ESR Consortium
HRTJ-0.7

Hard Realtime Java
Profile Specification

ESR009

Reference: ESR009-HRTJ
Version: 0.7
Rev: D

Contents

Tables

Illustrations

Copyright of The Software

Trademarks

1 PREFACE TO HRTJ 0.7 PROFILE, ESR009

This document defines the, *HRTJ 0.7* profile*,* targeting Java 2 Platforms. HRTJ *0.7* assumes B-ON 1.1[B-ON].

1.1 Hard Real-Time Java: definition

A hard real-time system (i.e. software & hardware), is a system that guaranties that expected behaviors will occur at the right times. The distinction between hard and soft real-time is between *guaranteeing* and *best-effort*.

Although hard timing constraints may be specified using any usefulness functions or probabilistic constraints, this specification uses *deterministic hard real time* constraints: hard real-time means determinism. An hard real-time operation is correct if and only if it computes the expected result within its given bounded time constraint.

This definition is not related to speed, but to have the different tasks of a system to complete by their respective deadlines for sure.

1.2 Who Should Use this Specification

This specification targets the following audiences:

- Platform Developer who want to build implementation that complies to the HRTJ profile specification;
- Application developers designing hard real-time applications;
- Java™ virtual machines providers deploying Java for hard real-time devices.

1.3 How This Specification is Organized

This specification is organized in three parts :

- **Introduction** is a short chapter explaining what is HRTJ, why it has been designed and what are its main assets.
- **Specification Core** describes the semantic of the concepts involved while designing a HRTJ system : scheduling, synchronization, task communication, rescue mode and memory management.
- **HRTJ API Documentation** lists the HRTJ APIs in as javadoc.

1.4 Comments

Your comments about HRTJ are welcome. Please send them by electronic mail to the following address: `comments@e-s-r.net` , with HRTJ in your subject line.

1.5 Related Literature

B-ON: , Beyond 1.1 - www.e-s-r.net, 2009
HRTJSLD: IS2T S.A., Education slides on Hard Real-time Java specification. Examples of understanding and of use cases., ,
SCHALG: Liu C. L., Layland J. W., Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment, 1973

PRIORT: Sha L., Rajkumar R., Lehoczky J. P., Priority Inheritance Protocols: An Approach to Real-Time Synchronization, 1987

1.6 Document Conventions

In this document, references to methods of a Java class are written as `ClassName.methodName(args)`. This applies to both static and instance methods. Where the method is static this will be made clear in the accompanying text.

1.7 Implementation Notes

The HRTJ specification does not include any implementation details. HRTJ implementors are free to use whatever techniques they deem appropriate to implement the specification, with (or without) collaboration of any Java virtual machine provider. HRTJ experts have taken great care not to mention any special Java virtual machines, nor any of their special features, in order to encourage fair competing implementations.

1.8 Application notes

The very first document to consider when new to hard real-time Java is [HRTJSLD].

2 INTRODUCTION

2.1 HRTJ: goal & a bit of history

[HRTJ] is designed with one goal in mind: to make it possible to have Java software certified at the highest possible certification levels, like DO178 level A for example. Therefore [HRTJ] targets libraries that “due to their sizes” can be certified at a reasonable cost, that is the embedded world. Also HRTJ is about hard real-time, not just real-time (see 1.1).

Garbage Collection (GC) is an essential part of the Java runtime system, which massively contributes to building reliable software faster at low costs, by freeing the programmer from complex and error prone explicit memory management. The HRTJ authors believes that for the acceptance of Java for hard real-time systems, applications must make use of GC (which copes with the determinism constraint).

More than a decade ago (roughly in 1998), experts started to work on introducing real-time capabilities to Java technology. Two consortium competed (`rtj.org` and `j-consortium.org`), and a great step has been made to capture what real-time Java could be: RTSJ (v1.0 is JSR001, v1.1 is JSR282[1]). In 2004-2006 the Open Group worked on defining a safety-critical Java (JSR302). All this previous work did not have in mind to reach *hard* real-time Java.

2.2 Requirements

The term MUST indicates that the associated definition is an absolute requirement, whereas MAY indicates that the item is optional. SHOULD indicates a highly recommended requirement.

This specification defines minimal requirements in order for implementations to employ resources (hardware and software) to their best advantages.

2.2.1 Hardware

There is no special hardware requirements for the HRTJ specification.

Depending on the target, some features may ease implementation like having at its disposal an hardware timer or watchdog.

2.2.2 Software

HRTJ relies on [B-ON], which defines

- (i) the start-up of a Java application that is the initialization sequence in a deterministic way,
- (ii) the persistent immutable read-only objects that may be placed into non-volatile memory areas, and do not require copies to be made in ram to be manipulated,
- (iii) immortal read-write objects that are always alive.

If the Java virtual machine is not baremetal[2], it is assumed that the operating system it relies on is an hard real-time operating system.

1The due date of the early draft review of the version 1.1 is beginning of May 2009.
2baremetal: a virtual machine is said to be baremetal when it does not require an OS/RTOS to run. A baremetal Java virtual machine is in fact an OS/RTOS that also embeds a Java engine. The device boots directly in Java.

3 SCHEDULING

3.1 Time

Time is not continuous but rather runs at a discrete rate, that is, one may count time with natural numbers. There are several way to get the time from an hardware board. Most modern processor have a special register that holds the number of cycles since the reset of the device. They are also RTC peripherals that provide time as a counter. At 8 Mhz, they are `8*1024*1024=8,388,608` cycles per second. Two cycles are separated by `0.000119209...` millisecond, i.e. `119.209...` nanoseconds.

HRTJ extents traditional Java time resolution (which is millisecond) to a finer resolution: the nanosecond resolution. The method `nanoTime` in the class `ej.hrt.HRTSystem` returns the current time of the system in nanoseconds as a `long` (64-bit). 2^{63}-1 is `java.lang.Long.MAX_VALUE`[3] : it represents a quantity of roughly 292 years.

3.2 HRT Tasks

An HRTJ application is exclusively made of HRT tasks, which are cyclic tasks launched by external triggers. A task, instance of the `ej.hrt.Task` class, is defined by its period T and a priority. Both period and priority are set by the program once for all (no modification possible at runtime). They MUST be compile-time constants. The priority is an `int` at least equal to 65 (see 7.3). The highest the priority is, the most important is the priority of the HRT task. The period is expressed in nanosecond.

Every HRTJ system is responsible for the computation of both the worst case execution time $WCET_i$ and the worst case allocation WCA_i of the n HRT tasks of the system. This computation can be made off-board the device before runtime. If the system is not able to compute one of these quantities, the application MUST not run. In particular, all methods that stops the HRT task for an undefined amount of time are forbidden (`sleep`, `wait`, native blocking read, etc.).

All HRT tasks MUST be identifiable by analysis of the application binary code. HRT system are based on the [B-ON] deterministic initialization phase. HRT task are started at the beginning of the mission phase [B-ON]. No HRT task can be added within the application once the application has started[4].

An HRT task defines its cycle of execution through its `run()` method. HRT tasks MUST not self suspend themselves (sleep, wait, etc.) nor block on some data read, etc.

3.3 States and start of HRT tasks

An HRT task may be in one of the next 3 states (Figure 3-1 presents state machine for an HRT task).

running	The processor is assigned to the HRT task, so that its instructions are executed. Only one HRT task can be in this state at any point in time.
preempted	All functional prerequisites for the HRT task to run exist, but another task is in the running state. Wait to be become the running task.
suspended	The task is passive and may be started. This is the initial state of all tasks.

3 9 223 372 036 854 775 807 nanoseconds

4 This excludes dynamically download of unknown code that could creates HRT tasks.

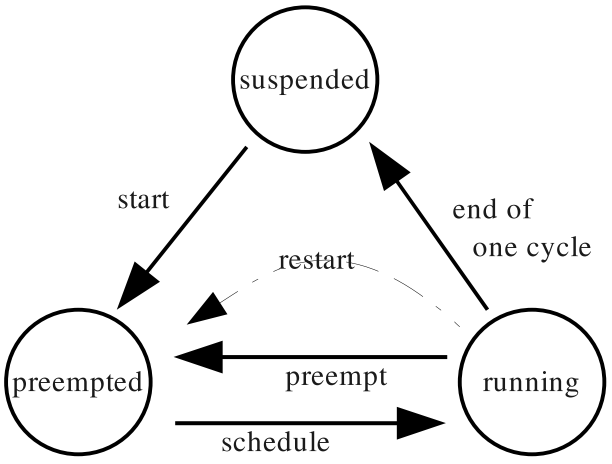
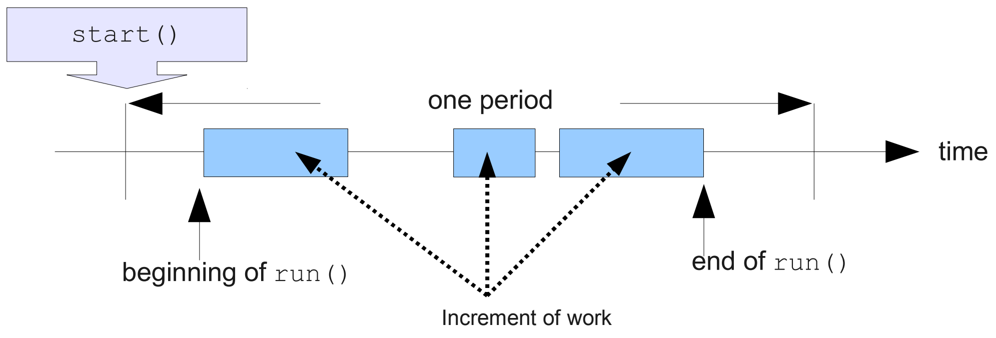


Figure 3-1: State model for HRT tasks

The start of an HRT task is done either by calling the `start()` method of a task, or by calling the `start(Task[])` method which starts all the provided tasks at the very same time. This last way of starting HRT tasks allow to have all tasks time-synchronized.

Once it has been started, an HRT task may restart itself automatically at the end of its cycle, if it has been registered to do so with the `Task.registerCyclic()` method. In such case, the next start occurs precisely after one period of the HRT task (the "restart" arrow of the Figure 3-1). `Task.unregisterCyclic()` switches off the automatic restart.

Note that starting an HRT task does not mean its `run()` method effectively runs immediately. It only turns the HRT task to be a task that is asking for the cpu resource, and that it has its `period` of time to finish the execution of its single cycle, i.e. its `run()` method.



If the start of an HRT task has been triggered (one time or several times) while the task is not in the `suspended` state (it either be running or preempted), the HRT task will restart one single new cycle at the end of its period (whatever the number of triggered `start()`). Triggering a start on an already started task that has been registered as cyclic has no effect.

Task priorities defines cpu resources attribution: higher priority tasks preempt lower priority tasks.

3.4 Startup

Initialization and mission phases definition are based on [B-ON] specification. In HRTJ specification initialization phase is extended to `main` method execution. Once `main` method is terminated, system enters in mission phase.

4 TASKS SYNCHRONIZATION

Critical sections are defined in Java by object's monitor (synchronized blocks or synchronized methods).

HRTJ uses the ceiling-priority protocol for critical sections[5]. HRT tasks can only synchronize on `ej.hrt.Monitor` objects. The ceiling priority of a `Monitor` may be set explicitly by the program. If unset, the HRTJ system is responsible for the computation of the ceiling priority of the monitor.

If an HRT task try to acquire an HRT monitor, and that HRT monitor ceiling priority is less that the HRT task priority, an HRT exception is thrown[6] (see 7.2).

5 MEMORY MANAGEMENT

An HRTJ system allows HRT tasks to allocate objects. An HRTJ system MUST guaranty that there will be no `OutOfMemoryError` during the mission phase for the HRT tasks.

HRTJ defines two properties that MUST be provided to the HRTJ system for it to run: CPB and HRTH. The CPB (Cross Period Buffer) defines the maximum amount of HRT objects that can potentially be alive when no HRT tasks run. HRT (Hard Real Time Heap), defines the available heap size for HRT allocation.

An HRTJ system is responsible for the computation of the WCA, Worst Case Allocation by cycle, of each HRT task of an application. If the system cannot determine such values, the application MUST not run.

With given CPB and HRTH, an HRTJ system MUST be able to qualify if there is enough memory to guaranty that none of the HRT tasks will ever run out of memory. If an HRTJ system is not capable to guaranty it, the application MUST not run.

6 INTER-TASK COMMUNICATION

HRT tasks can share data through `ObjectBuffers`. An ObjectBuffer is a structure that contains a limited number of objects. These elements are ordered in a first-in-first-out (FIFO) manner.

A task may share an object via the `ObjectBuffer.add` method and another task can retrieve this object via the `ObjectBuffer.remove` method.

Accesses to buffer are synchronized and not blocking. Behavior of adding an element to a full buffer may differ depending on the choosen policy (drop oldest value or don't add element).

Objects added to an ObjectBuffer are considered as elements of the CPB.

Primitive types such as integer or float can be shared with ByteBuffer, IntBuffer, FloatBuffer, LongBuffer or DoubleBuffer.

5 During initialization phase, there is only one running task. These task can always acquire any HRT monitor.

6 This can only happen if the ceiling priority has not been set automatically by the HRT system.

7 RESCUE MODE

7.1 Definition

A transient hardware failure or an undetected software flaw may cause an HRT task to execute longer than expected, or allocate more than its memory budget per period. When this happens, the "faulty" task is switched by the HRTJ system, at runtime, in the so called “rescue” mode.

7.2 HRT task behavior and rescue mode

The HRTJ system monitors that an HRT task does not execute a forbidden action. Once in the rescue mode, the HRT task suspends its execution and a specific handler is called. An HRT task that was registered as a cyclic task gets unregistered: it will not restart automatically. An explicit re-register needs to be applied.

The HRTJ system MUST switch the HRT task in the rescue mode and invokes the specific handler:

- `outOfResource(WCAException)`: when the HRT task tries to allocate some memory and it has already allocated all its available WCA budget.
- `outOfResource(DeadlineException)`: when the execution of cycle of the HRT is not finished at the end of its period. Handler is invoked immediately if the HRT task is not in a critical section, or as soon as it has exited all the critical sections the HRT task has entered in.
- `uncaughtException(Throwable)`: when the HRT task tries to acquire a monitor that is not HRT, when a runtime exception is thrown or when the HRT task tries to execute a unauthorized blocking operation (sleep, join, ...).

Execution in rescue mode is done at a lower priority than the GC. Objects created in rescue mode cannot be referenced by any heap object (reclaimable and immortal) nor static fields. When a task is in rescue mode it can access only immortal objects and objects created in rescue mode. An `ej.hrt.IllegalAssignmentError` is thrown when a task in rescue mode accesses a reclaimable object. The management of the objects created during rescue mode is implementation dependent[7].

7.3 Task priorities

The HRTJ specification defines 2 baskets of priorities, with the lowest priority as 65:

- [65..127]: priorities of HRT tasks,
- [1..63]: effective running priorities for HRT tasks in rescue mode.

When an HRT task switch to rescue mode, its priority is lowered by 64. When it has finished its execution in rescue mode, its priority regains its HRT value.

7 A typical implementation is to provide a fixed stack-based memory: when the task leaves the rescue mode, all objects created during the rescue mode are garbaged.

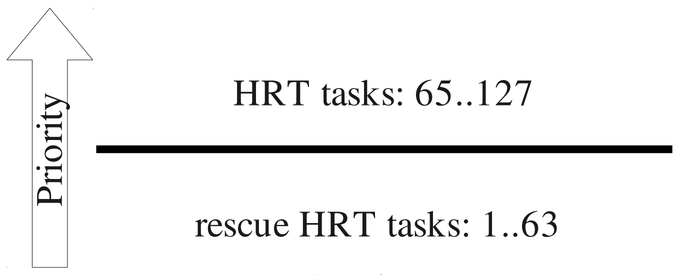


Figure 7-1: Task Priorities

8 JAVA VIRTUAL MACHINE CHARACTERISTICS

8.1 Bounded instructions and natives methods

Every instruction of the Java processor MUST be bounded. Instructions that cannot be executed in a constant time should be documented in order for the HRTJ system to detect them (and forbid their use while it computes the WCET of the HRT tasks). Same applies for native methods.

8.2 Execution model

There are several execution models that permit to compute the schedulability of a system. This specification defines a number of models, each with a given id. An application based on HRTJ will behave the same on two different HRTJ virtual machines if these two virtual machines share the same execution model. The global variable `EXECUTION_MODEL` states the execution model currently in used by a given implementation of this specification.

Next are the id for well known execution models :

- Rate Monotonic Algorithm : `1`

Specific execution models that are not yet in the specification start at `1000`.

9 SUMMARY OF RMA

Although this specification does not preclude any system schedulability method, there is one that is well know and easy to understand: the Rate Monotonic Algorithm. It results that it is often a good pragmatically choice. It also has the characteristic to manipulate quantities that can be computed by analysis over the binary Java code of applications.

9.1 Independent HRT Tasks

Let $WCET_i$ and T_i be the worst-case-execution-time and period of HRT task i respectively.

A set of n independent periodic HRT tasks scheduled by the rate monotonic algorithm will always meet its deadlines, if [SCHALG]:

$$\sum_{i=1}^{n} WCET_i / T_i <= n(2^{1/n} - 1)$$

9.2 HRT Tasks with critical sections

Let $WCET_i$ and T_i be the worst-case-execution-time and period of HRT task i respectively. Let B_i be the longest potentially blocking critical section of all task with a lower priority than the one of T_i (note: by construction, the B of the lowest priority HRT task is 0).

A set of n periodic tasks using the priority ceiling protocol can be scheduled by the rate monotonic algorithm, if [PRIORT]:

$$\left(\sum_{i=1}^{n} WCET_i / T_i\right) + \operatorname{Max}_{i=1}^{n}(B_i / T_i) <= n(2^{1/n} - 1)$$

10 JAVA DOCUMENTATION

Class Summary		***Page***
ByteBuffer	HRT tasks can share data through ByteBuffer instances. A ByteBuffer is a structure that contains a limited number of bytes.	*11*
DoubleBuffer	HRT tasks can share data through DoubleBuffer instances. A DoubleBuffer is a structure that contains a limited number of doubles.	*14*
FloatBuffer	HRT tasks can share data through FloatBuffer instances. A FloatBuffer is a structure that contains a limited number of floats.	*16*
HRTSystem		*19*
IntBuffer	HRT tasks can share data through IntBuffer instances. An IntBuffer is a structure that contains a limited number of integers.	*22*
LongBuffer	HRT tasks can share data through LongBuffer instances. A LongBuffer is a structure that contains a limited number of longs.	*24*
Monitor	Monitors may be used to synchronize critical sections using the ceiling priority protocol. Tasks can synchronized only on instances of this class.	*26*
ObjectBuffer	HRT tasks can share data through ObjectBuffer instances. An ObjectBuffer is a structure that contains a limited number of objects.	*27*
Task	A Task is a periodic cyclic task.	*30*

Exception Summary		***Page***
DeadlineException	This exception is thrown when the current Task has not ended its execution before its deadline.	*13*
HRTException	This exception is thrown when a Task does not respect an HRT constraint.	*18*
PeriodException	This exception is thrown when period of the current task is lowest than its minimum period.	*29*
WCAException	This exception is thrown when the current task allocates more than its WCA.	*38*

Error Summary		***Page***
IllegalAssignmentError	The exception thrown on an attempt to make an illegal assignment.	*21*

Class ByteBuffer

ej.hrt

```
java.lang.Object
  └
    ej.hrt.ByteBuffer
```

```
public class ByteBuffer
extends Object
```

HRT tasks can share data through ByteBuffer instances.
A ByteBuffer is a structure that contains a limited number of bytes. These elements are ordered in a first-in-first-out (FIFO) manner.

Field Summary		*Page*
`static int`	**DROP_OLDEST_ON_FULL_POLICY** Drop oldest element when buffer is full and an element is added.	*11*
`static int`	**SKIP_DATA_ON_FULL_POLICY** When buffer is full, no element can be added.	*11*

Constructor Summary	*Page*
ByteBuffer`(Monitor lock, int size, int policy)` Constructs a ByteBuffer initialized with a buffer of `size` elements.	*12*

Method Summary		*Page*
`boolean`	**add**`(byte b)` Add an element in this buffer.	*12*
`int`	**available**`()` Returns the number of bytes that are stored in this buffer. This method may be used before calling remove() to know if buffer is empty or not.	*12*
`byte`	**remove**`()` Remove the oldest element from this buffer and returns it. This method is not blocking.	*12*

Field Detail

DROP_OLDEST_ON_FULL_POLICY

```
public static final int DROP_OLDEST_ON_FULL_POLICY
```

Drop oldest element when buffer is full and an element is added.

SKIP_DATA_ON_FULL_POLICY

```
public static final int SKIP_DATA_ON_FULL_POLICY
```

When buffer is full, no element can be added.

Constructor Detail

ByteBuffer

```
public ByteBuffer(Monitor lock,
                  int size,
                  int policy)
```

Constructs a ByteBuffer initialized with a buffer of `size` elements.

Parameters:

`lock` - monitor used to synchronize accesses to the buffer.
`size` - maximum number of byte that can be stored in the buffer.
`policy` - DROP_OLDEST_ON_FULL_POLICY or SKIP_DATA_ON_FULL_POLICY

Method Detail

add

```
public boolean add(byte b)
```

Add an element in this buffer. This method is not blocking.
Behavior when this buffer is full depends on the buffer policy.

Parameters:

`b` - the byte to add in the buffer.

Returns:

true if the buffer was not full when adding the element, otherwise returns false.

---

remove

```
public byte remove()
```

Remove the oldest element from this buffer and returns it.
This method is not blocking.

Returns:

the oldest element of this buffer or 0 if the buffer is empty.

---

available

```
public int available()
```

Returns the number of bytes that are stored in this buffer.
This method may be used before calling remove() to know if buffer is empty or not.

Returns:

the number of bytes that are stored in this buffer.

Class DeadlineException

ej.hrt

```
java.lang.Object
  └─
    java.lang.Throwable
      └─
        java.lang.Exception
          └─
            java.lang.RuntimeException
              └─
                ej.hrt.HRTException
                  └─
                    ej.hrt.DeadlineException
```

All Implemented Interfaces:

Serializable

```
public class DeadlineException
extends HRTException
```

This exception is thrown when the current Task has not ended its execution before its deadline.

Constructor Summary	*Page*
DeadlineException`()`	*13*

Constructor Detail

DeadlineException

```
public DeadlineException()
```

Class DoubleBuffer

ej.hrt

```
java.lang.Object
  └
    ej.hrt.DoubleBuffer
```

```
public class DoubleBuffer
extends Object
```

HRT tasks can share data through DoubleBuffer instances.
A DoubleBuffer is a structure that contains a limited number of doubles. These elements are ordered in a first-in-first-out (FIFO) manner.

Field Summary		*Page*
`static int`	**DROP_OLDEST_ON_FULL_POLICY** Drop oldest element when buffer is full and an element is added.	14
`static int`	**SKIP_DATA_ON_FULL_POLICY** When buffer is full, no element can be added.	14

Constructor Summary	*Page*
DoubleBuffer`(`Monitor` lock, int size, int policy)` Constructs a DoubleBuffer initialized with a buffer of `size` elements.	15

Method Summary		*Page*
`boolean`	**add**`(double d)` Add an element in this buffer.	15
`int`	**available**`()` Returns the number of doubles that are stored in this buffer. This method may be used before calling remove() to know if buffer is empty or not.	15
`double`	**remove**`()` Remove the oldest element from this buffer and returns it. This method is not blocking.	15

Field Detail

DROP_OLDEST_ON_FULL_POLICY

`public static final int` **`DROP_OLDEST_ON_FULL_POLICY`**

Drop oldest element when buffer is full and an element is added.

SKIP_DATA_ON_FULL_POLICY

`public static final int` **`SKIP_DATA_ON_FULL_POLICY`**

When buffer is full, no element can be added.

Constructor Detail

DoubleBuffer

```
public DoubleBuffer(Monitor lock,
                    int size,
                    int policy)
```

Constructs a DoubleBuffer initialized with a buffer of `size` elements.

Parameters:

`lock` - monitor used to synchronize accesses to the buffer.
`size` - maximum number of doubles that can be stored in the buffer.
`policy` - DROP_OLDEST_ON_FULL_POLICY or SKIP_DATA_ON_FULL_POLICY

Method Detail

add

```
public boolean add(double d)
```

Add an element in this buffer. This method is not blocking.
Behavior when this buffer is full depends on the buffer policy.

Parameters:

`d` - the double to add in the buffer.

Returns:

true if the buffer was not full when adding the element, otherwise returns false.

---

remove

```
public double remove()
```

Remove the oldest element from this buffer and returns it.
This method is not blocking.

Returns:

the oldest element of this buffer or `Double.NaN` if the buffer is empty.

---

available

```
public int available()
```

Returns the number of doubles that are stored in this buffer.
This method may be used before calling remove() to know if buffer is empty or not.

Returns:

the number of doubles that are stored in this buffer.

Class FloatBuffer

ej.hrt

```
java.lang.Object
  └─
    ej.hrt.FloatBuffer
```

```
public class FloatBuffer
extends Object
```

HRT tasks can share data through FloatBuffer instances.
A FloatBuffer is a structure that contains a limited number of floats. These elements are ordered in a first-in-first-out (FIFO) manner.

Field Summary		Page
`static int`	**DROP_OLDEST_ON_FULL_POLICY** Drop oldest element when buffer is full and an element is added.	16
`static int`	**SKIP_DATA_ON_FULL_POLICY** When buffer is full, no element can be added.	16

Constructor Summary	Page
FloatBuffer`(`Monitor` lock, int size, int policy)` Constructs a FloatBuffer initialized with a buffer of `size` elements.	17

Method Summary		Page
`boolean`	**add**`(float f)` Add an element in this buffer.	17
`int`	**available**`()` Returns the number of floats that are stored in this buffer. This method may be used before calling remove() to know if buffer is empty or not.	17
`float`	**remove**`()` Remove the oldest element from this buffer and returns it. This method is not blocking.	17

Field Detail

DROP_OLDEST_ON_FULL_POLICY

```
public static final int DROP_OLDEST_ON_FULL_POLICY
```

Drop oldest element when buffer is full and an element is added.

SKIP_DATA_ON_FULL_POLICY

```
public static final int SKIP_DATA_ON_FULL_POLICY
```

When buffer is full, no element can be added.

Constructor Detail

FloatBuffer

```
public FloatBuffer(Monitor lock,
                   int size,
                   int policy)
```

Constructs a FloatBuffer initialized with a buffer of `size` elements.

Parameters:
`lock` - monitor used to synchronize accesses to the buffer.
`size` - maximum number of float that can be stored in the buffer.
`policy` - DROP_OLDEST_ON_FULL_POLICY or SKIP_DATA_ON_FULL_POLICY

Method Detail

add

```
public boolean add(float f)
```

Add an element in this buffer. This method is not blocking.
Behavior when this buffer is full depends on the buffer policy.

Parameters:
`f` - the float to add in the buffer.
Returns:
true if the buffer was not full when adding the element, otherwise returns false.

---

remove

```
public float remove()
```

Remove the oldest element from this buffer and returns it.
This method is not blocking.

Returns:
the oldest element of this buffer or `Float.NaN` if the buffer is empty.

---

available

```
public int available()
```

Returns the number of floats that are stored in this buffer.
This method may be used before calling remove() to know if buffer is empty or not.

Returns:
the number of floats that are stored in this buffer.

Class HRTException

ej.hrt

```
java.lang.Object
  └
    java.lang.Throwable
      └
        java.lang.Exception
          └
            java.lang.RuntimeException
              └
                ej.hrt.HRTException
```

All Implemented Interfaces:

Serializable

Direct Known Subclasses:

DeadlineException, PeriodException, WCAException

```
public class HRTException
extends RuntimeException
```

This exception is thrown when a Task does not respect an HRT constraint.

Constructor Summary	*Page*
HRTException ()	*18*
HRTException (String msg)	*18*

Constructor Detail

HRTException

```
public HRTException()
```

HRTException

```
public HRTException(String msg)
```

Class HRTSystem

ej.hrt

```
java.lang.Object
  └
    ej.hrt.HRTSystem
```

```
public class HRTSystem
extends Object
```

Constructor Summary	Page
HRTSystem()	19

Method Summary		Page
static long	freeHRTMemory() Returns the amount of free memory in the HRT heap.	19
static long	nanoTime() Returns the current value of the most precise available system timer, in nanoseconds. The value returned represents nanoseconds since some fixed but arbitrary time.	19
static long	totalHRTMemory() Returns the total amount of memory in the HRT heap.	19

Constructor Detail

HRTSystem

```
public HRTSystem()
```

Method Detail

freeHRTMemory

```
public static long freeHRTMemory()
```

Returns the amount of free memory in the HRT heap.

---

totalHRTMemory

```
public static long totalHRTMemory()
```

Returns the total amount of memory in the HRT heap. Note that the amount of memory required to hold an object of any given type may be implementation-dependent.

---

nanoTime

```
public static long nanoTime()
```

Returns the current value of the most precise available system timer, in nanoseconds.

The value returned represents nanoseconds since some fixed but arbitrary time.

Class IllegalAssignmentError

ej.hrt

```
java.lang.Object
  └─
    java.lang.Throwable
      └─
        java.lang.Error
          └─
            ej.hrt.IllegalAssignmentError
```

All Implemented Interfaces:

Serializable

```
public class IllegalAssignmentError
extends Error
```

The exception thrown on an attempt to make an illegal assignment.

Constructor Summary	*Page*
IllegalAssignmentError()	*21*

Constructor Detail

IllegalAssignmentError

```
public IllegalAssignmentError()
```

Class IntBuffer

ej.hrt

```
java.lang.Object
  └
    ej.hrt.IntBuffer
```

```
public class IntBuffer
extends Object
```

HRT tasks can share data through IntBuffer instances.
An IntBuffer is a structure that contains a limited number of integers. These elements are ordered in a first-in-first-out (FIFO) manner.

Field Summary		Page
`static int`	**DROP_OLDEST_ON_FULL_POLICY** Drop oldest element when buffer is full and an element is added.	22
`static int`	**SKIP_DATA_ON_FULL_POLICY** When buffer is full, no element can be added.	22

Constructor Summary	Page
IntBuffer`(`Monitor` lock, int size, int policy)` Constructs an IntBuffer initialized with a buffer of `size` elements.	23

Method Summary		Page
`boolean`	**add**`(int i)` Add an element in this buffer.	23
`int`	**available**`()` Returns the number of integers that are stored in this buffer. This method may be used before calling remove() to know if buffer is empty or not.	23
`int`	**remove**`()` Remove the oldest element from this buffer and returns it. This method is not blocking.	23

Field Detail

DROP_OLDEST_ON_FULL_POLICY

```
public static final int DROP_OLDEST_ON_FULL_POLICY
```

Drop oldest element when buffer is full and an element is added.

SKIP_DATA_ON_FULL_POLICY

```
public static final int SKIP_DATA_ON_FULL_POLICY
```

When buffer is full, no element can be added.

Constructor Detail

IntBuffer

```
public IntBuffer(Monitor lock,
                 int size,
                 int policy)
```

Constructs an IntBuffer initialized with a buffer of `size` elements.

Parameters:

`lock` - monitor used to synchronize accesses to the buffer.
`size` - maximum number of integer that can be stored in the buffer.
`policy` - DROP_OLDEST_ON_FULL_POLICY or SKIP_DATA_ON_FULL_POLICY

Method Detail

add

```
public boolean add(int i)
```

Add an element in this buffer. This method is not blocking.
Behavior when this buffer is full depends on the buffer policy.

Parameters:

`i` - the integer to add in the buffer.

Returns:

true if the buffer was not full when adding the element, otherwise returns false.

---

remove

```
public int remove()
```

Remove the oldest element from this buffer and returns it.
This method is not blocking.

Returns:

the oldest element of this buffer or 0 if the buffer is empty.

---

available

```
public int available()
```

Returns the number of integers that are stored in this buffer.
This method may be used before calling remove() to know if buffer is empty or not.

Returns:

the number of integers that are stored in this buffer.

Class LongBuffer

ej.hrt

```
java.lang.Object
  └
    ej.hrt.LongBuffer
```

```
public class LongBuffer
extends Object
```

HRT tasks can share data through LongBuffer instances.
A LongBuffer is a structure that contains a limited number of longs. These elements are ordered in a first-in-first-out (FIFO) manner.

Field Summary		Page
static int	**DROP_OLDEST_ON_FULL_POLICY** Drop oldest element when buffer is full and an element is added.	24
static int	**SKIP_DATA_ON_FULL_POLICY** When buffer is full, no element can be added.	24

Constructor Summary	Page
LongBuffer(Monitor lock, int size, int policy) Constructs a LongBuffer initialized with a buffer of `size` elements.	25

Method Summary		Page
boolean	**add**(long l) Add an element in this buffer.	25
int	**available**() Returns the number of longs that are stored in this buffer. This method may be used before calling remove() to know if buffer is empty or not.	25
long	**remove**() Remove the oldest element from this buffer and returns it. This method is not blocking.	25

Field Detail

DROP_OLDEST_ON_FULL_POLICY

```
public static final int DROP_OLDEST_ON_FULL_POLICY
```

Drop oldest element when buffer is full and an element is added.

SKIP_DATA_ON_FULL_POLICY

```
public static final int SKIP_DATA_ON_FULL_POLICY
```

When buffer is full, no element can be added.

Constructor Detail

LongBuffer

```
public LongBuffer(Monitor lock,
                  int size,
                  int policy)
```

Constructs a LongBuffer initialized with a buffer of `size` elements.

Parameters:

`lock` - monitor used to synchronize accesses to the buffer.
`size` - maximum number of longs that can be stored in the buffer.
`policy` - DROP_OLDEST_ON_FULL_POLICY or SKIP_DATA_ON_FULL_POLICY

Method Detail

add

```
public boolean add(long l)
```

Add an element in this buffer. This method is not blocking.
Behavior when this buffer is full depends on the buffer policy.

Parameters:

`l` - the long to add in the buffer.

Returns:

true if the buffer was not full when adding the element, otherwise returns false.

---

remove

```
public long remove()
```

Remove the oldest element from this buffer and returns it.
This method is not blocking.

Returns:

the oldest element of this buffer or 0 if the buffer is empty.

---

available

```
public int available()
```

Returns the number of longs that are stored in this buffer.
This method may be used before calling remove() to know if buffer is empty or not.

Returns:

the number of longs that are stored in this buffer.

Class Monitor

ej.hrt

```
java.lang.Object
  └─
    ej.hrt.Monitor
```

```
public class Monitor
extends Object
```

Monitors may be used to synchronize critical sections using the ceiling priority protocol.
Tasks can synchronized only on instances of this class.

Constructor Summary	Page
Monitor(int ceilingPriority) Creates a new monitor with the given ceilingPriority.	26

Method Summary		Page
int	getUID() Gets the unique ID of the monitor.	26

Constructor Detail

Monitor

```
public Monitor(int ceilingPriority)
```

Creates a new monitor with the given ceilingPriority. The ceiling priority of a monitor must be included between Task.MIN_HRT_TASK_PRIORITY and Task.MAX_HRT_TASK_PRIORITY.

Parameters:
`ceilingPriority` - the ceiling priority of this monitor

Throws:
`IllegalArgumentException` - if ceilingPriority is not between Task.MIN_HRT_TASK_PRIORITY and Task.MAX_HRT_TASK_PRIORITY included

Method Detail

getUID

```
public int getUID()
```

Gets the unique ID of the monitor.

Returns:
the unique ID of the monitor

Class ObjectBuffer

ej.hrt

```
java.lang.Object
  └
    ej.hrt.ObjectBuffer
```

```
public class ObjectBuffer
extends Object
```

HRT tasks can share data through ObjectBuffer instances.
An ObjectBuffer is a structure that contains a limited number of objects. These elements are ordered in a first-in-first-out (FIFO) manner.

Field Summary		*Page*
`static int`	**DROP_OLDEST_ON_FULL_POLICY** Drop oldest element when buffer is full and an element is added.	*27*
`static int`	**SKIP_DATA_ON_FULL_POLICY** When buffer is full, no element can be added.	*27*

Constructor Summary	*Page*
ObjectBuffer`(`Monitor` lock, int size, int policy)` Constructs an ObjectBuffer initialized with a buffer of `size` elements.	*28*

Method Summary		*Page*
`boolean`	**add**`(Object o)` Add an element in this buffer.	*28*
`Object`	**remove**`()` Remove the oldest element from this buffer and returns it. This method is not blocking.	*28*

Field Detail

DROP_OLDEST_ON_FULL_POLICY

`public static final int` **`DROP_OLDEST_ON_FULL_POLICY`**

Drop oldest element when buffer is full and an element is added.

SKIP_DATA_ON_FULL_POLICY

`public static final int` **`SKIP_DATA_ON_FULL_POLICY`**

When buffer is full, no element can be added.

Constructor Detail

ObjectBuffer

```
public ObjectBuffer(Monitor lock,
                    int size,
                    int policy)
```

Constructs an ObjectBuffer initialized with a buffer of `size` elements.

Parameters:
`lock` - monitor used to synchronize accesses to the buffer.
`size` - maximum number of objects that can be stored in the buffer.
`policy` - DROP_OLDEST_ON_FULL_POLICY or SKIP_DATA_ON_FULL_POLICY

Method Detail

add

```
public boolean add(Object o)
```

Add an element in this buffer. This method is not blocking.
Behavior when this buffer is full depends on the buffer policy.

Parameters:
`o` - the object to add in the buffer.
Returns:
true if the buffer was not full when adding the element, otherwise returns false.

---

remove

```
public Object remove()
```

Remove the oldest element from this buffer and returns it.
This method is not blocking.

Returns:
the oldest element of this buffer or null if the buffer is empty.

Class PeriodException

ej.hrt

```
java.lang.Object
  └─
    java.lang.Throwable
      └─
        java.lang.Exception
          └─
            java.lang.RuntimeException
              └─
                ej.hrt.HRTException
                  └─
                    ej.hrt.PeriodException
```

All Implemented Interfaces:

Serializable

```
public class PeriodException
extends HRTException
```

This exception is thrown when period of the current task is lowest than its minimum period.

Constructor Summary	*Page*
PeriodException()	*29*

Constructor Detail

PeriodException

```
public PeriodException()
```

Class Task

ej.hrt

```
java.lang.Object
  └─
    ej.hrt.Task
```

```
abstract public class Task
extends Object
```

A Task is a periodic cyclic task. Each task allocates a new thread of execution. This is used for real-time systems.

Field Summary		Page
static int	MAX_HRT_TASK_PRIORITY	31
static int	MIN_HRT_TASK_PRIORITY	31

Constructor Summary	Page
Task(String name, long periodMin, int priority) Creates a cyclic task with a deadline equals to `periodMin`.	31
Task(String name, long periodMin, long deadline, int priority) Creates a cyclic task.	32

Method Summary		Page
Thread	asThread()	35
static Task	currentTask() Returns a reference to the currently executing Task object. If the current thread is not a Task instance, this method returns null.	37
long	getCurrentWCA()	35
static long	getCurrentWCA(int taskUID) Calls the getCurrentWCA() method of the task with the specified UID.	36
long	getCurrentWCET()	35
static long	getCurrentWCET(int taskUID) Calls the getCurrentWCET() method of the task with the specified UID.	36
long	getDeadline() Gets the deadline of the task.	33
String	getName() Gets the name of the task.	32
int	getNbDeadlinesReached()	35
static int	getNbDeadlinesReached(int taskUID) Calls the getNbDeadlinesReached() method of the task with the specified UID.	37
long	getPeriod() Gets the minimum period of the task.	32
int	getPriority() Gets the priority of the task.	32
int	getUID() Gets the unique ID of this task.	34
boolean	isInRescueMode() Tests if this task is in rescue mode.	35

`static boolean`	**isInRescueMode**`(int taskUID)` Calls the isInRescueMode() method of the task with the specified UID.	*36*
`void`	**outOfResource**`(`DeadlineException` exception)` Method called when this Task terminates due to the given uncaught DeadlineException exception.	*33*
`void`	**outOfResource**`(`WCAException` exception)` Method called when this Task terminates due to the given uncaught WCAException exception.	*33*
`static void`	**registerCyclicTask**`(int taskUID, long increment, long period)` Calls the registerCyclicTask(long, long) method of the task with the specified UID.	*36*
`void`	**registerCyclicTask**`(long increment, long period)` Starts a task cycle in increment nanoseconds and then every period nanoseconds. If the task is already registered as a cyclic task, this method modifies timing configuration.	*34*
`abstract void`	**run**`()` Starting the task causes this method run() to be called in a separately executing thread.	*33*
`void`	**start**`()` Starts a task cycle.	*33*
`static void`	**start**`(`Task`[] tasks)` Starts all the provided tasks at the very same time.	*33*
`static void`	**start**`(int id)` Calls the start() method of the task with the specified UID.	*34*
`void`	**uncaughtException**`(Throwable exception)` Method called when this Task terminates due to the given uncaught exception.	*34*
`void`	**unregisterCyclicTask**`()` Disables the automatic start for this task.	*35*
`static void`	**unregisterCyclicTask**`(int taskUID)` Calls the unregisterCyclicTask() method of the task with the specified UID.	*36*

Field Detail

MIN_HRT_TASK_PRIORITY

```
public static final int MIN_HRT_TASK_PRIORITY
```

MAX_HRT_TASK_PRIORITY

```
public static final int MAX_HRT_TASK_PRIORITY
```

Constructor Detail

Task

```
public Task(String name,
            long periodMin,
            int priority)
```

Creates a cyclic task with a deadline equals to `periodMin`. Equivalent to `new Task(name, periodMin, periodMin, priority)`.

Parameters:

`name` - the name of the task
`periodMin` - the minimum period of the task (in nanoseconds)

Throws:

`IllegalArgumentException` - if the given priority is not in the range MIN_HRT_TASK_PRIORITY to MAX_HRT_TASK_PRIORITY.
HRTException - if the system is not in initialization mode.

Task

```
public Task(String name,
            long periodMin,
            long deadline,
            int priority)
```

Creates a cyclic task. If `deadline` is equals to 0, system will not check for deadline exception for this task.

Parameters:

`name` - the name of the task
`periodMin` - the minimum period of the task (in nanoseconds)
`deadline` - the duration of the deadline of the task (in nanoseconds)

Throws:

`IllegalArgumentException` - if deadline is higher than periodMin,
if the given priority is not in the range MIN_HRT_TASK_PRIORITY to MAX_HRT_TASK_PRIORITY.
HRTException - if the system is not in initialization mode.

Method Detail

getName

```
public String getName()
```

Gets the name of the task.

Returns:

the name of the task

---

getPriority

```
public int getPriority()
```

Gets the priority of the task.

Returns:

the priority of the task

---

getPeriod

```
public long getPeriod()
```

Gets the minimum period of the task.

Returns:

the minimum period of the task (in nanoseconds)

---

getDeadline

```
public long getDeadline()
```

Gets the deadline of the task.

Returns:
the duration of the deadline of the task (in nanoseconds)

---

start

```
public final void start()
```

Starts a task cycle. If this thread is running, the start is recorded and performed later.

Throws:
HRTException - if the task is registered as cyclic task

---

start

```
public static void start(Task[] tasks)
```

Starts all the provided tasks at the very same time.

Throws:
HRTException - if one the tasks is registered as cyclic task

---

run

```
public abstract void run()
```

Starting the task causes this method run() to be called in a separately executing thread. Subclasses of Task must override this method.

---

outOfResource

```
public void outOfResource(WCAException exception)
```

Method called when this Task terminates due to the given uncaught WCAException exception. This method invoke the uncaughtException() method with the given exception.
This method should be overridden by subclasses in order to define behavior of the task in downgraded mode.
Any exception thrown by this method will be ignored by the system.

Parameters:
`exception` - the exception

---

outOfResource

```
public void outOfResource(DeadlineException exception)
```

Method called when this Task terminates due to the given uncaught DeadlineException exception. This method invoke the uncaughtException() method with the given exception.
This method should be overridden by subclasses in order to define behavior of the task in downgraded mode.
Any exception thrown by this method will be ignored by the system.

Parameters:
`exception` - the exception

---

uncaughtException

```
public void uncaughtException(Throwable exception)
```

Method called when this Task terminates due to the given uncaught exception. This method invoke the printStackTrace() method on the given exception.
This method should be overridden by subclasses in order to define behavior of the task in downgraded mode.
Any exception thrown by this method will be ignored by the system.

Parameters:
`exception` - the exception

---

getUID

```
public int getUID()
```

Gets the unique ID of this task.

Returns:
the unique ID of this task

---

start

```
public static void start(int id)
```

Calls the start() method of the task with the specified UID.

Parameters:
`id` - the unique ID of the task to start
Throws:
`IllegalArgumentException` - if the specified UID is not valid
HRTException - if the task is registered as cyclic task

---

registerCyclicTask

```
public void registerCyclicTask(long increment,
                               long period)
```

Starts a task cycle in increment nanoseconds and then every period nanoseconds.
If the task is already registered as a cyclic task, this method modifies timing configuration.

Parameters:
`increment` - time before first start in nanoseconds
`period` - time between each starts in nanoseconds

Throws:

PeriodException - if `period` is lower than period of this task.

---

unregisterCyclicTask

`public void` **`unregisterCyclicTask`**`()`

Disables the automatic start for this task.

---

isInRescueMode

`public boolean` **`isInRescueMode`**`()`

Tests if this task is in rescue mode.

Returns:

true if the task is in rescue mode, false otherwise.

---

getCurrentWCET

`public long` **`getCurrentWCET`**`()`

Returns:

Maximum cycle time for current execution in nanoseconds.

---

getCurrentWCA

`public long` **`getCurrentWCA`**`()`

Returns:

Maximum cycle allocation for current execution in bytes.

---

getNbDeadlinesReached

`public int` **`getNbDeadlinesReached`**`()`

Returns:

number of deadlines reached for current execution.

---

asThread

`public Thread` **`asThread`**`()`

Returns:

the Thread instance that will execute this task.

---

registerCyclicTask

```
public static void registerCyclicTask(int taskUID,
                                      long increment,
                                      long period)
```

Calls the registerCyclicTask(long, long) method of the task with the specified UID.

Parameters:
`taskUID` - the unique ID of the task

Throws:
`IllegalArgumentException` - if the specified UID is not valid

---

unregisterCyclicTask

```
public static void unregisterCyclicTask(int taskUID)
```

Calls the unregisterCyclicTask() method of the task with the specified UID.

Parameters:
`taskUID` - the unique ID of the task

Throws:
`IllegalArgumentException` - if the specified UID is not valid

---

isInRescueMode

```
public static boolean isInRescueMode(int taskUID)
```

Calls the isInRescueMode() method of the task with the specified UID.

Parameters:
`taskUID` - the unique ID of the task

Throws:
`IllegalArgumentException` - if the specified UID is not valid

---

getCurrentWCET

```
public static long getCurrentWCET(int taskUID)
```

Calls the getCurrentWCET() method of the task with the specified UID.

Parameters:
`taskUID` - the unique ID of the task

Throws:
`IllegalArgumentException` - if the specified UID is not valid

---

getCurrentWCA

```
public static long getCurrentWCA(int taskUID)
```

Calls the getCurrentWCA() method of the task with the specified UID.

Parameters:
`taskUID` - the unique ID of the task
Throws:
`IllegalArgumentException` - if the specified UID is not valid

---

getNbDeadlinesReached

`public static int getNbDeadlinesReached(int taskUID)`

Calls the getNbDeadlinesReached() method of the task with the specified UID.

Parameters:
`taskUID` - the unique ID of the task
Throws:
`IllegalArgumentException` - if the specified UID is not valid

---

currentTask

`public static Task currentTask()`

Returns a reference to the currently executing Task object.
If the current thread is not a Task instance, this method returns null.

Class WCAException

ej.hrt

```
java.lang.Object
  └─ java.lang.Throwable
      └─ java.lang.Exception
          └─ java.lang.RuntimeException
              └─ ej.hrt.HRTException
                  └─ ej.hrt.WCAException
```

All Implemented Interfaces:

Serializable

```
public class WCAException
extends HRTException
```

This exception is thrown when the current task allocates more than its WCA.

Constructor Summary	*Page*
WCAException()	*38*

Constructor Detail

WCAException

```
public WCAException()
```