for `ArrivalTimeQueueOverflowException`, but user code should use `get()`⁷ instead.

15.2.2.1.2 Methods

get

Signature

```
public static javax.realtime.ArrivalTimeQueueOverflowException  
get()
```

Description

Obtain the singleton of this static throwable. It is prepared for immediate throwing.

Returns

the single instance of this throwable.

Available since RTSJ 2.0

15.2.2.2 CeilingViolationException

Inheritance

```
java.lang.Object  
  java.lang.Throwable  
    java.lang.Exception  
      java.lang.RuntimeException
```

⁷Section 15.2.2.1.2

java.lang.IllegalArgumentException
java.lang.IllegalThreadStateException
javax.realtime.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](#)⁸ and the thread or SO's base priority exceeds the policy's ceiling.

Available since RTSJ 2.0 implements StaticThrowable

15.2.2.2.1 Methods

get

Signature

```
public static javax.realtime.CeilingViolationException  
get()
```

Description

Obtain 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

⁸Section [7.3.1.2](#)

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.

initMessage(String)

Signature

```
public java.lang.Throwable  
initMessage(String message)
```

Description

Set the message in SO local storage. This is the only method not defined in java.lang.Throwable.

Parameters

message is the text to save.

getMessage

Signature

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

Description

get the message describing the problem from SO local memory.

Returns

the message given to the constructor or null when no message was set.

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

Initializes the cause to the given Throwable in SO local memory.

Parameters

causingThrowable the reason why this Throwable gets Thrown.

Throws

IllegalArgumentException when the cause is this Throwable itself.

Returns

the reference to this Throwable.

getCause

Signature

```
public java.lang.Throwable  
getCause()
```

Description

getCause returns the cause of this exception or null when no cause was set. The cause is another exception that was caught before this exception was created.

Returns

The cause or null.

fillInStackTrace

Signature

```
public java.lang.Throwable  
fillInStackTrace()
```

Description

Calls into the virtual machine to capture the current stack trace in SO local memory.

Returns

a reference to this Throwable.

setStackTrace(StackTraceElement)

Signature

```
public void  
setStackTrace(java.lang.StackTraceElement[] new_stackTrace)  
throws NullPointerException
```

Description

This method allows overriding the stack trace that was filled during construction of this object. It is intended to be used in a serialization context when the stack trace of a remote exception should be treated like a local.

Parameters

new_stackTrace the stack trace to replace be used.

Throws

NullPointerException when new_stackTrace or any element of new_stackTrace is null.

getStackTrace*Signature*

```
public java.lang.StackTraceElement[]  
getStackTrace()
```

Description

Get the stack trace created by fillInStackTrace for this Throwable as an array of StackTraceElements.

The stack trace does not need to contain entries for all methods that are actually on the call stack, the virtual machine may decide to skip some stack trace entries. Even an empty array is a valid result of this function.

Repeated calls of this function without intervening calls to fillInStackTrace will return the same result.

When memory areas of the RTSJ are used (see [MemoryArea⁹](#)), and this Throwable was allocated in a different memory area than the current allocation context, the resulting stack trace will be allocated in either the same memory area this was allocated in or the current memory area, depending on which is the least deeply nested, thereby creating objects that are assignment compatible with both areas.

Returns

array representing the stack trace, never null.

printStackTrace*Signature*

```
public void  
printStackTrace()
```

Description

Print stack trace of this Throwable to System.err.

The printed stack trace contains the result of toString() as the first line followed by one line for each stack trace element that contains the name of the

⁹Section [11.3.2.3](#)

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

Print the stack trace of this Throwable to the given stream.

The printed stack trace contains the result of `toString()` as the first line followed by one line for each stack trace element that contains the name of the method or constructor, optionally followed by the source file name and source file line number when available.

Parameters

stream the stream to print to.

printStackTrace(PrintWriter)

Signature

```
public void  
printStackTrace(PrintWriter writer)
```

Description

Print the stack trace of this Throwable to the given PrintWriter.

The printed stack trace contains the result of `toString()` as the first line followed by one line for each stack trace element that contains the name of the method or constructor, optionally followed by the source file name and source file line number when available.

Parameters

s the PrintWriter to write to.

15.2.2.3 DeregistrationException

Inheritance

java.lang.Object

java.lang.Throwable
 java.lang.Exception
 java.lang.RuntimeException
 javax.realtime.StaticRuntimeException
 javax.realtime.DeregistrationException

Description

An exception to throw when trying to deregister an [ActiveEvent](#)¹⁰ from an [ActiveEventDispatcher](#)¹¹ to which it is not registered.

Available since RTSJ 2.0

15.2.2.3.1 Methods

get

Signature

```
public static javax.realtime.DeregistrationException  
get()
```

Description

Obtain the singleton of this static throwable. It is prepared for immediate throwing.

Returns

the single instance of this throwable.

15.2.2.4 IllegalSchedulableStateException

Inheritance

```
java.lang.Object  
  java.lang.Throwable  
    java.lang.Exception  
      java.lang.RuntimeException  
        java.lang.IllegalArgumentException  
          java.lang.IllegalThreadStateException
```

¹⁰Section [8.3.1.1](#)

¹¹Section [8.3.3.1](#)

[javax.realtime.IllegalSchedulableStateException](#)*Interfaces*[javax.realtime.StaticThrowable](#)*Description*

The exception thrown when a [Schedulable](#)¹² 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

15.2.2.4.1 Methods

get

Signature

```
public static javax.realtime.IllegalSchedulableStateException  
get()
```

Description

Get the preallocated version of this Throwable. Allocation is done in memory that acts like [ImmortalMemory](#)¹³. 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

¹²Section [6.3.1.3](#)

¹³Section [11.3.2.2](#)

Set the message in SO local storage. This is the only method not defined in `java.lang.Throwable`.

Parameters

message is the text to save.

getMessage*Signature*

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

Description

get the message describing the problem from SO local memory.

Returns

the message given to the constructor or null when no message was set.

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

Initializes the cause to the given Throwable is SO local memory.

Parameters

causingThrowable the reason why this Throwable gets Thrown.

Throws

IllegalArgumentException when the cause is this Throwable itself.

Returns

the reference to this Throwable.

getCause*Signature*

```
public java.lang.Throwable  
    getCause()
```

Description

getCause returns the cause of this exception or null when no cause was set. The cause is another exception that was caught before this exception was created.

Returns

The cause or null.

fillInStackTrace*Signature*

```
public java.lang.Throwable  
    fillInStackTrace()
```

Description

Calls into the virtual machine to capture the current stack trace in SO local memory.

Returns

a reference to this Throwable.

setStackTrace(StackTraceElement)*Signature*

```
public void  
    setStackTrace(java.lang.StackTraceElement[] new_stackTrace)
```

throws `NullPointerException`

Description

This method allows overriding the stack trace that was filled during construction of this object. It is intended to be used in a serialization context when the stack trace of a remote exception should be treated like a local.

Parameters

`new_stackTrace` the stack trace to replace be used.

Throws

`NullPointerException` when `new_stackTrace` or any element of `new_stackTrace` is `null`.

getStackTrace

Signature

```
public java.lang.StackTraceElement[]  
getStackTrace()
```

Description

Get the stack trace created by `fillInStackTrace` for this `Throwable` as an array of `StackTraceElements`.

The stack trace does not need to contain entries for all methods that are actually on the call stack, the virtual machine may decide to skip some stack trace entries. Even an empty array is a valid result of this function.

Repeated calls of this function without intervening calls to `fillInStackTrace` will return the same result.

When memory areas of the RTSJ are used (see [MemoryArea¹⁴](#)), and this `Throwable` was allocated in a different memory area than the current allocation context, the resulting stack trace will be allocated in either the same memory area this was allocated in or the current memory area, depending on which is the least deeply nested, thereby creating objects that are assignment compatible with both areas.

Returns

array representing the stack trace, never `null`.

¹⁴Section [11.3.2.3](#)

printStackTrace

Signature

```
public void  
printStackTrace()
```

Description

Print stack trace of this Throwable to System.err.

The printed stack trace contains the result of toString() as the first line followed by one line for each stack trace element that contains the name of the method or constructor, optionally followed by the source file name and source file line number when available.

printStackTrace(PrintStream)

Signature

```
public void  
printStackTrace(PrintStream stream)
```

Description

Print the stack trace of this Throwable to the given stream.

The printed stack trace contains the result of toString() as the first line followed by one line for each stack trace element that contains the name of the method or constructor, optionally followed by the source file name and source file line number when available.

Parameters

stream the stream to print to.

printStackTrace(PrintWriter)

Signature

```
public void  
printStackTrace(PrintWriter writer)
```

Description

Print the stack trace of this Throwable to the given PrintWriter.

The printed stack trace contains the result of toString() as the first line followed by one line for each stack trace element that contains the name of the

method or constructor, optionally followed by the source file name and source file line number when available.

Parameters

s the `PrintWriter` to write to.

15.2.2.5 InaccessibleAreaException

Inheritance

```
java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      java.lang.RuntimeException
        javax.realtime.StaticRuntimeException
          javax.realtime.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`

15.2.2.5.1 Constructors

InaccessibleAreaException

Signature

```
public
InaccessibleAreaException()
```

Description

A constructor for `InaccessibleAreaException`, but application code should use `get()`¹⁵ instead.

¹⁵Section 15.2.2.5.2

InaccessibleAreaException(String)

Signature

```
public  
InaccessibleAreaException(String description)
```

Description

A descriptive constructor for InaccessibleAreaException.

Deprecated since RTSJ 2.0; application code should use `get()`¹⁶ instead.

Parameters

description Description of the error.

15.2.2.5.2 Methods

get

Signature

```
public static javax.realtime.InaccessibleAreaException  
get()
```

Description

Obtain the singleton of this static throwable. It is prepared for immediate throwing.

Returns

the single instance of this throwable.

Available since RTSJ 2.0

15.2.2.6 LateStartException

Inheritance

¹⁶Section 15.2.2.5.2

java.lang.Object
 java.lang.Throwable
 java.lang.Exception
 javafx.runtime.StaticCheckedException
 javafx.runtime.LateStartException

Description

Exception thrown when a periodic realtime thread or timer is started after its assigned, absolute, start time.

Available since RTSJ 2.0

Available since RTSJ 2.0 extends StaticRuntimeException

15.2.2.6.1 Methods

get

Signature

```
public static javafx.runtime.LateStartException  
get()
```

Description

Obtain the singleton of this static throwable. It is prepared for immediate throwing.

Returns

the single instance of this throwable.

15.2.2.7 MITViolationException

Inheritance

java.lang.Object
 java.lang.Throwable
 java.lang.Exception
 java.lang.RuntimeException
 javafx.runtime.StaticRuntimeException
 javafx.runtime.MITViolationException

Description

Thrown by the `AsyncEvent.fire()`¹⁷ 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 `StaticRuntimeException`

15.2.2.7.1 Constructors

MITViolationException

Signature

```
public  
MITViolationException()
```

Description

A constructor for `MITViolationException`.

MITViolationException(String)

Signature

```
public  
MITViolationException(String description)
```

Description

A descriptive constructor for `MITViolationException`.

Parameters

description Description of the error.

¹⁷Section [8.3.3.4.2](#)

15.2.2.7.2 Methods

get

Signature

```
public static javax.realtime.MITViolationException  
get()
```

Description

Obtain the singleton of this static throwable. It is prepared for immediate throwing.

Returns

the single instance of this throwable.

Available since RTSJ 2.0

15.2.2.8 MemoryInUseException

Inheritance

```
java.lang.Object  
  java.lang.Throwable  
    java.lang.Exception  
      java.lang.RuntimeException  
        javax.realtime.StaticRuntimeException  
          javax.realtime.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

15.2.2.8.1 Constructors

MemoryInUseException

Signature

```
public  
MemoryInUseException()
```

Description

A constructor for MemoryInUseException, but application code should use `get()`¹⁸ instead.

MemoryInUseException(String)

Signature

```
public  
MemoryInUseException(String description)
```

Description

A descriptive constructor for MemoryInUseException.

Deprecated since RTSJ 2.0; application code should use `get()`¹⁹ instead.

Parameters

description Description of the error.

15.2.2.8.2 Methods

get

Signature

```
public static javax.realtime.MemoryInUseException  
get()
```

Description

¹⁸Section 15.2.2.8.2

¹⁹Section 15.2.2.8.2

Obtain the singleton of this static throwable. It is prepared for immediate throwing.

Returns

the single instance of this throwable.

Available since RTSJ 2.0

15.2.2.9 MemoryScopeException

Inheritance

```
java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      java.lang.RuntimeException
        javax.realtime.StaticRuntimeException
          javax.realtime.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`

15.2.2.9.1 Constructors

MemoryScopeException

Signature

```
public
MemoryScopeException()
```

Description

A constructor for `MemoryScopeException`, but application code should use `get()`²⁰ instead.

²⁰Section 15.2.2.9.2

15.2.2.9.2 Methods

get

Signature

```
public static javax.realtime.MemoryScopeException  
get()
```

Description

Obtain the singleton of this static throwable. It is prepared for immediate throwing.

Returns

the single instance of this throwable.

Available since RTSJ 2.0

15.2.2.10 MemoryTypeConflictException

Inheritance

```
java.lang.Object  
  java.lang.Throwable  
    java.lang.Exception  
      java.lang.RuntimeException  
        javax.realtime.StaticRuntimeException  
          javax.realtime.MemoryTypeConflictException
```

Description

This exception is thrown when the [PhysicalMemoryManager](#)²¹ 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

²¹Section [A.2.3.19](#)

15.2.2.10.1 Constructors

MemoryTypeConflictException

Signature

```
public  
MemoryTypeConflictException()
```

Description

A constructor for MemoryTypeConflictException, but application code should use `get()`²² instead.

MemoryTypeConflictException(String)

Signature

```
public  
MemoryTypeConflictException(String description)
```

Description

A descriptive constructor for MemoryTypeConflictException.

Deprecated since RTSJ 2.0; application code should use `get()`²³ instead.

Parameters

description A description of the exception.

15.2.2.10.2 Methods

²²Section 15.2.2.10.2

²³Section 15.2.2.10.2

get*Signature*

```
public static javax.realtime.MemoryTypeConflictException  
get()
```

Description

Obtain the singleton of this static throwable. It is prepared for immediate throwing.

Returns

the single instance of this throwable.

Available since RTSJ 2.0

15.2.2.11 OffsetOutOfBoundsException

Inheritance

```
java.lang.Object  
  java.lang.Throwable  
    java.lang.Exception  
      java.lang.RuntimeException  
        javax.realtime.StaticRuntimeException  
          javax.realtime.OffsetOutOfBoundsException
```

Description

when the constructor of an [ImmortalPhysicalMemory](#)²⁴, [LTPhysicalMemory](#)²⁵, [VTPhysicalMemory](#)²⁶, [RawMemoryAccess](#)²⁷, or [RawMemoryFloatAccess](#)²⁸ is given an invalid address.

Available since RTSJ 1.0.1 became unchecked

Available since RTSJ 2.0 extends `StaticRuntimeException`

²⁴Section [A.2.3.10](#)

²⁵Section [A.2.3.12](#)

²⁶Section [A.2.3.36](#)

²⁷Section [A.2.3.24](#)

²⁸Section [A.2.3.25](#)

15.2.2.11.1 Constructors

OffsetOutOfBoundsException

Signature

```
public  
OffsetOutOfBoundsException()
```

Description

A constructor for `OffsetOutOfBoundsException`, application code should use `get()`²⁹ instead.

15.2.2.11.2 Methods

get

Signature

```
public static javax.realtime.OffsetOutOfBoundsException  
get()
```

Description

Obtain the singleton of this static throwable. It is prepared for immediate throwing.

Returns

the single instance of this throwable.

Available since RTSJ 2.0

15.2.2.12 POSIXException

Inheritance

²⁹Section [15.2.2.11.2](#)

```

java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      javax.realtime.StaticCheckedException
      javax.realtime.POSIXException

```

Description

A base class for all POSIX exceptions.

Available since RTSJ 2.0

15.2.2.13 POSIXInvalidSignalException

Inheritance

```

java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      javax.realtime.StaticCheckedException
      javax.realtime.POSIXException
      javax.realtime.POSIXInvalidSignalException

```

Description

An invalid POSIX signal number has been specified.

Available since RTSJ 2.0

15.2.2.13.1 Methods

get

Signature

```

public static javax.realtime.POSIXInvalidSignalException
get()

```

Description

Obtain the singleton of this static throwable. It is prepared for immediate throwing.

Returns

the single instance of this throwable.

15.2.2.14 POSIXInvalidTargetException

Inheritance

```
java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      javax.realtime.StaticCheckedException
        javax.realtime.POSIXException
          javax.realtime.POSIXInvalidTargetException
```

Description

The target of the signal does not exist.

Available since RTSJ 2.0

15.2.2.14.1 Methods

get

Signature

```
public static javax.realtime.POSIXInvalidTargetException
get()
```

Description

Obtain the singleton of this static throwable. It is prepared for immediate throwing.

Returns

the single instance of this throwable.

15.2.2.15 POSIXSignalPermissionException

Inheritance

```
java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      javax.realtime.StaticCheckedException
```

javax.realtime.POSIXException
javax.realtime.POSIXSignalPermissionException

Description

The process does not have permission to send the given signal to the given target.

Available since RTSJ 2.0

15.2.2.15.1 Methods

get

Signature

```
public static javax.realtime.POSIXSignalPermissionException  
get()
```

Description

Obtain the singleton of this static throwable. It is prepared for immediate throwing.

Returns

the single instance of this throwable.

15.2.2.16 ProcessorAffinityException

Inheritance

```
java.lang.Object  
  java.lang.Throwable  
    java.lang.Exception  
      javax.realtime.StaticCheckedException  
        javax.realtime.ProcessorAffinityException
```

Description

Exception used to report processor affinity-related errors.

Available since RTSJ 2.0

15.2.2.16.1 Methods

get*Signature*

```
public static javax.realtime.ProcessorAffinityException  
get()
```

Description

Obtain the singleton of this static throwable. It is prepared for immediate throwing.

Returns

the single instance of this throwable.

15.2.2.17 RangeOutOfBoundsException

Inheritance

```
java.lang.Object  
  java.lang.Throwable  
    java.lang.Exception  
      javax.realtime.StaticCheckedException  
        javax.realtime.RangeOutOfBoundsException
```

Description

Available since RTSJ 2.0

15.2.2.17.1 Methods

get*Signature*

```
public static javax.realtime.RangeOutOfBoundsException  
get()
```

Description

Obtain the singleton of this static throwable. It is prepared for immediate throwing.

Returns

the single instance of this throwable.

15.2.2.18 RegistrationException

Inheritance

```
java.lang.Object  
  java.lang.Throwable  
    java.lang.Exception  
      java.lang.RuntimeException  
        javax.realtime.StaticRuntimeException  
          javax.realtime.RegistrationException
```

Description

An exception to throw when trying to register an [ActiveEvent](#)³⁰ with an [ActiveEventDispatcher](#)³¹ to which it is already registered.

Available since RTSJ 2.0

15.2.2.18.1 Methods

get*Signature*

```
public static javax.realtime.RegistrationException  
get()
```

Description

³⁰Section [8.3.1.1](#)

³¹Section [8.3.3.1](#)

Obtain the singleton of this static throwable. It is prepared for immediate throwing.

Returns

the single instance of this throwable.

15.2.2.19 ScopedCycleException

Inheritance

```
java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      java.lang.RuntimeException
        javax.realtime.StaticRuntimeException
          javax.realtime.ScopedCycleException
```

Description

Thrown when a schedulable attempts to enter an instance of [ScopedMemory](#)³² where that operation would cause a violation of the single parent rule.

Available since RTSJ 2.0 extends StaticRuntimeException

15.2.2.19.1 Constructors

ScopedCycleException*Signature*

```
public
  ScopedCycleException()
```

Description

A constructor for ScopedCycleException, but application code should use [get\(\)](#)³³ instead.

³²Section [A.2.3.31](#)

³³Section [15.2.2.19.2](#)

ScopedCycleException(String)

Signature

```
public  
    ScopedCycleException(String description)
```

Description

A descriptive constructor for ScopedCycleException.

Deprecated since RTSJ 2.0; application code should use `get()`³⁴ instead.

Parameters

description Description of the error.

15.2.2.19.2 Methods

get

Signature

```
public static javax.realtime.ScopedCycleException  
    get()
```

Description

Obtain the singleton of this static throwable. It is prepared for immediate throwing.

Returns

the single instance of this throwable.

Available since RTSJ 2.0

15.2.2.20 SizeOutOfBoundsException

Inheritance

³⁴Section 15.2.2.19.2

```

java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      java.lang.RuntimeException
        javax.realtime.StaticRuntimeException
          javax.realtime.SizeOutOfBoundsException

```

Description

To throw when the constructor of an [ImmutablePhysicalMemory](#)³⁵, [LTPhysicalMemory](#)³⁶, or [VTPhysicalMemory](#)³⁷ is given an invalid size or when a memory access generated by a raw memory accessor instance (See [javax.realtime.device.RawMemory](#)³⁸.) would cause access to an invalid address.

Available since RTSJ 1.0.1 became unchecked

Available since RTSJ 2.0 extends `StaticRuntimeException`

15.2.2.20.1 Constructors

SizeOutOfBoundsException*Signature*

```

public
SizeOutOfBoundsException()

```

Description

A constructor for `SizeOutOfBoundsException`, but application code should use [get\(\)](#)³⁹ instead.

SizeOutOfBoundsException(String)*Signature*

³⁵Section [A.2.3.10](#)
³⁶Section [A.2.3.12](#)
³⁷Section [A.2.3.36](#)
³⁸Section [12.3.1.16](#)
³⁹Section [15.2.2.20.2](#)

```
public  
SizeOutOfBoundsException(String description)
```

Description

A descriptive constructor for SizeOutOfBoundsException.

Deprecated since RTSJ 2.0; application code should use `get()`⁴⁰ instead.

Parameters

description The description of the exception.

15.2.2.20.2 Methods

get

Signature

```
public static javax.realtime.SizeOutOfBoundsException  
get()
```

Description

Obtain the singleton of this static throwable. It is prepared for immediate throwing.

Returns

the single instance of this throwable.

Available since RTSJ 2.0

15.2.2.21 StaticCheckedException

Inheritance

```
java.lang.Object  
  java.lang.Throwable  
    java.lang.Exception  
      javax.realtime.StaticCheckedException
```

Interfaces

⁴⁰Section 15.2.2.20.2

`javax.realtime.StaticThrowable`

Description

A base class for all checked exceptions defined in the specification and do not extend a conventional Java exception.

Available since RTSJ 2.0

15.2.2.21.1 Constructors

StaticCheckedException

Signature

protected
StaticCheckedException()

Description

15.2.2.21.2 Methods

initMessage(String)

Signature

public java.lang.Throwable
initMessage(String message)

Description

Set the message in SO local storage. This is the only method not defined in java.lang.Throwable.

Parameters

message is the text to save.

getMessage

Signature

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

Description

get the message describing the problem from SO local memory.

Returns

the message given to the constructor or null when no message was set.

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

Initializes the cause to the given Throwable is SO local memory.

Parameters

causingThrowable the reason why this Throwable gets Thrown.

Throws

IllegalArgumentException when the cause is this Throwable itself.

Returns

the reference to this Throwable.

getCause

Signature

```
public java.lang.Throwable  
    getCause()
```

Description

getCause returns the cause of this exception or null when no cause was set. The cause is another exception that was caught before this exception was created.

Returns

The cause or null.

fillInStackTrace

Signature

```
public java.lang.Throwable  
    fillInStackTrace()
```

Description

Calls into the virtual machine to capture the current stack trace in SO local memory.

Returns

a reference to this Throwable.

setStackTrace(StackTraceElement)

Signature

```
public void  
    setStackTrace(java.lang.StackTraceElement[] new_stackTrace)  
    throws NullPointerException
```

Description

This method allows overriding the stack trace that was filled during construction of this object. It is intended to be used in a serialization context when the stack trace of a remote exception should be treated like a local.

Parameters

new_stackTrace the stack trace to replace be used.

Throws

NullPointerException when new_stackTrace or any element of new_stackTrace is null.

getStackTrace*Signature*

```
public java.lang.StackTraceElement[]  
getStackTrace()
```

Description

Get the stack trace created by fillInStackTrace for this Throwable as an array of StackTraceElements.

The stack trace does not need to contain entries for all methods that are actually on the call stack, the virtual machine may decide to skip some stack trace entries. Even an empty array is a valid result of this function.

Repeated calls of this function without intervening calls to fillInStackTrace will return the same result.

When memory areas of the RTSJ are used (see [MemoryArea⁴¹](#)), and this Throwable was allocated in a different memory area than the current allocation context, the resulting stack trace will be allocated in either the same memory area this was allocated in or the current memory area, depending on which is the least deeply nested, thereby creating objects that are assignment compatible with both areas.

Returns

array representing the stack trace, never null.

printStackTrace*Signature*

```
public void  
printStackTrace()
```

Description

Print stack trace of this Throwable to System.err.

The printed stack trace contains the result of toString() as the first line followed by one line for each stack trace element that contains the name of the

⁴¹Section [11.3.2.3](#)

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

Print the stack trace of this Throwable to the given stream.

The printed stack trace contains the result of `toString()` as the first line followed by one line for each stack trace element that contains the name of the method or constructor, optionally followed by the source file name and source file line number when available.

Parameters

stream the stream to print to.

printStackTrace(PrintWriter)

Signature

```
public void  
printStackTrace(PrintWriter writer)
```

Description

Print the stack trace of this Throwable to the given PrintWriter.

The printed stack trace contains the result of `toString()` as the first line followed by one line for each stack trace element that contains the name of the method or constructor, optionally followed by the source file name and source file line number when available.

Parameters

s the PrintWriter to write to.

15.2.2.22 StaticRuntimeException

Inheritance

java.lang.Object

java.lang.Throwable
 java.lang.Exception
 java.lang.RuntimeException
 javax.realtime.StaticRuntimeException

Interfaces

javax.realtime.StaticThrowable

Description

A base class for all unchecked exceptions defined in the specification and do not extend a conventional Java exception.

Available since RTSJ 2.0

15.2.2.22.1 Constructors

StaticRuntimeException

Signature

protected
StaticRuntimeException()

Description

15.2.2.22.2 Methods

initMessage(String)

Signature

public java.lang.Throwable
initMessage(String message)

Description

Set the message in SO local storage. This is the only method not defined in java.lang.Throwable.

Parameters

message is the text to save.

getMessage*Signature*

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

Description

get the message describing the problem from SO local memory.

Returns

the message given to the constructor or null when no message was set.

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

Initializes the cause to the given Throwable is SO local memory.

Parameters

causingThrowable the reason why this Throwable gets Thrown.

Throws

IllegalArgumentException when the cause is this Throwable itself.

Returns

the reference to this Throwable.

getCause*Signature*

```
public java.lang.Throwable  
    getCause()
```

Description

getCause returns the cause of this exception or null when no cause was set. The cause is another exception that was caught before this exception was created.

Returns

The cause or null.

fillInStackTrace*Signature*

```
public java.lang.Throwable  
    fillInStackTrace()
```

Description

Calls into the virtual machine to capture the current stack trace in SO local memory.

Returns

a reference to this Throwable.

setStackTrace(StackTraceElement)*Signature*

```
public void  
    setStackTrace(java.lang.StackTraceElement[] new_stackTrace)  
    throws NullPointerException
```

Description

This method allows overriding the stack trace that was filled during construction of this object. It is intended to be used in a serialization context when the stack trace of a remote exception should be treated like a local.

Parameters

`new_stackTrace` the stack trace to replace be used.

Throws

`NullPointerException` when `new_stackTrace` or any element of `new_stackTrace` is null.

getStackTrace

Signature

```
public java.lang.StackTraceElement[]  
getStackTrace()
```

Description

Get the stack trace created by `fillInStackTrace` for this `Throwable` as an array of `StackTraceElements`.

The stack trace does not need to contain entries for all methods that are actually on the call stack, the virtual machine may decide to skip some stack trace entries. Even an empty array is a valid result of this function.

Repeated calls of this function without intervening calls to `fillInStackTrace` will return the same result.

When memory areas of the RTSJ are used (see [MemoryArea⁴²](#)), and this `Throwable` was allocated in a different memory area than the current allocation context, the resulting stack trace will be allocated in either the same memory area this was allocated in or the current memory area, depending on which is the least deeply nested, thereby creating objects that are assignment compatible with both areas.

Returns

array representing the stack trace, never null.

printStackTrace

Signature

⁴²Section [11.3.2.3](#)

```
public void  
printStackTrace()
```

Description

Print stack trace of this Throwable to System.err.

The printed stack trace contains the result of toString() as the first line followed by one line for each stack trace element that contains the name of the method or constructor, optionally followed by the source file name and source file line number when available.

printStackTrace(PrintStream)*Signature*

```
public void  
printStackTrace(PrintStream stream)
```

Description

Print the stack trace of this Throwable to the given stream.

The printed stack trace contains the result of toString() as the first line followed by one line for each stack trace element that contains the name of the method or constructor, optionally followed by the source file name and source file line number when available.

Parameters

stream the stream to print to.

printStackTrace(PrintWriter)*Signature*

```
public void  
printStackTrace(PrintWriter writer)
```

Description

Print the stack trace of this Throwable to the given PrintWriter.

The printed stack trace contains the result of toString() as the first line followed by one line for each stack trace element that contains the name of the method or constructor, optionally followed by the source file name and source file line number when available.

Parameters

s the PrintWriter to write to.

15.2.2.23 UnsupportedPhysicalMemoryException

Inheritance

```
java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      java.lang.RuntimeException
        javax.realtime.StaticRuntimeException
          javax.realtime.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

15.2.2.23.1 Constructors

UnsupportedPhysicalMemoryException

Signature

```
public
  UnsupportedPhysicalMemoryException()
```

Description

A constructor for UnsupportedPhysicalMemoryException, but application code should use [get\(\)](#)⁴³ instead.

15.2.2.23.2 Methods

⁴³Section [15.2.2.23.2](#)

get*Signature*

```
public static javax.realtime.UnsupportedPhysicalMemoryException  
get()
```

Description

Obtain the singleton of this static throwable. It is prepared for immediate throwing.

Returns

the single instance of this throwable.

Available since RTSJ 2.0

15.2.2.24 UnsupportedRawMemoryRegionException

Inheritance

```
java.lang.Object  
  java.lang.Throwable  
    java.lang.Exception  
      java.lang.RuntimeException  
        javax.realtime.StaticRuntimeException  
          javax.realtime.UnsupportedRawMemoryRegionException
```

Description

Indicates an invalid raw memory region.

Available since RTSJ 2.0

15.2.2.24.1 Methods

get*Signature*

```
public static javax.realtime.UnsupportedRawMemoryRegionException  
get()
```

Description

Obtain the singleton of this static throwable. It is prepared for immediate throwing.

Returns

the single instance of this throwable.

15.2.3 Classes

15.2.3.1 AlignmentError

Inheritance

```
java.lang.Object
  java.lang.Throwable
    java.lang.Error
      javax.realtime.StaticError
        javax.realtime.AlignmentError
```

Description

The exception thrown on an on a request for a raw memory factory to return memory for a base address that is aligned such that the factory cannot guarantee that loads and stores based on that address will meet the factory's specifications. For instance, on many processors, odd addresses are unsuitable for anything but byte access.

Available since RTSJ 2.0 extends StaticError

15.2.3.1.1 Methods

get*Signature*

```
public static javax.realtime.AlignmentError
get()
```

Description

Obtain the singleton of this static throwable. It is prepared for immediate throwing.

Returns

the single instance of this throwable.

Available since RTSJ 2.0

15.2.3.2 IllegalAssignmentError

Inheritance

```
java.lang.Object
  java.lang.Throwable
    java.lang.Error
      javax.realtime.StaticError
        javax.realtime.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 by an instance of [ScopedMemory](#)⁴⁴) to a field of an object in immortal memory.

Available since RTSJ 2.0 extends [StaticError](#)

15.2.3.2.1 Constructors

IllegalAssignmentError

Signature

```
public
IllegalAssignmentError()
```

Description

A constructor for [IllegalAssignmentError](#), but the application should use [get\(\)](#)⁴⁵ instead.

⁴⁴Section [A.2.3.31](#)

⁴⁵Section [15.2.3.2.2](#)

15.2.3.2.2 Methods

get

Signature

```
public static javax.realtime.IllegalAssignmentError  
get()
```

Description

Obtain the singleton of this static throwable. It is prepared for immediate throwing.

Returns

the single instance of this throwable.

Available since RTSJ 2.0

15.2.3.3 MemoryAccessError

Inheritance

```
java.lang.Object  
  java.lang.Throwable  
    java.lang.Error  
      javax.realtime.StaticError  
        javax.realtime.MemoryAccessError
```

Description

This error is thrown on an attempt to refer to an object in an inaccessible [MemoryArea](#)⁴⁶. For example this will be when logic in a [NoHeapRealtimeThread](#)⁴⁷ attempts to refer to an object in the traditional Java heap.

Available since RTSJ 2.0 extends StaticError

15.2.3.3.1 Constructors

⁴⁶Section [11.3.2.3](#)

⁴⁷Section [A.2.3.14](#)

MemoryAccessError

Signature

```
public  
MemoryAccessError()
```

Description

A constructor for MemoryAccessError, but application code should use `get()`⁴⁸ instead.

15.2.3.3.2 Methods

get

Signature

```
public static javax.realtime.MemoryAccessError  
get()
```

Description

Obtain the singleton of this static throwable. It is prepared for immediate throwing.

Returns

the single instance of this throwable.

Available since RTSJ 2.0

15.2.3.4 ResourceLimitError

Inheritance

```
java.lang.Object  
  java.lang.Throwable  
    java.lang.Error  
      javax.realtime.StaticError
```

⁴⁸Section 15.2.3.3.2

`javax.realtime.ResourceLimitError`

Description

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`

15.2.3.4.1 Constructors

ResourceLimitError

Signature

```
public  
ResourceLimitError()
```

Description

A constructor for `ResourceLimitError`, but application code should use `get()`⁴⁹ instead.

ResourceLimitError(String)

Signature

```
public  
ResourceLimitError(String description)
```

Description

A descriptive constructor for `ResourceLimitError`.

Deprecated since RTSJ 2.0; application code should use `get()`⁵⁰ instead.

Parameters

description The description of the exception.

⁴⁹Section [15.2.3.4.2](#)

⁵⁰Section [15.2.3.4.2](#)

15.2.3.4.2 Methods

get

Signature

```
public static javax.realtime.ResourceLimitError  
get()
```

Description

Obtain the singleton of this static throwable. It is prepared for immediate throwing.

Returns

the single instance of this throwable.

Available since RTSJ 2.0

15.2.3.5 StaticError

Inheritance

```
java.lang.Object  
  java.lang.Throwable  
    java.lang.Error  
      javax.realtime.StaticError
```

Interfaces

```
javax.realtime.StaticThrowable
```

Description

A base class for all errors defined in the specification and do not extend a conventional Java error.

Available since RTSJ 2.0

15.2.3.5.1 Constructors

StaticError

Signature

```
protected  
StaticError()
```

Description

15.2.3.5.2 Methods

initMessage(String)

Signature

```
public java.lang.Throwable  
initMessage(String message)
```

Description

Set the message in SO local storage. This is the only method not defined in `java.lang.Throwable`.

Parameters

message is the text to save.

getMessage

Signature

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

Description

get the message describing the problem from SO local memory.

Returns

the message given to the constructor or null when no message was set.

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

Initializes the cause to the given Throwable is SO local memory.

Parameters

causingThrowable the reason why this Throwable gets Thrown.

Throws

IllegalArgumentException when the cause is this Throwable itself.

Returns

the reference to this Throwable.

getCause

Signature

```
public java.lang.Throwable  
getCause()
```

Description

getCause returns the cause of this exception or null when no cause was set. The cause is another exception that was caught before this exception was created.

Returns

The cause or null.

fillInStackTrace*Signature*

```
public java.lang.Throwable  
fillInStackTrace()
```

Description

Calls into the virtual machine to capture the current stack trace in SO local memory.

Returns

a reference to this Throwable.

setStackTrace(StackTraceElement)*Signature*

```
public void  
setStackTrace(java.lang.StackTraceElement[] new_stackTrace)  
throws NullPointerException
```

Description

This method allows overriding the stack trace that was filled during construction of this object. It is intended to be used in a serialization context when the stack trace of a remote exception should be treated like a local.

Parameters

`new_stackTrace` the stack trace to replace be used.

Throws

`NullPointerException` when `new_stackTrace` or any element of `new_stackTrace` is null.

getStackTrace*Signature*

```
public java.lang.StackTraceElement[]  
getStackTrace()
```


Description

Get the stack trace created by `fillInStackTrace` for this `Throwable` as an array of `StackTraceElements`.

The stack trace does not need to contain entries for all methods that are actually on the call stack, the virtual machine may decide to skip some stack trace entries. Even an empty array is a valid result of this function.

Repeated calls of this function without intervening calls to `fillInStackTrace` will return the same result.

When memory areas of the RTSJ are used (see [MemoryArea⁵¹](#)), and this `Throwable` was allocated in a different memory area than the current allocation context, the resulting stack trace will be allocated in either the same memory area this was allocated in or the current memory area, depending on which is the least deeply nested, thereby creating objects that are assignment compatible with both areas.

Returns

array representing the stack trace, never null.

printStackTrace*Signature*

```
public void  
printStackTrace()
```

Description

Print stack trace of this `Throwable` to `System.err`.

The printed stack trace contains the result of `toString()` as the first line followed by one line for each stack trace element that contains the name of the method or constructor, optionally followed by the source file name and source file line number when available.

printStackTrace(PrintStream)*Signature*

```
public void  
printStackTrace(PrintStream stream)
```

Description

⁵¹Section [11.3.2.3](#)

Print the stack trace of this Throwable to the given stream.

The printed stack trace contains the result of `toString()` as the first line followed by one line for each stack trace element that contains the name of the method or constructor, optionally followed by the source file name and source file line number when available.

Parameters

stream the stream to print to.

printStackTrace(PrintWriter)

Signature

```
public void  
printStackTrace(PrintWriter writer)
```

Description

Print the stack trace of this Throwable to the given PrintWriter.

The printed stack trace contains the result of `toString()` as the first line followed by one line for each stack trace element that contains the name of the method or constructor, optionally followed by the source file name and source file line number when available.

Parameters

s the PrintWriter to write to.

15.2.3.6 StaticOutOfMemoryError

Inheritance

```
java.lang.Object  
  java.lang.Throwable  
    java.lang.Error  
      java.lang.VirtualMachineError  
        java.lang.OutOfMemoryError  
          javax.realtime.StaticOutOfMemoryError
```

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**⁵². It is up to the

⁵²Section 11.3.2.1

implementation as to whether HeapMemory throws this exception or its parent.

Available since RTSJ 2.0

15.2.3.6.1 Methods

get

Signature

```
public static javax.realtime.StaticOutOfMemoryError  
get()
```

Description

Get the preallocated version of this Throwable. Allocation is done in memory that acts like [ImmortalMemory](#)⁵³. 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

Set the message in SO local storage. This is the only method not defined in java.lang.Throwable.

Parameters

message is the text to save.

⁵³Section [11.3.2.2](#)

getMessage

Signature

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

Description

get the message describing the problem from SO local memory.

Returns

the message given to the constructor or null when no message was set.

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

Initializes the cause to the given Throwable is SO local memory.

Parameters

causingThrowable the reason why this Throwable gets Thrown.

Throws

IllegalArgumentException when the cause is this Throwable itself.

Returns

the reference to this Throwable.

getCause

Signature

```
public java.lang.Throwable  
getCause()
```

Description

getCause returns the cause of this exception or null when no cause was set. The cause is another exception that was caught before this exception was created.

Returns

The cause or null.

fillInStackTrace

Signature

```
public java.lang.Throwable  
fillInStackTrace()
```

Description

Calls into the virtual machine to capture the current stack trace in SO local memory.

Returns

a reference to this Throwable.

setStackTrace(StackTraceElement)

Signature

```
public void  
setStackTrace(java.lang.StackTraceElement[] new_stackTrace)  
throws NullPointerException
```

Description

This method allows overriding the stack trace that was filled during construction of this object. It is intended to be used in a serialization context when the stack trace of a remote exception should be treated like a local.

Parameters

new_stackTrace the stack trace to replace be used.

Throws

NullPointerException when new_stackTrace or any element of new_stackTrace is null.

getStackTrace*Signature*

```
public java.lang.StackTraceElement[]  
getStackTrace()
```

Description

Get the stack trace created by fillInStackTrace for this Throwable as an array of StackTraceElements.

The stack trace does not need to contain entries for all methods that are actually on the call stack, the virtual machine may decide to skip some stack trace entries. Even an empty array is a valid result of this function.

Repeated calls of this function without intervening calls to fillInStackTrace will return the same result.

When memory areas of the RTSJ are used (see [MemoryArea](#)⁵⁴), and this Throwable was allocated in a different memory area than the current allocation context, the resulting stack trace will be allocated in either the same memory area this was allocated in or the current memory area, depending on which is the least deeply nested, thereby creating objects that are assignment compatible with both areas.

Returns

array representing the stack trace, never null.

printStackTrace*Signature*

```
public void  
printStackTrace()
```

Description

Print stack trace of this Throwable to System.err.

The printed stack trace contains the result of toString() as the first line followed by one line for each stack trace element that contains the name of the

⁵⁴Section [11.3.2.3](#)

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

Print the stack trace of this Throwable to the given stream.

The printed stack trace contains the result of `toString()` as the first line followed by one line for each stack trace element that contains the name of the method or constructor, optionally followed by the source file name and source file line number when available.

Parameters

stream the stream to print to.

printStackTrace(PrintWriter)

Signature

```
public void  
printStackTrace(PrintWriter s)
```

Description

Print the stack trace of this Throwable to the given PrintWriter.

The printed stack trace contains the result of `toString()` as the first line followed by one line for each stack trace element that contains the name of the method or constructor, optionally followed by the source file name and source file line number when available.

Parameters

s the PrintWriter to write to.

15.2.3.7 StaticThrowableStorage

Inheritance

java.lang.Object

java.lang.Throwable
 javafx.runtime.StaticThrowableStorage

Interfaces

 javafx.runtime.StaticThrowable

Description

Provide the methods for managing the thread local memory used for storing the data needed by preallocated throwables, i.e., exceptions and errors which implement [StaticThrowable](#)⁵⁵. This call is visible so that an application can extend an existing conventional Java throwable and still implement `StaticThrowable`; its methods can be implemented using the methods defined in this class. An application defined throwable that does not need to extend an existing conventional Java throwable should extend on of [StaticCheckedException](#)⁵⁶, [StaticRuntimeException](#)⁵⁷, or [StaticError](#)⁵⁸ instead.

Available since RTSJ 2.0

15.2.3.7.1 Methods

getCurrent

Signature

```
public static javafx.runtime.StaticThrowableStorage  
    getCurrent()
```

Description

A means of obtaining the storage object for the current task.

Returns

the storage object for the current task.

fillInStackTrace

Signature

⁵⁵Section [15.2.1.1](#)

⁵⁶Section [15.2.2.21](#)

⁵⁷Section [15.2.2.22](#)

⁵⁸Section [15.2.3.5](#)


```
public java.lang.Throwable  
fillInStackTrace()
```

Description

Capture the current thread's stack trace and save it in thread local storage. Only the part of the stack trace that fits in the preallocated buffer is stored. This method should be called by a preallocated exception to implement its method of the same name.

Returns

this

getMessage

Signature

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

Description

Get the message from thread local storage that was saved by the last preallocated exception thrown. This method should be called by a preallocated exception to implement its method of the same name.

Returns

the message

initMessage(String)

Signature

```
public java.lang.Throwable  
initMessage(String message)
```

Description

Save the message in thread local storage for later retrieval. Only the part of the message that fits in the preallocated buffer is stored. This method should be called by a preallocated exception to implement its method of the same name.

Parameters

message the message to save.

getCause

Signature

```
public java.lang.Throwable  
getCause()
```

Description

Get the cause from thread local storage that was saved by the last preallocated exception thrown. The actual exception that of the cause is not saved, but just a reference to its type. This returns a newly allocated exception without any valid content, i.e., no valid stack trace. This method should be called by a preallocated exception to implement its method of the same name.

Returns

the message

initCause(Throwable)

Signature

```
public java.lang.Throwable  
initCause(Throwable causingThrowable)
```

Description

Save the message in thread local storage for later retrieval. Only a reference to the exception class is stored. The rest of its information is lost. This method should be called by a preallocated exception to implement its method of the same name.

Parameters

causingThrowable

Returns

this

getStackTrace

Signature

```
public java.lang.StackTraceElement[]  
getStackTrace()
```

Description

Get 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 allows overriding the stack trace that was filled during construction of this object. It is intended to be used in a serialization context when the stack trace of a remote exception should be treated like a local.

Parameters

`new_stackTrace` the stack trace to replace be used.

Throws

`NullPointerException` when `new_stackTrace` or any element of `new_stackTrace` is `null`.

printStackTrace

Signature

```
public void  
printStackTrace()
```

Description

Print stack trace of this Throwable to System.err.

The printed stack trace contains the result of `toString()` as the first line followed by one line for each stack trace element that contains the name of the method or constructor, optionally followed by the source file name and source file line number when available.

printStackTrace(PrintStream)

Signature

```
public void  
printStackTrace(PrintStream stream)
```

Description

Print the stack trace of this Throwable to the given stream.

The printed stack trace contains the result of `toString()` as the first line followed by one line for each stack trace element that contains the name of the method or constructor, optionally followed by the source file name and source file line number when available.

Parameters

stream the stream to print to.

printStackTrace(PrintWriter)

Signature

```
public void  
printStackTrace(PrintWriter writer)
```

Description

Print the stack trace of this Throwable to the given PrintWriter.

The printed stack trace contains the result of `toString()` as the first line followed by one line for each stack trace element that contains the name of the

method or constructor, optionally followed by the source file name and source file line number when available.

Parameters

s the PrintWriter to write to.

15.2.3.8 ThrowBoundaryError

Inheritance

```
java.lang.Object
  java.lang.Throwable
    java.lang.Error
      javax.realtime.StaticError
        javax.realtime.ThrowBoundaryError
```

Description

The error thrown by `MemoryArea.enter(Runnable logic)`⁵⁹ 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

15.2.3.8.1 Constructors

ThrowBoundaryError

Signature

```
public
  ThrowBoundaryError()
```

Description

A constructor for ThrowBoundaryError, but application code should use `get()`⁶⁰ instead.

⁵⁹Section 11.3.2.3.2

⁶⁰Section 15.2.3.8.2

15.2.3.8.2 Methods

get

Signature

```
public static javax.realtime.ThrowBoundaryError  
get()
```

Description

Get the preallocated instance of this exception.

Returns

the preallocated instance of this exception.

initMessage(String)

Signature

```
public java.lang.Throwable  
initMessage(String message)
```

Description

Set the message in SO local storage. This is the only method not defined in `java.lang.Throwable`.

Parameters

`message` is the text to save.

15.3 Rationale

The need for additional exceptions given the new semantics added by the other sections of this specification is obvious. That the specification attaches new, nonconventional, exception semantics to `AsynchronouslyInterruptedException` is, perhaps, not so obvious. However, after careful thought, and given our self-imposed directive that only well-defined code blocks would be subject to having their control asynchronously transferred, the chosen mechanism is logical.

Open Issues

Editorial Issues

- The `\classref` tag should suppress multiple footnotes on the same {page,section,whatever}.
- We should consider numbered paragraphs.
- Make sure all other constructors are defined in terms of the complete constructor.
(Mostly done –jjh)
- Make the code environment use a monospaced font.
- Method reference do not generate footnotes
- We have a general issue of whether we signify when Illegal Assignment Exceptions can occur

List of Semantic Issues

Issue 6.2.1	117
Issue 6.2.2	119
Issue 6.3.1	165
Issue 6.3.2	213
Issue 6.4.1 (all)	222
Issue 11.4.1	503
Issue 12.1.1 (elb)	561
Issue 12.3.1	636
Issue 12.3.2	655
Issue 14.1.1 (jjh)	716
Issue 14.1.2 (jjh)	716
Issue 14.1.3 (elb)	716

List of Review Requests

InterruptedIOException and ATC (elb)	262
--	-----

Appendix A

Deprecated APIs

Since modules are new in 2.0 and this version introduces new ways of handling happening, POSIX signals, and raw memory access, there is no need to include the old API in the RTSJ subsets. Therefore the old classes are deprecated and appear only here. Other deprecated methods, constructors, and fields are also here. Only full implementation of the RTSJ should implement them.

A.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.

A.2 javax.realtime

A.2.1 Interfaces

A.2.1.1 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](#)¹, not directly from application code.

Deprecated as of RTSJ 2.0

A.2.1.1.1 Methods

contains(long, long)

Signature

```
public boolean  
contains(long base,  
         long size)
```

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.

¹Section [A.2.3.19](#)

Returns

true when the specified range contains ANY of this type of memory.

See [Section PhysicalMemoryManager.isRemovable](#)

find(long, long)*Signature*

```
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

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

```
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

true 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](#)² 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

onInsertion(long, long, AsyncEventHandler)

Signature

```
public void  
onInsertion(long base,  
            long size,  
            AsyncEventHandler aeh)
```

Description

Register the specified [AsyncEventHandler](#)³ to run when any memory of this type, and in the range is added to the system. When the size or the base is less than 0, unregister all "onInsertion" references to the handler.

²Section [8.3.3.4](#)

³Section [8.3.3.5](#)

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.

Deprecated as of RTSJ 1.0.1 Replace with `onInsertion(long, long, AsyncEvent)`

onRemoval(long, long, AsyncEvent)

Signature

```
public void  
onRemoval(long base,  
           long size,  
           AsyncEvent ae)
```

Description

Register 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 memory of this type, when ae is null, or when size is less than zero.

`OffsetOutOfBoundsException` when base is less than zero.

`SizeOutOfBoundsException` when base plus size would be greater than the physical addressing range of the processor.

Available since RTSJ 1.0.1

onRemoval(long, long, AsyncEventHandler)

Signature

```
public void  
onRemoval(long base,  
           long size,  
           AsyncEventHandler aeh)
```

Description

Register the specified AEH to run when any memory in the range is removed from the system. When size or base is less than 0, unregister 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 known to this filter, 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.

Deprecated as of RTSJ 1.0.1

unregisterInsertionEvent(long, long, AsyncEvent)

Signature

```
public boolean  
unregisterInsertionEvent(long base,  
                         long size,  
                         AsyncEvent ae)
```

Description

Unregister the specified insertion event. The event is only unregistered when all three arguments match the arguments used to register the event, except that ae

of null matches all values of ae and will unregister every ae that matches the address range.

Note 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 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

```
public boolean  
unregisterRemovalEvent(long base,  
                        long size,  
                        AsyncEvent ae)
```

Description

Unregister the specified removal event. The async event is only unregistered when all three arguments match the arguments used to register the event, except that ae of null matches all values of ae and will unregister every ae that matches the address range. Note 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

vFind(long, long)*Signature*

```
public long  
vFind(long base,  
      long size)
```

Description

Search 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.

A.2.1.2 Schedulable

The following elements of `Schedulable` are deprecated. The required elements are documented in Section 6.3.1.3 above.

A.2.1.2.1 Methods

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

Signature

```
public void  
setScheduler(Scheduler scheduler,  
              SchedulingParameters scheduling,  
              javax.realtime.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](#)⁴ 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](#)⁵.)
- `release` A reference to the [ReleaseParameters](#)⁶ 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](#)⁷.)
- `memoryParameters` A reference to the [MemoryParameters](#)⁸ which will be associated with this. When null, the default value is governed by scheduler (a new object

⁴Section 6.3.3.14

⁵Section 6.3.3.8

⁶Section 6.3.3.10

⁷Section 6.3.3.8

⁸Section 11.3.2.4

is created when the default value is not null). (See [PriorityScheduler](#)⁹.)

group A reference to the [ProcessingGroupParameters](#)¹⁰ which will be associated with this. When null, the default value is governed by scheduler (a new object is created). (See [PriorityScheduler](#)¹¹.)

Throws

`IllegalArgumentException` Thrown 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.

`SecurityException` 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](#)¹² object for this schedulable.

Returns

A reference to the current [ProcessingGroupParameters](#)¹³ object.

Deprecated since RTSJ 2.0

setProcessingGroupParameters(ProcessingGroupParameters)

⁹Section [6.3.3.8](#)

¹⁰Section [A.2.3.22](#)

¹¹Section [6.3.3.8](#)

¹²Section [A.2.3.22](#)

¹³Section [A.2.3.22](#)

Signature

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

Description

Sets the [ProcessingGroupParameters](#)¹⁴ 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](#)¹⁵ 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](#)¹⁶.)

Throws

[IllegalArgumentException](#) Thrown 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](#)¹⁷

setProcessingGroupParametersIfFeasible(ProcessingGroupParameters)

Signature

```
public boolean  
setProcessingGroupParametersIfFeasible(ProcessingGroupParameters group)
```

Description

¹⁴Section [A.2.3.22](#)

¹⁵Section [A.2.3.22](#)

¹⁶Section [6.3.3.8](#)

¹⁷Section [6.3.3.9](#)

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 proposed processing group parameters. 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](#)¹⁸.)

Throws

IllegalArgumentException Thrown when the parameter value is not compatible with the schedulable's scheduler. Also when this schedulable may not use the heap and the proposed parameter object is located in heap memory.

IllegalAssignmentError when this cannot hold a reference to the proposed 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

¹⁸Section [6.3.3.8](#)

This method first performs a feasibility analysis with this added to the system. When the resulting system is feasible, inform the scheduler and cooperating facilities that this instance of [Schedulable](#)¹⁹ should be considered in feasibility analysis until further notified. When the analysis showed 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, do 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

```
public boolean  
addToFeasibility()
```

Description

Inform the scheduler and cooperating facilities that this instance of [Schedulable](#)²⁰ should be considered in feasibility analysis until further notified.

When the object is already included in the feasibility set, do 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

¹⁹Section [6.3.1.3](#)

²⁰Section [6.3.1.3](#)

```
public boolean
removeFromFeasibility()
```

Description

Inform the scheduler and cooperating facilities that this instance of [Schedulable](#)²¹ 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(javax.realtime.ReleaseParameters<?> release,
              MemoryParameters memory)
throws IllegalArgumentException,
      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

²¹Section [6.3.1.3](#)

release The proposed release parameters. When null, the default value is governed by the associated scheduler (a new object is created when the default value is not null). (See [PriorityScheduler](#)²².)

memory The proposed memory parameters. 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](#)²³.)

Throws

IllegalArgumentException Thrown 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 proposed parameter objects are located in heap memory.

IllegalAssignmentError when this cannot hold references to the proposed 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

```
public boolean  
setIfFeasible(javax.realtime.ReleaseParameters<?> release,  
              MemoryParameters memory,  
              ProcessingGroupParameters group)  
throws IllegalArgumentException,  
      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

²²Section [6.3.3.8](#)

²³Section [6.3.3.8](#)

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 proposed release parameters. 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](#)²⁴.)

memory The proposed memory parameters. 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](#)²⁵.)

group The proposed processing group parameters. 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](#)²⁶.)

Throws

IllegalArgumentException Thrown 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 proposed parameter objects are located in heap memory.

IllegalAssignmentError when this cannot hold references to the proposed 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

²⁴Section [6.3.3.8](#)

²⁵Section [6.3.3.8](#)

²⁶Section [6.3.3.8](#)

setIfFeasible(ReleaseParameters, ProcessingGroupParameters)

Signature

```
public boolean  
setIfFeasible(javax.realtime.ReleaseParameters<?> release,  
              ProcessingGroupParameters group)  
throws IllegalArgumentException,  
      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 proposed release parameters. When null, the default value is governed by the associated scheduler (a new object is created when the default value is not null). (See [PriorityScheduler²⁷](#).)
- group The proposed processing group parameters. When null, the default value is governed by the associated scheduler (a new object is created when the default value is not null). (See [PriorityScheduler²⁸](#).)

Throws

- [IllegalArgumentException](#) Thrown 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 proposed parameter objects are located in heap memory.
- [IllegalAssignmentError](#) when this cannot hold references to the proposed parameter objects, or the parameter objects cannot hold a reference to this.

²⁷Section [6.3.3.8](#)

²⁸Section [6.3.3.8](#)

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,
              javax.realtime.ReleaseParameters<?> release,
              MemoryParameters memory)
throws IllegalArgumentException,
      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 proposed scheduling parameters. 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](#)²⁹.)

²⁹Section 6.3.3.8

release The proposed release parameters. 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](#)³⁰.)

memory The proposed memory parameters. 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](#)³¹.)

Throws

IllegalArgumentException Thrown 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 proposed parameter objects are located in heap memory.

IllegalAssignmentError when this cannot hold references to the proposed 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

setIfFeasible(SchedulingParameters, ReleaseParameters, MemoryParameters, ProcessingGroupParameters)

Signature

```
public boolean
setIfFeasible(SchedulingParameters scheduling,
              javax.realtime.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.

³⁰Section [6.3.3.8](#)

³¹Section [6.3.3.8](#)

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 proposed scheduling parameters. When null, the default value is governed by the associated scheduler (a new object is created when the default value is not null). (See [PriorityScheduler](#)³².)

release The proposed release parameters. 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](#)³³.)

memory The proposed memory parameters. 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](#)³⁴.)

group The proposed processing group parameters. 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](#)³⁵.)

Throws

`IllegalArgumentException` Thrown 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 proposed parameter objects are located in heap memory.

`IllegalAssignmentError` when this cannot hold references to the proposed 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

³²Section [6.3.3.8](#)

³³Section [6.3.3.8](#)

³⁴Section [6.3.3.8](#)

³⁵Section [6.3.3.8](#)

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 proposed memory parameters. 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](#)³⁶.)

Throws

IllegalArgumentException Thrown when the parameter value is not compatible with the schedulable's scheduler. Also when this schedulable may not use the heap and the proposed parameter object is located in heap memory.

IllegalAssignmentError when this cannot hold a reference to the proposed 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

³⁶Section [6.3.3.8](#)

setReleaseParametersIfFeasible(ReleaseParameters)

Signature

```
public boolean  
setReleaseParametersIfFeasible(javax.realtime.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 proposed release parameters. 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](#)³⁷.)

Throws

IllegalArgumentException Thrown when the parameter value is not compatible with the schedulable's scheduler. Also when this schedulable may not use the heap and the proposed parameter object is located in heap memory.

IllegalAssignmentError when this cannot hold a reference to the proposed 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.

³⁷Section [6.3.3.8](#)

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 proposed scheduling parameters. 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](#)³⁸.)

Throws

[IllegalArgumentException](#) Thrown when the parameter value is not compatible with the schedulable's scheduler. Also when this schedulable may not use the heap and the proposed parameter object is located in heap memory.

[IllegalAssignmentError](#) when this cannot hold a reference to the proposed 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

³⁸Section [6.3.3.8](#)

A.2.2 Exceptions

A.2.2.1 *ArrivalTimeQueueOverflowException*

The following elements of `ArrivalTimeQueueOverflowException` are deprecated. The required elements are documented in Section 15.2.2.1 above.

A.2.2.1.1 Constructors

`ArrivalTimeQueueOverflowException(String)`

Signature

```
public  
ArrivalTimeQueueOverflowException(String description)
```

Description

A descriptive constructor for `ArrivalTimeQueueOverflowException`.

Deprecated since RTSJ 2.0; application code should use `get()`³⁹ instead.

Parameters

description A description of the exception.

A.2.2.2 *AsynchronouslyInterruptedException*

The following elements of `AsynchronouslyInterruptedException` are deprecated. The required elements are documented in Section 8.3.2.1 above.

A.2.2.2.1 Methods

³⁹Section 15.2.2.1.2

happened(boolean)

Signature

```
public boolean  
happened(boolean propagate)
```

Description

Used with an instance of this exception to see if the current exception is this exception. When an `AsynchronouslyInterruptedException` is caught, the catch clause may invoke the `happened()` method on the `AsynchronouslyInterruptedException` in which it is interested to see if it matches the pending `AsynchronouslyInterruptedException`. When so, the pending `AsynchronouslyInterruptedException` is cleared for the schedulable and `happened` returns true. Otherwise, the behavior of `happened` depends on its propagation parameter. When propagation parameter is true, the `AsynchronouslyInterruptedException` will continue to propagate outward; i.e., it will be re-thrown by a mechanism that bypassed the normal requirement that the checked exception be identified in the method's signature. When propagation parameter is false, `happened` will return false and the `AsynchronouslyInterruptedException` remains pending.

Parameters

`propagate` Control the behavior when this is not the current exception:

- When true and this exception is the current one, set the state of this to non pending and return true.
- When true and this exception is not the current one, propagate the exception; i.e., rethrow it.
- When false and this exception is the current one, the state of this is set to nonpending (i.e., it will stop propagating) and return true.
- When false and this exception is not the current one, return false.

Throws

`IllegalThreadStateException` when called from a Java thread.

Returns

true, when this is the current exception, and false, when this is not the current exception.

Deprecated as of RTSJ 1.0.1. This method seriously violates standard Java exception semantics, and while it is a convenience it is not required. The `happened` method can be replaced with the `clear` method and application logic.

propagate

Signature

```
public static void  
propagate()
```

Description

Cause the pending exception to continue up the stack. The current AIE remains pending and control is transferred immediately to the next suitable catch or finally clause under the normal rules for AIE propagation.

When there is no current AIE, the method does nothing and simply returns.

This method is normally used in a catch clause that is handling an AIE, but that is not required. The method may be invoked at any time (from a schedulable).

Throws

IllegalThreadStateException when called from a Java thread.

Deprecated as of RTSJ 1.0.1. This method seriously violates standard Java exception semantics, and while it is a convenience it is not required. It should be replaced with throw of an instance of `AsynchronouslyInterruptedException`.

A.2.2.3 DuplicateFilterException

Inheritance

```
java.lang.Object  
  java.lang.Throwable  
    java.lang.Exception  
      javax.realtime.DuplicateFilterException
```

Description

`PhysicalMemoryManager`⁴⁰ 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

⁴⁰Section [A.2.3.19](#)

A.2.2.3.1 Constructors

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.

A.2.2.4 *MemoryScopeException*

The following elements of MemoryScopeException are deprecated. The required elements are documented in Section [15.2.2.9](#) above.

A.2.2.4.1 Constructors

MemoryScopeException(String)

Signature

```
public  
MemoryScopeException(String description)
```

Description

A descriptive constructor for MemoryScopeException.

Deprecated since RTSJ 2.0; application code should use `get()`⁴¹ instead.

Parameters

description A description of the exception.

A.2.2.5 *OffsetOutOfBoundsException*

The following elements of OffsetOutOfBoundsException are deprecated. The required elements are documented in Section 15.2.2.11 above.

A.2.2.5.1 Constructors

OffsetOutOfBoundsException(String)

Signature

```
public  
OffsetOutOfBoundsException(String description)
```

Description

A descriptive constructor for OffsetOutOfBoundsException.

Deprecated since RTSJ 2.0; pplication code should use `get()`⁴² instead.

Parameters

⁴¹Section 15.2.2.9.2

⁴²Section 15.2.2.11.2

description A description of the exception.

A.2.2.6 UnknownHappeningException

Inheritance

```
java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      java.lang.RuntimeException
        javax.realtime.UnknownHappeningException
```

Description

This exception is used to indicate a situation where an instance of [AsyncEvent](#)⁴³ attempts to bind to a happening that does not exist.

Deprecated since RTSJ 2.0

A.2.2.6.1 Constructors

UnknownHappeningException

Signature

```
public
UnknownHappeningException()
```

Description

A constructor for UnknownHappeningException.

UnknownHappeningException(String)

Signature

```
public
UnknownHappeningException(String description)
```

⁴³Section [8.3.3.4](#)

Description

A descriptive constructor for UnknownHappeningException.

Parameters

description Description of the error.

A.2.2.7 *UnsupportedPhysicalMemoryException*

The following elements of UnsupportedPhysicalMemoryException are deprecated. The required elements are documented in Section [15.2.2.23](#) above.

A.2.2.7.1 Constructors

UnsupportedPhysicalMemoryException(String)*Signature*

```
public
  UnsupportedPhysicalMemoryException(String description)
```

Description

A descriptive constructor for UnsupportedPhysicalMemoryException.

Deprecated since RTSJ 2.0; application code should use `get()`⁴⁴ instead.

Parameters

description The description of the exception.

A.2.3 Classes**A.2.3.1 *AbsoluteTime***

The following elements of AbsoluteTime are deprecated. The required elements are documented in Section [9.3.1.1](#) above.

⁴⁴Section [15.2.2.23.2](#)

A.2.3.1.1 Constructors

AbsoluteTime(long, int, Clock)

Signature

```
public  
AbsoluteTime(long millis,  
              int nanos,  
              Clock clock)  
throws IllegalArgumentException
```

Description

Superceded by and equivalent to [AbsoluteTime\(long, int, Chronograph\)](#)⁴⁵

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

IllegalArgumentException when there is an overflow in the millisecond component when normalizing.

AbsoluteTime(AbsoluteTime, Clock)

Signature

```
public  
AbsoluteTime(AbsoluteTime time,  
              Clock clock)
```

⁴⁵Section [9.3.1.1.1](#)

throws `IllegalArgumentException`

Description

Equivalent to `AbsoluteTime(long, int, Chronograph)`⁴⁶ with the arguments `time.getMilliseconds()`, `time.getNanoseconds()`, `clock()`.

Available since RTSJ 1.0.1

Deprecated since version 2.0

Parameters

`time` is the `AbsoluteTime` object which is the source for the copy.

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

Throws

`IllegalArgumentException` when the `time` parameter is null.

AbsoluteTime(Date, Clock)

Signature

```
public
AbsoluteTime(Date date,
              Clock clock)
throws IllegalArgumentException
```

Description

Superceded by and equivalent to `AbsoluteTime(Date, Chronograph)`⁴⁷

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

`IllegalArgumentException` when the `date` parameter is null.

⁴⁶Section 9.3.1.1.1

⁴⁷Section 9.3.1.1.1

AbsoluteTime(Clock)

Signature

```
public  
AbsoluteTime(Clock clock)
```

Description

Superceeded by and equivalent to [AbsoluteTime\(Chronograph\)](#)⁴⁸

Available since RTSJ 1.0.1

Deprecated since version 2.0

Parameters

clock The clock providing the association for the newly constructed object.

A.2.3.1.2 Methods

absolute(Clock)

Signature

```
public javax.realtime.AbsoluteTime  
absolute(Clock clock)
```

Description

Superceeded by and equivalent to [absolute\(Chronograph\)](#)⁴⁹.

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

⁴⁸Section [9.3.1.1.1](#)

⁴⁹Section [9.3.1.1.2](#)

absolute(Clock, AbsoluteTime)

Signature

```
public javax.realtime.AbsoluteTime  
absolute(Clock clock,  
         AbsoluteTime dest)
```

Description

Superceded by and equivalent to [absolute\(Chronograph, AbsoluteTime\)](#)⁵⁰.

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. Otherwise, a new object is allocated for the result.

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

```
public javax.realtime.RelativeTime  
relative(Clock clock)  
throws ArithmeticException
```

Description

Superceded by and equivalent to [relative\(Chronograph\)](#)⁵¹.

Parameters

clock The instance of [Clock](#)⁵² 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

⁵⁰Section [9.3.1.1.2](#)

⁵¹Section [9.3.1.1.2](#)

⁵²Section [10.3.2.1](#)

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

Superceeded by and equivalent to [relative\(Chronograph, RelativeTime\)](#)⁵³.

Parameters

clock The instance of [Clock](#)⁵⁴ used to convert the time of this into relative time, and the new clock association for the result.

dest When dest is not null, the result is placed there and returned. Otherwise, a new object is allocated for the result.

Throws

ArithmeticException when the result does not fit in the normalized format.

Returns

The RelativeTime conversion in dest when dest is not null, otherwise the result is returned in a newly allocated object. It is associated with the clock parameter.

Deprecated since version 2.0

A.2.3.2 AperiodicParameters

The following elements of AperiodicParameters are deprecated. The required elements are documented in [Section 6.3.3.2](#) above.

⁵³Section [9.3.1.1.2](#)

⁵⁴Section [10.3.2.1](#)

A.2.3.2.1 Fields

arrivalTimeQueueOverflowExcept

public static final 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 [ArrivalTimeQueueOverflowException](#)⁵⁵. 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](#)⁵⁶ 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 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

⁵⁵Section [15.2.2.1](#)

⁵⁶Section [8.3.3.5](#)

arrivalTimeQueueOverflowReplace

public static final 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 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” as it alters the actual queue length. Since the SAVE policy grows the arrival time queue as necessary, for the SAVE policy the initial queue length is only an optimization.

Available since RTSJ 1.0.1 Moved here from SporadicParameters.

Deprecated since RTSJ 2.0

A.2.3.2.2 Methods

getInitialArrivalTimeQueueLength*Signature*

```
public int  
getInitialArrivalTimeQueueLength()
```

Description

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\(\)](#)⁵⁷.

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

IllegalArgumentException 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\(int initial\)](#)⁵⁸.

⁵⁷Section [6.3.3.10.2](#)

⁵⁸Section [6.3.3.10.2](#)

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.getEventQueueOverflowPolicy](#)⁵⁹

setArrivalTimeQueueOverflowBehavior(String)

Signature

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

Description

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

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

Parameters

behavior A string representing the behavior.

Throws

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

Available since RTSJ 1.0.1 Moved here from SporadicParameters.

Deprecated Since RTSJ 2.0

⁵⁹Section [6.3.3.10.2](#)

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. 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

`IllegalArgumentException` when the time value of cost is less than zero, or the time value of deadline is less than or equal to zero, or the 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`⁶⁰, need not return false.)

Deprecated as of RTSJ 2.0

A.2.3.3 AsyncEvent

The following elements of `AsyncEvent` are deprecated. The required elements are documented in Section 8.3.3.4 above.

⁶⁰Section 6.3.3.15

A.2.3.3.1 Methods

handledBy(AsyncEventHandler)

Signature

```
public boolean  
handledBy(AsyncEventHandler handler)
```

Description

Replaced by [AsyncBaseEvent.handledBy\(AsyncBaseEventHandler\)](#)⁶¹

Parameters

handler to query 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\(AsyncBaseEventHandler\)](#)⁶²

Parameters

handler to add to this.

Deprecated since RTSJ 2.0

setHandler(AsyncEventHandler)

Signature

⁶¹Section [8.3.3.2.1](#)

⁶²Section [8.3.3.2.1](#)

```
public void  
setHandler(AsyncEventHandler handler)
```

Description

Replaced by [AsyncBaseEvent.setHandler\(AsyncBaseEventHandler\)](#)⁶³

Parameters

handler to be the sole handler for this.

Deprecated since RTSJ 2.0

removeHandler(AsyncEventHandler)

Signature

```
public void  
removeHandler(AsyncEventHandler handler)
```

Description

Replaced by [AsyncBaseEvent.removeHandler\(AsyncBaseEventHandler\)](#)⁶⁴

Parameters

handler to be removed.

Deprecated since RTSJ 2.0

bindTo(String)

Signature

```
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.

⁶³Section [8.3.3.2.1](#)

⁶⁴Section [8.3.3.2.1](#)

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.

IllegalArgumentException when happening is null.

Deprecated since RTSJ 2.0

unbindTo(String)

Signature

```
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.

Throws

UnknownHappeningException when this instance of AsyncEvent is not bound to the given happening or the given happening is not supported by the implementation.

IllegalArgumentException when happening is null.

Deprecated since RTSJ 2.0

A.2.3.4 AsyncEventHandler

The following elements of AsyncEventHandler are deprecated. The required elements are documented in Section 8.3.3.5 above.

A.2.3.4.1 Constructors

AsyncEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, SchedulingGroup, boolean, Runnable)

Signature

```
public
AsyncEventHandler(SchedulingParameters scheduling,
                 javax.realtime.ReleaseParameters<?> release,
                 MemoryParameters memory,
                 MemoryArea area,
                 SchedulingGroup group,
                 boolean nonheap,
                 Runnable logic)
```

Description

Calling this constructor is equivalent to calling [AsyncEventHandler\(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, SchedulingGroup, ConfigurationParameters, Runnable\)](#)⁶⁵ with arguments (scheduling, release, memory, clone(!nonheap), group, null, logic).

Deprecated in version 2.0.

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

Signature

```
public
AsyncEventHandler(SchedulingParameters scheduling,
```

⁶⁵Section 8.3.3.5.1

```
javax.realtime.ReleaseParameters<?> release,  
MemoryParameters memory,  
MemoryArea area,  
ProcessingGroupParameters group,  
Runnable logic)
```

Description

Calling this constructor is equivalent to calling `AsyncEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, SchedulingGroup, ConfigurationParameters, Runnable)`⁶⁶ 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,  
                 javax.realtime.ReleaseParameters<?> release,  
                 MemoryParameters memory,  
                 MemoryArea area,  
                 ProcessingGroupParameters group,  
                 boolean nonheap)
```

Description

Calling this constructor is equivalent to calling `AsyncEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, SchedulingGroup, ConfigurationParameters, Runnable)`⁶⁷ with arguments (scheduling, release, memory, area, group.getProcessingGroup(), null, null).

Deprecated in version 2.0.

⁶⁶Section 8.3.3.5.1

⁶⁷Section 8.3.3.5.1

AsyncEventHandler(boolean, Runnable)

Signature

```
public
AsyncEventHandler(boolean nonheap,
                  Runnable logic)
```

Description

Calling this constructor is equivalent to calling `AsyncEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, SchedulingGroup, ConfigurationParameters, Runnable)`⁶⁸ with arguments (null, null, new MemoryParameters(!nonheap), null, null, null, logic).

Deprecated in version 2.0.

AsyncEventHandler(boolean)

Signature

```
public
AsyncEventHandler(boolean nonheap)
```

Description

Calling this constructor is equivalent to calling `AsyncEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, SchedulingGroup, ConfigurationParameters, Runnable)`⁶⁹ with arguments (null, null, new MemoryParameters(!nonheap), null, null, null, null).

Deprecated in version 2.0.

A.2.3.4.2 Methods

⁶⁸Section 8.3.3.5.1

⁶⁹Section 8.3.3.5.1

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.

The effect of a call to getAndIncrementPendingFireCount on the arrival-time queue and the scheduling of this AEH depends on the semantics of the scheduler controlling this AEH.

Throws

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

`scheduling` scheduling A reference to the [SchedulingParameters](#)⁷⁰ 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](#)⁷¹.)

`release` release A reference to the [ReleaseParameters](#)⁷² 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](#)⁷³.)

`memoryParameters` memoryParameters A reference to the [MemoryParameters](#)⁷⁴ 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](#)⁷⁵.)

`group` group A reference to the [ProcessingGroupParameters](#)⁷⁶ which will be associated with this. When null, the default value is governed by scheduler (a new object is created). (See [PriorityScheduler](#)⁷⁷.)

Throws

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

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

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

⁷⁰Section [6.3.3.14](#)

⁷¹Section [6.3.3.8](#)

⁷²Section [6.3.3.10](#)

⁷³Section [6.3.3.8](#)

⁷⁴Section [11.3.2.4](#)

⁷⁵Section [6.3.3.8](#)

⁷⁶Section [A.2.3.22](#)

⁷⁷Section [6.3.3.8](#)

Deprecated since RTSJ 2.0

getProcessingGroupParameters

Signature

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

Description

Gets a reference to the [ProcessingGroupParameters](#)⁷⁸ object for this schedulable.

Returns

A reference to the current [ProcessingGroupParameters](#)⁷⁹ object.

Deprecated since RTSJ 2.0

setProcessingGroupParameters(ProcessingGroupParameters)

Signature

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

Description

Sets the [ProcessingGroupParameters](#)⁸⁰ 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 group A [ProcessingGroupParameters](#)⁸¹ 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](#)⁸².)

⁷⁸Section [A.2.3.22](#)

⁷⁹Section [A.2.3.22](#)

⁸⁰Section [A.2.3.22](#)

⁸¹Section [A.2.3.22](#)

⁸²Section [6.3.3.8](#)

Throws

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

`IllegalThreadStateException` `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

addToFeasibility

Signature

```
public boolean  
addToFeasibility()
```

Description

Inform the scheduler and cooperating facilities that this instance of `Schedulable`⁸³ should be considered in feasibility analysis until further notified.

When the object is already included in the feasibility set, do 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

```
public boolean  
addIfFeasible()
```

Description

This method first performs a feasibility analysis with this added to the system. When the resulting system is feasible, inform the scheduler and cooperating facilities that this instance of `Schedulable`⁸⁴ should be considered in feasibility

⁸³Section 6.3.1.3

⁸⁴Section 6.3.1.3

analysis until further notified. When the analysis showed 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, do 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

Inform the scheduler and cooperating facilities that this instance of [Schedulable](#)⁸⁵ 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(javax.realtime.ReleaseParameters<?> release,  
              MemoryParameters memory)
```

Description

⁸⁵Section [6.3.1.3](#)

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 *release* The proposed release parameters. 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](#)⁸⁶.)

memory *memory* The proposed memory parameters. 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](#)⁸⁷.)

Throws

IllegalArgumentException *IllegalArgumentException* Thrown 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 proposed parameter objects are located in heap memory.

IllegalAssignmentError *IllegalAssignmentError* when this cannot hold references to the proposed parameter objects, or the parameter objects cannot hold a reference to this.

IllegalThreadStateException *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)

⁸⁶Section [6.3.3.8](#)

⁸⁷Section [6.3.3.8](#)

Signature

```
public boolean  
setIfFeasible(SchedulingParameters scheduling,  
              javax.realtime.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 scheduling The proposed scheduling parameters. 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](#)⁸⁸.)

release release The proposed release parameters. 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](#)⁸⁹.)

memory memory The proposed memory parameters. 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](#)⁹⁰.)

Throws

IllegalArgumentException **IllegalArgumentException** Thrown 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 proposed parameter objects are located in heap memory.

IllegalAssignmentError **IllegalAssignmentError** when this cannot hold references to the proposed parameter objects, or the parameter objects cannot hold a

⁸⁸Section 6.3.3.8

⁸⁹Section 6.3.3.8

⁹⁰Section 6.3.3.8

reference to this.

IllegalThreadStateException *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(javax.realtime.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 *release* The proposed release parameters. 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](#)⁹¹.)

⁹¹Section 6.3.3.8

memory memory The proposed memory parameters. 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](#)⁹².)

group group The proposed processing group parameters. 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](#)⁹³.)

Throws

`IllegalArgumentException` `IllegalArgumentException` Thrown 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 proposed parameter objects are located in heap memory.

`IllegalAssignmentError` `IllegalAssignmentError` when this cannot hold references to the proposed parameter objects, or the parameter objects cannot hold a reference to this.

`IllegalThreadStateException` `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

```
public boolean
setIfFeasible(SchedulingParameters scheduling,
              javax.realtime.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

⁹²Section [6.3.3.8](#)

⁹³Section [6.3.3.8](#)

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 scheduling The proposed scheduling parameters. When null, the default value is governed by the associated scheduler (a new object is created when the default value is not null). (See [PriorityScheduler⁹⁴](#).)

release release The proposed release parameters. When null, the default value is governed by the associated scheduler (a new object is created when the default value is not null). (See [PriorityScheduler⁹⁵](#).)

memory memory The proposed memory parameters. When null, the default value is governed by the associated scheduler (a new object is created when the default value is not null). (See [PriorityScheduler⁹⁶](#).)

group group The proposed processing group parameters. When null, the default value is governed by the associated scheduler (a new object is created when the default value is not null). (See [PriorityScheduler⁹⁷](#).)

Throws

IllegalArgumentException **IllegalArgumentException** Thrown 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 proposed parameter objects are located in heap memory.

IllegalAssignmentError **IllegalAssignmentError** when this cannot hold references to the proposed parameter objects, or the parameter objects cannot hold a reference to this.

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

Returns

⁹⁴Section [6.3.3.8](#)

⁹⁵Section [6.3.3.8](#)

⁹⁶Section [6.3.3.8](#)

⁹⁷Section [6.3.3.8](#)

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

```
public boolean  
setReleaseParametersIfFeasible(javax.realtime.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 release The proposed release parameters. 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](#)⁹⁸.)

Throws

IllegalArgumentException IllegalArgumentException Thrown when the parameter value is not compatible with the schedulable's scheduler. Also when this schedulable may not use the heap and the proposed parameter object is located in heap memory.

IllegalAssignmentError IllegalAssignmentError when this cannot hold a reference to the proposed parameter object, or the parameter object cannot hold a reference to this.

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

⁹⁸Section 6.3.3.8

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

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 group The proposed processing group parameters. When null, the default value is governed by the associated scheduler (a new object is created when the default value is not null). (See [PriorityScheduler⁹⁹](#).)

Throws

IllegalArgumentException **IllegalArgumentException** Thrown when the parameter value is not compatible with the schedulable's scheduler. Also when this schedulable may not use the heap and the proposed parameter object is located in heap memory.

IllegalAssignmentError **IllegalAssignmentError** when this cannot hold a reference to the proposed parameter object, or the parameter object cannot hold a reference to this.

⁹⁹Section 6.3.3.8

IllegalThreadStateException 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

```
public boolean  
setIfFeasible(javax.realtime.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 *release* The proposed release parameters. 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](#)¹⁰⁰.)

group *group* The proposed processing group parameters. 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](#)¹⁰¹.)

¹⁰⁰Section [6.3.3.8](#)

¹⁰¹Section [6.3.3.8](#)

Throws

`IllegalArgumentException` `IllegalArgumentException` Thrown 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 proposed parameter objects are located in heap memory.

`IllegalAssignmentError` `IllegalAssignmentError` when this cannot hold references to the proposed parameter objects, or the parameter objects cannot hold a reference to this.

`IllegalThreadStateException` `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

```
public final 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` `memory` The proposed memory parameters. 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](#)¹⁰².)

Throws

IllegalArgumentException **IllegalArgumentException** Thrown when the parameter value is not compatible with the schedulable's scheduler. Also when this schedulable may not use the heap and the proposed parameter object is located in heap memory.

IllegalAssignmentError **IllegalAssignmentError** when this cannot hold a reference to the proposed parameter object, or the parameter object cannot hold a reference to this.

IllegalThreadStateException **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

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

¹⁰²Section [6.3.3.8](#)

scheduling scheduling The proposed scheduling parameters. 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](#)¹⁰³.)

Throws

IllegalArgumentException **IllegalArgumentException** Thrown when the parameter value is not compatible with the schedulable's scheduler. Also when this schedulable may not use the heap and the proposed parameter object is located in heap memory.

IllegalAssignmentError **IllegalAssignmentError** when this cannot hold a reference to the proposed parameter object, or the parameter object cannot hold a reference to this.

IllegalThreadStateException **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

A.2.3.5 *BoundAsyncEventHandler*

The following elements of *BoundAsyncEventHandler* are deprecated. The required elements are documented in Section [8.3.3.10](#) above.

A.2.3.5.1 Constructors

BoundAsyncEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, ProcessingGroupParameters, boolean, Runnable)

Signature

¹⁰³Section [6.3.3.8](#)


```

public
BoundAsyncEventHandler(SchedulingParameters scheduling,
                      javax.realtime.ReleaseParameters<?> release,
                      MemoryParameters memory,
                      MemoryArea area,
                      ProcessingGroupParameters group,
                      boolean nonheap,
                      Runnable logic)

```

Description

Create 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](#)¹⁰⁴ object which will be associated with the constructed instance. When null, and the creator is a Java thread, a SchedulingParameters object is created which has the default SchedulingParameters for the scheduler associated with the current thread. When null, and the creator is a schedulable object, the SchedulingParameters are inherited from the current schedulable (a new SchedulingParameters object is cloned).

release A [ReleaseParameters](#)¹⁰⁵ object which will be associated with the constructed instance. When null, this will have default ReleaseParameters for the BAEH's scheduler.

memory A [MemoryParameters](#)¹⁰⁶ object which will be associated with the constructed instance. When null, this will have no MemoryParameters.

area The [MemoryArea](#)¹⁰⁷ for this. When null, the memory area will be that of the current thread/schedulable.

group A [ProcessingGroupParameters](#)¹⁰⁸ 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\(\)](#)¹⁰⁹. When null, the default handleAsyncEvent() method invokes nothing.

¹⁰⁴Section 6.3.3.14

¹⁰⁵Section 6.3.3.10

¹⁰⁶Section 11.3.2.4

¹⁰⁷Section 11.3.2.3

¹⁰⁸Section A.2.3.22

¹⁰⁹Section 8.3.3.5.2

nonheap when true, the code executed by this handler may not reference or store objects in [HeapMemory](#)¹¹⁰; otherwise, that code may do so. When true and the current handler tries to reference or store objects in HeapMemory or enter the HeapMemory a `IllegalArgumentException` is thrown.

Throws

`IllegalArgumentException` when nonheap is true and logic, any parameter object, or this is in heap memory. Also when noheap 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.

A.2.3.6 Clock

The following elements of Clock are deprecated. The required elements are documented in [Section 10.3.2.1](#) above.

A.2.3.6.1 Methods

setResolution(RelativeTime)

Signature

```
public abstract void
setResolution(RelativeTime resolution)
```

Description

Set the resolution of this. For some hardware clocks setting resolution is impossible and when this method is called on those clocks, then an `UnsupportedOperationException` is thrown.

Parameters

resolution The new resolution of this, when the requested value is supported by this clock. When resolution is smaller than the minimum resolution supported by this clock then it throws `IllegalArgumentException`. When the requested

¹¹⁰Section [11.3.2.1](#)

resolution is not available and it is larger than the minimum resolution, then the clock will be set to the closest resolution that the clock supports, via truncation. The value of the resolution parameter is not altered. The clock association of the resolution parameter is ignored.

Throws

`IllegalArgumentException` when resolution is null, or when the requested resolution is smaller than the minimum resolution supported by this clock.

`UnsupportedOperationException` when the clock does not support setting its resolution.

Deprecated since RTSJ 2.0

getResolution

Signature

```
public final javax.realtime.RelativeTime  
getResolution()
```

Description

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

Returns

A newly allocated [RelativeTime](#)¹¹¹ object representing the resolution of this. The returned object is associated with this clock.

Deprecated since RTSJ 2.0

See [Section getDrivePrecision](#)

See [Section getQueryPrecision](#)

A.2.3.7 GarbageCollector

The following elements of `GarbageCollector` are deprecated. The required elements are documented in [Section 14.2.2.1](#) above.

¹¹¹Section [9.3.1.3](#)

A.2.3.7.1 Constructors

GarbageCollector

Signature

```
public
GarbageCollector()
```

Description

Create an instance of this.

Deprecated as of RTSJ 1.0.1 This class and any subclasses should be singletons.

A.2.3.8 HighResolutionTime

The following elements of HighResolutionTime are deprecated. The required elements are documented in Section 9.3.1.2 above.

A.2.3.8.1 Methods

absolute(Clock, AbsoluteTime)

Signature

```
public abstract javax.realtime.AbsoluteTime
absolute(Clock clock,
         AbsoluteTime dest)
```

Description

Equivalent to and superseded by [absolute\(Chronograph, AbsoluteTime\)](#)¹¹². When dest is not null, the result is placed there and returned. Otherwise, a new object

¹¹²Section 9.3.1.2.1

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](#)¹¹³ 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. Otherwise, a new object is allocated for the result.

Returns

The AbsoluteTime conversion in dest when dest is not null; otherwise the result is returned in a newly allocated object. It is associated with the 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\(Chronograph\)](#)¹¹⁴.

Parameters

clock is the instance of [Clock](#)¹¹⁵ 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

¹¹³Section [10.3.2.1](#)

¹¹⁴Section [9.3.1.2.1](#)

¹¹⁵Section [10.3.2.1](#)

```
public abstract javax.realtime.RelativeTime
relative(Clock clock,
        RelativeTime dest)
```

Description

Equivalent to and superseded by [relative\(Chronograph, RelativeTime\)](#)¹¹⁶

Parameters

clock The instance of [Clock](#)¹¹⁷ used to convert the time of this into relative time, and the new clock association for the result.

dest When dest is not null, the result is placed there and returned. Otherwise, a new object is allocated for the result.

Returns

The [RelativeTime](#)¹¹⁸ 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*

```
public abstract javax.realtime.RelativeTime
relative(Clock clock)
```

Description

Equivalent to and superseded by [relative\(Chronograph\)](#)¹¹⁹

Parameters

clock The instance of [Clock](#)¹²⁰ 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

¹¹⁶Section [9.3.1.2.1](#)

¹¹⁷Section [10.3.2.1](#)

¹¹⁸Section [9.3.1.3](#)

¹¹⁹Section [9.3.1.2.1](#)

¹²⁰Section [10.3.2.1](#)

A.2.3.9 *IllegalAssignmentError*

The following elements of `IllegalAssignmentError` are deprecated. The required elements are documented in Section 15.2.3.2 above.

A.2.3.9.1 Constructors

`IllegalAssignmentError(String)`

Signature

```
public  
IllegalAssignmentError(String description)
```

Description

A descriptive constructor for `IllegalAssignmentError`.

Deprecated since RTSJ 2.0; application code should use `get()`¹²¹ instead.

Parameters

description Description of the error.

A.2.3.10 `ImmortalPhysicalMemory`

Inheritance

```
java.lang.Object  
  javax.realtime.MemoryArea  
    javax.realtime.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

¹²¹Section 15.2.3.2.2

[ImmortalMemory](#)¹²² 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`.

Deprecated since RTSJ 2.0

A.2.3.10.1 Constructors

`ImmortalPhysicalMemory(Object, long, long, Runnable)`

Signature

```
public
ImmortalPhysicalMemory(Object type,
                        long base,
                        long size,
                        Runnable logic)
```

Description

Create 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.

¹²²Section [11.3.2.2](#)

logic The run() method of this object will be called whenever `MemoryArea.enter()`¹²³ is called. When logic is null, logic must be supplied when the memory area is entered.

Throws

`SecurityException` when the application doesn't have permissions to access physical memory or the given 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`¹²⁴ has been registered with the `PhysicalMemoryManager`¹²⁵.

`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.

`OutOfMemoryError` 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, SizeEstimator, Runnable)

Signature

```
public
ImmortalPhysicalMemory(Object type,
                        long base,
                        SizeEstimator size,
                        Runnable logic)
```

Description

¹²³Section 11.3.2.3.2

¹²⁴Section A.2.1.1

¹²⁵Section A.2.3.19

Create 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.

logic The `run()` method of this object will be called whenever `MemoryArea.enter()`¹²⁶ is called. When logic is null, logic must be supplied when the memory area is entered.

Throws

`SecurityException` when the application doesn't have permissions to access physical memory or the given 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`¹²⁷ has been registered with the `PhysicalMemoryManager`¹²⁸.

`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.

`OutOfMemoryError` 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.

¹²⁶Section 11.3.2.3.2

¹²⁷Section A.2.1.1

¹²⁸Section A.2.3.19

ImmortalPhysicalMemory(Object, long, Runnable)

Signature

```
public  
ImmortalPhysicalMemory(Object type,  
                        long size,  
                        Runnable logic)
```

Description

Create 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()`¹²⁹ is called. When logic is null, logic must be supplied when the memory area is entered.

Throws

`SecurityException` when the application doesn't have permissions to access physical memory or the given 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`¹³⁰ has been registered with the `PhysicalMemoryManager`¹³¹.

`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.

`OutOfMemoryError` when there is insufficient memory for the `ImmortalPhysicalMemory` object or for the backing memory.

¹²⁹Section 11.3.2.3.2

¹³⁰Section A.2.1.1

¹³¹Section A.2.3.19

IllegalAssignmentError when storing logic in this would violate the assignment rules.

ImmortalPhysicalMemory(Object, SizeEstimator, Runnable)

Signature

```
public
ImmortalPhysicalMemory(Object type,
                        SizeEstimator size,
                        Runnable logic)
```

Description

Create 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()`¹³² is called. When logic is null, logic must be supplied when the memory area is entered.

Throws

`SecurityException` when the application doesn't have permissions to access physical memory or the given 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`¹³³ has been registered with the `PhysicalMemoryManager`¹³⁴.

`IllegalArgumentException` when size is null, or `size.getEstimate()` is negative.

MemoryTypeConflictException when type specifies incompatible memory attributes.

¹³²Section 11.3.2.3.2

¹³³Section A.2.1.1

¹³⁴Section A.2.3.19

`OutOfMemoryError` 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, long)

Signature

```
public  
ImmortalPhysicalMemory(Object type,  
                        long base,  
                        long size)
```

Description

Create 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

`SecurityException` when the application doesn't have permissions to access physical memory or the given range of memory.

`OffsetOutOfBoundsException` when the base address is invalid.

`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`¹³⁵ has been registered with the `PhysicalMemoryManager`¹³⁶.

`MemoryTypeConflictException` when the specified base does not point to memory that matches the requested type, or when type specifies incompatible memory attributes.

¹³⁵Section A.2.1.1

¹³⁶Section A.2.3.19

`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.

`OutOfMemoryError` when there is insufficient memory for the `ImmortalPhysicalMemory` object or for the backing memory.

ImmortalPhysicalMemory(Object, long, SizeEstimator)

Signature

```
public
ImmortalPhysicalMemory(Object type,
                        long base,
                        SizeEstimator size)
```

Description

Create 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

`SecurityException` 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`¹³⁷ has been registered with the `PhysicalMemoryManager`¹³⁸.

¹³⁷Section [A.2.1.1](#)

¹³⁸Section [A.2.3.19](#)

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.

OutOfMemoryError when there is insufficient memory for the **ImmortalPhysicalMemory** object or for the backing memory.

ImmortalPhysicalMemory(Object, long)

Signature

```
public  
    ImmortalPhysicalMemory(Object type,  
                           long size)
```

Description

Create 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

SecurityException 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**¹³⁹ has been registered with the **PhysicalMemoryManager**¹⁴⁰.

MemoryTypeConflictException when type specifies incompatible memory attributes.

¹³⁹Section A.2.1.1

¹⁴⁰Section A.2.3.19

`IllegalArgumentException` when size is less than zero.

`OutOfMemoryError` when there is insufficient memory for the `ImmutablePhysicalMemory` object or for the backing memory.

`SizeOutOfBoundsException` when the size extends into an invalid range of memory.

ImmutablePhysicalMemory(Object, SizeEstimator)

Signature

```
public
ImmutablePhysicalMemory(Object type,
                        SizeEstimator size)
```

Description

Create 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

`SecurityException` 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`¹⁴¹ has been registered with the `PhysicalMemoryManager`¹⁴².

`MemoryTypeConflictException` when `type` specifies incompatible memory attributes.

`IllegalArgumentException` when `size` is null, or `size.getEstimate()` is negative.

`OutOfMemoryError` when there is insufficient memory for the `ImmutablePhysicalMemory` object or for the backing memory.

¹⁴¹Section [A.2.1.1](#)

¹⁴²Section [A.2.3.19](#)

A.2.3.11 LTMemory

Inheritance

java.lang.Object
 javax.realtime.MemoryArea
 javax.realtime.ScopedMemory
 javax.realtime.LTMemory

Description

Equivalent to and superseded by [javax.realtime.memory.LTMemory](#)¹⁴³.

Deprecated since RTSJ 2.0; moved to package javax.realtime.memory.

A.2.3.11.1 Constructors

LTMemory(long, long, Runnable)

Signature

```
public
LTMemory(long initial,
          long maximum,
          Runnable logic)
```

Description

Create 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\(long initial, long maximum\)](#)¹⁴⁴.

Throws

¹⁴³Section [11.4.3.1](#)

¹⁴⁴Section [A.2.3.11.1](#)

`IllegalArgumentException` when `initial` is greater than `maximum`, or when `initial` or `maximum` is less than zero.

`OutOfMemoryError` when there is insufficient memory for the `LTMemory` object or for the backing memory.

`IllegalAssignmentError` when storing logic in this would violate the assignment rules.

LTMemory(SizeEstimator, SizeEstimator, Runnable)

Signature

```
public
LTMemory(SizeEstimator initial,
         SizeEstimator maximum,
         Runnable logic)
```

Description

Equivalent to `LTMemory(long, long, Runnable)`¹⁴⁵ with the argument list (`initial.getEstimate()`, `maximum.getEstimate()`, `logic`).

Parameters

`initial` An instance of `SizeEstimator`¹⁴⁶ 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`¹⁴⁷ 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)`¹⁴⁸.

Throws

`IllegalArgumentException` when `initial` is null, `maximum` is null, `initial.getEstimate()` is greater than `maximum.getEstimate()`, or when `initial.getEstimate()` is less than zero.

`OutOfMemoryError` when there is insufficient memory for the `LTMemory` object or for the backing memory.

`IllegalAssignmentError` when storing logic in this would violate the assignment rules.

¹⁴⁵Section [A.2.3.11.1](#)

¹⁴⁶Section [11.3.2.5](#)

¹⁴⁷Section [11.3.2.5](#)

¹⁴⁸Section [A.2.3.11.1](#)

LTMemory(long, long)

Signature

```
public
LTMemory(long initial,
          long maximum)
```

Description

Equivalent to [LTMemory\(long, long, Runnable\)](#)¹⁴⁹ 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.

OutOfMemoryError when there is insufficient memory for the LTMemory object or for the backing memory.

LTMemory(SizeEstimator, SizeEstimator)

Signature

```
public
LTMemory(SizeEstimator initial,
          SizeEstimator maximum)
```

Description

Equivalent to [LTMemory\(long, long, Runnable\)](#)¹⁵⁰ with the argument list (initial.getEstimate(), maximum.getEstimate(), null).

Parameters

initial An instance of [SizeEstimator](#)¹⁵¹ used to give an estimate of the initial size. This memory must be committed before the completion of the constructor.

¹⁴⁹Section [A.2.3.11.1](#)

¹⁵⁰Section [A.2.3.11.1](#)

¹⁵¹Section [11.3.2.5](#)

maximum An instance of [SizeEstimator](#)¹⁵² used to give an estimate for the maximum bytes to allocate for this area.

Throws

`IllegalArgumentException` when `initial` is null, `maximum` is null, `initial.getEstimate()` is greater than `maximum.getEstimate()`, or when `initial.getEstimate()` is less than zero.

`OutOfMemoryError` when there is insufficient memory for the `LTMemory` object or for the backing memory.

LTMemory(long, Runnable)

Signature

```
public
LTMemory(long size,
         Runnable logic)
```

Description

Equivalent to `LTMemory(long, long, Runnable)`¹⁵³ 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 initial memory area. When `logic` is null, this constructor is equivalent to `LTMemory(long size)`¹⁵⁴.

Throws

`IllegalArgumentException` when `size` is less than zero.

`OutOfMemoryError` when there is insufficient memory for the `LTMemory` object or for the backing memory.

`IllegalAssignmentError` when storing `logic` in this would violate the assignment rules.

¹⁵²Section [11.3.2.5](#)

¹⁵³Section [A.2.3.11.1](#)

¹⁵⁴Section [A.2.3.11.1](#)

LTMemory(SizeEstimator, Runnable)

Signature

```
public  
LTMemory(SizeEstimator size,  
          Runnable logic)
```

Description

Equivalent to `LTMemory(long, long, Runnable)`¹⁵⁵ with the argument list (size, getEstimate(), size.getEstimate(), logic).

Available since RTSJ 1.0.1

Parameters

size An instance of `SizeEstimator`¹⁵⁶ used to give an estimate of the initial size. This memory must be committed before the completion of the constructor.

logic The run() of the given Runnable will be executed using this as its initial memory area. When logic is null, this constructor is equivalent to `LTMemory(SizeEstimator initial)`¹⁵⁷.

Throws

`IllegalArgumentException` when size is null, or size.getEstimate() is less than zero.

`OutOfMemoryError` when there is insufficient memory for the LTMemory object or for the backing memory.

`IllegalAssignmentError` when storing logic in this would violate the assignment rules.

LTMemory(long)

Signature

```
public  
LTMemory(long size)
```

Description

¹⁵⁵Section [A.2.3.11.1](#)

¹⁵⁶Section [11.3.2.5](#)

¹⁵⁷Section [A.2.3.11.1](#)

Equivalent to `LTMemory(long, long, Runnable)`¹⁵⁸ 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.

`OutOfMemoryError` when there is insufficient memory for the `LTMemory` object or for the backing memory.

LTMemory(SizeEstimator)

Signature

```
public
LTMemory(SizeEstimator size)
```

Description

Equivalent to `LTMemory(long, long, Runnable)`¹⁵⁹ with the argument list (size, `size.getEstimate()`, `size.getEstimate()`, null).

Available since RTSJ 1.0.1

Parameters

size An instance of `SizeEstimator`¹⁶⁰ 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.

`OutOfMemoryError` when there is insufficient memory for the `LTMemory` object or for the backing memory.

A.2.3.11.2 Methods

¹⁵⁸Section [A.2.3.11.1](#)

¹⁵⁹Section [A.2.3.11.1](#)

¹⁶⁰Section [11.3.2.5](#)

toString

Signature

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

Description

Create a string representation of this object. The string is of the form

(LTMemory) Scoped memory # num

where num uniquely identifies the LTMemory area.

Returns

A string representing the value of this.

A.2.3.12 LTPhysicalMemory

Inheritance

```
java.lang.Object  
  javax.realtime.MemoryArea  
    javax.realtime.ScopedMemory  
      javax.realtime.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](#)¹⁶¹ memory areas, and the same performance restrictions as [LTMemory](#)¹⁶².

No provision is made for sharing object in LTPhysicalMemory with entities outside the JVM that creates them, and, while the memory backing an instance of LTPhysicalMemory could be shared by multiple JVMs, the class does not support such sharing.

Methods from LTPhysicalMemory should be overridden only by methods that use super.

Deprecated since RTSJ 2.0

¹⁶¹Section [A.2.3.31](#)

¹⁶²Section [A.2.3.11](#)

A.2.3.12.1 Constructors

LTPhysicalMemory(Object, long, long, Runnable)*Signature*

```

public
LTPhysicalMemory(Object type,
                  long base,
                  long size,
                  Runnable logic)

```

Description

Create 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\(\)](#)¹⁶³ 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.

[SecurityException](#) 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.

¹⁶³Section [11.3.2.3.2](#)

UnsupportedPhysicalMemoryException when the underlying hardware does not support the given type, or when no matching **PhysicalMemoryTypeFilter**¹⁶⁴ has been registered with the **PhysicalMemoryManager**¹⁶⁵.

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)**¹⁶⁶ 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.

¹⁶⁴Section **A.2.1.1**

¹⁶⁵Section **A.2.3.19**

¹⁶⁶Section **A.2.3.12.1**

logic The `run()` method of this object will be called whenever `MemoryArea.enter()`¹⁶⁷ is called. When logic is null, logic must be supplied when the memory area is entered.

Throws

`SecurityException` when the application doesn't have permissions to access physical memory or the given 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`¹⁶⁸ has been registered with the `PhysicalMemoryManager`¹⁶⁹.

`MemoryTypeConflictException` when the specified base does not point to memory that matches the requested type, or when type specifies incompatible memory attributes.

`MemoryInUseException` when the specified memory is already in use.

`IllegalArgumentException` when size is null, or `size.getEstimate()` is negative.

`IllegalAssignmentError` when storing logic in this would violate the assignment rules.

LTPhysicalMemory(Object, long, long)

Signature

```
public
LTPhysicalMemory(Object type,
                  long base,
                  long size)
```

Description

Equivalent to `LTPhysicalMemory(Object, long, long, Runnable)`¹⁷⁰ with the the argument list (type, base, size, null).

See Section `PhysicalMemoryManager`

¹⁶⁷Section 11.3.2.3.2

¹⁶⁸Section A.2.1.1

¹⁶⁹Section A.2.3.19

¹⁷⁰Section A.2.3.12.1

Parameters

type An instance of `Object` representing the type of memory required (e.g., *dma*, *shared*) - used to define the base address and control the mapping. When the required memory has more than one attribute, **type** may be an array of objects. When **type** is null or a reference to an array with no entries, any type of memory is acceptable. Note that **type** values are compared by reference (`==`), not by value (`equals`).

base The physical memory address of the area.

size The size of the area in bytes.

Throws

`SecurityException` when the application doesn't have permissions to access physical memory or the given 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`¹⁷¹ has been registered with the `PhysicalMemoryManager`¹⁷².

`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*

```
public
LTPhysicalMemory(Object type,
                  long base,
                  SizeEstimator size)
```

Description

Equivalent to `LTPhysicalMemory(Object, long, long, Runnable)`¹⁷³ with the argument list (**type**, **base**, **size.getEstimate()**, null).

¹⁷¹Section A.2.1.1

¹⁷²Section A.2.3.19

¹⁷³Section A.2.3.12.1

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

`SecurityException` 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](#)¹⁷⁴ has been registered with the [PhysicalMemoryManager](#)¹⁷⁵.

[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

¹⁷⁴Section [A.2.1.1](#)

¹⁷⁵Section [A.2.3.19](#)

Equivalent to `LTPhysicalMemory(Object, long, long, Runnable)`¹⁷⁶ with the argument list (type, 0, size, 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).

size The size of the area in bytes.

logic The `run()` method of this object will be called whenever `MemoryArea.enter()`¹⁷⁷ is called. When logic is null, logic must be supplied when the memory area is entered.

Throws

`SecurityException` when the application doesn't have permissions to access physical memory or the given 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`¹⁷⁸ has been registered with the `PhysicalMemoryManager`¹⁷⁹.

`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(Object, SizeEstimator, Runnable)

Signature

¹⁷⁶Section [A.2.3.12.1](#)

¹⁷⁷Section [11.3.2.3.2](#)

¹⁷⁸Section [A.2.1.1](#)

¹⁷⁹Section [A.2.3.19](#)

```
public
LTPhysicalMemory(Object type,
                  SizeEstimator size,
                  Runnable logic)
```

Description

Equivalent to `LTPhysicalMemory(Object, long, long, Runnable)`¹⁸⁰ with the argument list (type, 0, 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).

size A size estimator for this area.

logic The `run()` method of this object will be called whenever `MemoryArea.enter()`¹⁸¹ is called. When logic is null, logic must be supplied when the memory area is entered.

Throws

`SecurityException` when the application doesn't have permissions to access physical memory or the given 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`¹⁸² has been registered with the `PhysicalMemoryManager`¹⁸³.

`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 would violate the assignment rules.

¹⁸⁰Section [A.2.3.12.1](#)

¹⁸¹Section [11.3.2.3.2](#)

¹⁸²Section [A.2.1.1](#)

¹⁸³Section [A.2.3.19](#)

LTPhysicalMemory(Object, long)

Signature

```
public  
LTPhysicalMemory(Object type,  
                  long size)
```

Description

Equivalent to [LTPhysicalMemory\(Object, long, long, Runnable\)](#)¹⁸⁴ with the argument list (type, 0, 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).

size The size of the area in bytes.

Throws

[SecurityException](#) 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](#)¹⁸⁵ has been registered with the [PhysicalMemoryManager](#)¹⁸⁶.

[MemoryTypeConflictException](#) when type specifies incompatible memory attributes.

LTPhysicalMemory(Object, SizeEstimator)

Signature

¹⁸⁴Section [A.2.3.12.1](#)

¹⁸⁵Section [A.2.1.1](#)

¹⁸⁶Section [A.2.3.19](#)

```
public
LTPhysicalMemory(Object type,
                  SizeEstimator size)
```

Description

Equivalent to `LTPhysicalMemory(Object, long, long, Runnable)`¹⁸⁷ with the argument list (type, 0, 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).

size A size estimator for this area.

Throws

`SecurityException` 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`¹⁸⁸ has been registered with the `PhysicalMemoryManager`¹⁸⁹.

`MemoryTypeConflictException` when type specifies incompatible memory attributes.

`IllegalArgumentException` when size is null, or size.getEstimate() is negative.

A.2.3.12.2 Methods

toString*Signature*

¹⁸⁷Section [A.2.3.12.1](#)

¹⁸⁸Section [A.2.1.1](#)

¹⁸⁹Section [A.2.3.19](#)


```
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.

A.2.3.13 *MemoryAccessError*

The following elements of MemoryAccessError are deprecated. The required elements are documented in Section 15.2.3.3 above.

A.2.3.13.1 Constructors

MemoryAccessError(String)*Signature*

```
public  
MemoryAccessError(String description)
```

Description

A descriptive constructor for MemoryAccessError.

Deprecated since RTSJ 2.0; application code should use `get()`¹⁹⁰ instead.

Parameters

description Description of the error.

¹⁹⁰Section 15.2.3.3.2

A.2.3.14 NoHeapRealtimeThread

Inheritance

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

Description

A `NoHeapRealtimeThread` is a specialized form of `RealtimeThread`¹⁹¹. 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` and `join`) complete normally except where execution would cause access violations. The constructors of `NoHeapRealtimeThread` require a reference to `ScopedMemory`¹⁹² or `ImmortalMemory`¹⁹³.

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

A.2.3.14.1 Constructors

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

¹⁹¹Section 5.3.2.2

¹⁹²Section A.2.3.31

¹⁹³Section 11.3.2.2

Signature

```
public  
NoHeapRealtimeThread(SchedulingParameters scheduling,  
                      javax.realtime.ReleaseParameters release,  
                      MemoryParameters memory,  
                      MemoryArea area,  
                      ProcessingGroupParameters group,  
                      Runnable logic)
```

Description

Create 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 it defaults to the a copy of the creator's release parameters created in the same memory area as the new NoHeapRealtimeThread.

memory the MemoryParameters associated with this (and possibly other instances of Schedulable). When memory is null, the new NoHeapRealtimeThread will have a null value for its memory parameters, and the amount or rate of memory allocation is unrestricted.

area the MemoryArea associated with this. When area is null, an IllegalArgumentException is thrown.

group the ProcessingGroupParameters associated with this (and possibly other instances of Schedulable). When null, the new NoHeapRealtimeThread will not be associated with any processing group.

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

Throws

`IllegalArgumentException` when the parameters are not compatible with the associated scheduler, when area is null, when area is heap memory, when area, scheduling release, memory or group is allocated in heap memory. when this is in heap memory, or when logic is in heap memory.

`IllegalAssignmentError` when the new `NoHeapRealtimeThread` instance cannot hold references to non-null values of the scheduling release, memory and group, or when those parameters cannot hold a reference to the new `NoHeapRealtimeThread`.

NoHeapRealtimeThread(SchedulingParameters, ReleaseParameters, MemoryArea)

Signature

```
public
NoHeapRealtimeThread(SchedulingParameters scheduling,
                      javax.realtime.ReleaseParameters release,
                      MemoryArea area)
```

Description

Create a realtime thread which may not use the heap with the given [SchedulingParameters](#)¹⁹⁴, [ReleaseParameters](#)¹⁹⁵ and [MemoryArea](#)¹⁹⁶, and default values for all other parameters. This constructor is equivalent to `NoHeapRealtimeThread(scheduling, release, null, area, null, null, null)`.

NoHeapRealtimeThread(SchedulingParameters, MemoryArea)

Signature

```
public
NoHeapRealtimeThread(SchedulingParameters scheduling,
                      MemoryArea area)
```

Description

¹⁹⁴Section [6.3.3.14](#)

¹⁹⁵Section [6.3.3.10](#)

¹⁹⁶Section [11.3.2.3](#)

Create a realtime thread with the given [SchedulingParameters](#)¹⁹⁷ and [MemoryArea](#)¹⁹⁸ and default values for all other parameters.

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

A.2.3.14.2 Methods

start

Signature

```
public void  
start()
```

Description

Set up the realtime thread's environment and start it. The set up might include delaying it until the assigned start time and initializing the thread's scope stack. (See [ScopedMemory](#)¹⁹⁹.)

Throws

`IllegalStateException` when the configured Scheduler and `SchedulingParameters` for this `RealtimeThread` are not compatible.

Available since RTSJ 2.0 adds new exception

startPeriodic(PhasingPolicy)

Signature

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

Description

Start the thread with the specified phasing policy.

Available since RTSJ 2.0

¹⁹⁷Section [6.3.3.14](#)

¹⁹⁸Section [11.3.2.3](#)

¹⁹⁹Section [A.2.3.31](#)

A.2.3.15 *OneShotTimer*

The following elements of `OneShotTimer` are deprecated. The required elements are documented in Section 10.3.2.2 above.

A.2.3.15.1 Constructors

`OneShotTimer(HighResolutionTime, Clock, AsyncEventHandler)`

Signature

```
public
OneShotTimer(javax.realtime.HighResolutionTime<?> time,
              Clock clock,
              AsyncEventHandler handler)
throws IllegalArgumentException,
      UnsupportedOperationException,
      IllegalAssignmentError
```

Description

Create an instance of `OneShotTimer`²⁰⁰, based on the given clock, that will execute its `fire` method according to the given time. The `Clock`²⁰¹ association of the parameter `time` is ignored.

Deprecated 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()`.
- `clock` The clock on which to base this timer, overriding the clock associated with the parameter `time`. When `null`, the system `Realtime` clock is used. The clock associated with the parameter `time` is always ignored.

²⁰⁰Section 10.3.2.2

²⁰¹Section 10.3.2.1

handler The [AsyncEventHandler](#)²⁰² that will be released when fire is invoked. When null, no handler is associated with this Timer and nothing will happen when this event fires unless a handler is subsequently associated with the timer using the addHandler() or setHandler() method.

Throws

IllegalArgumentException when time is a RelativeTime instance less than zero.

UnsupportedOperationException when the [Chronograph](#)²⁰³ associated with time is not a [Clock](#)²⁰⁴.

[IllegalAssignmentError](#) when this OneShotTimer cannot hold references to time, handler, or clock.

A.2.3.16 POSIXSignalHandler

Inheritance

java.lang.Object

[java.x.realtime.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

A.2.3.16.1 Fields

SIGHUP

public static final SIGHUP

Description

²⁰²Section [8.3.3.5](#)

²⁰³Section [10.3.1.2](#)

²⁰⁴Section [10.3.2.1](#)

Hangup (POSIX).

SIGINT

public static final SIGINT

Description

interrupt (ANSI)

SIGQUIT

public static final SIGQUIT

Description

quit (POSIX)

SIGILL

public static final SIGILL

Description

illegal instruction (ANSI)

SIGTRAP

public static final SIGTRAP

Description

trace trap (POSIX), optional signal.

SIGABRT

public static final SIGABRT

Description

Abort (ANSI).

SIGBUS

public static final SIGBUS

Description

BUS error (4.2 BSD), optional signal.

SIGFPE

public static final SIGFPE

Description

floating point exception

SIGKILL

public static final SIGKILL

Description

Kill, unblockable (POSIX).

SIGUSR1

public static final SIGUSR1

Description

User-defined signal 1 (POSIX).

SIGSEGV

public static final SIGSEGV

Description

Segmentation violation (ANSI).

SIGUSR2

public static final SIGUSR2

Description

User-defined signal 2 (POSIX).

SIGPIPE

public static final SIGPIPE

Description

Broken pipe (POSIX).

SIGALRM

public static final SIGALRM

Description

Alarm clock (POSIX).

SIGTERM

public static final SIGTERM

Description

Termination (ANSI).

SIGCHLD

public static final SIGCHLD

Description

Child status has changed (POSIX).

SIGCONT

public static final SIGCONT

Description

Continue (POSIX), optional signal.

SIGSTOP

public static final SIGSTOP

Description

Stop, unblockable (POSIX), optional signal.

SIGTSTP

public static final SIGTSTP

Description

Keyboard stop (POSIX), optional signal.

SIGTTIN

public static final SIGTTIN

Description

Background read from tty (POSIX), optional signal.

SIGTTOU

public static final SIGTTOU

Description

Background write to tty (POSIX), optional signal.

SIGURG

public static final SIGURG

Description

Urgent condition on socket (4.2 BSD).

Deprecated as of RTSJ 1.0.1 not part of POSIX 9945-1-1996 standard

SIGXCPU

public static final SIGXCPU

Description

CPU limit exceeded (4.2 BSD).

Deprecated as of RTSJ 1.0.1 not part of POSIX 9945-1-1996 standard

SIGXFSZ

public static final SIGXFSZ

Description

File size limit exceeded (4.2 BSD).

Deprecated as of RTSJ 1.0.1 not part of POSIX 9945-1-1996 standard

SIGVTALRM

public static final SIGVTALRM

Description

Virtual alarm clock (4.2 BSD).

Deprecated as of RTSJ 1.0.1 not part of POSIX 9945-1-1996 standard

SIGPROF

public static final SIGPROF

Description

Profiling alarm clock (4.2 BSD).

Deprecated as of RTSJ 1.0.1 not part of POSIX 9945-1-1996 standard

SIGWINCH

public static final SIGWINCH

Description

Window size change (4.3 BSD, Sun).

Deprecated as of RTSJ 1.0.1 not part of POSIX 9945-1-1996 standard

SIGIO

public static final SIGIO

Description

I/O now possible (4.2 BSD).

Deprecated as of RTSJ 1.0.1 not part of POSIX 9945-1-1996 standard

SIGPWR

public static final SIGPWR

Description

Power failure restart (System V).

Deprecated as of RTSJ 1.0.1 not part of POSIX 9945-1-1996 standard

SIGSYS

public static final SIGSYS

Description

Bad system call, optional signal.

SIGIOT

public static final SIGIOT

Description

IOT instruction (4.2 BSD), optional signal.

SIGPOLL

public static final SIGPOLL

Description

Pollable event occurred (System V).

Deprecated as of RTSJ 1.0.1 not part of POSIX 9945-1-1996 standard

SIGCLD

public static final SIGCLD

Description

Same as SIGCHLD (System V), optional signal.

SIGEMT

public static final SIGEMT

Description

EMT instruction, optional signal.

SIGLOST

public static final SIGLOST

Description

Deprecated as of RTSJ 1.0.1 not part of POSIX 9945-1-1996 standard

SIGCANCEL

public static final SIGCANCEL

Description

Deprecated as of RTSJ 1.0.1 not part of POSIX 9945-1-1996 standard

SIGFREEZE

public static final SIGFREEZE

Description

Deprecated as of RTSJ 1.0.1 not part of POSIX 9945-1-1996 standard

SIGLWP

public static final SIGLWP

Description

Deprecated as of RTSJ 1.0.1 not part of POSIX 9945-1-1996 standard

SIGTHAW

public static final SIGTHAW

Description

Deprecated as of RTSJ 1.0.1 not part of POSIX 9945-1-1996 standard

SIGWAITING

public static final SIGWAITING

Description

Deprecated as of RTSJ 1.0.1 not part of POSIX 9945-1-1996 standard

A.2.3.16.2 Methods

addHandler(int, AsyncEventHandler)

Signature

```
public static void
addHandler(int signal,
           AsyncEventHandler handler)
```

Description

`addHandler` 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` iff `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

`removeHandler` 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` iff `signal` is not defined by any of the constants in this class.

`setHandler(int, AsyncEventHandler)`

Signature

```
public static void  
setHandler(int signal,  
            AsyncEventHandler handler)
```

Description

`setHandler` sets the set of handlers that will be released on the provided signal to the set with the provided handler being the single element.

Parameters

signal The POSIX signal as defined in the constants SIG*.

handler the handler to be released on the given signal, null to remove all handlers for the given signal.

Throws

IllegalArgumentException iff signal is not defined by any of the constants in this class.

A.2.3.17 *PeriodicParameters*

The following elements of PeriodicParameters are deprecated. The required elements are documented in Section 6.3.3.6 above.

A.2.3.17.1 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 this. 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.setReleaseParameters\(ReleaseParameters\)](#)²⁰⁵) the parameters of those schedulables are altered as specified by each schedulable's respective scheduler.

Parameters

period The proposed period. There is no default value. When period is null an exception is thrown.

²⁰⁵Section 5.3.2.2.2

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 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. Also when the values are incompatible with the scheduler for any of the schedulables which are presently using this parameter object.

`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.

Deprecated as of RTSJ 2.0; the framework for feasibility analysis is inadequate

A.2.3.18 *PeriodicTimer*

The following elements of `PeriodicTimer` are deprecated. The required elements are documented in Section 10.3.2.3 above.

A.2.3.18.1 Constructors

PeriodicTimer(HighResolutionTime, RelativeTime, Clock, Async-EventHandler)

Signature

```
public
PeriodicTimer(javax.realtime.HighResolutionTime<?> start,
               RelativeTime interval,
               Clock clock,
               AsyncEventHandler handler)
```

throws `IllegalArgumentException`,
`UnsupportedOperationException`,
`IllegalAssignmentError`

Description

Create a timer that executes its fire method periodically.

Deprecated 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 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`²⁰⁶.
- clock The clock to be used to time the start and interval. When null, the system Realtime clock is used. The `Clock`²⁰⁷ association of the parameters start and interval is always ignored.
- handler The `AsyncEventHandler`²⁰⁸ that will be released when fire is invoked. When null, no handler is associated with this `Timer` and nothing will happen when this event fires unless a handler is subsequently associated with the timer using the `addHandler()` or `setHandler()` method.

Throws

- `IllegalArgumentException` when start or interval is a `RelativeTime` instance with a value less than zero; or the clocks associated with start and interval are not the identical.
- `IllegalAssignmentError` when this `PeriodicTimer` cannot hold references to handler, clock and interval.
- `UnsupportedOperationException` when the `Chronograph`²⁰⁹ associated with time is not a `Clock`²¹⁰.

²⁰⁶Section 10.3.2.2

²⁰⁷Section 10.3.2.1

²⁰⁸Section 8.3.3.5

²⁰⁹Section 10.3.1.2

²¹⁰Section 10.3.2.1

A.2.3.19 PhysicalMemoryManager

Inheritance

java.lang.Object
 [javax.realtime.PhysicalMemoryManager](#)

Description

The PhysicalMemoryManager is not ordinarily used by applications, except that the implementation may require the application to use the [registerFilter](#)²¹¹ 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](#)²¹², [LTPhysicalMemory](#)²¹³, and [ImmortalPhysicalMemory](#)²¹⁴) 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 holes in the memory space. Some of these holes may be filled with removable memory.

The physical memory is partitioned into chunks (pages, segments, etc.). Each chunk of memory has a base address and a length.

Each chunk of memory has certain properties. Some of these properties may require actions to be performed by the Physical Memory Manager when the memory is accessed. For example, access to IO_PAGE may require the use of special instructions to even reach the devices, or it may require special code sequences to ensure proper handling of processor write queues and caches.

Filters tell the Physical Memory Manager about the properties of the memory that are available on the machine by registering with the Physical Memory Manager.

When the program requests a physical memory area with particular properties, the constructor communicates with the Physical Memory Manager through a private interface. The Physical Memory Manager asks the filter if the address specified has the required properties and whether it is free, or asks for a chunk of memory with the requested size.

²¹¹Section [A.2.3.19.2](#)

²¹²Section [A.2.3.36](#)

²¹³Section [A.2.3.12](#)

²¹⁴Section [A.2.3.10](#)

The Physical Memory Manager then maps the physical memory chunk into virtual memory (on systems that support virtual memory). and locks the virtual memory to the memory chunk.

Examples of characteristics that might be specified are DMA memory, hardware byte swapping, 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 or other interested parties may provide [PhysicalMemoryTypeFilter](#)²¹⁵ classes that allow additional characteristics of memory devices to be specified.

Deprecated as of RTSJ 2.0

A.2.3.19.1 Fields

ALIGNED

public static final 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

public static final BYTESWAP

Description

²¹⁵Section [A.2.1.1](#)

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

public static final DMA

Description

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

public static final 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

public static final 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.

A.2.3.19.2 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

```
public static void  
onInsertion(long base,  
            long size,  
            AsyncEvent ae)
```

Description

Register the specified `AsyncEvent`²¹⁶ 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

²¹⁶Section 8.3.3.4

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

onInsertion(long, long, AsyncEventHandler)

Signature

```
public static void  
onInsertion(long base,  
            long size,  
            AsyncEventHandler aeh)
```

Description

Register the specified `AsyncEventHandler`²¹⁷ to run when any memory in the range is added to the system. When the specified range of physical memory contains multiple different types of removable memory, the AEH will be registered with each of them. When the size or the base is less than 0, unregister all "onInsertion" references to the handler.

Note 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 contains no removable memory, or when aeh is null and size and base are both greater than or equal to zero.

²¹⁷Section 8.3.3.5

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

Register the specified AE to fire when any memory in the range is removed from the system. When the specified range of physical memory contains multiple different types of removable memory, the AE will be registered with each of them.

Parameters

base The starting address in physical memory.

size The size of the memory area.

ae The async event to register.

Throws

IllegalArgumentException when the specified range contains no removable memory, when ae is null, or when size is less than zero.

OffsetOutOfBoundsException when base is less than zero.

SizeOutOfBoundsException when base plus size would be greater than the physical addressing range of the processor.

onRemoval(long, long, AsyncEventHandler)

Signature

```
public static void  
onRemoval(long base,  
           long size,  
           AsyncEventHandler aeh)
```

Description

Register the specified AEH to run when any memory in the range is removed from the system. When the specified range of physical memory contains multiple different types of removable memory, the AEH will be registered with each of

them. When size or base is less than 0, unregister all "onRemoval" references to the handler parameter.

Note 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

```
public static final void  
registerFilter(Object name,  
               PhysicalMemoryTypeFilter filter)  
throws DuplicateFilterException
```

Description

Register 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.

`SecurityException` when this operation is not permitted.

removeFilter(Object)*Signature*

```
public static final void  
removeFilter(Object name)
```

Description

Remove the identified filter from the set of registered filters. When the filter is not registered, silently do nothing.

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

Parameters

name The identifying object for this memory attribute.

Throws

IllegalArgumentException when name is null.

SecurityException when this operation is not permitted.

unregisterInsertionEvent(long, long, AsyncEvent)*Signature*

```
public static boolean  
unregisterInsertionEvent(long base,  
                           long size,  
                           AsyncEvent ae)
```

Description

Unregister the specified insertion event. The event is only unregistered when all three arguments match the arguments used to register the event, except that ae of null matches all values of ae and will unregister every ae that matches the address range.

Note 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 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

```
public static boolean  
unregisterRemovalEvent(long base,  
                        long size,  
                        AsyncEvent ae)
```

Description

Unregister the specified removal event. The async event is only unregistered when all three arguments match the arguments used to register the event, except that ae of null matches all values of ae and will unregister every ae that matches the address range.

Note 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

A.2.3.20 *PriorityCeilingEmulation*

The following elements of `PriorityCeilingEmulation` are deprecated. The required elements are documented in Section [7.3.1.2](#) above.

A.2.3.20.1 Methods

getDefaultCeiling

Signature

```
public int  
getDefaultCeiling()
```

Description

Gets the priority ceiling for this `PriorityCeilingEmulation` object.

Returns

The priority ceiling.

Deprecated as of RTSJ 1.0.1. The method name is misleading. Replaced with `getCeiling()`

A.2.3.21 *PriorityScheduler*

The following elements of `PriorityScheduler` are deprecated. The required elements are documented in Section [6.3.3.8](#) above.

A.2.3.21.1 Fields

MAX_PRIORITY

```
public static final MAX_PRIORITY
```

Description

The maximum priority value used by the implementation.

Deprecated as of RTSJ 1.0.1 Use the [getMaxPriority²¹⁸](#) method instead.

MIN_PRIORITY

```
public static final MIN_PRIORITY
```

Description

The minimum priority value used by the implementation.

Deprecated as of RTSJ 1.0.1 Use the [getMinPriority²¹⁹](#) method instead.

A.2.3.21.2 Methods

getMaxPriority(Thread)*Signature*

```
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 a Java thread then 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

IllegalArgumentException when thread is a realtime thread that is not scheduled by an instance of PriorityScheduler.

²¹⁸Section [6.3.3.8.3](#)

²¹⁹Section [6.3.3.8.3](#)

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 a Java thread then 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

IllegalArgumentException 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 a Java

thread then 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

IllegalArgumentException 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

Return a reference to the distinguished instance of PriorityScheduler which is the system's base scheduler.

Returns

A reference to the distinguished instance PriorityScheduler.

Deprecated since RTSJ 2.0

isFeasible

Signature

```
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](#)²²⁰ and [SporadicParameters](#)²²¹.

When any schedulable within the feasibility set has release parameters of the exact type [AperiodicParameters](#)²²² (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](#)²²³, or passes proposed release parameters of type [AperiodicParameters](#)²²⁴, 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](#)²²⁵, is when the method will change those release parameters to not be [AperiodicParameters](#)²²⁶.

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

```
public boolean
setIfFeasible(javax.realtime.Schedulable<?> schedulable,
              javax.realtime.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

²²⁰Section [6.3.3.6](#)

²²¹Section [6.3.3.15](#)

²²²Section [6.3.3.2](#)

²²³Section [6.3.3.2](#)

²²⁴Section [6.3.3.2](#)

²²⁵Section [6.3.3.2](#)

²²⁶Section [6.3.3.2](#)

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](#)²²⁷.)

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](#)²²⁸.)

Throws

IllegalArgumentException 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`²²⁹ or `RealtimeThread.waitForNextPeriodInterruptible`²³⁰.

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

²²⁷Section [6.3.3.8](#)

²²⁸Section [6.3.3.8](#)

²²⁹Section ??

²³⁰Section ??

```

public boolean
setIfFeasible(javax.realtime.Schedulable<?> schedulable,
              javax.realtime.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 [PriorityScheduler²³¹](#).)

`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 [PriorityScheduler²³²](#).)

`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 [PriorityScheduler²³³](#).)

Throws

`IllegalArgumentException` 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.waitForNextPeriod234` or

²³¹Section [6.3.3.8](#)

²³²Section [6.3.3.8](#)

²³³Section [6.3.3.8](#)

²³⁴Section [??](#)

RealtimeThread.waitForNextPeriodInterruptible²³⁵.

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, SchedulingParameters, ReleaseParameters, MemoryParameters, ProcessingGroupParameters)

Signature

```
public boolean
setIfFeasible(javax.realtime.Schedulable<?> schedulable,
              SchedulingParameters scheduling,
              javax.realtime.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](#)²³⁶.)

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](#)²³⁷.)

²³⁵Section ??

²³⁶Section 6.3.3.8

²³⁷Section 6.3.3.8

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](#)²³⁸.)

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](#)²³⁹.)

Throws

`IllegalArgumentException` 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`²⁴⁰ or `RealtimeThread.waitForNextPeriodInterruptible`²⁴¹.

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(javax.realtime.Schedulable<?> schedulable)
```

Description

Inform this scheduler and cooperating facilities that the resource demands of the given instance of [Schedulable](#)²⁴² will be considered in the feasibility analysis of the associated [Scheduler](#)²⁴³ until further notice. Whether the resulting system is

²³⁸Section [6.3.3.8](#)

²³⁹Section [6.3.3.8](#)

²⁴⁰Section ??

²⁴¹Section ??

²⁴²Section [6.3.1.3](#)

²⁴³Section [6.3.3.12](#)

feasible or not, the addition is completed. When the object is already included in the feasibility set, do nothing.

Parameters

schedulable A reference to the given instance of [Schedulable](#)²⁴⁴

Throws

IllegalArgumentException 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. False, when not.

Deprecated as of RTSJ 2.0 The framework for feasibility analysis is inadequate

removeFromFeasibility(Schedulable)

Signature

protected boolean

removeFromFeasibility(javax.realtime.Schedulable<?> schedulable)

Description

Inform this scheduler and cooperating facilities that the resource demands of the given instance of [Schedulable](#)²⁴⁵ should no longer be considered in the feasibility analysis of the associated [Scheduler](#)²⁴⁶. Whether the resulting system is feasible or not, the removal is completed.

Parameters

schedulable A reference to the given instance of [Schedulable](#)²⁴⁷

Throws

IllegalArgumentException 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

²⁴⁴Section [6.3.1.3](#)

²⁴⁵Section [6.3.1.3](#)

²⁴⁶Section [6.3.3.12](#)

²⁴⁷Section [6.3.1.3](#)

fireSchedulable(Schedulable)

Signature

```
public void
fireSchedulable(javax.realtime.Schedulable<?> schedulable)
```

Description

Trigger the execution of a schedulable (like an [AsyncEventHandler²⁴⁸](#)).

Parameters

schedulable schedulable The schedulable to make active. When null, nothing happens.

Throws

UnsupportedOperationException Thrown in all cases by the PriorityScheduler

Deprecated RTSJ 2.0

A.2.3.22 ProcessingGroupParameters

Inheritance

```
java.lang.Object
  javax.realtime.ProcessingGroupParameters
```

Interfaces

```
Cloneable
Serializable
```

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. The motivation for this class is to allow the execution demands of one or more aperiodic schedulables to be bound. However, periodic or sporadic schedulables can also be associated with a processing group.

Processing groups have an associated affinity set that must contain only a single processor. The default affinity set is given by `Affinity.getGroupDefaultAffinity()`.

For all schedulables with a reference to an instance of `ProcessingGroupParameters` `p` no more than `p.cost` will be allocated to the execution of these schedulables on the processor associated with its processing group in each interval of time given by `p.period` after the time indicated by `p.start`. No execution of the

²⁴⁸Section [8.3.3.5](#)

schedulables will be allowed on any processor other than this processor. When there is no intersection between the a schedulable objects affinity set and its processing group's affinity set, then the schedulable execution is constrained by the default processing group's 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 (budget) parameter, (b) one of its associated schedulables is executable and is the most eligible of the executable schedulables. When the server is logically executable, the associated schedulable is executed. When the cost has been consumed, any overrunHandler is released, and the server is not eligible for logical execution until its next period is due. At this point, its allocated cost (budget) is replenished. When the server is logically executing when its deadline expires, any associated missHandler is released. The deadline and cost parameters of all the associated schedulable objects have the same impact as they would if the objects were not bound to a processing group.

Processing group parameters use [HighResolutionTime](#)²⁴⁹ values for cost, deadline, period and start time. Since those times are expressed as a [HighResolutionTime](#)²⁵⁰, 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](#)²⁵¹ parameter value. The value of each time object should be treated as if it were copied at the time it is passed to the parameter object, but the object reference must also be retained. Only changes to a ProcessingGroupParameters object caused by methods on that object are immediately visible to the scheduler. For instance, invoking setPeriod() on a ProcessingGroupParameters object will make the change, then notify that the scheduler that the parameter object has changed. At that point the scheduler's view of the processing group parameters

²⁴⁹Section [9.3.1.2](#)

²⁵⁰Section [9.3.1.2](#)

²⁵¹Section [9.3.1.2](#)

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.

Table A.1: `ProcessingGroupParameter` Default Values

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

Caution: This class is explicitly unsafe in multithreaded situations when it is being changed. No synchronization is done. It is assumed that users of this class who are mutating instances will be doing their own synchronization at a higher level.

Caution: The cost parameter time should be considered to be measured against the target platform.

Deprecated as of RTSJ 2.0; replaced by `ProcessingGroup`²⁵².

A.2.3.22.1 Constructors

²⁵²Section 6.3.3.9

ProcessingGroupParameters(HighResolutionTime, RelativeTime, RelativeTime, AsyncEventHandler, AsyncEventHandler)

Signature

```
public
ProcessingGroupParameters(javax.realtime.HighResolutionTime<?> start,
                           RelativeTime period,
                           RelativeTime cost,
                           RelativeTime deadline,
                           AsyncEventHandler overrunHandler,
                           AsyncEventHandler missHandler)
throws IllegalArgumentException,
      IllegalAssignmentError
```

Description

Create 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 `misHandler` cannot be stored in this.

A.2.3.22.2 Methods

clone

Signature

```
public java.lang.Object  
clone()  
throws CloneNotSupportedException
```

Description

Create 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.)

Throws

`CloneNotSupportedException` never

Returns

the clone of this

Available since RTSJ 1.0.1

getCost

Signature

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

Description

Gets the value of cost.

Returns

a reference to the value of cost.

getCostOverrunHandler

Signature

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

Description

Gets the cost overrun handler.

Returns

A reference to an instance of [AsyncEventHandler](#)²⁵³ that is cost overrun handler of this.

getDeadline

Signature

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

Description

Gets the value of deadline.

Returns

A reference to an instance of [RelativeTime](#)²⁵⁴ that is the deadline of this.

²⁵³Section [8.3.3.5](#)

²⁵⁴Section [9.3.1.3](#)

getDeadlineMissHandler

Signature

```
public javax.realtime.AsyncEventHandler  
getDeadlineMissHandler()
```

Description

Gets the deadline miss handler.

Returns

A reference to an instance of [AsyncEventHandler](#)²⁵⁵ 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](#)²⁵⁶ 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

²⁵⁵Section [8.3.3.5](#)

²⁵⁶Section [9.3.1.3](#)

A reference to an instance of [HighResolutionTime](#)²⁵⁷ 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.

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

```
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.

²⁵⁷Section [9.3.1.2](#)

setDeadline(RelativeTime)

Signature

```
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.

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

```
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*

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

A.2.3.23 RationalTime

Inheritance

```
java.lang.Object  
  javax.realtime.HighResolutionTime
```

javax.realtime.RelativeTime
javax.realtime.RationalTime

Description

An object that represents a time interval milliseconds/10³ + nanoseconds/10⁹ seconds long that is divided into subintervals by some frequency. This is generally used in periodic events, threads, and feasibility analysis to specify periods where there is a basic period that must be adhered to strictly (the interval), but within that interval the periodic events are supposed to happen frequency times, as uniformly spaced as possible, but clock and scheduling jitter is moderately acceptable.

Caution: This class is explicitly unsafe for multithreading when being changed. Code that mutates instances of this class should synchronize at a higher level. **Deprecated** as of RTSJ 1.0.1

A.2.3.23.1 Constructors

RationalTime(int, long, int)

Signature

```
public  
RationalTime(int frequency,  
              long millis,  
              int nanos)
```

Description

Construct an instance of RationalTime. All arguments must be greater than or equal to zero.

Parameters

frequency The frequency value.
millis The milliseconds value.
nanos The nanoseconds value.

Throws

IllegalArgumentException When any of the argument values are less than zero, or when frequency is equal to zero.

RationalTime(int, RelativeTime)

Signature

```
public  
RationalTime(int frequency,  
              RelativeTime interval)
```

Description

Construct an instance of RationalTime from the given [RelativeTime](#)²⁵⁸.

Parameters

frequency The frequency value.

interval The given instance of [RelativeTime](#)²⁵⁹.

Throws

IllegalArgumentException When either of the argument values are less than zero, or when frequency is equal to zero.

RationalTime(int)

Signature

```
public  
RationalTime(int frequency)
```

Description

Construct an instance of RationalTime. Equivalent to new RationalTime(1000, 0, frequency)—essentially a cycles-per-second value.

Throws

IllegalArgumentException when frequency is less than or equal to zero.

A.2.3.23.2 Methods

²⁵⁸Section [9.3.1.3](#)

²⁵⁹Section [9.3.1.3](#)

absolute(Clock, AbsoluteTime)

Signature

```
public javax.realtime.AbsoluteTime  
absolute(Clock clock,  
         AbsoluteTime destination)
```

Description

Convert time of this to an absolute time.

Parameters

clock The reference clock. When null, Clock.getRealTimeClock() is used.
destination A reference to the destination instance.

addInterarrivalTo(AbsoluteTime)

Signature

```
public void  
addInterarrivalTo(AbsoluteTime destination)
```

Description

Add the time of this to an [AbsoluteTime](#)²⁶⁰ It is almost the same dest.add(this, dest) except that it accounts for (i.e. divides by) the frequency.

Parameters

destination A reference to the destination instance.

getFrequency

Signature

```
public int  
getFrequency()
```

Description

Gets the value of frequency.

Returns

The value of frequency as an integer.

²⁶⁰Section [9.3.1.1](#)

getInterarrivalTime

Signature

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

Description

Gets the interarrival time. This time is $(\text{milliseconds}/10^3 + \text{nanoseconds}/10^9)/\text{frequency}$ rounded down to the nearest expressible value of the fields and their types of [RelativeTime](#)²⁶¹.

getInterarrivalTime(RelativeTime)

Signature

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

Description

Gets the interarrival time. This time is $(\text{milliseconds}/10^3 + \text{nanoseconds}/10^9)/\text{frequency}$ rounded down to the nearest expressible value of the fields and their types of [RelativeTime](#)²⁶².

Parameters

dest Result is stored in dest and returned, when null, a new object is returned.

set(long, int)

Signature

```
public javax.realtime.RationalTime  
set(long millis,  
    int nanos)
```

Description

Sets the indicated fields to the given values.

Parameters

millis The new value for the millisecond field.

²⁶¹Section [9.3.1.3](#)

²⁶²Section [9.3.1.3](#)

nanos The new value for the nanosecond field.

Returns

this

setFrequency(int)

Signature

```
public javax.realtime.RationalTime  
setFrequency(int frequency)
```

Description

Sets the value of the frequency field.

Parameters

frequency The new value for the frequency.

Throws

IllegalArgumentException when frequency is less than or equal to zero.

Returns

this

toString

Signature

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

Description

Create a printable string of the time given by this.

The string shall be a decimal representation of the frequency, milliseconds and nanosecond values; formatted as follows "(100, 2251 ms, 750000 ns)"

Returns

String object converted from the time given by this.

A.2.3.24 RawMemoryAccess

Inheritance

java.lang.Object
[javax.realtime.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](#)²⁶³ static byte variable in class [RealtimeSystem](#)²⁶⁴. 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](#)²⁶⁵ 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](#)²⁶⁶. 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](#)²⁶⁷. 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.

²⁶³Section [14.2.2.3.1](#)

²⁶⁴Section [14.2.2.3](#)

²⁶⁵Section [A.2.1.1](#)

²⁶⁶Section [15.2.2.11](#)

²⁶⁷Section [15.2.2.20](#)

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

The true values in the table are represented by properties of the following form.
javafx.realtime.atomicaccess_<access>_<type>_<alignment>_atomicity=true
for example:

```
javafx.realtime.atomicaccess_read_byte_0_memory=true
```

Table entries with a value of false may be explicitly represented, but since false is the default value, such properties are redundant.

All raw memory access is treated as volatile, and *serialized*. The run-time must be forced to re-read memory or write to memory on each call to a raw memory getxxx or putxxx method, and to complete the reads and writes in the order they appear in the program order.

Table A.2: Properties Array

Attribute	Values	Comment
Access type	read, write	
Data type	byte, short, int, long, float, double	
Alignment	0	aligned
	1 to one less than data type size	the first byte of the data is <i>alignment</i> bytes away from natural alignment.
Atomicity	processor	means access is atomic with respect to other tasks on processor.
	smp	means access is <i>processor</i> atomic, and atomic with respect to all processors in an SMP.
	memory	means that access is <i>smp</i> atomic, and atomic with respect to all access to the memory including DMA.

Deprecated as of RTSJ 2.0. Use [javax.realtime.device.RawMemoryFactory](#)²⁶⁸ to create the appropriate [javax.realtime.device.RawMemory](#)²⁶⁹ object.

A.2.3.24.1 Constructors

RawMemoryAccess(Object, long, long)

Signature

```
public
RawMemoryAccess(Object type,
                 long base,
                 long size)
```

Description

²⁶⁸Section [12.3.2.6](#)

²⁶⁹Section [12.3.1.16](#)

Construct an instance of `RawMemoryAccess` with the given parameters, and set the object to the mapped state. When the platform supports virtual memory, map the raw memory into virtual memory.

The run time environment is allowed to choose the virtual address where the raw memory area corresponding to this object will be mapped. The attributes of the mapping operation are controlled by the `VMFlags` and `VMAttributes` of the `PhysicalMemoryTypeFilter` objects that matched this object's type parameter. (See `PhysicalMemoryTypeFilter.getVMAttributes`²⁷⁰ and `PhysicalMemoryTypeFilter.getVMFlags`²⁷¹).

Parameters

`type` An instance of `Object` representing the type of memory required (e.g., *dma*, *shared*) - used to define the base address and control the mapping. When the required memory has more than one attribute, `type` may be an array of objects. When `type` is null or a reference to an array with no entries, any type of memory is acceptable. Note that `type` values are compared by reference (`==`), not by value (`equals`).

`base` The physical memory address of the region.

`size` The size of the area in bytes.

Throws

`SecurityException` when application doesn't have permissions to access physical memory, the specified range of addresses, or the given type of memory.

`OffsetOutOfBoundsException` when the address is invalid.

`SizeOutOfBoundsException` when the size is negative or extends into an invalid range of memory.

`UnsupportedPhysicalMemoryException` when the underlying hardware does not support the given type, or when no matching `PhysicalMemoryTypeFilter`²⁷² has been registered with the `PhysicalMemoryManager`²⁷³.

`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)

²⁷⁰Section [A.2.1.1.1](#)

²⁷¹Section [A.2.1.1.1](#)

²⁷²Section [A.2.1.1](#)

²⁷³Section [A.2.3.19](#)

Signature

```
public  
RawMemoryAccess(Object type,  
                  long size)
```

Description

Construct an instance of `RawMemoryAccess` with the given parameters, and set the object to the mapped state. When the platform supports virtual memory, map the raw memory into virtual memory.

The run time environment is allowed to choose the virtual address where the raw memory area corresponding to this object will be mapped. The attributes of the mapping operation are controlled by the `vmFlags` and `vmAttributes` of the `PhysicalMemoryTypeFilter` objects that matched this object's type parameter. (See `PhysicalMemoryTypeFilter.getVMAttributes`²⁷⁴ and `PhysicalMemoryTypeFilter.getVMFlags`²⁷⁵).

Parameters

`type` An instance of `Object` representing the type of memory required (e.g., *dma*, *shared*) - used to define the base address and control the mapping. When the required memory has more than one attribute, `type` may be an array of objects. When `type` is null or a reference to an array with no entries, any type of memory is acceptable. Note that type values are compared by reference (`==`), not by value (`equals`).

`size` The size of the area in bytes.

Throws

`SecurityException` when the application doesn't have permissions to access physical memory, the specified range of addresses, or the given type of memory.

`SizeOutOfBoundsException` when the size is negative or extends into an invalid range of memory.

`UnsupportedPhysicalMemoryException` when the underlying hardware does not support the given type, or when no matching `PhysicalMemoryTypeFilter`²⁷⁶ has been registered with the `PhysicalMemoryManager`²⁷⁷.

`MemoryTypeConflictException` when the specified base does not point to memory that matches the request type, or when type specifies incompatible memory attributes.

²⁷⁴Section A.2.1.1.1

²⁷⁵Section A.2.1.1.1

²⁷⁶Section A.2.1.1

²⁷⁷Section A.2.3.19

OutOfMemoryError when the requested type of memory exists, but there is not enough of it free to satisfy the request.

SecurityException when the application doesn't have permissions to access physical memory or the given range of memory.

A.2.3.24.2 Methods

getBytes(long)

Signature

```
public byte  
getBytes(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](#)²⁷⁸ somewhat overlaps this exception since it is when the offset is within the object but outside the mapped area.

SecurityException when this access is not permitted by the security manager.

Returns

The byte from raw memory.

²⁷⁸Section [15.2.2.20](#)

See [Section RawMemoryAccess.map\(long,long\)](#)

getBytes(long, byte, int, int)

Signature

```
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](#)²⁷⁹ somewhat overlaps this exception since it is when the offset is within the object but outside the mapped area.

[SizeOutOfBoundsException](#) when the object is not mapped, or when the byte falls in an invalid address range. This is checked at every entry in the array to allow for the possibility that the memory area could be unmapped or remapped. The

²⁷⁹Section [15.2.2.20](#)

bytes array could, therefore, be partially updated when the raw memory is unmapped or remapped mid-method.

`ArrayIndexOutOfBoundsException` when `low` is less than 0 or greater than `bytes.length - 1`, or when `low + number` is greater than or equal to `bytes.length`.

`SecurityException` when this access is not permitted by the security manager.

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](#)²⁸⁰ somewhat overlaps this exception since it is when the offset is within the object but outside the mapped area.

[SizeOutOfBoundsException](#) when the object is not mapped, or when the integer falls in an invalid address range.

`SecurityException` when this access is not permitted by the security manager.

Returns

²⁸⁰Section [15.2.2.20](#)

The integer from raw memory.

See [Section RawMemoryAccess.map\(long,long\)](#)

getInts(long, int, int, int)

Signature

```
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.

ints The array into which the integers read from the raw memory are placed.

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

number The number of integers to loaded.

Throws

[OffsetOutOfBoundsException](#) when the offset is negative or greater than the size of the raw memory area. The role of the [SizeOutOfBoundsException](#)²⁸¹ somewhat overlaps this exception since it is when the offset is within the object but outside the mapped area.

²⁸¹Section [15.2.2.20](#)

SizeOutOfBoundsException when the object is not mapped, or when the integers fall in an invalid address range. This is checked at every entry in the array to allow for the possibility that the memory area could be unmapped or remapped. The ints array could, therefore, be partially updated when the raw memory is unmapped or remapped mid-method.

ArrayIndexOutOfBoundsException when low is less than 0 or greater than bytes.length - 1, or when low + number is greater than or equal to bytes.length.

SecurityException when this access is not permitted by the security manager.

See Section [RawMemoryAccess.map\(long,long\)](#)

getLong(long)

Signature

```
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 double falls in an invalid address range.

SecurityException when this access is not permitted by the security manager.

Returns

The long from raw memory.

getLongs(long, long, int, int)*Signature*

```
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 long array passed starting at position low.

The loads are not required to be atomic even when they are located on natural boundaries.

Caching of the memory access is controlled by the memory type requested when the RawMemoryAccess instance was created. When the memory is not cached, this method guarantees serialized access (that is, the memory access at the memory occurs in the same order as in the program. Multiple writes to the same location may not be coalesced.)

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**²⁸² somewhat overlaps this exception since it is when the offset is within the object but outside the mapped area.

SizeOutOfBoundsException when the object is not mapped, or when a long falls in an invalid address range. This is checked at every entry in the array to allow for the possibility that the memory area could be unmapped or remapped. The longs array could, therefore, be partially updated when the raw memory is unmapped or remapped mid-method.

²⁸²Section [15.2.2.20](#)

ArrayIndexOutOfBoundsException when low is less than 0 or greater than bytes.length - 1, or when low + number is greater than or equal to bytes.length.

SecurityException when this access is not permitted by the security manager.

See Section [RawMemoryAccess.map\(long,long\)](#)

getMappedAddress

Signature

```
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). Same as the base address when virtual memory is not supported.

getShort(long)

Signature

```
public short  
getShort(long offset)
```

Description

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](#)²⁸³ somewhat overlaps this exception since it is when the offset is within the object but outside the mapped area.

[SizeOutOfBoundsException](#) when the object is not mapped, or when the short falls in an invalid address range.

[SecurityException](#) when this access is not permitted by the security manager.

Returns

The short loaded from raw memory.

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.)

²⁸³Section [15.2.2.20](#)

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](#)²⁸⁴ somewhat overlaps this exception since it is when the offset is within the object but outside the mapped area.

[SizeOutOfBoundsException](#) when the object is not mapped, or when a short falls in an invalid address range. This is checked at every entry in the array to allow for the possibility that the memory area could be unmapped or remapped. The shorts array could, therefore, be partially updated when the raw memory is unmapped or remapped mid-method.

[ArrayIndexOutOfBoundsException](#) when low is less than 0 or greater than bytes.length - 1, or when low + number is greater than or equal to bytes.length.

[SecurityException](#) when this access is not permitted by the security manager.

See [Section RawMemoryAccess.map\(long,long\)](#)

map*Signature*

```
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 `VMAttributes` of the `PhysicalMemoryTypeFilter` objects that matched this object's type parameter.

(See [PhysicalMemoryTypeFilter.getVMAttributes](#)²⁸⁵ and [PhysicalMemoryTypeFilter](#).)

²⁸⁴Section [15.2.2.20](#)

²⁸⁵Section [A.2.1.1.1](#)

`getVMFlags`²⁸⁶.

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 `vmAttributes` of the `PhysicalMemoryTypeFilter` objects that matched this object's type parameter. (See `PhysicalMemoryTypeFilter.getVMAttributes`²⁸⁷ and `PhysicalMemoryTypeFilter.getVMFlags`²⁸⁸).

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.

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.

²⁸⁶Section [A.2.1.1.1](#)

²⁸⁷Section [A.2.1.1.1](#)

²⁸⁸Section [A.2.1.1.1](#)

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 `vmAttributes` of the `PhysicalMemoryTypeFilter` objects that matched this object's type parameter. (See [PhysicalMemoryTypeFilter.getVMAttributes](#)²⁸⁹ and [PhysicalMemoryTypeFilter.getVMFlags](#)²⁹⁰).

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](#)²⁹¹. 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

`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.

²⁸⁹Section [A.2.1.1.1](#)

²⁹⁰Section [A.2.1.1.1](#)

²⁹¹Section [15.2.2.20](#)

setByte(long, byte)

Signature

```
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](#)²⁹² somewhat overlaps this exception since it is when the offset is within the object but outside the mapped area.

[SizeOutOfBoundsException](#) when the object is not mapped, or when the byte falls in an invalid address range.

[SecurityException](#) when this access is not permitted by the security manager.

See [Section RawMemoryAccess.map\(long,long\)](#)

setBytes(long, byte, int, int)

Signature

```
public void  
setBytes(long offset,  
        byte[] bytes,
```

²⁹²Section [15.2.2.20](#)


```
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](#)²⁹³ somewhat overlaps this exception since it is when the offset is within the object but outside the mapped area.

[SizeOutOfBoundsException](#) when the object is not mapped, or when the a short falls in an invalid address range. This is checked at every entry in the array to allow for the possibility that the memory area could be unmapped or remapped. The store of the array into memory could, therefore, be only partially complete when the raw memory is unmapped or remapped mid-method.

[ArrayIndexOutOfBoundsException](#) when low is less than 0 or greater than bytes.length - 1, or when low + number is greater than or equal to bytes.length.

[SecurityException](#) when this access is not permitted by the security manager.

See [Section RawMemoryAccess.map\(long,long\)](#)

²⁹³Section [15.2.2.20](#)

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

[OffsetOutOfBoundsException](#) when the offset is negative or greater than the size of the raw memory area. The role of the [SizeOutOfBoundsException](#)²⁹⁴ somewhat overlaps this exception since it is when the offset is within the object but outside the mapped area.

[SizeOutOfBoundsException](#) when the object is not mapped, or when the integer falls in an invalid address range.

[SecurityException](#) when this access is not permitted by the security manager.

See Section [RawMemoryAccess.map\(long,long\)](#)

setInts(long, int, int, int)*Signature*

²⁹⁴Section [15.2.2.20](#)

```
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**²⁹⁵ somewhat overlaps this exception since it is when the offset is within the object but outside the mapped area.

SizeOutOfBoundsException when the object is not mapped, or when an int falls in an invalid address range. This is checked at every entry in the array to allow for the possibility that the memory area could be unmapped or remapped. The store of the array into memory could, therefore, be only partially complete when the raw memory is unmapped or remapped mid-method.

ArrayIndexOutOfBoundsException when low is less than 0 or greater than bytes.length - 1, or when low + number is greater than or equal to bytes.length.

²⁹⁵Section [15.2.2.20](#)

SecurityException 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](#)²⁹⁶ somewhat overlaps this exception since it is when the offset is within the object but outside the mapped area.

[SizeOutOfBoundsException](#) when the object is not mapped, or when the long falls in an invalid address range.

SecurityException when this access is not permitted by the security manager.

See Section [RawMemoryAccess.map\(long,long\)](#)

²⁹⁶Section [15.2.2.20](#)

setLongs(long, long, int, int)

Signature

```
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](#)²⁹⁷ somewhat overlaps this exception since it is when the offset is within the object but outside the mapped area.

[SizeOutOfBoundsException](#) when the object is not mapped, or when the a short falls in an invalid address range. This is checked at every entry in the array to allow for the possibility that the memory area could be unmapped or remapped.

²⁹⁷Section [15.2.2.20](#)

The store of the array into memory could, therefore, be only partially complete when the raw memory is unmapped or remapped mid-method.

`ArrayIndexOutOfBoundsException` when `low` is less than 0 or greater than `bytes.length - 1`, or when `low + number` is greater than or equal to `bytes.length`.

`SecurityException` when this access is not permitted by the security manager.

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](#)²⁹⁸ somewhat overlaps this exception since it is when the offset is within the object but outside the mapped area.

[SizeOutOfBoundsException](#) when the object is not mapped, or when the short falls in an invalid address range.

`SecurityException` when this access is not permitted by the security manager.

²⁹⁸Section [15.2.2.20](#)

See Section [RawMemoryAccess.map\(long,long\)](#)

setShorts(long, short, int, int)

Signature

```
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](#)²⁹⁹ somewhat overlaps this exception since it is when the offset is within the object but outside the mapped area.

[SizeOutOfBoundsException](#) when the object is not mapped, or when the a short falls in an invalid address range. This is checked at every entry in the array to

²⁹⁹Section [15.2.2.20](#)

allow for the possibility that the memory area could be unmapped or remapped. The store of the array into memory could, therefore, be only partially complete when the raw memory is unmapped or remapped mid-method.

`ArrayIndexOutOfBoundsException` when `low` is less than 0 or greater than `bytes.length - 1`, or when `low + number` is greater than or equal to `bytes.length`.

`SecurityException` when this access is not permitted by the security manager.

See Section [RawMemoryAccess.map\(long,long\)](#)

unmap

Signature

```
public void
unmap()
```

Description

Unmap the physical memory range from virtual memory. This changes the raw memory from the mapped state to the unmapped state. When the platform supports virtual memory, this operation frees the virtual addresses used for the raw memory region.

When the object is already in the unmapped state, this method has no effect.

While a raw memory object is unmapped all attempts to set or get values in the raw memory will throw [SizeOutOfBoundsException](#)³⁰⁰.

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.

A.2.3.25 RawMemoryFloatAccess

Inheritance

```
java.lang.Object
  javax.realtime.RawMemoryAccess
    javax.realtime.RawMemoryFloatAccess
```

Description

³⁰⁰Section [15.2.2.20](#)

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](#)³⁰¹ 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](#)³⁰² 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 a optional system properties that identify a platform's level of support for atomic raw put and get. (See [RawMemoryAccess](#)³⁰³.) The properties represent a four-dimensional sparse array with boolean values whether that

³⁰¹Section [A.2.3.24](#)

³⁰²Section [A.2.1.1](#)

³⁰³Section [A.2.3.24](#)

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](#)³⁰⁴. 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](#)³⁰⁵. 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](#)³⁰⁶.

A.2.3.25.1 Constructors

RawMemoryFloatAccess(Object, long, long)

Signature

```
public
RawMemoryFloatAccess(Object type,
                      long base,
                      long size)
```

Description

Construct an instance of `RawMemoryFloatAccess` with the given parameters, and set the object to the mapped state. When the platform supports virtual memory, map the raw memory into virtual memory.

The run time environment is allowed to choose the virtual address where the raw memory area corresponding to this object will be mapped. The attributes of the mapping operation are controlled by the `VMFlags` and `VMAttributes` of the `PhysicalMemoryTypeFilter` objects that matched this object's type parameter. (See [PhysicalMemoryTypeFilter.getVMAttributes](#)³⁰⁷ and [PhysicalMemoryTypeFilter.getVMFlags](#)³⁰⁸).

³⁰⁴Section [15.2.2.11](#)

³⁰⁵Section [15.2.2.20](#)

³⁰⁶Section [12.3.1.16](#)

³⁰⁷Section [A.2.1.1.1](#)

³⁰⁸Section [A.2.1.1.1](#)

Parameters

type An instance of `Object` representing the type of memory required (e.g., *dma*, *shared*) - used to define the base address and control the mapping. When the required memory has more than one attribute, **type** may be an array of objects. When **type** is null or a reference to an array with no entries, any type of memory is acceptable. Note that **type** values are compared by reference (`==`), not by value (`equals`).

base The physical memory address of the region.

size The size of the area in bytes.

Throws

`SecurityException` when 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`³⁰⁹ has been registered with the `PhysicalMemoryManager`³¹⁰.

`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

RawMemoryFloatAccess(Object, long)

Signature

```
public  
RawMemoryFloatAccess(Object type,  
                      long size)
```

Description

Construct an instance of `RawMemoryFloatAccess` with the given parameters, and set the object to the mapped state. When the platform supports virtual memory, map the raw memory into virtual memory.

The run time environment is allowed to choose the virtual address where the raw memory area corresponding to this object will be mapped. The attributes

³⁰⁹Section [A.2.1.1](#)

³¹⁰Section [A.2.3.19](#)

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`³¹¹ and `PhysicalMemoryTypeFilter.getVMFlags`³¹².

Parameters

`type` An instance of `Object` representing the type of memory required (e.g., *dma*, *shared*) - used to define the base address and control the mapping. When the required memory has more than one attribute, `type` may be an array of objects. When `type` is null or a reference to an array with no entries, any type of memory is acceptable. Note that type values are compared by reference (`==`), not by value (equals).

`size` The size of the area in bytes.

Throws

`SecurityException` when the application doesn't have permissions to access physical memory, the specified range of addresses, or the given type of memory.

`SizeOutOfBoundsException` when the size is negative or extends into an invalid range of memory.

`UnsupportedPhysicalMemoryException` when the underlying hardware does not support the given type, or when no matching `PhysicalMemoryTypeFilter`³¹³ has been registered with the `PhysicalMemoryManager`³¹⁴.

`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.

A.2.3.25.2 Methods

getDouble(long)

Signature

public double

³¹¹Section A.2.1.1.1

³¹²Section A.2.1.1.1

³¹³Section A.2.1.1

³¹⁴Section A.2.3.19

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 long.

Throws

[OffsetOutOfBoundsException](#) when the offset is invalid.

[SizeOutOfBoundsException](#) when the object is not mapped, or when the double falls in an invalid address range.

[SecurityException](#) when this access is not permitted by the security manager.

Returns

The double from raw memory.

getDoubles(long, double, int, int)

Signature

```
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 assign them to the double array passed starting at position low.

The loads are not required to be atomic even when they are located on natural boundaries.

Caching of the memory access is controlled by the memory type requested when the RawMemoryAccess instance was created. When the memory is not

cached, this method guarantees serialized access (that is, the memory access at the memory occurs in the same order as in the program. Multiple writes to the same location may not be coalesced.)

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\(long,long\)](#)³¹⁵).

[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.

[SecurityException](#) 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

³¹⁵Section [A.2.3.24.2](#)

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)**³¹⁶).

SizeOutOfBoundsException when the object is not mapped, or when the float falls in an invalid address range.

SecurityException when this access is not permitted by the security manager.

Returns

The float from raw memory.

getFloats(long, float, int, int)

Signature

```
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 int array passed starting at position low.

When the floats are aligned on natural boundaries each float is loaded from memory in a single atomic operation. Groups of floats may be loaded together, but this is unspecified.

When the floats are not aligned on natural boundaries they may not be loaded atomically and the number and order of load operations is unspecified.

Caching of the memory access is controlled by the memory type requested when the **RawMemoryAccess** instance was created. When the memory is not cached, this method guarantees serialized access (that is, the memory access at

³¹⁶Section [A.2.3.24.2](#)

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.
- floats The array into which the floats loaded from the raw memory are placed.
- low The offset which is the starting point in the given array for the loaded items to be placed.
- number The number of floats to loaded.

Throws

OffsetOutOfBoundsException when the offset is negative or greater than the size of the raw memory area. The role of the **SizeOutOfBoundsException** somewhat overlaps this exception since it is when the offset is within the object but outside the mapped area. (See **RawMemoryAccess.map(long,long)**³¹⁷).

SizeOutOfBoundsException when the object is not mapped, or when a float falls in an invalid address range. This is checked at every entry in the array to allow for the possibility that the memory area could be unmapped or remapped. The floats array could, therefore, be partially updated when the raw memory is unmapped or remapped mid-method.

ArrayIndexOutOfBoundsException when low is less than 0 or greater than bytes.length - 1, or when low + number is greater than or equal to bytes.length.

SecurityException when this access is not permitted by the security manager.

setDouble(long, double)

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

³¹⁷Section [A.2.3.24.2](#)

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 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\(long,long\)](#)³¹⁸).

[SizeOutOfBoundsException](#) when the object is not mapped, or when the double falls in an invalid address range.

[SecurityException](#) when this access is not permitted by the security manager.

setDoubles(long, double, int, int)

Signature

```
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 double array passed starting at position low. Even when they are aligned, the double values may not be updated atomically. It is unspecified how many load and store operations will be used or in what order.

Caching of the memory access is controlled by the memory type requested when the [RawMemoryAccess](#) instance was created. When the memory is not cached, this method guarantees serialized access (that is, the memory access at the memory occurs in the same order as in the program. Multiple writes to the same location may not be coalesced.)

Parameters

offset The offset in bytes from the beginning of the raw memory area at which to start writing.

³¹⁸Section [A.2.3.24.2](#)

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)**³¹⁹).

SizeOutOfBoundsException when the object is not mapped, or when the a short falls in an invalid address range. This is checked at every entry in the array to allow for the possibility that the memory area could be unmapped or remapped. The doubles array could, therefore, be partially updated when the raw memory is unmapped or remapped mid-method.

ArrayIndexOutOfBoundsException when low is less than 0 or greater than bytes.length - 1, or when low + number is greater than or equal to bytes.length.

SecurityException when this access is not permitted by the security manager.

setFloat(long, float)

Signature

```
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 integer.

³¹⁹Section [A.2.3.24.2](#)

value The float to write.

Throws

[OffsetOutOfBoundsException](#) when the offset is negative or greater than the size of the raw memory area. The role of the [SizeOutOfBoundsException](#) somewhat overlaps this exception since it is when the offset is within the object but outside the mapped area. (See [RawMemoryAccess.map\(long,long\)](#)³²⁰).

[SizeOutOfBoundsException](#) when the object is not mapped, or when the float falls in an invalid address range.

[SecurityException](#) when this access is not permitted by the security manager.

setFloats(long, float, int, int)

Signature

```
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

³²⁰Section [A.2.3.24.2](#)

OffsetOutOfBoundsException when the offset is negative or greater than the size of the raw memory area. The role of the **SizeOutOfBoundsException** somewhat overlaps this exception since it is when the offset is within the object but outside the mapped area. (See **RawMemoryAccess.map(long, long)**³²¹).

SizeOutOfBoundsException when the object is not mapped, or when the float falls in an invalid address range. This is checked at every entry in the array to allow for the possibility that the memory area could be unmapped or remapped. The store of the array into memory could, therefore, be only partially complete when the raw memory is unmapped or remapped mid-method.

ArrayIndexOutOfBoundsException when low is less than 0 or greater than bytes.length - 1, or when low + number is greater than or equal to bytes.length.

SecurityException when this access is not permitted by the security manager.

A.2.3.26 *RealtimeSystem*

The following elements of **RealtimeSystem** are deprecated. The required elements are documented in Section 14.2.2.3 above.

A.2.3.26.1 Fields

A.2.3.27 *RealtimeThread*

The following elements of **RealtimeThread** are deprecated. The required elements are documented in Section 5.3.2.2 above.

A.2.3.27.1 Constructors

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

³²¹Section A.2.3.24.2

Signature

```
public
RealtimeThread(SchedulingParameters scheduling,
                javax.realtime.ReleaseParameters<?> release,
                MemoryParameters memory,
                MemoryArea area,
                ProcessingGroupParameters group,
                Runnable logic)
```

Description

Create 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

A.2.3.27.2 Methods

sleep(Clock, HighResolutionTime)

Signature

```
public static void
sleep(Clock clock,
      javax.realtime.HighResolutionTime<?> time)
throws InterruptedException,
      ClassCastException,
      UnsupportedOperationException,
      IllegalArgumentException
```

Description

A sleep method that is controlled by a generalized clock. Since the time is expressed as a [HighResolutionTime](#)³²², 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

³²²Section [9.3.1.2](#)

is the given [Clock](#)³²³ 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.

It is permissible to call sleep when control is in an [AsyncEventHandler](#)³²⁴. The method cause 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](#)³²⁵ used as the base. When clock is null the realtime clock (see [Clock.getRealtimeClock](#)³²⁶) 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\(\)](#)³²⁷ or [AsynchronouslyInterruptedException](#)³²⁸ during the time between calling this method and returning from it.
`ClassCastException` when the current execution context is that of a Java thread.
`UnsupportedOperationException` when the sleep operation is not supported by clock.
`IllegalArgumentException` when time is null, or when time is a relative time less than zero.

Deprecated in RTSJ 2.0

waitForNextPeriod

Signature

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

³²³Section [10.3.2.1](#)

³²⁴Section [8.3.3.5](#)

³²⁵Section [10.3.2.1](#)

³²⁶Section [10.3.2.1.2](#)

³²⁷Section [5.3.2.2.2](#)

³²⁸Section [8.3.2.1.2](#)

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](#)³²⁹ type of [PeriodicParameters](#)³³⁰ 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](#)³³¹.)

Throws

`IllegalThreadStateException` when this does not have a reference to a [ReleaseParameters](#)³³² type of [PeriodicParameters](#)³³³.

`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\(\)](#)³³⁴

waitForNextPeriodInterruptible

Signature

```
public static boolean
waitForNextPeriodInterruptible()
throws InterruptedException,
    ClassCastException,
    IllegalThreadStateException
```

Description

³²⁹Section [6.3.3.10](#)

³³⁰Section [6.3.3.6](#)

³³¹Section [6.3.3.8](#)

³³²Section [6.3.3.10](#)

³³³Section [6.3.3.6](#)

³³⁴Section [5.3.2.2.2](#)

The `waitForNextPeriodInterruptible()` method is a duplicate of `waitForNextPeriod()`³³⁵ except that `waitForNextPeriodInterruptible` is able to throw `InterruptedException`.

Used by threads that have a reference to a `ReleaseParameters`³³⁶ type of `PeriodicParameters`³³⁷ 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`³³⁸)

Throws

`InterruptedException` when the thread is interrupted by `interrupt()`³³⁹ or `AsynchronouslyInterruptedException`³⁴⁰ 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`³⁴¹ type of `PeriodicParameters`³⁴².

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

Deprecated RTSJ 2.0 Replaced by `waitForNextRelease`³⁴³

addIfFeasible

Signature

³³⁵Section ??

³³⁶Section 6.3.3.10

³³⁷Section 6.3.3.6

³³⁸Section 6.3.3.8

³³⁹Section 5.3.2.2.2

³⁴⁰Section 8.3.2.1.2

³⁴¹Section 6.3.3.10

³⁴²Section 6.3.3.6

³⁴³Section 5.3.2.2.2


```
public boolean  
addIfFeasible()
```

Description

This method first performs a feasibility analysis with this added to the system. When the resulting system is feasible, inform the scheduler and cooperating facilities that this instance of [Schedulable](#)³⁴⁴ should be considered in feasibility analysis until further notified. When the analysis showed 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, do 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 anlaysis is inadequate

addToFeasibility

Signature

```
public boolean  
addToFeasibility()
```

Description

Inform the scheduler and cooperating facilities that this instance of [Schedulable](#)³⁴⁵ should be considered in feasibility analysis until further notified.

When the object is already included in the feasibility set, do nothing.

Returns

True, when the resulting system is feasible. False, when not.

Deprecated as of RTSJ 2.0 The framework for feasibility anlaysis is inadequate

³⁴⁴Section [6.3.1.3](#)

³⁴⁵Section [6.3.1.3](#)

deschedulePeriodic

Signature

```
public void  
deschedulePeriodic()
```

Description

When the [ReleaseParameters](#)³⁴⁶ object associated with this RealtimeThread is an instance of [PeriodicParameters](#)³⁴⁷, perform any *deschedulePeriodic* actions specified by this thread's scheduler. When the type of the associated instance of [ReleaseParameters](#)³⁴⁸ is not [PeriodicParameters](#)³⁴⁹ nothing happens.

Deprecated since RTSJ 2.0

getProcessingGroupParameters

Signature

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

Description

Gets a reference to the [ProcessingGroupParameters](#)³⁵⁰ object for this schedulable.

Returns

A reference to the current [ProcessingGroupParameters](#)³⁵¹ object.

Deprecated since RTSJ 2.0

removeFromFeasibility

Signature

```
public boolean  
removeFromFeasibility()
```

³⁴⁶Section [6.3.3.10](#)

³⁴⁷Section [6.3.3.6](#)

³⁴⁸Section [6.3.3.10](#)

³⁴⁹Section [6.3.3.6](#)

³⁵⁰Section [A.2.3.22](#)

³⁵¹Section [A.2.3.22](#)

Description

Inform the scheduler and cooperating facilities that this instance of [Schedulable](#)³⁵² 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

Begin unblocking `RealtimeThread.waitForNextPeriod`³⁵³ for a periodic thread. When deadline miss detection is disabled, enable it. Typically used when a periodic schedulable is in a deadline miss condition. The details of the interaction of this method with `deschedulePeriodic`³⁵⁴ and `waitForNextPeriod`³⁵⁵ are dictated by this thread's scheduler.

When this `RealtimeThread` does not have a type of [PeriodicParameters](#)³⁵⁶ as its [ReleaseParameters](#)³⁵⁷ nothing happens.

Deprecated since RTSJ 2.0

setIfFeasible(ReleaseParameters, MemoryParameters)*Signature*

```
public boolean
setIfFeasible(javafx.realtime.ReleaseParameters<?> release,
              MemoryParameters memory)
```

³⁵²Section [6.3.1.3](#)

³⁵³Section ??

³⁵⁴Section ??

³⁵⁵Section ??

³⁵⁶Section [6.3.3.6](#)

³⁵⁷Section [6.3.3.10](#)

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 release The proposed release parameters. When null, the default value is governed by the associated scheduler (a new object is created when the default value is not null). (See [PriorityScheduler³⁵⁸](#).)

memory memory The proposed memory parameters. When null, the default value is governed by the associated scheduler (a new object is created when the default value is not null). (See [PriorityScheduler³⁵⁹](#).)

Throws

`IllegalArgumentException` `IllegalArgumentException` Thrown 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 proposed parameter objects are located in heap memory.

`IllegalAssignmentError` `IllegalAssignmentError` when this cannot hold references to the proposed parameter objects, or the parameter objects cannot hold a reference to this.

`IllegalThreadStateException` `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

³⁵⁸Section [6.3.3.8](#)

³⁵⁹Section [6.3.3.8](#)

setIfFeasible(ReleaseParameters, MemoryParameters, ProcessingGroupParameters)

Signature

```
public boolean  
setIfFeasible(javax.realtime.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 release The proposed release parameters. When null, the default value is governed by the associated scheduler (a new object is created when the default value is not null). (See [PriorityScheduler³⁶⁰](#).)

memory memory The proposed memory parameters. When null, the default value is governed by the associated scheduler (a new object is created when the default value is not null). (See [PriorityScheduler³⁶¹](#).)

group group The proposed processing group parameters. When null, the default value is governed by the associated scheduler (a new object is created when the default value is not null). (See [PriorityScheduler³⁶²](#).)

Throws

IllegalArgumentException IllegalArgumentException Thrown 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 proposed parameter objects are located in heap memory.

³⁶⁰Section [6.3.3.8](#)

³⁶¹Section [6.3.3.8](#)

³⁶²Section [6.3.3.8](#)

IllegalAssignmentError IllegalAssignmentError when this cannot hold references to the proposed parameter objects, or the parameter objects cannot hold a reference to this.

IllegalThreadStateException 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, ProcessingGroupParameters)

Signature

```
public boolean
setIfFeasible(javax.realtime.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 release The proposed release parameters. 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](#)³⁶³.)

³⁶³Section 6.3.3.8

group group The proposed processing group parameters. 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](#)³⁶⁴.)

Throws

`IllegalArgumentException` `IllegalArgumentException` Thrown 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 proposed parameter objects are located in heap memory.

`IllegalAssignmentError` `IllegalAssignmentError` when this cannot hold references to the proposed parameter objects, or the parameter objects cannot hold a reference to this.

`IllegalThreadStateException` `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,
              javax.realtime.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.

³⁶⁴Section [6.3.3.8](#)

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 scheduling The proposed scheduling parameters. 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](#)³⁶⁵.)

release release The proposed release parameters. 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](#)³⁶⁶.)

memory memory The proposed memory parameters. 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](#)³⁶⁷.)

Throws

`IllegalArgumentException` `IllegalArgumentException` Thrown 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 proposed parameter objects are located in heap memory.

`IllegalAssignmentError` `IllegalAssignmentError` when this cannot hold references to the proposed parameter objects, or the parameter objects cannot hold a reference to this.

`IllegalThreadStateException` `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

³⁶⁵Section [6.3.3.8](#)

³⁶⁶Section [6.3.3.8](#)

³⁶⁷Section [6.3.3.8](#)


```

public boolean
setIfFeasible(SchedulingParameters scheduling,
              javax.realtime.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 scheduling The proposed scheduling parameters. 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](#)³⁶⁸.)
- release release The proposed release parameters. 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](#)³⁶⁹.)
- memory memory The proposed memory parameters. 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](#)³⁷⁰.)
- group group The proposed processing group parameters. 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](#)³⁷¹.)

Throws

- IllegalArgumentException IllegalArgumentException Thrown 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 proposed parameter objects are located in heap memory.

³⁶⁸Section [6.3.3.8](#)

³⁶⁹Section [6.3.3.8](#)

³⁷⁰Section [6.3.3.8](#)

³⁷¹Section [6.3.3.8](#)

IllegalAssignmentError IllegalAssignmentError when this cannot hold references to the proposed parameter objects, or the parameter objects cannot hold a reference to this.

IllegalThreadStateException 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

```
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 memory The proposed memory parameters. 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](#)³⁷².)

Throws

IllegalArgumentException IllegalArgumentException Thrown when the parameter value is not compatible with the schedulable's scheduler. Also when this

³⁷²Section [6.3.3.8](#)

schedulable may not use the heap and the proposed parameter object is located in heap memory.

IllegalAssignmentError IllegalAssignmentError when this cannot hold a reference to the proposed parameter object, or the parameter object cannot hold a reference to this.

IllegalThreadStateException 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**³⁷³ 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 group A **ProcessingGroupParameters**³⁷⁴ 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**³⁷⁵.)

Throws

IllegalArgumentException IllegalArgumentException Thrown 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.

³⁷³Section [A.2.3.22](#)

³⁷⁴Section [A.2.3.22](#)

³⁷⁵Section [6.3.3.8](#)

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

IllegalThreadStateException 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 group The proposed processing group parameters. 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](#)³⁷⁶.)

Throws

IllegalArgumentException IllegalArgumentException Thrown when the parameter value is not compatible with the schedulable's scheduler. Also when this schedulable may not use the heap and the proposed parameter object is located in heap memory.

³⁷⁶Section [6.3.3.8](#)

IllegalAssignmentError IllegalAssignmentError when this cannot hold a reference to the proposed parameter object, or the parameter object cannot hold a reference to this.

IllegalThreadStateException 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

```
public boolean
setReleaseParametersIfFeasible(javax.realtime.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 **release** The proposed release parameters. 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](#)³⁷⁷.)

Throws

IllegalArgumentException IllegalArgumentException Thrown when the parameter value is not compatible with the schedulable's scheduler. Also when this

³⁷⁷Section 6.3.3.8

schedulable may not use the heap and the proposed parameter object is located in heap memory.

IllegalAssignmentError IllegalAssignmentError when this cannot hold a reference to the proposed parameter object, or the parameter object cannot hold a reference to this.

IllegalThreadStateException 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,
             javax.realtime.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 scheduler A reference to the scheduler that will manage the execution of this schedulable. Null is not a permissible value.

scheduling scheduling A reference to the [SchedulingParameters](#)³⁷⁸ 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](#)³⁷⁹.)

³⁷⁸Section [6.3.3.14](#)

³⁷⁹Section [6.3.3.8](#)

release release A reference to the [ReleaseParameters](#)³⁸⁰ 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](#)³⁸¹.)

memoryParameters memoryParameters A reference to the [MemoryParameters](#)³⁸² 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](#)³⁸³.)

group group A reference to the [ProcessingGroupParameters](#)³⁸⁴ which will be associated with this. When null, the default value is governed by scheduler (a new object is created). (See [PriorityScheduler](#)³⁸⁵.)

Throws

[IllegalArgumentException](#) [IllegalArgumentException](#) Thrown 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](#) [IllegalAssignmentError](#) when this object cannot hold references to all the parameter objects or the parameters cannot hold references to this.

[IllegalThreadStateException](#) [IllegalThreadStateException](#) when scheduler prohibits the changing of the scheduler or a parameter at this time due to the state of the schedulable.

[SecurityException](#) [SecurityException](#) 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

³⁸⁰Section [6.3.3.10](#)

³⁸¹Section [6.3.3.8](#)

³⁸²Section [11.3.2.4](#)

³⁸³Section [6.3.3.8](#)

³⁸⁴Section [A.2.3.22](#)

³⁸⁵Section [6.3.3.8](#)

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 scheduling The proposed scheduling parameters. 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](#)³⁸⁶.)

Throws

IllegalArgumentException IllegalArgumentException Thrown when the parameter value is not compatible with the schedulable's scheduler. Also when this schedulable may not use the heap and the proposed parameter object is located in heap memory.

IllegalAssignmentError IllegalAssignmentError when this cannot hold a reference to the proposed parameter object, or the parameter object cannot hold a reference to this.

IllegalThreadStateException 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

A.2.3.28 *RelativeTime*

The following elements of *RelativeTime* are deprecated. The required elements are documented in Section [9.3.1.3](#) above.

³⁸⁶Section [6.3.3.8](#)

A.2.3.28.1 Constructors

RelativeTime(long, int, Clock)*Signature*

```
public  
RelativeTime(long millis,  
              int nanos,  
              Clock clock)  
throws IllegalArgumentException
```

Description

Construct a RelativeTime object representing an interval based on the parameter millis plus the parameter nanos. The construction is subject to millis and nanos parameters normalization. When there is an overflow in the millisecond component when normalizing then an IllegalArgumentException will be thrown.

The clock association is 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

IllegalArgumentException when there is an overflow in the millisecond component when normalizing.

RelativeTime(RelativeTime, Clock)*Signature*

```
public  
RelativeTime(RelativeTime time,  
             Clock clock)  
throws IllegalArgumentException
```

Description

Make 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 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

IllegalArgumentException 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.

A.2.3.28.2 Methods

absolute(Clock)

Signature

```
public javax.realtime.AbsoluteTime  
absolute(Clock clock)  
throws ArithmeticException
```

Description

Convert the time of this to an absolute time, using the given instance of [Clock](#)³⁸⁷ to determine the current time. The calculation is the current time indicated by the given instance of [Clock](#)³⁸⁸ 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](#)³⁸⁹ used to convert the time of this into absolute time, and the new clock association for the result.

Throws

ArithmeticException when the result does not fit in the normalized format.

Returns

The 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
```

³⁸⁷Section [10.3.2.1](#)

³⁸⁸Section [10.3.2.1](#)

³⁸⁹Section [10.3.2.1](#)

Description

Convert the time of this to an absolute time, using the given instance of [Clock](#)³⁹⁰ 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](#)³⁹¹ 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. Otherwise, a new object is allocated for the result.

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)

Signature

```
public javax.realtime.RelativeTime  
    relative(Clock clock)
```

Description

Return 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.

³⁹⁰Section [10.3.2.1](#)

³⁹¹Section [10.3.2.1](#)

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

Return 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. Otherwise, a new object is allocated for the result.

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

addInterarrivalTo(AbsoluteTime)*Signature*

```
public void  
addInterarrivalTo(AbsoluteTime timeAndDestination)
```

Description

Add the interval of this to the given instance of [AbsoluteTime](#)³⁹².

Parameters

timeAndDestination A reference to the given instance of [AbsoluteTime](#)³⁹³ and the result.

Deprecated as of RTSJ 1.0.1

getInterarrivalTime

Signature

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

Description

Gets the interval defined by this. For an instance of [RationalTime](#)³⁹⁴ it is the interval divided by the frequency.

Returns

A reference to a new instance of [RelativeTime](#)³⁹⁵ with the same interval as this.

Deprecated as of RTSJ 1.0.1

getInterarrivalTime(RelativeTime)

Signature

```
public javax.realtime.RelativeTime  
getInterarrivalTime(RelativeTime destination)
```

Description

Gets the interval defined by this. For an instance of [RationalTime](#)³⁹⁶ it is the interval divided by the frequency.

Parameters

destination A reference to the new object holding the result.

Returns

³⁹²Section [9.3.1.1](#)

³⁹³Section [9.3.1.1](#)

³⁹⁴Section [A.2.3.23](#)

³⁹⁵Section [9.3.1.3](#)

³⁹⁶Section [A.2.3.23](#)

A reference to an object holding the result.

Deprecated as of RTSJ 1.0.1

A.2.3.29 *ReleaseParameters*

The following elements of ReleaseParameters are deprecated. The required elements are documented in Section 6.3.3.10 above.

A.2.3.29.1 **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](#)³⁹⁷ 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

³⁹⁷Section 9.3.1.3

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

`IllegalAssignmentError` when cost 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

A.2.3.30 Scheduler

The following elements of Scheduler are deprecated. The required elements are documented in Section 6.3.3.12 above.

A.2.3.30.1 Methods

addToFeasibility(Schedulable)

Signature

```
protected abstract boolean
addToFeasibility(javax.realtime.Schedulable<?> schedulable)
```

Description

Inform this scheduler and cooperating facilities that the resource demands of the given instance of `Schedulable`³⁹⁸ will be considered in the feasibility analysis of the associated `Scheduler`³⁹⁹ 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, do nothing.

Parameters

schedulable A reference to the given instance of `Schedulable`⁴⁰⁰

Throws

³⁹⁸Section 6.3.1.3

³⁹⁹Section 6.3.3.12

⁴⁰⁰Section 6.3.1.3

IllegalArgumentException when schedulable is null, or when schedulable is not associated with this; that is schedulable.getSchedular() != this.

Returns

True, when the system is feasible after the addition. 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(javax.realtime.Schedulable<?> schedulable,  
              javax.realtime.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](#)⁴⁰¹.)

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](#)⁴⁰².)

Throws

`IllegalArgumentException` 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`⁴⁰³ or `RealtimeThread.waitForNextPeriodInterruptible`⁴⁰⁴.

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(javax.realtime.Schedulable<?> schedulable,
              javax.realtime.ReleaseParameters<?> release,
              MemoryParameters memory,
              ProcessingGroupParameters group)
```

⁴⁰¹Section [6.3.3.8](#)

⁴⁰²Section [6.3.3.8](#)

⁴⁰³Section ??

⁴⁰⁴Section ??

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](#)⁴⁰⁵.)

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](#)⁴⁰⁶.)

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](#)⁴⁰⁷.)

Throws

IllegalArgumentException 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`⁴⁰⁸ or `RealtimeThread.waitForNextPeriodInterruptible`⁴⁰⁹.

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.

⁴⁰⁵Section [6.3.3.8](#)

⁴⁰⁶Section [6.3.3.8](#)

⁴⁰⁷Section [6.3.3.8](#)

⁴⁰⁸Section ??

⁴⁰⁹Section ??

Deprecated as of RTSJ 2.0 The framework for feasibility analysis is inadequate

setIfFeasible(Schedulable, SchedulingParameters, ReleaseParameters, MemoryParameters, ProcessingGroupParameters)

Signature

```
public abstract boolean
setIfFeasible(javax.realtime.Schedulable<?> schedulable,
              SchedulingParameters scheduling,
              javax.realtime.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](#)⁴¹⁰.)

`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](#)⁴¹¹.)

`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](#)⁴¹².)

`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

⁴¹⁰Section [6.3.3.8](#)

⁴¹¹Section [6.3.3.8](#)

⁴¹²Section [6.3.3.8](#)

null). (See [PriorityScheduler](#)⁴¹³.)

Throws

`IllegalArgumentException` 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`⁴¹⁴ or `RealtimeThread.waitForNextPeriodInterruptible`⁴¹⁵.

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(javax.realtime.Schedulable<?> schedulable)
```

Description

Inform this scheduler and cooperating facilities that the resource demands of the given instance of [Schedulable](#)⁴¹⁶ should no longer be considered in the feasibility analysis of the associated [Scheduler](#)⁴¹⁷. Whether the resulting system is feasible or not, the removal is completed.

Parameters

`schedulable` A reference to the given instance of [Schedulable](#)⁴¹⁸

Throws

`IllegalArgumentException` when `schedulable` is null.

⁴¹³Section 6.3.3.8

⁴¹⁴Section ??

⁴¹⁵Section ??

⁴¹⁶Section 6.3.1.3

⁴¹⁷Section 6.3.3.12

⁴¹⁸Section 6.3.1.3

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 abstract void
fireSchedulable(javax.realtime.Schedulable<?> schedulable)
```

Description

Trigger the execution of a schedulable (like an [AsyncEventHandler](#)⁴¹⁹).

Parameters

schedulable The schedulable to make active. When null, nothing happens.

Throws

UnsupportedOperationException when the scheduler cannot release schedulable for execution.

Deprecated RTSJ 2.0

A.2.3.31 ScopedMemory

Inheritance

```
java.lang.Object
  javax.realtime.MemoryArea
    javax.realtime.ScopedMemory
```

Description

Equivalent to and superseded by [javax.realtime.memory.ScopedMemory](#)⁴²⁰.

Deprecated in RTSJ 2.0; moved to package javax.realtime.memory

⁴¹⁹Section [8.3.3.5](#)

⁴²⁰Section [11.4.3.6](#)

A.2.3.31.1 Constructors

ScopedMemory(long, Runnable)

Signature

```
public  
    ScopedMemory(long size,  
                  Runnable logic)
```

Description

Create a new ScopedMemory area with the given parameters.

Deprecated since RTSJ 2.0 no longer visible

Parameters

size The size of the new ScopedMemory area in bytes.

logic The Runnable to execute when this ScopedMemory is entered. When logic is null, this constructor is equivalent to constructing the memory area without a logic value.

Throws

IllegalArgumentException when size is less than zero.

IllegalAssignmentError when storing logic in this would violate the assignment rules.

OutOfMemoryError when there is insufficient memory for the ScopedMemory object or for the backing memory.

ScopedMemory(SizeEstimator, Runnable)

Signature

```
public  
    ScopedMemory(SizeEstimator size,  
                  Runnable logic)
```

Description

Equivalent to [ScopedMemory\(long, Runnable\)](#)⁴²¹ with the argument list (size.

⁴²¹Section [A.2.3.31.1](#)

getEstimate(), logic).

Deprecated since RTSJ 2.0

Parameters

size The size of the new ScopedMemory area estimated by an instance of [SizeEstimator](#)⁴²².

logic The logic which will use the memory represented by this as its initial memory area. When logic is null, this constructor is equivalent to constructing the memory area without a logic value.

Throws

IllegalArgumentException when size is null, or size.getEstimate() is negative.

OutOfMemoryError when there is insufficient memory for the ScopedMemory object or for the backing memory.

[IllegalAssignmentError](#) when storing logic in this would violate the assignment rules.

ScopedMemory(long)

Signature

```
public  
ScopedMemory(long size)
```

Description

Equivalent to [ScopedMemory\(long, Runnable\)](#)⁴²³ with the argument list (size, null).

Deprecated since RTSJ 2.0 no longer visible.

Parameters

size of the new ScopedMemory area in bytes.

Throws

IllegalArgumentException when size is less than zero.

OutOfMemoryError when there is insufficient memory for the ScopedMemory object or for the backing memory.

⁴²²Section [11.3.2.5](#)

⁴²³Section [A.2.3.31.1](#)

ScopedMemory(SizeEstimator)

Signature

```
public  
    ScopedMemory(SizeEstimator size)
```

Description

Equivalent to `ScopedMemory(long, Runnable)`⁴²⁴ with the argument list (size, getEstimate(), null).

Deprecated since RTSJ 2.0.

Parameters

size The size of the new ScopedMemory area estimated by an instance of `SizeEstimator`⁴²⁵.

Throws

`IllegalArgumentException` when size is null, or size.getEstimate() is negative.

`OutOfMemoryError` when there is insufficient memory for the ScopedMemory object or for the backing memory.

A.2.3.31.2 Methods

enter

Signature

```
public void  
    enter()
```

Description

Associate this memory area with the current schedulable for the duration of the execution of the run() method of the instance of Runnable given in the constructor. During this period of execution, this memory area becomes the default allocation context until another default allocation context is selected (using enter, or `executeInArea`⁴²⁶) or the enter method exits.

⁴²⁴Section [A.2.3.31.1](#)

⁴²⁵Section [11.3.2.5](#)

⁴²⁶Section [A.2.3.31.2](#)

Throws

ScopedCycleException when this invocation would break the single parent rule.

ThrowBoundaryError Thrown when the JVM needs to propagate an exception allocated in this scope to (or through) the memory area of the caller. Storing a reference to that exception would cause an **IllegalAssignmentError**⁴²⁷, so the JVM cannot be permitted to deliver the exception. The **ThrowBoundaryError**⁴²⁸ 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 **IllegalArgumentException** when the caller is a schedulable and a null value for logic was supplied when the memory area was constructed.

MemoryAccessError **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

Associate this memory area with the current schedulable for the duration of the execution of the `run()` method of the given `Runnable`. During this period of execution, this memory area becomes the default allocation context until another default allocation context is selected (using `enter`, or **executeInArea**⁴²⁹) or the `enter` method exits.

Parameters

logic logic The `Runnable` object whose `run()` method should be invoked.

Throws

ScopedCycleException when this invocation would break the single parent rule.

⁴²⁷Section 15.2.3.2

⁴²⁸Section 15.2.3.8

⁴²⁹Section A.2.3.31.2

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**⁴³⁰, so the JVM cannot be permitted to deliver the exception. The **ThrowBoundaryError**⁴³¹ 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 **IllegalArgumentException** when the caller is a schedulable and logic is null.

executeInArea(Runnable)

Signature

```
public void  
executeInArea(Runnable logic)
```

Description

Execute the run method from the logic parameter using this memory area as the current allocation context. This method behaves as if it moves the allocation context down the scope stack to the occurrence of this.

Parameters

logic The runnable object whose run() method should be executed.

Throws

IllegalThreadStateException when the caller is a Java thread.

InaccessibleAreaException when the memory area is not in the schedulable's scope stack.

IllegalArgumentException when the caller is a schedulable and logic is null.

getPortal

Signature

⁴³⁰Section 15.2.3.2

⁴³¹Section 15.2.3.8

```
public java.lang.Object  
getPortal()
```

Description

Return a reference to the portal object in this instance of ScopedMemory.

Assignment rules are enforced on the value returned by `getPortal` as if the return value were first stored in an object allocated in the current allocation context, then moved to its final destination.

Throws

IllegalAssignmentError when a reference to the portal object cannot be stored in the caller's allocation context; that is, when this is "inner" relative to the current allocation context or not on the caller's scope stack.

IllegalThreadStateException when the caller is a Java thread.

Returns

A reference to the portal object or null when there is no portal object. The portal value is always set to null when the contents of the memory are deleted.

getReferenceCount

Signature

```
public int  
getReferenceCount()
```

Description

Returns the reference count of this ScopedMemory.

Note, a reference count of 0 reliably means that the scope is not referenced, but other reference counts are subject to artifacts of lazy/eager maintenance by the implementation.

Returns

The reference count of this ScopedMemory.

join

Signature

```
public void  
join()
```

throws InterruptedException

Description

Wait until the reference count of this ScopedMemory goes down to zero. Return immediately when the memory is unreferenced.

Throws

InterruptedException When this schedulable is interrupted by [RealtimeThread.interrupt\(\)](#)⁴³² or [AsynchronouslyInterruptedException.fire\(\)](#)⁴³³ while waiting for the reference count to go to zero.

IllegalThreadStateException when the caller is a Java thread.

join(HighResolutionTime)

Signature

```
public void  
join(javax.realtime.HighResolutionTime<?> time)  
throws InterruptedException
```

Description

Wait at most until the time designated by the time parameter for the reference count of this ScopedMemory to drop to zero. Return immediately when the memory area is unreferenced.

Since the time is expressed as a [HighResolutionTime](#)⁴³⁴, 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.

⁴³²Section [5.3.2.2.2](#)

⁴³³Section [8.3.2.1.2](#)

⁴³⁴Section [9.3.1.2](#)

Throws

InterruptedException When this schedulable is interrupted by `RealtimeThread.interrupt()`⁴³⁵ or `AsynchronouslyInterruptedException.fire()`⁴³⁶ while waiting for the reference count to go to zero.

IllegalThreadStateException when the caller is a Java thread.

IllegalArgumentException when the caller is a schedulable and time is null.

UnsupportedOperationException when the wait operation is not supported using the clock associated with time.

joinAndEnter*Signature*

```
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 enter the `ScopedMemory` and execute the run method from logic passed in the constructor. When no instance of `Runnable` was passed to the memory area's constructor, the method throws `IllegalArgumentException` immediately.

When multiple threads are waiting in `joinAndEnter` family methods for a memory area, at most *one* of them will be released each time the reference count goes to zero.

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()`⁴³⁷ or `AsynchronouslyInterruptedException.fire()`⁴³⁸ while waiting for the reference count to go to zero.

⁴³⁵Section 5.3.2.2.2

⁴³⁶Section 8.3.2.1.2

⁴³⁷Section 5.3.2.2.2

⁴³⁸Section 8.3.2.1.2

IllegalThreadStateException when the caller is a Java thread, or when this method is invoked during finalization of objects in scoped memory and entering this scoped memory area would force deletion of the SO that triggered finalization. This would include the scope containing the SO, and the scope (if any) containing the scope containing the SO.

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**⁴³⁹, so the JVM cannot be permitted to deliver the exception. The **ThrowBoundaryError**⁴⁴⁰ 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

```
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 `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**⁴⁴¹, this method has an

⁴³⁹Section 15.2.3.2

⁴⁴⁰Section 15.2.3.8

⁴⁴¹Section 9.3.1.2

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`⁴⁴².

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 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**⁴⁴³, so the JVM cannot be permitted to deliver the exception. The **ThrowBoundaryError**⁴⁴⁴ is allocated in the current allocation context and contains information about the exception it replaces.

InterruptedException When this schedulable is interrupted by **RealtimeThread.interrupt()**⁴⁴⁵ or **AsynchronouslyInterruptedException.fire()**⁴⁴⁶ 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.

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 using the clock associated with time.

MemoryAccessError when the calling schedulable may not use the heap and this

⁴⁴²Section A.2.3.31.2

⁴⁴³Section 15.2.3.2

⁴⁴⁴Section 15.2.3.8

⁴⁴⁵Section 5.3.2.2.2

⁴⁴⁶Section 8.3.2.1.2

memory area's logic value is allocated in heap memory.

joinAndEnter(Runnable)

Signature

```
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()`⁴⁴⁷ or `AsynchronouslyInterruptedException.fire()`⁴⁴⁸ while waiting for the reference count to go to zero.

`IllegalThreadStateException` when the caller is a Java thread, or when this method is invoked during finalization of objects in scoped memory and entering this scoped memory area would force deletion of the SO that triggered finalization. This would include the scope containing the SO, and the scope (if any) containing the scope containing the SO.

`ThrowBoundaryError` Thrown when the JVM needs to propagate an exception allocated in this scope to (or through) the memory area of the caller. Storing a reference to that exception would cause an `IllegalAssignmentError`⁴⁴⁹, so the

⁴⁴⁷Section 5.3.2.2.2

⁴⁴⁸Section 8.3.2.1.2

⁴⁴⁹Section 15.2.3.2

JVM cannot be permitted to deliver the exception. The [ThrowBoundaryError](#)⁴⁵⁰ 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

```
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 [HighResolutionTime](#)⁴⁵¹, 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\(Runnable\)](#)⁴⁵².

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.

⁴⁵⁰Section [15.2.3.8](#)

⁴⁵¹Section [9.3.1.2](#)

⁴⁵²Section [A.2.3.31.2](#)

time The time that bounds the wait.

Throws

`InterruptedException` When this schedulable is interrupted by `RealtimeThread.interrupt()`⁴⁵³ or `AsynchronouslyInterruptedException.fire()`⁴⁵⁴ while waiting for the reference count to go to zero.

`IllegalThreadStateException` when the caller is a Java thread, or when this method is invoked during finalization of objects in scoped memory and entering this scoped memory area would force deletion of the SO that triggered finalization. This would include the scope containing the SO, and the scope (if any) containing the scope containing the SO.

`ThrowBoundaryError` Thrown when the JVM needs to propagate an exception allocated in this scope to (or through) the memory area of the caller. Storing a reference to that exception would cause an `IllegalAssignmentError`⁴⁵⁵, so the JVM cannot be permitted to deliver the exception. The `ThrowBoundaryError`⁴⁵⁶ 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

```
public java.lang.Object
newArray(java.lang.Class<?> type,
         int number)
```

Description

Allocate an array of the given type in this memory area. This method may be concurrently used by multiple threads.

Parameters

⁴⁵³Section 5.3.2.2.2

⁴⁵⁴Section 8.3.2.1.2

⁴⁵⁵Section 15.2.3.2

⁴⁵⁶Section 15.2.3.8

type 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 number The number of elements in the new array.

Throws

IllegalArgumentException IllegalArgumentException when number is less than zero, type is null, or type is java.lang.Void.TYPE.

OutOfMemoryError OutOfMemoryError when space in the memory area is exhausted.

IllegalThreadStateException when the caller is a Java thread.

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

```
public T
newInstance(java.lang.Class<T> type)
throws IllegalAccessException,
        InstantiationException
```

Description

Allocate an object in this memory area. This method may be concurrently used by multiple threads.

Parameters

type type The class of which to create a new instance.

Throws

IllegalAccessException IllegalAccessException The class or initializer is inaccessible.

IllegalArgumentException IllegalArgumentException when type is null.

ExceptionInInitializerError ExceptionInInitializerError when an unexpected exception has occurred in a static initializer.

OutOfMemoryError OutOfMemoryError when space in the memory area is exhausted.

InstantiationException 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 is a Java thread.

InaccessibleAreaException when the memory area is not in the schedulable's scope stack.

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 IllegalAccessException,  
       InstantiationException,  
       InvocationTargetException
```

Description

Allocate an object in this memory area. This method may be concurrently used by multiple threads.

Parameters

c Tc The constructor for the new instance.

args args An array of arguments to pass to the constructor.

Throws

IllegalAccessException IllegalAccessException when the class or initializer is inaccessible under Java access control.

InstantiationException InstantiationException when the specified class object could not be instantiated. Possible causes are it is an interface, it is abstract, it is an array.

OutOfMemoryError OutOfMemoryError when space in the memory area is exhausted.

IllegalArgumentException IllegalArgumentException when c is null, or the args array does not contain the number of arguments required by c. A null value of args is treated like an array of length 0.

IllegalThreadStateException when the caller is a Java thread.

InvocationTargetException 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)
```

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

IllegalThreadStateException when the caller is a Java Thread.

IllegalAssignmentError when the caller is a schedulable, and object is not allocated in this scoped memory instance and not null.

InaccessibleAreaException when the caller is a schedulable, this memory area is not in the caller's scope stack and object is not null.

toString*Signature*

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

Description

Returns a user-friendly representation of this ScopedMemory of the form ScopedMemory#<num> where <num> is a number that uniquely identifies this scoped memory area.

Returns

The string representation

A.2.3.32 SporadicParameters

The following elements of SporadicParameters are deprecated. The required elements are documented in Section 6.3.3.15 above.

A.2.3.32.1 Fields

mitViolationExcept

public static final 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](#)⁴⁵⁷ which has this as its instance of [ReleaseParameters](#)⁴⁵⁸ occurs at a time less than the minimum interarrival time defined here then the fire() method shall throw [MITViolationException](#)⁴⁵⁹. 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](#)⁴⁶⁰ is bound then the arrival time is ignored.

Deprecated since RTSJ 2.0

mitViolationIgnore

public static final 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](#)⁴⁶¹ which has this as its instance of [ReleaseParameters](#)⁴⁶² occurs at a time

⁴⁵⁷Section 6.3.1.3

⁴⁵⁸Section 6.3.3.10

⁴⁵⁹Section 15.2.2.7

⁴⁶⁰Section 8.3.3.5

⁴⁶¹Section 6.3.1.3

⁴⁶²Section 6.3.3.10

less than 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

public static final mitViolationSave

Description

Represents the “SAVE” policy for dealing with minimum interarrival time violations. Under this policy the arrival time for any instance of [Schedulable](#)⁴⁶³ which has this as its instance of [ReleaseParameters](#)⁴⁶⁴ 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

public static final 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](#)⁴⁶⁵ which has this as its instance of [ReleaseParameters](#)⁴⁶⁶ occurs at a time less than 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

A.2.3.32.2 Methods

⁴⁶³Section [6.3.1.3](#)

⁴⁶⁴Section [6.3.3.10](#)

⁴⁶⁵Section [6.3.1.3](#)

⁴⁶⁶Section [6.3.3.10](#)

setMitViolationBehavior(String)

Signature

```
public void  
setMitViolationBehavior(String behavior)
```

Description

Sets the behavior of the arrival time queue in the case where the new arrival time is closer to the previous arrival time than the minimum interarrival time given in this.

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

Parameters

behavior A string representing the behavior.

Throws

IllegalArgumentException when behavior is not one of the final MIT violation behavior values defined in this class.

Deprecated since RTSJ 2.0 and replaced by [setMinimumInterarrivalPolicy](#)⁴⁶⁷.

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](#)⁴⁶⁸.

⁴⁶⁷Section [6.3.3.15.3](#)

⁴⁶⁸Section [6.3.3.15.3](#)

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 *cost* The proposed cost. to determine when any particular object exceeds cost. When null, the default value is a new instance of `RelativeTime(0,0)`.
deadline *deadline* The proposed deadline. When null, the default value is a new instance of `RelativeTime(mit)`.

Throws

IllegalArgumentException *IllegalArgumentException* when the time value of cost is less than zero, or the time value of deadline is less than or equal to zero, or the values are incompatible with the scheduler for any of the schedulables which are presently using this parameter object.

IllegalAssignmentError *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 anlaysis is inadequate

setIfFeasible(RelativeTime, RelativeTime, RelativeTime)

Signature

```
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

`IllegalArgumentException` when **minInterarrival** is null or its time value is not greater than zero, or the time value of **cost** is less than zero, or the time value of **deadline** is not greater than zero.

`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

A.2.3.33 *ThrowBoundaryError*

The following elements of `ThrowBoundaryError` are deprecated. The required elements are documented in Section 15.2.3.8 above.

A.2.3.33.1 Constructors

ThrowBoundaryError(String)

Signature

```
public
  ThrowBoundaryError(String description)
```

Description

A descriptive constructor for `ThrowBoundaryError`.

Deprecated since RTSJ 2.0; application code should use `get()`⁴⁶⁹ instead.

Parameters

`description` Description of the error.

A.2.3.34 *Timer*

The following elements of `Timer` are deprecated. The required elements are documented in Section 10.3.2.6 above.

A.2.3.34.1 *Methods***destroy***Signature*

```
public void
  destroy()
  throws IllegalStateException
```

Description

Stop this from counting or comparing when *active*, remove from it all the associated handlers if any, and release as many of its resources as possible back to the system. Every method invoked on a `Timer` that has been *destroyed* will throw `IllegalStateException`.

Throws

`IllegalStateException` when this `Timer` has been *destroyed*.

Deprecated since RTSJ 2.0

⁴⁶⁹Section 15.2.3.8.2

bindTo(String)

Signature

```
public void  
bindTo(String happening)  
throws UnsupportedOperationException
```

Description

Should not be called.

Parameters

happening to which to bind

Throws

UnsupportedOperationException when bindTo is called on a Timer.

Available since RTSJ 1.0.1

Deprecated RTSJ 2.0

A.2.3.35 VTMemory

Inheritance

```
java.lang.Object  
  javax.realtime.MemoryArea  
    javax.realtime.ScopedMemory  
      javax.realtime.VTMemory
```

Description

VTMemory is similar to [LTMemory](#)⁴⁷⁰ 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

A.2.3.35.1 Constructors

⁴⁷⁰Section [A.2.3.11](#)

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 this memory area to which the size may grow.

logic An instance of Runnable whose run() method will use this as its initial memory area. When logic is null, this constructor is equivalent to [VTMemory\(long initial, long maximum\)](#)⁴⁷¹.

Throws

[IllegalArgumentException](#) when initial is greater than maximum, or when initial or maximum is less than zero.

[OutOfMemoryError](#) when there is insufficient memory for the VTMemory object or for the backing memory.

[IllegalAssignmentError](#) when storing logic in this would violate the assignment rules.

VTMemory(SizeEstimator, SizeEstimator, Runnable)

Signature

```
public
    VTMemory(SizeEstimator initial,
              SizeEstimator maximum,
              Runnable logic)
```

Description

Equivalent to [VTMemory\(long, long, Runnable\)](#)⁴⁷² with the argument list (initial.

⁴⁷¹Section [A.2.3.35.1](#)

⁴⁷²Section [A.2.3.35.1](#)

getEstimate(), maximum.getEstimate(), logic).

Parameters

initial The size in bytes of the memory to initially allocate for this area.

maximum The maximum size in bytes this memory area to which the size may grow estimated by an instance of [SizeEstimator](#)⁴⁷³.

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\(SizeEstimator initial, SizeEstimator maximum\)](#)⁴⁷⁴.

Throws

`IllegalArgumentException` when `initial` is null, `maximum` is null, `initial.getEstimate()` is greater than `maximum.getEstimate()`, or when `initial.getEstimate()` is less than zero.

`OutOfMemoryError` when there is insufficient memory for the `VTMemory` object or for the backing memory.

[IllegalAssignmentError](#) when storing `logic` in this would violate the assignment rules.

VTMemory(long, long)

Signature

```
public
    VTMemory(long initial,
              long maximum)
```

Description

Equivalent to [VTMemory\(long, long, Runnable\)](#)⁴⁷⁵ 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 this memory area to which the size may grow.

Throws

`IllegalArgumentException` when `initial` is greater than `maximum` or when `initial` or `maximum` is less than zero.

⁴⁷³Section [11.3.2.5](#)

⁴⁷⁴Section [A.2.3.35.1](#)

⁴⁷⁵Section [A.2.3.35.1](#)

OutOfMemoryError when there is insufficient memory for the VTMemory object or for the backing memory.

VTMemory(SizeEstimator, SizeEstimator)

Signature

```
public  
VTMemory(SizeEstimator initial,  
          SizeEstimator maximum)
```

Description

Equivalent to `VTMemory(long, long, Runnable)`⁴⁷⁶ with the argument list (initial.getEstimate(), maximum.getEstimate(), null).

Parameters

initial The size in bytes of the memory to initially allocate for this area.

maximum The maximum size in bytes this memory area to which the size may grow estimated by an instance of `SizeEstimator`⁴⁷⁷.

Throws

IllegalArgumentException when initial is null, maximum is null, initial.getEstimate() is greater than maximum.getEstimate(), or when initial.getEstimate() is less than zero.

OutOfMemoryError when there is insufficient memory for the VTMemory object or for the backing memory.

VTMemory(long, Runnable)

Signature

```
public  
VTMemory(long size,  
          Runnable logic)
```

Description

Equivalent to `VTMemory(long, long, Runnable)`⁴⁷⁸ with the argument list (size, size, logic).

⁴⁷⁶Section [A.2.3.35.1](#)

⁴⁷⁷Section [11.3.2.5](#)

⁴⁷⁸Section [A.2.3.35.1](#)

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 initial memory area. When logic is null, this constructor is equivalent to `VTMemory(long size)`⁴⁷⁹.

Throws

`IllegalArgumentException` when size is less than zero.

`OutOfMemoryError` when there is insufficient memory for the VTMemory object or for the backing memory.

`IllegalAssignmentError` when storing logic in this would violate the assignment rules.

VTMemory(SizeEstimator, Runnable)

Signature

```
public  
VTMemory(SizeEstimator size,  
          Runnable logic)
```

Description

Equivalent to `VTMemory(long, long, Runnable)`⁴⁸⁰ with the argument list (size, getEstimate(), size.getEstimate(), logic).

Available since RTSJ 1.0.1

Parameters

size An instance of `SizeEstimator`⁴⁸¹ used to give an estimate of the initial size. This memory must be committed before the completion of the constructor.

logic The run() of the given Runnable will be executed using this as its initial memory area. When logic is null, this constructor is equivalent to `VTMemory(SizeEstimator initial)`⁴⁸².

Throws

⁴⁷⁹Section A.2.3.35.1

⁴⁸⁰Section A.2.3.35.1

⁴⁸¹Section 11.3.2.5

⁴⁸²Section A.2.3.35.1

`IllegalArgumentException` when `size` is null, or `size.getEstimate()` is less than zero.

`OutOfMemoryError` when there is insufficient memory for the `VTMemory` object or for the backing memory.

`IllegalAssignmentError` when storing logic in this would violate the assignment rules.

VTMemory(long)

Signature

```
public  
VTMemory(long size)
```

Description

Equivalent to `VTMemory(long, long, Runnable)`⁴⁸³ 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.

`OutOfMemoryError` when there is insufficient memory for the `VTMemory` object or for the backing memory.

VTMemory(SizeEstimator)

Signature

```
public  
VTMemory(SizeEstimator size)
```

Description

Equivalent to `VTMemory(long, long, Runnable)`⁴⁸⁴ with the argument list (`size.getEstimate()`, `size.getEstimate()`, `null`).

⁴⁸³Section [A.2.3.35.1](#)

⁴⁸⁴Section [A.2.3.35.1](#)

Available since RTSJ 1.0.1

Parameters

size An instance of [SizeEstimator](#)⁴⁸⁵ 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.
OutOfMemoryError when there is insufficient memory for the VTMemory object
or for the backing memory.

A.2.3.35.2 Methods

toString

Signature

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

Description

Create a string representing this object. The string is of the form
(VTMemory) Scoped memory # num
where num uniquely identifies the VTMemory area.

Returns

A string representing the value of this.

A.2.3.36 VTPhysicalMemory

Inheritance

```
java.lang.Object  
  javafx.runtime.MemoryArea  
    javafx.runtime.ScopedMemory  
      javafx.runtime.VTPhysicalMemory
```

Description

⁴⁸⁵Section [11.3.2.5](#)

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](#)⁴⁸⁶ 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

A.2.3.36.1 Constructors

VTPhysicalMemory(Object, long, long, Runnable)

Signature

⁴⁸⁶Section [A.2.3.31](#)

```
public
VTPhysicalMemory(Object type,
                  long base,
                  long size,
                  Runnable logic)
```

Description

Create 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\(\)](#)⁴⁸⁷ is called. When logic is null, logic must be supplied when the memory area is entered.

Throws

[SizeOutOfBoundsException](#) when the implementation detects that size extends beyond physically addressable memory.

[SecurityException](#) when the application 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](#)⁴⁸⁸ has been registered with the [PhysicalMemoryManager](#)⁴⁸⁹.

[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.

⁴⁸⁷Section [11.3.2.3.2](#)

⁴⁸⁸Section [A.2.1.1](#)

⁴⁸⁹Section [A.2.3.19](#)

IllegalAssignmentError when storing logic in this would violate the assignment rules.

VTPhysicalMemory(Object, long, SizeEstimator, Runnable)

Signature

```
public
VTPhysicalMemory(Object type,
                  long base,
                  SizeEstimator size,
                  Runnable logic)
```

Description

Equivalent to **VTPhysicalMemory(Object, long, long, Runnable)**⁴⁹⁰ 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()**⁴⁹¹ is called. When logic is null, logic must be supplied when the memory area is entered.

Throws

SecurityException when the application doesn't have permissions to access physical memory or the given range of memory.

SizeOutOfBoundsException when the implementation detects that the size estimate from size extends beyond physically addressable memory.

⁴⁹⁰Section [A.2.3.36.1](#)

⁴⁹¹Section [11.3.2.3.2](#)

[OffsetOutOfBoundsException](#) when the base address is invalid.

[UnsupportedPhysicalMemoryException](#) when the underlying hardware does not support the given type, or when no matching [PhysicalMemoryTypeFilter](#)⁴⁹² has been registered with the [PhysicalMemoryManager](#)⁴⁹³.

[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\(Object, long, long, Runnable\)](#)⁴⁹⁴ 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.

⁴⁹²Section [A.2.1.1](#)

⁴⁹³Section [A.2.3.19](#)

⁴⁹⁴Section [A.2.3.36.1](#)

Throws

`SecurityException` when the application doesn't have permissions to access physical memory or the given range of memory.

`SizeOutOfBoundsException` when the implementation detects that size extends beyond physically addressable memory.

`OffsetOutOfBoundsException` when the base address is invalid.

`UnsupportedPhysicalMemoryException` when the underlying hardware does not support the given type, or when no matching `PhysicalMemoryTypeFilter`⁴⁹⁵ has been registered with the `PhysicalMemoryManager`⁴⁹⁶.

`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*

```
public
VTPhysicalMemory(Object type,
                  long base,
                  SizeEstimator size)
```

Description

Equivalent to `VTPhysicalMemory(Object, long, long, Runnable)`⁴⁹⁷ 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).

⁴⁹⁵Section [A.2.1.1](#)

⁴⁹⁶Section [A.2.3.19](#)

⁴⁹⁷Section [A.2.3.36.1](#)

base The physical memory address of the area.

size A size estimator for this memory area.

Throws

SecurityException when the application doesn't have permissions to access physical memory or the given range of memory.

SizeOutOfBoundsException when the implementation detects that the size estimate from size extends beyond physically addressable memory.

OffsetOutOfBoundsException when the base address is invalid.

UnsupportedPhysicalMemoryException when the underlying hardware does not support the given type, or when no matching PhysicalMemoryTypeFilter⁴⁹⁸ has been registered with the PhysicalMemoryManager⁴⁹⁹.

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)⁵⁰⁰ with the argument list (type, 0, size, 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

⁴⁹⁸Section A.2.1.1

⁴⁹⁹Section A.2.3.19

⁵⁰⁰Section A.2.3.36.1

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\(\)](#)⁵⁰¹ is called. When logic is null, logic must be supplied when the memory area is entered.

Throws

[SecurityException](#) 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](#)⁵⁰² has been registered with the [PhysicalMemoryManager](#)⁵⁰³.

[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\(Object, long, long, Runnable\)](#)⁵⁰⁴ with the argument list (type, 0, size.getEstimate(), logic).

See [Section PhysicalMemoryManager](#)

⁵⁰¹Section [11.3.2.3.2](#)

⁵⁰²Section [A.2.1.1](#)

⁵⁰³Section [A.2.3.19](#)

⁵⁰⁴Section [A.2.3.36.1](#)

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()`⁵⁰⁵ is called. When **logic** is null, **logic** must be supplied when the memory area is entered.

Throws

`SecurityException` when the application doesn't have permissions to access physical memory or the given range of memory.

`SizeOutOfBoundsException` when the implementation detects that the size estimate from **size** extends beyond physically addressable memory.

`UnsupportedPhysicalMemoryException` when the underlying hardware does not support the given type, or when no matching `PhysicalMemoryTypeFilter`⁵⁰⁶ has been registered with the `PhysicalMemoryManager`⁵⁰⁷.

`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.

`IllegalAssignmentError` when storing **logic** in this would violate the assignment rules.

VTPhysicalMemory(Object, long)

Signature

```
public
VTPhysicalMemory(Object type,
                  long size)
```

Description

⁵⁰⁵Section 11.3.2.3.2

⁵⁰⁶Section A.2.1.1

⁵⁰⁷Section A.2.3.19

Equivalent to `VTPhysicalMemory(Object, long, long, Runnable)`⁵⁰⁸ with the argument list (type, 0, 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`).

`size` The size of the area in bytes.

Throws

`SecurityException` when the application doesn't have permissions to access physical memory or the given range of memory.

`SizeOutOfBoundsException` when the implementation detects that `size` extends beyond physically addressable memory.

`UnsupportedPhysicalMemoryException` when the underlying hardware does not support the given `type`, or when no matching `PhysicalMemoryTypeFilter`⁵⁰⁹ has been registered with the `PhysicalMemoryManager`⁵¹⁰.

`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

⁵⁰⁸Section [A.2.3.36.1](#)

⁵⁰⁹Section [A.2.1.1](#)

⁵¹⁰Section [A.2.3.19](#)

Equivalent to `VTPhysicalMemory(Object, long, long, Runnable)`⁵¹¹ with the argument list (type, 0, 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).

size A size estimator for this area.

Throws

`SecurityException` when the application doesn't have permissions to access physical memory or the given range of memory.

`SizeOutOfBoundsException` when the implementation detects that the size estimate from size extends beyond physically addressable memory.

`UnsupportedPhysicalMemoryException` when the underlying hardware does not support the given type, or when no matching `PhysicalMemoryTypeFilter`⁵¹² has been registered with the `PhysicalMemoryManager`⁵¹³.

`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.

A.2.3.36.2 Methods

toString

Signature

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

⁵¹¹Section [A.2.3.36.1](#)

⁵¹²Section [A.2.1.1](#)

⁵¹³Section [A.2.3.19](#)

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.

A.2.3.37 WaitFreeDeque**Inheritance**

java.lang.Object
[javax.realtime.WaitFreeDeque](#)

Description

A WaitFreeDeque encapsulates a WaitFreeWriteQueue and a WaitFreeReadQueue. Each method on a WaitFreeDeque corresponds to an equivalent operation on the underlying WaitFreeWriteQueue or WaitFreeReadQueue.

Incompatibility with V1.0: Three exceptions previously thrown by the constructor have been deleted from the throws clause. These are

- java.lang.IllegalAccessException,
- java.lang.ClassNotFoundException, and
- java.lang.InstantiationException.

Including these exceptions on the throws clause was an error. Their deletion may cause compile-time errors in code using the previous constructor. The repair is to remove the exceptions from the catch clause around the constructor invocation.

WaitFreeDeque is one of the classes allowing NoHeapRealtimeThreads and regular Java threads to synchronize on an object without the risk of a NoHeapRealtimeThread incurring Garbage Collector latency due to priority inversion avoidance management.

Deprecated as of RTSJ 1.0.1

A.2.3.37.1 Constructors

WaitFreeDeque(Runnable, Runnable, int, MemoryArea)

Signature

```
public
WaitFreeDeque(Runnable writer,
               Runnable reader,
               int maximum,
               MemoryArea memory)
```

Description

Constructs a queue, in memory, with an underlying [WaitFreeWriteQueue](#)⁵¹⁴ and [WaitFreeReadQueue](#)⁵¹⁵, each of size maximum.

The writer and reader parameters, when non-null, are checked to insure that they are compatible with the MemoryArea specified by memory (when non-null.) When memory is null and both Runnables are non-null, the constructor will select the nearest common scoped parent memory area, or when there is no such scope it will use immortal memory. When all three parameters are null, the queue will be allocated in immortal memory.

reader and writer are not necessarily the only threads or schedulables that will access the queue; moreover, there is no check that they actually access the queue at all.

Note that the wait free queues' internal queues are allocated in memory, but the memory area of the wait free dequeue 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 Then maximum number of elements in the both the [WaitFreeReadQueue](#)⁵¹⁶ and the [WaitFreeWriteQueue](#)⁵¹⁷.

memory The [MemoryArea](#)⁵¹⁸ in which internal elements are allocated.

Throws

[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.

⁵¹⁴Section [7.3.1.5](#)

⁵¹⁵Section [7.3.1.4](#)

⁵¹⁶Section [7.3.1.4](#)

⁵¹⁷Section [7.3.1.5](#)

⁵¹⁸Section [11.3.2.3](#)

IllegalArgumentException When an argument holds an invalid value. The writer argument must be null, a reference to a `Thread`, or a reference to a schedulable (a `RealtimeThread`, or an `AsyncEventHandler`.) The reader argument must be null, a reference to a `Thread`, or a reference to a schedulable object. The maximum argument must be greater than zero.

InaccessibleAreaException when memory is a scoped memory that is not on the caller's scope stack.

A.2.3.37.2 Methods

nonBlockingRead

Signature

```
public java.lang.Object
nonBlockingRead()
```

Description

An unsynchronized call of the `read()` method of the underlying [WaitFreeReadQueue](#)⁵¹⁹.

Returns

A `java.lang.Object` object read from this. When there are no elements in this then `null` is returned.

blockingWrite(Object)

Signature

```
public void
blockingWrite(Object object)
throws InterruptedException
```

Description

A synchronized call of the `write()` method of the underlying [WaitFreeReadQueue](#)⁵²⁰. This call blocks on queue full and waits until there is space in this.

⁵¹⁹Section [7.3.1.4](#)

⁵²⁰Section [7.3.1.4](#)

Parameters

object The java.lang.Object to place in this.

Throws

[MemoryScopeException](#) when a memory access error or illegal assignment error would occur while storing object in the queue.

InterruptedException when the thread is interrupted by interrupt() or [AsynchronouslyInterruptedException](#)⁵²¹ during the time between calling this method and returning from it.

Available since RTSJ 1.0.1 Return type changed from boolean to void because this method *always* returned true, and added InterruptedException.

nonBlockingWrite(Object)

Signature

```
public boolean  
nonBlockingWrite(Object object)
```

Description

An unsynchronized call of the write() method of the underlying [WaitFreeWriteQueue](#)⁵²². This call does not block on queue full.

Parameters

object The Object to attempt to place in this.

Throws

[MemoryScopeException](#) when a memory access error or illegal assignment error would occur while storing object in the queue.

Returns

true when object was inserted (i.e., the queue was not full), false otherwise.

blockingRead

Signature

```
public java.lang.Object  
blockingRead()  
throws InterruptedException
```

Description

⁵²¹Section [8.3.2.1.2](#)

⁵²²Section [7.3.1.5](#)

A synchronized call of the `read()` method of the underlying `WaitFreeWriteQueue`⁵²³. This call blocks on queue empty and will wait until there is an element in the queue to return.

Throws

`InterruptedException` when the thread is interrupted by `interrupt()` or `AsynchronouslyInterruptedException`⁵²⁴ during the time between calling this method and returning from it.

Returns

The `java.lang.Object` read.

Available since RTSJ 1.0.1 Added throws `InterruptedException`.

force(Object)

Signature

```
public boolean  
force(Object object)
```

Description

When this's underlying `WaitFreeWriteQueue`⁵²⁵ is full, then overwrite with object the most recently inserted element. Otherwise this call is equivalent to `nonBlockingWrite()`.

Parameters

object The object to be written.

Throws

`MemoryScopeException` when a memory access error or illegal assignment error would occur while storing object in the queue.

Returns

true when an element was overwritten; false when there as an empty element into which the write occurred.

A.3 Rationale

These are interface and classes that have been shown to be less the ideal. They have been replaced by elements that better fulfill the requirements. Compatibility can

⁵²³Section 7.3.1.5

⁵²⁴Section 8.3.2.1.2

⁵²⁵Section 7.3.1.5

be provided by implemenations that use existing facilities so there is not reason to continue requiring their inclusion new implementations.

Appendix B

Bibliography

- [1] *Portable Operating System Interface (POSIX®) Part 1: System Application Program Interface, International Standard ISO/IEC 9945-1, 1996 (E) IEEE Std 1003.1*, 1996 edition ed. The Institute of Electrical and Electronics Engineers, Inc., 1996.
- [2] BARR, M. Memory types. *Embedded Systems Programming* (2001), 103–104.
- [3] BURNS, A., AND WELLINGS, A. J. *Real-Time Systems and Programming Languages*; 4th ed. Addison Wesley, 2010.
- [4] DOS SANTOS, O. M., AND WELLINGS, A. Cost enforcement in the real-time specification for java. *Real-Time Syst.* 37, 2 (Nov. 2007), 139–179.
- [5] GOSLING, J., JOY, B., STEELE, G., BRACHA, G., AND BUCKLEY, A. *The Java Language Specification Java SE 8 Edition*. Oracle, 2014.
- [6] LINDHOLM, T., YELLIN, F., BRACHA, G., AND BUCKLEY, A. *The Java Virtual Machine Specification Java SE 8 Edition*. Oracle, 2014.
- [7] REGEHR, J. Safe and structured use of interrupts in real-time and embedded software. In *Handbook of Real-Time and Embedded Systems*, I. Lee, J. Y.-T. Leug, and S. H. Son, Eds. Chapman and Hall/CRC, 2007, pp. 16–1–16–12.

