AERO: Architecture for Enhanced Reprogrammability and Operability

Contract ESTEC 15750/02/NL/LVH

Technical Note 2
Enhancements of the JVM

Fridtjof Siebert, aicas

Tobias Ritzau, Linköping Universitet
Frederic Deladerriere, Astrium

Reference: AERO/TN2

Issue: 0.2

Date: **2003-2-13**

Abstract:

This document presents the modifications and add-ons made to the JamaicaVM virtual machine during the AERO project that result in the JamaicaVM/AERO JVM. The AERO JVM was based on the JamaicaVM and extended by features to be compliant with the requirements of the space domain.

In an validation process, the AERO JVM was validated and evaluated to verify that it is compliant with these requirements.

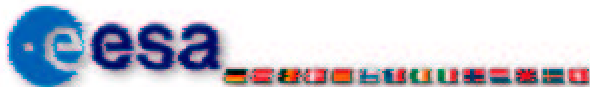
The major enhancements made on the basis JamaicaVM are in two areas: First, support for execution on an VxWorks/ERC32 environment via a target specific layer for this environment. Second, support for standard APIs as required in the space domain. Particularly, a subset of the Java standard libraries plus an implementation of a subset of the Real-Time Specification for Java (RTSJ) classes.

During the project, enhancements were also made within the framework of the EC IST-funded HIDOORS project . This projet contributed significantly to the RTSJ support within AERO JVM.

Written by:	**Name**	**Company**	**Signature**	**Internal reference**
	Fridtjof Siebert	aicas		

AERO (Architecture for Enhanced Reprogrammability and Operability) is an ESA project (Contract ESTEC 15750/02/NL/LVH) conducted by a consortium led by ASTRIUM-SAS with aicas GmbH and Linköping Universitet.

For more information please contact:



Frank J. de Bruin
ESTEC. Keplerlaan 1. PO Box 299
2200 AG Noordwijk ZH - The Netherlands
Tel: +31 (0) 71 565 4951. Fax: +31 (0) 71 565 5420
e-mail fdebruin@estec.esa.nl

Frédéric Deladerrière
ASTRIUM
31, avenue des cosmonautes
F-31 402 Toulouse Cedex 4, France
Tel: +33 5 62 19 56 49. Fax: +33 5 62 19 78 97
e-mail: frederic.deladerriere@astrium-space.com

Fridtjof Siebert
AICAS GmbH
Haid-und-Neu-Str. 18
D-76131 Karlsruhe, Germany
Tel: +49 721 663 968-23 Fax: +49 721 663 968-93
e-mail: siebert@aicas.com

Tobias Ritzau
Linköping Universitet
Dep. Of Computer and Information Science
SE-58183 Linköping, Sweden
Tel: +46 13 28 4494. Fax: +46 13 28 5899
e-mail: tobri@ida.liu.se

Revision History

Version	Date	Paragraphs modified	Comments
0.1	2002-11-30		First issue
0.2	2003-2-14	2.1.1, 2.2.1, 3.3.3, A.1, A.3, A.5, B	Changes as requested at CDR meeting

Table of Contents

1. Introduction....7
1.1 Scope....7
1.1.1 Scope of the Project....7
1.1.2 Scope of the Document....7
1.2 Related Documentation....7
1.3 Definition of Terms and Acronyms....8
1.3.1 Definition of Terms....8
1.3.2 Acronyms and Abbreviations....8
2. Modifications for the Builder tool 9
2.1 Overview over the Builder tool....10
2.1.1 Design of the Builder....11
2.2 Modifications and Additions made on the Builder Tool....12
2.2.1 Lists of included/excluded methods and fields after Smart Linking....12
2.2.2 New Option -showIncludedFeatures 12
2.2.3 New Option -showExcludedFeatures 13
2.2.4 Java Priority Mapping to System Priorities....14
2.2.5 Strict Semantics of Real-Time Specification for Java's....15
2.2.6 Static Garbage Collection support....15
3. Real-Time Specification for Java Implementation....16
3.1 Scheduling....17
3.1.1 Realtime and NoHeapRealtimeThreads....17
3.1.2 NoHeapRealtimeThread scheduling....18
3.2 Asychronous Events....20
3.2 Asynchronous Transfer of Control....21
3.3 Memory....23
3.3.1 Scoped Memory....23
3.3.2 Immortal Memory....24
3.3.3 Memory Types for Objects....24
3.3.4 Assignment checks for reference stores....25
3.3.5 Memory Areas and NoHeapRealtimeThreads....26
3.3.6 Physical Memory....26
3.3.4 Allocation budgets and allocation rates....27
4. VxWorks/ERC32 support....28
4.1 Overview over Target OS dependent layer....28
4.1.1 Hierarchical structure of target dependent code....28
4.2 VxWorks/ERC32 port....30
4.2.1 config.h for VxWorks/ERC32....30

4.2.2 Native threads for VxWorks/ERC32 31
4.2.3 Native semaphores for VxWorks/ERC32 31
4.3 RTEMS/ERC32 port 32
Appendix A: Builder Usage 33
A.1 General 33
A.2 Classes, files, and paths 34
A.3 Smart linking and compaction 35
A.4 Java standard API configurations 37
A.5 Compilation and optimization 37
A.6 Memory and threads settings 40
A.7 Profiling 44
A.8 Native code 44
Appendix B: The Real-Time Specification for Java APIs 46
B.1 High Precision Timers and Clocks 46
B.2 Realtime Threads 47
B.3 Asynchronous Events 48
B.4 Scheduler and Schedulability 49
B.5 Asynchronous Transfer of Control and Thread Termination 51
B.6 Control over Java Monitor Behaviour 52
B.7 Memory Management Related APIs 52
B.8 Additional Runtime Errors and Exceptions 55
B.9 Other APIs 56

page intentionally left blank

1. Introduction

1.1 Scope

1.1.1 Scope of the Project

AERO (Architecture for Enhanced Reprogrammability and Operability) is an ESA project (contract ESTEC 15750/02/NL/LVH). The objectives of the project are to investigate on a real-time Java virtual machine for ERC32. Special attention is put on the garbage collection mechanism and deterministic execution model.

The project is split in two phases. The first phase investigates existing virtual machines to choose a potential candidate that will be customised, are then investigated the definition of requirements concerning a real-time interpreter in on-board systems. An implementation plan is proposed for the second phase. This second phase is dedicated to the definition of software functions of the real-time Java virtual machine and to their implementation and assessment through validation tests.

1.1.2 Scope of the Document

This document is output of task 1.b.1 "Software JVM Development".

This document presents the modifications and add-ons made to the JamaicaVM virtual machine during the AERO project that result in the JamaicaVM/AERO JVM. The AERO JVM was based on the JamaicaVM and extended by features to be compliant with the requirements of the space domain.

In an validation process, the AERO JVM was validated and evaluated to verify that it is compliant with these requirements.

The major enhancements made on the basis JamaicaVM are in two areas: First, support for execution on an VxWorks/ERC32 environment via a target specific layer for this environment. Second, support for standard APIs as required in the space domain. Particularly, a subset of the Java standard libraries plus an implementation of a subset of the Real-Time Specification for Java (RTSJ) classes.

During the project, enhancements were also made within the framework of the EC IST-funded HIDOORS project . This projet contributed significantly to the RTSJ support within AERO JVM.

1.2 Related Documentation

[AERO]	Architecture for Enhanced Reprogrammability and Operability, ESTEC Contract n°15750/02/NL/LVH.
[AERO-DDD]	AERO DDD: JVM Detailed Design Document
[AERO-SP1]	AERO SP1: Specification
[AERO-TN3]	AERO TN3: Real-time/Garbage collection strategy.
[Prop]	Architecture for Enhanced Reprogrammability and Operability, Proposal for ESA ITT AO/1-3959/01/NL/PB. Astrium EEA.PR.FD.3682269.01.
[BKMS98]	David F. Bacon, Ravi Konuru, Chet Murthy, Mauricio Serrano: Thin Locks: Featherweight Synchronization for Java, PLDI, 1998
[DS84]	L. Peter Deutsch and Allan M. Schiff-man: Efficient Implementation of the Smalltalk-80 System, Conference Record of the Eleventh Annual ACM

Symposium on Principles of Programming Languages, pp. 297-302, Salt Lake City, UT, January, 1984

[Peren] http://www.peren.com

[PH] http://www.plumhall.com

[Jacks] http://www-124.ibm.com/developerworks/oss/cvs/jikes/~checkout~/jacks/

[Mauve] http://sources.redhat.com/mauve/

[RTSJ] The Real-Time Java Experts Group: The Real-Time Specification for Java, final version, December 2001. http://www.rtj.org/

[Strou87] Bjarne Stroustrup: Multiple Inheritance for C++, Proceedings of the European Unix Users Group Conference, pp. 189-207, Helsinki, May, 1987

[SW01] Fridtjof Siebert and Andy Walter: Deterministic Execution of Java's Primitive Bytecode Operations, Java Virtual Machine Research and Technology Symposium (JVM'01), Monterey, California, April 2001.

[Yang99] Byung-Sun Yang, Junpyo Lee, Jinpyo Park, Soo-Mook Moon, Kemal Ebcioglu, Erik Altmann: Lightweight Moni-tor for Java VM, ACMComputer Ar-chitecture News, Vol 27/1, March 1999.

1.3 Definition of Terms and Acronyms

1.3.1 Definition of Terms

Real-Time Specification of Java

A specification for realtime extensions of the Java standards APIs as defined in [RTSJ] and further discussed by the Real-Time Java Experts Group [RTJEG].

1.3.2 Acronyms and Abbreviations

ESA European Space Agency

ESTEC European Space Technological Centre

AERO Architecture for Enhanced Reprogrammability and Operability

AOT Ahead-of-Time (compiler)

API Application Program Interface

CDC Connected Device Configuration

CLDC Connected Limited Device Configuration

J2ME Java-2 Micro Edition

JDK Java Development Kit

JIT Just-in-Time (compiler)

JNI Java Native Interface

JVM Java Virtual Machine

RTSJ Real-Time Specification for Java

2. Modifications for the Builder tool

The central tool used for development is the builder. A comprehensive description of the usage of this tool is given in Appendix A. This section only presents the additions made during the AERO project and the options relevant for the Realtime Specification for Java support.

2.1 Overview over the Builder tool

The builder is an executable application that provides a common interface to all functions needed for building a Java application such that it can be executed on the target system. The use of this tool to build an image out of Java classes and put this image onto a target system is illustrated in **Figure 2.1.**

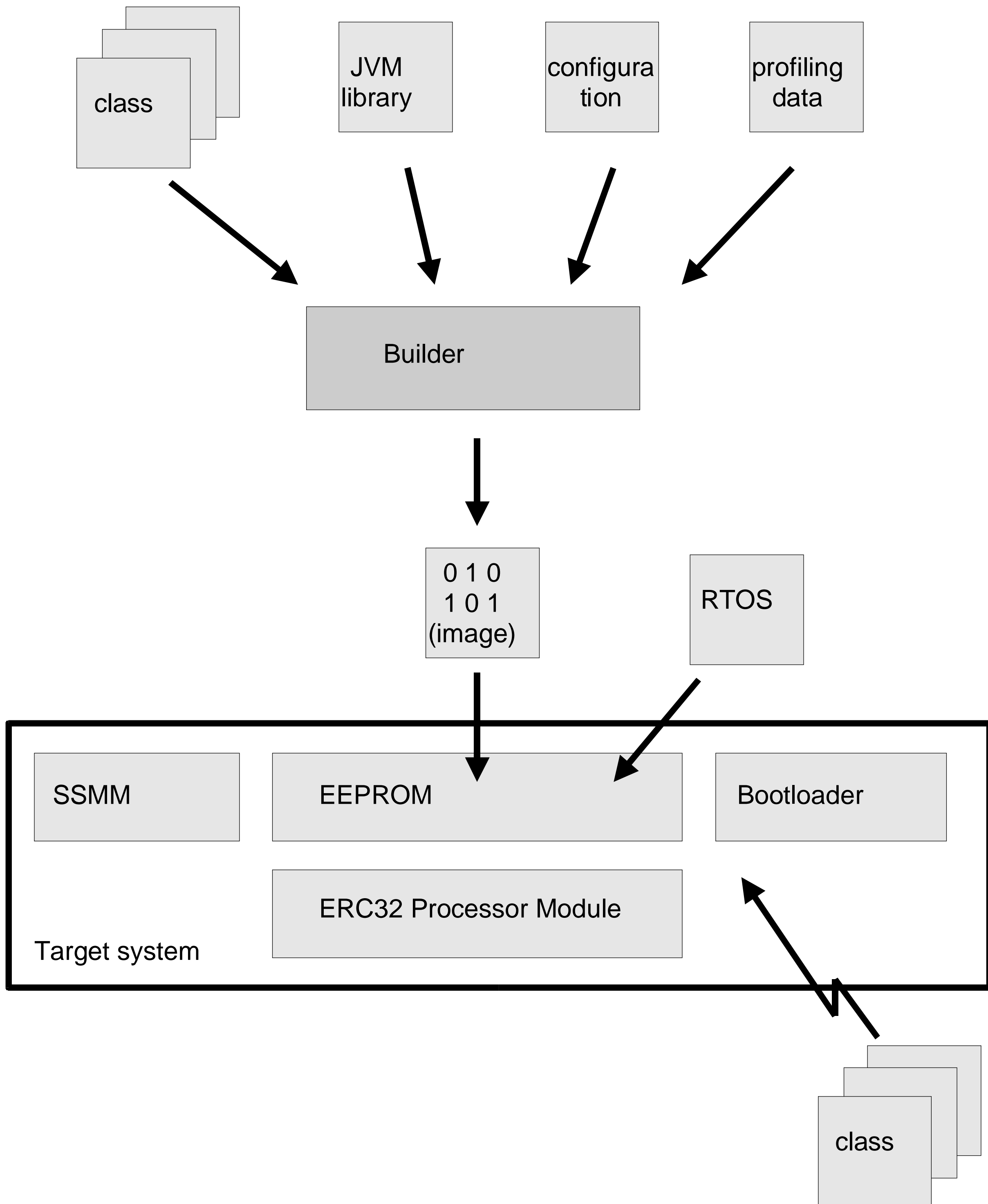


Figure 2.1: Building an image out of class files using the builder tool.

2.1.1 Design of the Builder

The process of building the image of the static part of the on-board Java software is illustrated in more detail in **Figure 2.2**.

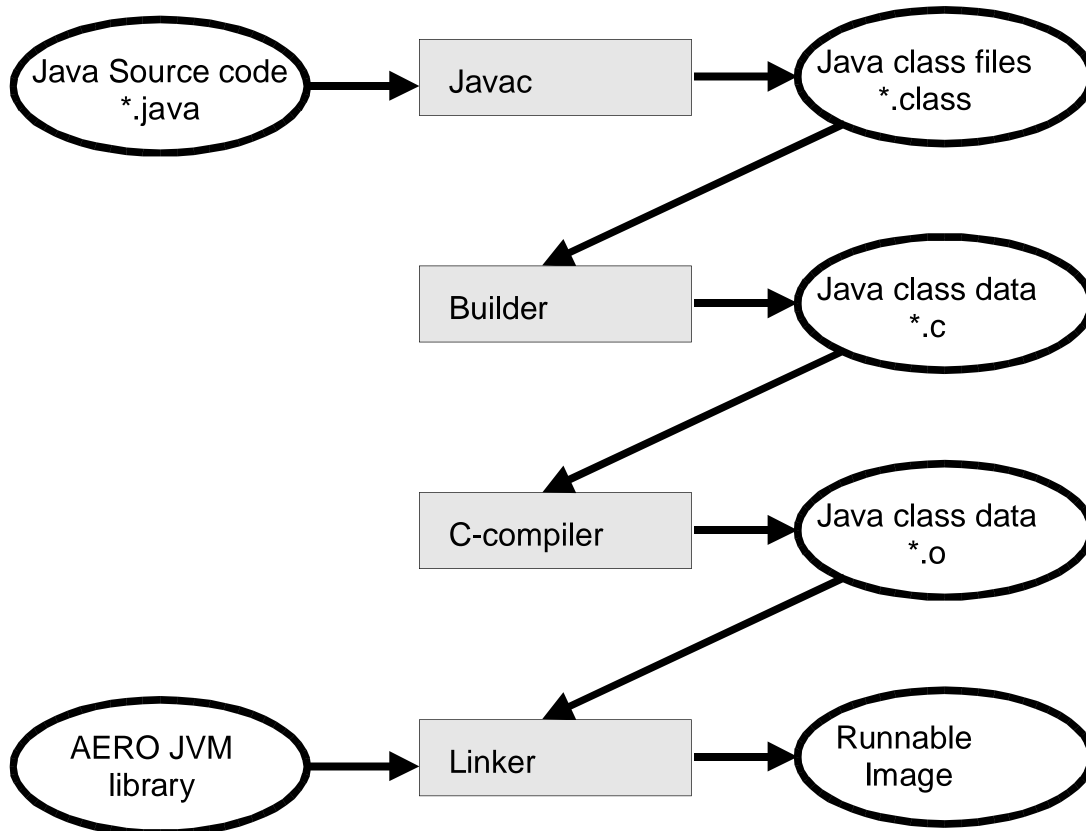


Figure 2.2: Operation of the Builder when creating an image for the static on-board code

The builder starts with the Java class files that were generated from the Java source code using a Java compiler such as *javac*. These files are then analysed and converted in C source code that contains the class file information and additional data such as the result of static garbage collection analysis and C source code for statically compiled Java methods. So the C code contains the information from the class files plus additional data structures to hold metadata plus, for all compiled methods, the C code generated from the original bytecode instructions by the static compiler.

The C code can then be translated in to target-dependent object code using a C-compiler. The result will be linked with a precompiled library containing the JVM to be executable on the target system.

The resulting image is stand alone, i.e., it can be executed without requiring to load additional data from a file system etc. This means this image can be put into ROM or FLASH memory on the target system.

Nevertheless, the dynamic class loading features of Java that are implemented in the AERO JVM permit that additional Java classes are added dynamically at run time. The code can be extended dynamically by new features.

The performance enhancement due to optimisations made by the builder, the static garbage collector and the static compiler are not directly available to code that will be loaded dynamically. But annotations of this code can bring these advantages even to dynamic code. Additionally, the dynamic code benefits from the optimisations made to the standard libraries that are in the static part.

2.2 Modifications and Additions made on the Builder Tool

Most modifications to the builder caused the introduction of additional command line arguments to this tool. The new features are detailed in the following text.

2.2.1 Lists of included/excluded methods and fields after Smart Linking

Two new options have been included to permit better documentation of the smart linking process. During smart linking, the application is analysed to determine what features of the application are actually used and need to be part of the built application. Features are methods or fields declared in the classes of the application.

If smart linking is enabled for a project, the builder removes unused code and performs better optimization to obtain smaller binary files, smaller memory usage and faster execution.

Smart linking must only be used for closed applications that do not use Java's reflection API (including reflection via the Java Native Interface JNI) and that do not download classes dynamically or use functions like java.lang.Class.forName(className) that cause loading of classes that is not visible to the builder tool. If smart linking is used for these applications, code that is referenced by classes that are loaded dynamically or through the reflection API might have been removed and methods that are redefined by dynamically loaded code might not be called correctly since -smart causes static binding of virtual or interface methods that are not redefined. The builder issues a warning if smart linking is used for such an application.

When dynamic class loading or reflection is used, the option -notSmart can be used to restrict the effect of smart linking and still profit from better optimization. All methods and fields of a class listed as an argument of the option -notSmart and all of its subclasses referenced by the application will be excluded from smart linking.

The new options -showIncludedFeatures and -showExcludedFeatures serve to document which methods and fields were included or excluded, respectively, during the smart linking process.

The smart linking process starts with a set of methods that are known to be potentially called during the program execution. This set contains the method main() of the main class of the application, since this method is the entry point into the application. The set also contains any code that is referenced internally by the virtual machine such as ClassLoader.loadClass, Thread.go or the initializers of internally thrown exceptions.

The set of called methods is then continuously extended by any methods that can be called from methods that are already in the set. In this process, dynamic binding is considered, i.e., any virtual call might cause the addition of all methods that redefine the called method and that are part of a class that might be instantiated at runtime.

The set of called methods grows until a complete call graph is generated and now new methods can be found. Then, no calls to any method that is not yet in the set of called methods can be performed during runtime. These methods are dead code and they can be removed.

Furthermore, a similar analysis can be performed for fields. All fields that are not accessed by any of the methods in the set of called methods will not be accessed during the execution of the application. These fields can consequently be removed from the resulting application. In addition to that, all fields that are written by a method in the set of called methods, but that are never read by any of these methods can also be removed since writing of these fields does not have an effect on the execution of the application. The write operations to fields that are never read must consequently be removed as well.

2.2.2 New Option -showIncludedFeatures

This option will cause the builder to display a list of all classes, methods and fields that will be part of the built application. Any classes, methods or fields removed from the target application through

mechanisms such as smart linking (as activated through option -smart) will not be displayed. In conjunction with option -smart and -notSmart, this helps one to identify what methods are included into the application

The output of this option consists of lines starting with the string INCLUDED CLASS, INCLUDED METHOD or INCLUDED FIELD followed by the name of a class or the name and signature of a method or field, respectively.

Example: The output of this option when applied to a small HelloWorld-Appliation looks like this:

```
> jamaica -smart HelloWorld -showIncludedFeatures
Jamaica Builder Tool 1.0.7 Release 1
Generating code for target 'linux-gnu-i686'
 + HelloWorld__.c
INCLUDED CLASS HelloWorld
INCLUDED METHOD HelloWorld.main([Ljava.lang.String;)V
INCLUDED CLASS java.io.BufferedReader
INCLUDED FIELD java.io.BufferedReader.inLjava.io.Reader;
INCLUDED FIELD java.io.BufferedReader.buffer[C
INCLUDED FIELD java.io.BufferedReader.posI
INCLUDED FIELD java.io.BufferedReader.limitI
INCLUDED FIELD java.io.BufferedReader.markPosI
INCLUDED METHOD java.io.BufferedReader.<init>(Ljava.io.Reader;)V
INCLUDED METHOD java.io.BufferedReader.<init>(Ljava.io.Reader;I)V
INCLUDED METHOD java.io.BufferedReader.read([CII)I
INCLUDED METHOD java.io.BufferedReader.fill()I
INCLUDED METHOD java.io.BufferedReader.read()I
INCLUDED METHOD java.io.BufferedReader.lineEnd(I)I
INCLUDED METHOD java.io.BufferedReader.readLine()Ljava.lang.String;
INCLUDED METHOD java.io.BufferedReader.checkStatus()V
INCLUDED CLASS java.io.ByteArrayInputStream
INCLUDED FIELD java.io.ByteArrayInputStream.buf[B
INCLUDED FIELD java.io.ByteArrayInputStream.posI
INCLUDED FIELD java.io.ByteArrayInputStream.countI
INCLUDED METHOD java.io.ByteArrayInputStream.<init>([B)V
INCLUDED METHOD java.io.ByteArrayInputStream.<init>([BII)V
INCLUDED METHOD java.io.ByteArrayInputStream.read()I
INCLUDED METHOD java.io.ByteArrayInputStream.read([BII)I
INCLUDED CLASS java.io.CharConversionException
INCLUDED METHOD java.io.CharConversionException.<init>(Ljava.lang.String;)V
INCLUDED CLASS java.io.File
INCLUDED FIELD java.io.File.separatorLjava.lang.String;
INCLUDED FIELD java.io.File.separatorCharC
INCLUDED FIELD java.io.File.pathSeparatorLjava.lang.String;
INCLUDED FIELD java.io.File.pathLjava.lang.String;
INCLUDED METHOD java.io.File.<init>(Ljava.lang.String;)V
INCLUDED METHOD java.io.File.getName()Ljava.lang.String;
INCLUDED METHOD java.io.File.getPath()Ljava.lang.String;
INCLUDED METHOD java.io.File.getAbsolutePath()Ljava.lang.String;
INCLUDED METHOD java.io.File.isAbsolute()Z
INCLUDED METHOD java.io.File.exists()Z
INCLUDED METHOD java.io.File.existsInternal(Ljava.lang.String;)Z
INCLUDED METHOD java.io.File.isFile()Z
INCLUDED METHOD java.io.File.isFileInternal(Ljava.lang.String;)Z
INCLUDED METHOD java.io.File.equals(Ljava.lang.Object;)Z
INCLUDED METHOD java.io.File.hashCode()I
INCLUDED METHOD java.io.File.toString()Ljava.lang.String;
INCLUDED METHOD java.io.File.<clinit>()V
INCLUDED CLASS java.io.FileDescriptor
INCLUDED FIELD java.io.FileDescriptor.inLjava.io.FileDescriptor;
INCLUDED FIELD java.io.FileDescriptor.outLjava.io.FileDescriptor;
INCLUDED FIELD java.io.FileDescriptor.errLjava.io.FileDescriptor;
INCLUDED FIELD java.io.FileDescriptor.native_fdI
INCLUDED METHOD java.io.FileDescriptor.<init>(I)V
INCLUDED METHOD java.io.FileDescriptor.valid()Z
INCLUDED METHOD java.io.FileDescriptor.validInternal(I)Z
...
```

2.2.3 New Option -showExcludedFeatures

This option will cause the builder to display a list of all classes, methods and fields that where removed from the built application. Any classes, methods or fields removed from the target application through mechanisms such as smart linking (as activated through option -smart) will be displayed. In conjunction with option -smart and -notSmart, this helps one to identify what methods are excluded from the application

The output of this option consists of lines starting with the string EXCLUDED CLASS, EXCLUDED METHOD or EXCLUDED FIELD followed by the name of a class or the name and signature of a method or field, respectively.

Example: The output of this option when applied to a small HelloWorld-Appliation looks like this:

```
> jamaica -smart HelloWorld -showExcludedFeatures
Jamaica Builder Tool 1.0.7 Release 1
Generating code for target 'linux-gnu-i686'
 + HelloWorld__.c
EXCLUDED METHOD HelloWorld.<init>()V
EXCLUDED CLASS com.aicas.jamaica.NYIException
EXCLUDED CLASS jamaica.lang.Process
EXCLUDED CLASS java.awt.AWTPermission
EXCLUDED CLASS java.io.BufferedInputStream
EXCLUDED FIELD java.io.BufferedReader.DEFAULT_BUFFER_SIZEI
EXCLUDED METHOD java.io.BufferedReader.close()V
EXCLUDED METHOD java.io.BufferedReader.markSupported()Z
EXCLUDED METHOD java.io.BufferedReader.mark(I)V
EXCLUDED METHOD java.io.BufferedReader.reset()V
EXCLUDED METHOD java.io.BufferedReader.ready()Z
EXCLUDED METHOD java.io.BufferedReader.skip(J)J
EXCLUDED FIELD java.io.ByteArrayInputStream.markI
EXCLUDED METHOD java.io.ByteArrayInputStream.available()I
EXCLUDED METHOD java.io.ByteArrayInputStream.mark(I)V
EXCLUDED METHOD java.io.ByteArrayInputStream.markSupported()Z
EXCLUDED METHOD java.io.ByteArrayInputStream.reset()V
EXCLUDED METHOD java.io.ByteArrayInputStream.skip(J)J
EXCLUDED CLASS java.io.ByteArrayOutputStream
EXCLUDED FIELD java.io.CharConversionException.serialVersionUIDJ
EXCLUDED METHOD java.io.CharConversionException.<init>()V
EXCLUDED CLASS java.io.DataInput
EXCLUDED CLASS java.io.DataInputStream
EXCLUDED CLASS java.io.DataOutput
EXCLUDED CLASS java.io.DataOutputStream
EXCLUDED CLASS java.io.EOFException
EXCLUDED CLASS java.io.Externalizable
EXCLUDED FIELD java.io.File.pathSeparatorCharC
EXCLUDED METHOD java.io.File.createTempFile(Ljava.lang.String;Ljava.lang.String;)Ljava.io.File;
EXCLUDED METHOD java.io.File.createTempFile(Ljava.lang.String;Ljava.lang.String;Ljava.io.File;)
EXCLUDED METHOD java.io.File.createInternal(Ljava.lang.String;)Z
EXCLUDED METHOD java.io.File.listRoots()[Ljava.io.File;
EXCLUDED METHOD java.io.File.<init>(Ljava.io.File;Ljava.lang.String;)V
EXCLUDED METHOD java.io.File.<init>(Ljava.lang.String;Ljava.lang.String;)V
EXCLUDED METHOD java.io.File.getAbsoluteFile()Ljava.io.File;
EXCLUDED METHOD java.io.File.getCanonicalPath()Ljava.lang.String;
EXCLUDED METHOD java.io.File.getCanonicalFile()Ljava.io.File;
EXCLUDED METHOD java.io.File.getParent()Ljava.lang.String;
...
```

2.2.4 Java Priority Mapping to System Priorities

-priMap <jp=sp{,jp=sp}>

Java threads are mapped directly to threads of the operating system used on the target system. The Java priorities are mapped to system-level priorities for this purpose. This options permit one to replace the default mapping used for a target system by a specific priority mapping.

The Java thread priorities are integer values in the range 1 through 255, where 1 corresponds to the lowest priority and 255 to the highest priority. The Java priorities 1 through 10 correspond to the ten priority levels of java.lang.Thread threads, while priorities staring at 11 represent the priority levels of javax.realtime.RealtimeThread threads. The highest priority is not available for Java threads, it is used for the synchronization thread that permits round robin scheduling of threads of equal priorities.

Each single Java priority can and has to be mapped to a system priority. The mapping has to contain an entry of the form <java-priority>=<system-priority>. To simplify the description of a mapping a range of priorities can be described using <from>..<to>.

Example 1: "-priMap 1..11=50,12..39=51..78,40=85" will cause all java.lang.Thread threads to use system priority 50, while the realtime threads will be mapped to priorities 51 through 78 and the synchronization thread will use priority 85. There will be 28 priority levels available for javax.realtime.RealtimeThread threads.

Example 2: "-priMap 1..52=22..104" will cause the use of system priorities 2, 4, 6, through 102 for the Java priorities 1 through 51. The synchronization thread will use priority 104. There will be 40 priority levels available for javax.realtime.RealtimeThread threads.

2.2.5 Strict Semantics of Real-Time Specification for Java's

-strictRTSJ

The Real-Time Specification for Java (RTSJ) defines a number of classes in the package javax.realtime. These classes can be used to create realtime threads with stricter semantics than normal Java threads. In particular, these threads can run in their own memory areas (scoped memory) that are not part of the Java heap, such that memory allocation is independent of garbage collector intervention. It is even possible to create threads of class javax.realtime.NoHeapRealtimeThread that may not access any objects stored on the Java heap.

In JamaicaVM, normal Java Threads do not suffer from these restrictions, thread priorities of normal threads can be in the range permitted for RealtimeThreads (see option -priMap). Furthermore, any thread can access objects allocated on the heap without having to fear being delayed by the garbage collector. Any thread is safe from being interrupted or delayed by garbage collector activity, only higher priority threads can interrupt lower priority threads.

When using JamaicaVM, it is hence not required to use non-heap memory areas for realtime tasks and it is possible for any thread to access objects on the heap. Furthermore, scoped memory provided by the classes defined in the RTSJ are available to normal threads as well.

The strict semantics of the RTSJ require a significant runtime overhead to check that an access to an object is legal. Since these checks are not needed by JamaicaVM, they are disabled by default. However, setting the option -strictRTSJ forces JamaicaVM to perform this checks.

If -strictRTSJ is set, the following checks are performed and the corresponding exceptions are thrown: MemoryAccessError: If a NoHeapRealtimeThread attempts to access an object stored in the normal Java heap, a MemoryAccessError is thrown

IllegalStateException: If a non-RealtimeThread attempts to enter a javax.realtime.MemoryArea or tries to access the scope stack through the methods getCurrentMemoryArea, getMemoryAreaStackDepth, getOuterMemoryArea or getInitialMemoryAreaIndex defined in class javax.realtime.RealtimeThread.

2.2.6 Static Garbage Collection support

-staticGC

A new builder option -staticGC enables the use of the static garbage collector.

The static garbage collector permit the analysis of dynamic memory allocations performed with an application and the replacement of the dynamic allocations by static allocations that do not require memory recycling by the garbage collector. Consequently, these allocations to not suffer from the performance penalty or any indeterminism introduced by the underlying runtime garbage collection mechanism.

The builder was equipped with a new option -staticGC that activates the static garbage collector. As for the smart linking option -smart, this analysis requires a closed application. Code that uses dynamically loaded code that is not present at the time of the analysis has to be excluded from this optimization using the option -notSmart.

A in-depth discussion of the static garbage collection technology is presented in the TN3: Real-time/Garbage Collection strategy document [AERO_TN3].

3. Real-Time Specification for Java Implementation

A subset of the functionality of the classes of the Real-Time Specification are an important requirement for AERO JVM and the major enhancement to the functionality made during the AERO JVM project. A significant part of this work was performed within the HIDOORS. One of the goals of HIDOORS is to provide a comprehensive set of realtime APIs. These requirements perfectly complements most requirements for the AERO project.

The Real-Time Specification for Java is provided through the package

javax.realtime

and as defined in the SP1 document were be implemented in the AERO JVM.

The design of the implementation of the required functions are detailed in the following sections. The implementation required parts to be written in Java and low-level parts that reference internal structures of the virtual machine to be implemented in C-code. The Java source files implemented for the RTSJ are

AbsoluteTime.java
ArrivalTimeQueueOverflowException.java
AsyncEventHandler.java
BoundAsyncEventHandler.java
DuplicateFilterException.java
GarbageCollector.java
HighResolutionTime.java
ImmortalMemory.java
ImportanceParameters.java
Interruptible.java
LTMemory.java
MITViolationException.java
MemoryArea.java
MemoryParameters.java
MemoryTypeConflictException.java
NoHeapRealtimeThread.java
OneShotTimer.java
PeriodicParameters.java
PhysicalMemoryManager.java
PriorityCeilingEmulation.java
PriorityParameters.java
ProcessingGroupParameters.java
RawMemoryAccess.java
RealtimeClock.java
RealtimeSystem.java
RelativeTime.java
ResourceLimitError.java
Schedulables.java
SchedulingParameters.java
ScopedMemory.java
SizeOutOfBoundsException.java
ThrowBoundaryError.java
Timer.java
UnsupportedPhysicalMemoryException.java
VTPhysicalMemory.java
WaitFreeReadQueue.java
AperiodicParameters.java
AsyncEvent.java
AsynchronouslyInterruptedException.java
Clock.java
FinalizeNode.java
HeapMemory.java
IllegalAssignmentError.java
ImmortalPhysicalMemory.java
InaccessibleAreaException.java
JamaicaGC.java
LTPhysicalMemory.java
MemoryAccessError.java
MemoryInUseException.java
MemoryScopeException.java
MonitorControl.java
OffsetOutOfBoundsException.java
POSIXSignalHandler.java
PeriodicTimer.java
PhysicalMemoryTypeFilter.java
PriorityInheritance.java
PriorityScheduler.java
RationalTime.java
RawMemoryFloatAccess.java
RealtimeSecurity.java
RealtimeThread.java
ReleaseParameters.java
Schedulable.java
Scheduler.java
ScopedCycleException.java
SizeEstimator.java
SporadicParameters.java
Timed.java
UnknownHappeningException.java
VTMemory.java
WaitFreeDequeue.java
WaitFreeWriteQueue.java

A detailed list of the public and protected methods provided by these classes is given in Appendix B of this document.

The C-code is mainly concentrated on native methods in class MemoryArea, since the handling of MemoryAreas is central to the RTSJ memory, thread and event models. The C-code for the RTSJ implementation is contained in the classes

javax_realtime_POSIXSignalHandler.c
javax_realtime_POSIXSignalHandler.c
javax_realtime_RawMemoryFloatAccess.c
javax_realtime_PriorityScheduler.c
javax_realtime_RealtimeClock.c
javax_realtime_RawMemoryAccess.c
javax_realtime_MemoryArea.c

3.1 Scheduling

Priority based scheduling is provided by the AERO JVM through the RTSJ interface. New thread classes RealTimeThread and NoHeapRealtimeThread can be used together with the interface PriorityScheduler.

This scheduling is enabled by use of the underlying operating system's scheduler, in this case VxWorks' priority based preemptive scheduler.

3.1.1 Realtime and NoHeapRealtimeThreads

The implementation provides at least 28 distinct priority levels for these threads. These are mapped to distinct priorities of the underlying operating system to provide the OS's realtime threads to Java code. Any 'normal' non realtime thread will have a priority in the standard priority range 1 through 10 of Java threads. These threads are also mapped to OS's threads, but with priorities lower than those of the realtime threads.

The mapping of the Java priorities to system-level priorities is performed via the Builder option -priMap that is described in chapter 2 of this document.

The RTSJ requires that NoHeapRealtimeThreads be executed at a priority logically higher than that of the garbage collector. Since the AERO JVM does not execute the garbage collector as an independent thread that might interrupt the rest of the application, all realtime threads automatically execute on a priority logically higher than that of the garbage collector. Garbage collection work is performed only within threads that allocate memory on the Java heap. This garbage collection work is pre-emptable by higher priority threads such that realtime threads are not affected by it.

Figure 3.1 illustrates the mapping of Java threads to OS level threads, with native threads that use priorities higher or lower than the Java threads. These native threads can be used for legacy code or low-level system services that are outside of the AERO JVM. The priority based preemptive scheduling of all threads in the system is performed by the underlying operating systems.

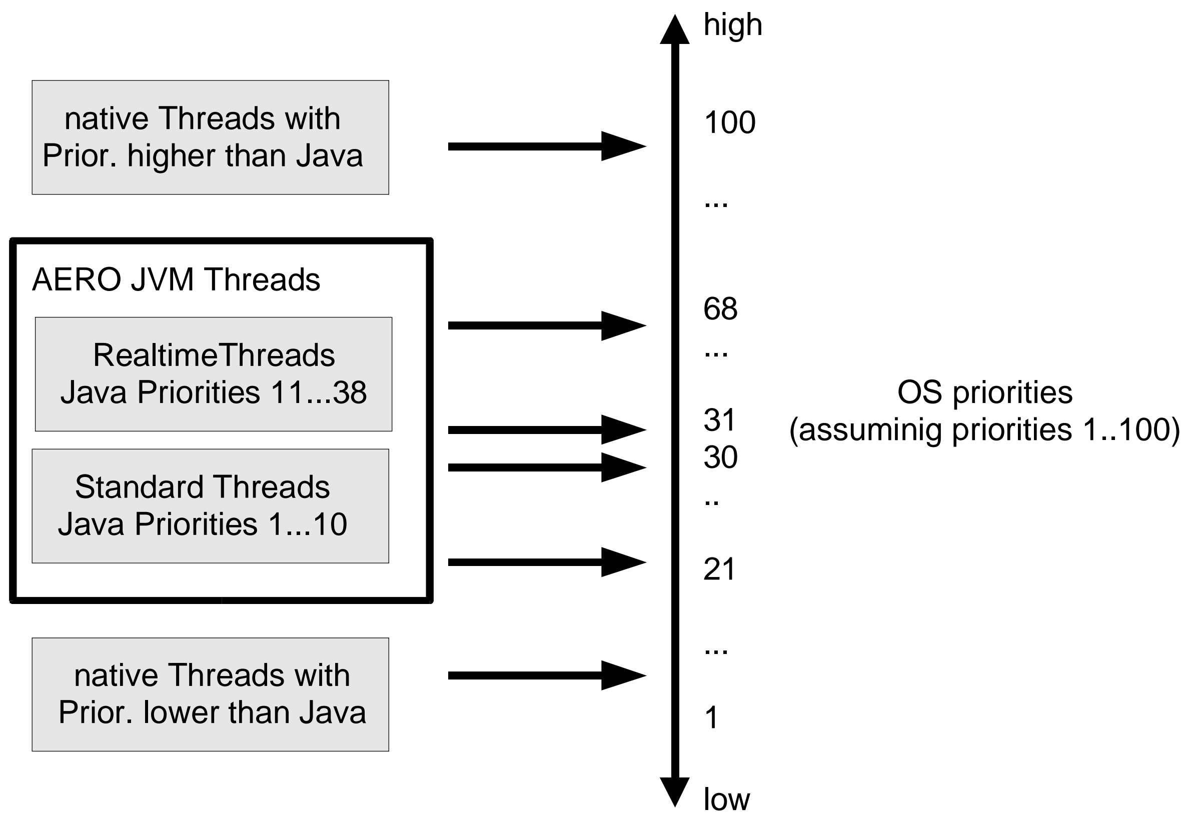


Figure 3.1: Mapping of Java and native thread priorities to RTOS priorities.

3.1.2 NoHeapRealtimeThread scheduling

The Realtime Specification for Java requires that NoHeapRealtimeThreads are essentially equal to RealtimeThreads in the way they are scheduled and the priority range they use. However, NoHeapRealtimeThreads are guaranteed to run at a priority that is logically higher than that of the garbage collector if and only if all other threads have priorities lower than the NoHeapRealtimeThread. This must even hold for dynamic priorities that might be raised in a priority inversion avoidance mechanism like priority inheritance or priority ceiling.

Within the AERO JVM, there is no dedicated garbage collector thread. Consequently, NoHeapRealtimeThreads are automatically executing at a priority that is logically higher than that of the garbage collector. All garbage collection work is performed within the application threads, i.e., within RealtimeThread and normal Java Thread threads. Since these threads must have priorities lower than the NoHeapRealtimeThread, the requirement form the Realtime Specification for Java that all NoHeapRealtimeThreads run at priorities logically higher than the garbage collector is trivially satisfied even without any additional action by the implementation.

However, one aspect of NoHeapRealtimeThreads needs to be taken care of: According to the specification, these threads are not permitted to access any memory on the garbage collected heap. This is to ensure the consistency of traditional garbage collectors in conjunction with NoHeapRealTimeThreads. In AERO JVM, this restriction is technically not required. Furthermore, enforcing this restriction will cause additional runtime checks that severely affect the performance of the overall system. The AERO JVM therefore does not enforce the restriction to access only memory areas that are not under the control of the garbage collector from within NoHeapRealtimeThreads.

For an application it might however be desirable to be portable to other implementation of the Realtime Specification for Java. To be able to develop code using NoHeapRealtimeThreads in a portable way, a compatible mode that enforces the restriction to non-garbage-collected memory was introduced in AERO JVM. This mode is activated by the option -strictRTSJ and it enables the required runtime checks.

3.2 Asychronous Events

Asynchronous events that execute Java code need proper synchronisation with the Java runtime environment not to corrupt the internal state of the AERO JVM. To provide asynchronous event handling, a high priority thread will be used that pre-empts all other Java threads to react on an asynchronous event.

The asynchronous event handling implementation activates this high priority thread whenever external events occur.

Since the specification explicitly requires the ability to handle large numbers of different instances of *AsyncEvent* and *AsyncEventHandler*, we cannot create different contexts for each such event. Instead, a few threads must be able to handle different events. Since the number of fired handlers can be expected to be smaller, this will be a feasible solution.

The asynchronous event handlers are hence mapped to RealtimeThreads that serve as execution context of these events. A pool of these threads is maintained by the Asynchronous event mechanism, each event handler will be assigned to a specific thread as its context.

Not all event handlers can be mapped to such a generic event handler. The important exceptions are event handlers of class BoundAsyncEventHandler that must by specification of this event class be assigned to a single thread context as execution environment and event handlers that have specific execution parameters that differ from other event handlers. These specific execution parameters might be a memory area (such as scoped or immortal memory) that is to be used by the event handler, specific memory parameters (such as budgets on the allocation permitted by the handler) or the requirement to execute the event handler in a NoHeapRealtimeThread context.

Figure 3.2 illustrates the mapping of event handlers to the realtime threads pool and the one-to-one mapping for BoundAsyncEventHandlers and event handlers with specific execution parameters.

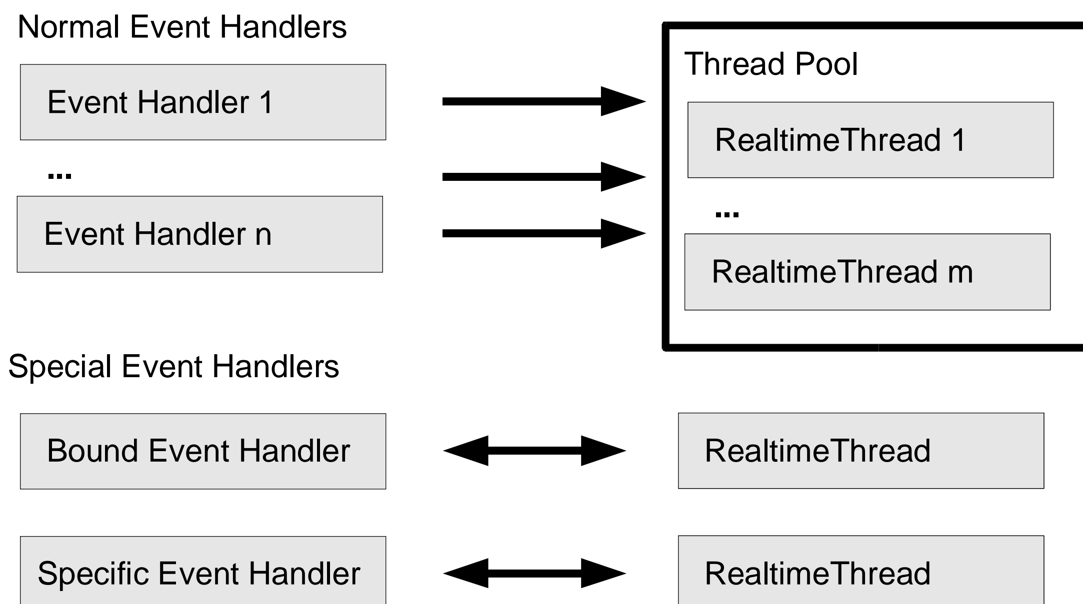


Figure 3.2: Mapping of asynchronous events to realtime threads from thread pool and special events one-to-one mapping to realtime threads.

3.2 Asynchronous Transfer of Control

For the asynchronous transfer of control mechanisms of the RTSJ the specification defines asynchronously-interruptible and non-asynchronously interruptible methods. The asynchronously interruptible code sequences must be equipped with additional checks for asynchronous transfer of control. These checks can be added similar to the way normal Java runtime checks are inserted in the code. They are included at

- synchronization points with asynchronously interruptible methods
- the entry of asynchronously interruptible methods
- after calls to non-asynchronously interruptible methods
- at the end of synchronised blocks within asynchronously interruptible methods
- After a sequence of bytecode instructions

By adding the checks for asynchronous transfer of control only at these points, the overall overhead of this code is minimal.

Different to normal exceptions, AsynchronouslyInterruptedException objects remain pending even after a catch-clause in the Java code. The only means to remove a pending Asynchronously-InterruptedException is a call to the exceptions' happened() method.

Java thread objects contain a private field to hold pending exceptions such that these exceptions can be passed between different code sections (interpreted, compiled, native code). This pending exception field is erased when the exception is caught. To deal with asynchronous exceptions, a new field was added to the thread structure that holds a pending asynchronous exception. This exception remains pending until the exception's happend() method is called. Checks for asynchronous transfer of control consequently check for the existence of a pending asynchronous exception. If one such exception is found, it is thrown, i.e., it is made the new pending exception. A catch-clause removes this pending exception, but does not remove the pending asynchronous exception.

In summary, every thread has two fields for pending exceptions:

```
pendingException
pendingAsyncException
```

The basic operations on exceptions are:

Throwing an exception e (throw e):

```
currentThread().pendingException = e;
jump to next catch-clause
```

Throwing an async exception e in thread t (t.interrupt() or e.fire()):

```
t.pendingAsyncException = e;
```

Catching an exception e (catch (Exception e)):

```
e = currentThread().pendingException;
currentThread().pendingException = null;
```

Checking for pending async exception (within asynchronously interruptible methods):

```
if (currentThread().pendingAsyncException != null) {
      throw currentThread().pendingAsyncException;
      jump to next catch clause;
}
```

Removing a pending async exception e (e.happened()):

```
if (currentThread().pendingAsyncException == e) {
      currentThread.pendingAsyncException = null;
}
```

The resulting code sequences are fairly simple such that no major performance penalty is imposed on the application for support of asynchronous exceptions.

3.3 Memory

Even though the AERO JVM garbage collector does not require special treatment of memory allocations performed within realtime code, the special memory classes defined in the RTSJ are provided by the AERO JVM to ensure inter-operability with other Java implementations and tools.

3.3.1 Scoped Memory

Any realtime thread can be equipped with a scoped memory. The corresponding thread structure will refer to this scope and perform allocations from this memory pool.

These scoped memory areas permit to use special heaps that are not under the control of the garbage collector. In a strict implementation of the Real-Time Specification for Java, only threads that do not access the garbage collected heap can be guaranteed not be to be interrupted by garbage collection activity and can hence give realtime guarantees.

To allow communications between realtime and non-realtime threads, objects can be stored in the shared memory area ImmortalMemory. Objects allocated in this area are never subject to garbage collection and will survive until the system will be shut down.

Scoped memory areas allow stack-like allocation and deallocation of objects. These memory areas can be nested to model nested life spans of Java objects allocated in different scoped memory areas.

In the AERO JVM, the restriction to non-garbage-collected heaps for realtime activities is not required. However, compatibility with the RTSJ specification is desired to be able to execute code developed according to this specification. Consequently, the defined memory areas shall be provided by the AERO JVM.

Threads need to be equipped with a stack of memory areas the thread has entered. Any scoped memory areas that are used in a nested way be several threads must be entered in the same order by all of these threads. However, non-scoped memory areas such as ImmortalMemory or HeapMemory can form part of the scope stack without disturbing the nesting order of the scoped areas.

To record the scope stack, each thread is equipped with an array that holds this stack. Scoped memory areas are additionally linked through a parent reference. The situation of two threads that use the same nested scoped memories but different non-scoped memory areas on their scope stacks is illustrated in **Figure 3.3.**

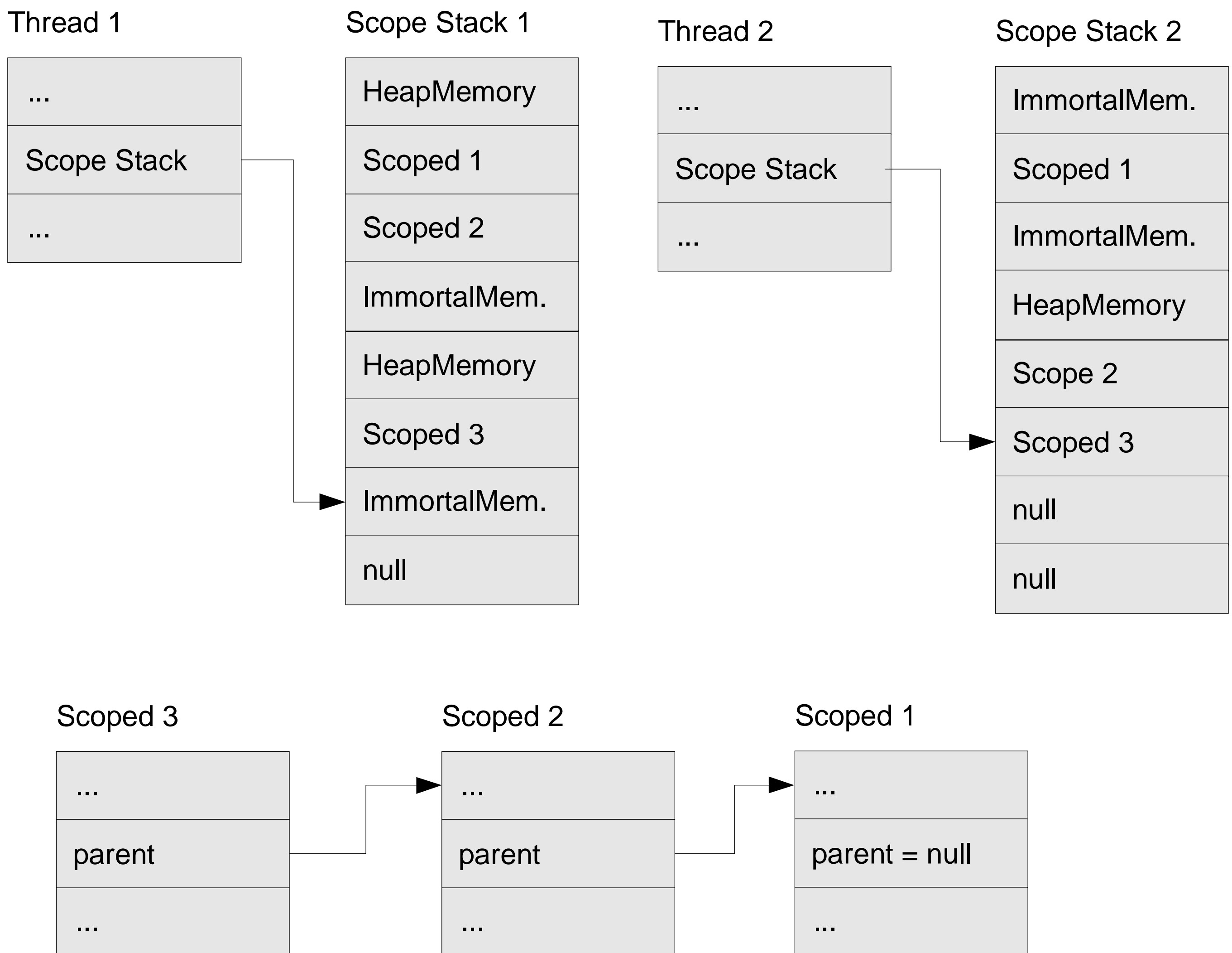


Figure 3.3: Two threads using different scope stacks but the same nesting of scoped memory areas.

3.3.2 Immortal Memory

Immortal memory support in the AERO JVM is straightforward, objects allocated in such an area will be excluded from garbage collection. Their finalizers will not be executed.

3.3.3 Memory Types for Objects

The existence of different memory areas in the Realtime Specification for Java requires to implement a means to determine the memory area an object is stored in. This functionality is provided to the user of the MemoryArea class through the function getMemoryArea(o). This function is performance critical since the determination of the memory area is required for assignment checks as described in the next section.

To permit an efficient implementation, every object is equipped with an 8-bit index value that represent the memory area the object was allocated in. The number of different memory areas in use is hence restricted to 256 with appears to be a reasonable limit for typical space applications.

The determination of the memory area of an object o uses this index and a global array of all used memory areas to find the memory area this object was created in. The code is very simple and efficient.

3.3.4 Assignment checks for reference stores

The existence of nested scoped memory areas with restrictions on the references permitted between two memory areas requires additional runtime checks whenever a reference is stored in an object. More particularly, the runtime checks must ensure that no reference to an object allocated in scoped memory is stored in an object that is stored in heap memory, in immortal memory or in another scope whose nesting level is outside of the current scope. To permit an efficient implementation of this check, all memory areas are equipped with an index in the scope stack, while inner scopes have higher indices than outer scopes. Special indices are assigned to heap and immortal memory areas as follows:

memory area	**scope stack index (idx)**
heap	0
immortal	1
scope	2, 3, 4, etc.

Additionally, all memory areas have a field that gives the maximum assignable scope stack index for any objects reference that is to be stored into this area. These maximum assignable scope stack indices are 1 for heap and immortal memory. Consequently, assignments between both these areas are allowed but no reference to any object on a scope can be stored in an object on the heap or on immortal memory.

For scoped memory, the maximum scope stack index is equal to its scope stack index, assignments of references to objects on heap, immortal, outer scopes or this scope are allowed, but not assignments of references to inner scopes.

The values for the maximum assignable scope stack indices:

memory area	**maximum assignable scope stack index (max_idx)**
heap	1
immortal	1
scope	2, 3, 4, etc.

The code that needs to be performed for a reference assignment of the form

```
p.f = o;
```

is the code

```
MemoryArea o_ma = getMemoryArea(o);
MemoryArea p_ma = getMemoryArea(p);
if (p_ma.max_idx < o_ma.idx) {
      throw new IllegalAssignmentError();
}
p.f = o;
```

The runtime overhead is limited to a few memory accesses per reference assignment, it is reasonably low.

3.3.5 Memory Areas and NoHeapRealtimeThreads

When running AERO JVM in compatible mode that is activated through option -strictRTSJ, the implementation must ensure via runtime checks that no NoHeapRealtimeThread ever accesses memory that is under the control of the garbage collector. The implementation must trap all accesses to heap memory by these threads.

To implement this restrictions, simple assignment rules as presented above for the consistency of scope memory areas are not sufficient. Instead, any instruction that reads reference from memory must be checked. These instructions are the following bytecode instructions

getstatic

getfield

aaload

In addition to these bytecode instruction, the new bytecodes introduced into JamaicaVM need be considered as well. The affected instruction is

getfield_R

On all these instructions, it must be determined whether the current thread is a NoHeapRealtime-Thread. If this is the case, it must be checked that a reference value loaded by this instruction does not refer to heap memory. If a reference to heap memory is loaded, a MemoryAccessError exception must be thrown at runtime.

Even though the test is relatively efficient with the memory area information stored with all Java objects, the tests need to be executed very frequently at all load operations (compared to only write accesses to the heap for the checks required to ensure the consistency of nested scoped memory areas). The result is a significant runtime overhead that affects not only NoHeapRealtimeThreads but all threads.

An alternative would be to duplicate all code of the virtual machine and code generated by the compiler to have normal and NoHeapRealtimeThread-versions of all methods. This would take the overhead from normal threads, but it would cause significantly higher memory demand for the application (approximately a factor 2 larger code size).

This high overhead explains the design decision to make these checks that are technically not required for AERO JVM option through the option -strictRTSJ. Applications that require all runtime checks for conformity with the Realtime Specification for Java can chose to run with these checks enabled (at least during development/validation period), but applications that do not require these checks at runtime do not suffer from a degradation in performance during runtime.

3.3.6 Physical Memory

The use of specific memory areas for allocation of objects requires the support of a non-contiguous heap for the allocation of objects. The AERO JVM does not require support for physical memory (as apposed to raw memory, which is required for I/O accesses etc.), so the classes were implemented to behave as specified for systems without physical memory or with physical memory features deactivated.

User code attempting to create physical memory areas will see the corresponding runtime exceptions such as UnsupportedPhysicalMemoryException. Future versions of AERO JVM might support physical memory areas, but this will require significant changes in the overall memory management code (garbage collector, write barrier code). The reason is that the current implementation relies on the fact that the heap memory is a large contiguous range of memory that is under the control of the

AERO JVM. To support physical memory ranges, all code accessing the heap must be change to cope with the fact that the heap might consist of several non-contiguous ranges.

The classes that deal with physical memory and that are consequently not implemented completely for the AERO JVM are listed here:

ImmortalPhysicalMemory.java
LTPhysicalMemory.java
PhysicalMemoryManager.java
PhysicalMemoryTypeFilter.java
UnsupportedPhysicalMemoryException.java
VTPhysicalMemory.java

3.3.4 Allocation budgets and allocation rates

Threads that are limited by allocation budgets and allocation rates have to contain additional data describing these budgets within the internal thread data. Also, the amount of memory allocated by each thread needs to be recorded to check the budget and to be able to test whether reducing the budget of a running thread would be feasible (i.e., whether the thread has not yet exceeded the more restrictive budget).

The budgets can be specified for allocations in a thread's memory area (heap or scoped memory) or within the global immortal memory. Consequently, each thread is equipped with budget values for allocations in memory areas and in immortal memory.

All allocation routines were extended to first check the available budget and to update the amount of memory allocation performed by the thread. These checks and bookkeeping take into account whether the allocation is performed in a memory area or in immortal memory to check against the applicable budget.

4. VxWorks/ERC32 support

The AERO JVM is based on top of system dependent layer. To keep the system portable, all target dependent parts need to be separated into specific files. The target dependent functions use POSIX standard functions as far as possible.

4.1 Overview over Target OS dependent layer

For the implementation of the target OS dependent layers, several options were considered:

a) basing the system on an existing realtime operating system. This has the advantage that existing code and realtime thread and scheduling mechanisms can be reused. The realtime kernel does not need to be developed from scratch for the use by the AERO JVM. Candidates for the underlying operating systems are

- VxWorks
- RTEMS

This solution will permit integration of legacy C-code that. A hybrid system consisting of a mixture of Java code and traditional C code, a solution based on an existing realtime OS will be preferable.

b) Develop a AERO-JVM specific target OS layer that directly implements the basic system functions required from the realtime operating system. This resulting system will run directly on top of the hardware, no additional operating system level will be needed.

4.1.1 Hierarchical structure of target dependent code

Even though target dependent parts of the application are separated from the target-independent code, the amount of target-specific code is fairly large considering the wide range of very different realtime- and non-realtime operating systems and processor architectures supported by the base Java implementation JamaicaVM. The target specific code consequently contains parts that are very similar or even equal between different architecture.

Particularly, most system functions even for realtime operating systems are based on equal or similar functions in Unix-style systems. The target specific layer hence contains standard code for Unix systems, only those parts that differ for a specific target systems —be it Linux, QNX, or, as in this case, VxWorks—need to be adopted.

The target dependent code is hence split into a generic file for standard Unix-like systems and a target dependent file that might overwrite these declarations. The generic files defined are

file	file i/o functions
io	basic i/o functions
math	mathematical functions (isnan, etc.)
math_int64	mathematical functions for 64-bit integers
memory	memory allocation/deallocation
misc	miscellaneous (system name, etc.)
network	network functions
semaphore	semaphore functions
signal	POSIX signals
thread	thread functions

All declarations within these generic files are made conditionally in the format

```
#ifndef JAMAICA_NATIVE_<targetspecific>
  #define JAMAICA_NATIVE_<targetspecic> <default_implementation>
#endif
```

Consequently, the target specific files only need to contain declarations that differ from the default implementation and include the generic files after these specific declarations were made. The generic files will then use the standard declaration for all symbols that were not already declared in the target specific file.

The amount of duplicated code and hence the problem of maintaining a set of different but similar target specific code layers is minimized and any changes within the target specific layer are restricted to the generic target specific code and those target specific files that redefine the generic code. For most targets, a large part of the declarations in the generic part can be used.

4.2 VxWorks/ERC32 port

For the port to VxWorks/ERC32, a relatively large part of the target specific code needed to be reimplemented in the target specific files. Particularly, the POSIX-based definitions for threads and semaphores were not used directly in the VxWorks port even though VxWorks support the POSIX thread interfaces. The reason for this decision lies in the additional overhead introduced by VxWorks's POSIX interface that itself is mapped to VxWorks's internal thread functions.

The heavy use of thread and semaphore functions and their importance for the overall systems performance lead to the design decision to redeclare the thread interface based on VxWorks's native thread and semaphore mechanisms and to circumvent the generic POSIX-based implementation.

The platform dependent files that were written for this platform are

Include files:

config.h
jamaica_native_math.h
jamaica_native_misc.h
jamaica_native_thread.h
jamaica_types.h
jamaica_native_file.h
jamaica_native_math_int64.h
jamaica_native_semaphore.h
jamaica_specific_native.h
jamaica_native_io.h
jamaica_native_memory.h
jamaica_native_signal.h
jamaica_target.h

C files:

jamaica_native_io.c
jamaica_native_misc.c
jamaica_native_signal.c
jamaica_specific_native.c
jamaica_native_file.c
jamaica_native_math_int64.c
jamaica_native_semaphore.c
jamaica_native_thread.c
jamaica_target.c

4.2.1 config.h for VxWorks/ERC32

The config.h file lists low-level declarations that are required to support the target system. It is generated automatically by a configure stript, which itself is generated by the tool autoconf from a file configure.in.

For VxWorks/ERC32, the confgure.in file needed to be extended by the following settings:

- WIND_BASE needs to be checked for VxWorks configuration
- VxWorks libraries need to be included
- the target architecture is BIG_ENDIAN
- WindRiver include files and compiler/linker flags need to be used
- There is no home Jamaica home directory on the target system
- There is no graphics or mouse mouse supported
- Function System.exec() is not available.

4.2.2 Native threads for VxWorks/ERC32

The native thread support for VxWorks is based directly on the tasks provided by the VxWorks operating system. The target specific file jamaica_native_threads.c for VxWorks uses these system functions for this:

- taskSpawn
- taskDelete
- taskIdVerify
- taskDelay
- taskPrioritySet

4.2.3 Native semaphores for VxWorks/ERC32

Functions for inter-task communication are based on VxWorks semaphores using the system functions

- semBCreate,
- semDelete,
- semGive, and
- semTake.

The corresponding target-specific code is provided in files jamaica_native_semaphore.h and jamaica_native_semaphore.c.

4.3 RTEMS/ERC32 port

The RTEMS/ERC32 port was not performed by the time of writing of this document. Due to the similarity to Unix-like systems, the port will also be based on the Unix-like interface. In addition, specific code will be required, essentially requiring new implementations of all those C header files and C source files that needed adoption for the VxWorks/ERC32 port.

Appendix A: Builder Usage

To control the work of the builder tool, a variety of arguments can be provided. The arguments can be provided directly to the builder, or using the configuration file in $JAMAICA/etc.

The syntax is as follows:

```
jamaica [-help] [-Xhelp] [-Xdoc] [-version] [-verbose] [-showSettings]
        [-XjamaicaHome <path>] [-Xbootclasspath <path>] [-classpath <path>]
        [-XnoMain] [-XnoClasses] [-XvarMemModel] [-main <class>] [-tmpdir
        <name>] [-destination <name>] [-loadPath <path>] [-resource <names>]
        [-smart] [-notSmart <classes>] [-showIncludedFeatures]
        [-showExcludedFeatures] [-numDynamicTypes <n>]
        [-numDynamicTypesFromEnv <var>] [-noCompaction] [-cldc] [-compile]
        [-inline <n>] [-trace <level>] [-debug] [-excludeLongerThan <n>]
        [-Xcc <cc>] [-XO <optflag>] [-XCFLAGS <cflags>] [-XLDFLAGS <ldflags>]
        [-Xld <linker>] [-Xlib <lib>] [-Xstaticlib <lib>] [-Xlinkerprefix
        <prefix>] [-Xlibpath <libpath>] [-target <platform>]
        [-XavailableTargets] [-XnewTarget <original_target>] [-Xstrip
        <strip>] [-heapSize <n>] [-stackSize <n>] [-immortalMemorySize <n>]
        [-numThreads <n>] [-heapSizeFromEnv <var>] [-stackSizeFromEnv <var>]
        [-immortalMemorySizeFromEnv <var>] [-numThreadsFromEnv <var>]
        [-analyse <acc>] [-analyseFromEnv <var>] [-staticGC <gc-work>]
        [-staticGCFromEnv <var>] [-blockSize <n>] [-threadPreemption <n>]
        [-timeSliceNanos <n>] [-finalizerPri <pri>] [-priMap <jp=sp{,jp=sp}>]
        [-strictRTSJ] [-profile] [-useProfile <p>] [-percentageCompiled <n>]
        [-object <files>] [-include <dirs>] [-XTRACTPROJECT] [-Xstats]
        [-XcountCalls] [-XcountArrays] [-XcountAllocs] [-XcountFields] [-Xpg]
        class1 [... classn]
```

A.1 General

These are general options providing information about the builder itself or enabling the use of script files that specify further options

-help

Display the usage of the Builder tool and a short description of all possible standard command line options

-Xhelp

Display the usage of the Builder tool and a short description of all possible standard and extended command line options. The extended command line options are not needed for normal control of the builder command. They provide means to configure the tools and options to be used and provide tools required internally for development of the JamaicaVM

-Xdoc

Setting this option causes the creation of docbook documentation file for this command.

-version

Prints the version of the Jamaica Builder Tool and exits.

-verbose

Enable verbose mode for the builder. In verbose mode, additional output on the state of the build process is printed to standard out. The information provided in verbose mode includes the list of all class files that are loaded and the methods that are compiled in case compilation is switched on.

-showSettings

The currently used options of the JamaicaVM Builder are written to stdout in property file format. To make these the default settings, copy these options into your jamaica.conf file.

A.2 Classes, files, and paths

These options allow to specify classes and paths to be used by the builder.

-XjamaicaHome <path>

Specifies the path to the Jamaica directory.

-Xbootclasspath <path>

Specifies path used for loading system classes

-classpath <path>

Specify the paths that are to be used to search for class files. A list of paths separated by the path separator char (":" on Unix systems) can be specified. This list will be traversed from left to right when the builder tries to load a class.

-XnoMain

Do not select a main class for the built application. Instead, the first argument of the argument list passed to the application will be interpreted as the main class.

-XnoClasses

Do not include any classes in the built application. Setting this option is only needed when building the jamaicavm command itself.

-XvarMemModel

Create C code that is not specific to a certain block size or object model.

-main <class>

Specify the main class of the application that is to be built. This class must contain a static method "int main(String[] args)". This method is the main entry point of the Java application.

If -main is not specified, the first entry of the classes list that is provided to the builder is used as main class.

-tmpdir <name>

This option can be used to specify a name of the directory for temporary files generated by the builder (such as C-source files, object files and make files).

If -tmpdir is not specified, the current directory is used.

-destination <name>

Specify the name of the destination executable to be generated by the builder. If this option is not present, the name of the main class is used as the name of the destination executable.

The destination name can be a path into a different directory. E.g., "-destination myproject/bin/application" can be used to save to create the executable "application" in "myproject/bin"

-loadPath <path>

Load all classes that can be found in the given directory. This option can be used to automatically include all classes in this directory to a project.

The classes must be stored in sub-directories whose name corresponds to the package name, e.g., class java.lang.Object must reside in file <path>/java/lang/Object.class on a system that uses "/" as file separator character.

The path specified must not include the package directory.

-resource <names>

This option causes the inclusion of additional resources in the built application. A resource is additional data (such as images, sound etc.) that can be accessed by the Java application

Within the Java application, the resource data can be accessed using the resource name specified as an argument to -resource. To load the resource, a call to Class.getResourceAsStream(<name>) can be used.

If a resource is supposed to be in a certain package, the resource name must include the package name. Any "." must be replaced by "/". E.g., the resource ABC from package foo.bar can be added using "-resource foo/bar/ABC".

The builder uses the class path provided through the option "-classpath" to search for resources. Any path containing resource that are provided using "-resource" must hence be added to the path provided to "-classpath".

A.3 Smart linking and compaction

Smart linking and compaction are techniques to reduce the code size and heap memory required by the generated application. These techniques are controlled by the following options.

-smart

If this option is set, smart linking is enabled for this project. Smart linking permits the builder to remove unused code and to perform better optimization to obtain smaller binary files, smaller memory usage and faster execution.

Smart linking must only be used for closed applications that do not use Java's reflection API (including reflection via the Java Native Interface JNI) and that do not download classes dynamically or use functions like java.lang.Class.forName(className) that cause loading of classes that is not visible to the builder tool. If -smart is used for these applications, code that is referenced by classes that are loaded dynamically or through the reflection API might have been removed and methods that are redefined by dynamically loaded code might not be called correctly since -smart causes static binding of virtual or interface methods that are not redefined.

When dynamic class loading or reflection is used, the option -notSmart can be used to restrict the effect of -smart and still profit from better optimization

-notSmart <classes>

This option causes all classes provided as an argument to this option to be excluded from the optimizations enabled by option -smart.

For reflection (including reflection through JNI) to work properly, all classes that are used by the reflection API must be excluded from smart linking using this option. Similarly, all classes that might be referenced by classes that are downloaded dynamically must be excluded form smart linking as well.

Since serialization requires the reflection API, smart linking can change the format of the data generated for serialized objects. This can cause conflicts if serialized classes need to be exchanged between different versions of an application or different Java runtime environments. To avoid these difficulies, all classes that are subject to serialization should be excluded from smart linking

If several classes are specified as an argument for -notSmart, the classes must be separated using commas. E.g., specifying "-notSmart pack1.Foo,pack2.Bar" excludes classes pack1.Foo and pack2.Bar from smart linking.

-showIncludedFeatures

This option will cause the builder to display a list of all classes, methods and fields that will be part of the built application. Any classes, methods or fields removed from the target application through mechanisms such as smart linking (as activated through option -smart) will not be displayed. In conjunction with option -smart and -notSmart, this helps one to identify what methods are included into the application

The output of this option consists of lines starting with the string INCLUDED CLASS, INCLUDED METHOD or INCLUDED FIELD followed by the name of a class or the name and signature of a method or field, respectively.

-showExcludedFeatures

This option will cause the builder to display a list of all classes, methods and fields that where removed from the built application. Any classes, methods or fields removed from the target application through mechanisms such as smart linking (as activated through option -smart) will be displayed. In conjunction with option -smart and -notSmart, this helps one to identify what methods are excluded from the application

The output of this option consists of lines starting with the string EXCLUDED CLASS, EXCLUDED METHOD or EXCLUDED FIELD followed by the name of a class or the name and signature of a method or field, respectively.

-numDynamicTypes <n>

When -smart is set, the number of types that can be used by the application is limited. Types are all classes used in the system and all arrays of different dimensions and of different element types. The number of classes used by the internal classes of the application is determined automatically by the Jamaica Builder. It is therefore only required to specify "-numDynamicTypes" in the case dynamic loading of classes is used.

This option can only specified in conjunction with "-smart". If smart linking is disabled, the number of types is not limited statically. In this case, only the memory available in the Java heap limits the number of types that can be created dynamically.

The value can be set to zero in case dynamic loading is not used. Lower values require less memory. The maximum allowed number of types is 65535.

-numDynamicTypesFromEnv <var>

Specifying this options causes the creation of an application that reads the number of dynamically loadable types from the environment variable specified using this option. If this variable is not set, the number specified using "-numDynamicTypes <n>" will be used.

-noCompaction

Specifying this option disables class file compaction. Disabling compaction forces the Jamaica Builder not to use Jamaica's compact class file format for all the classes in this project. Not compacting the classes will cause a higher memory overhead since classes are left in the original format.

A.4 Java standard API configurations

These options permit the selection of API configurations to be used by the built application.

-cldc

This option forces the inclusion of all standard library code within the CLDC configuration. This 'Connected Limited Device Configuration' is supposed to be used in small connected devices.

Providing this option forces the builder to include all classes, fields, methods and constructors that form part of the CLDC configuration. These will be added to the built application even if smart linking is activated using -smart.

The resulting application will hence be able to access all code in the CLDC configuration through dynamically loaded classes or the reflection API even if this code would otherwise not be included in the application.

A.5 Compilation and optimization

Compilation and different optimization techniques are used for optimal runtime performance of Jamaica applications. These techniques are controlled using the following options.

-compile

Enable static compilation for this project. All methods in this project are compiled into native code causing a significant speedup at runtime compared to the interpreted code that is executed by the virtual machine. It is recommended to use compilation whenever execution time is important. However, it is often sufficient to compile about 10% of the classes, which results in much smaller

executables of comparable speed. You can achieve this by using -profile / -useProfile instead of -compile.

-inline <n>

When -compile is set, this option can be used to set the level of inlining to be used by the compiler. Inlining typically causes a significant speedup at runtime since overhead to perform method calls is avoided. Nevertheless, inlining causes duplication of code and hence might increase the binary size of the application. In systems with tight memory resources, inlinling might hence not be acceptable

Eleven levels of inlining are supported by the Jamaica compiler ranging from 0 (no inlining) to 10 (aggressive inlining).

-trace <level>

Enable the generation of TRACE-code at the given level. Trace code can be used for low-level debugging of compiled code. In trace mode, information on the currently executed method or instruction is printed to standard output.

There are three levels of trace information: no tracing (-trace 0), tracing of method calls (-trace 1), and tracing of single instructions (-trace 2).

The amount of data printed to standard output when trace is enabled is typically significant. Care must be taken when this option is set, it is recommended to be used for problems that are hard to debug otherwise.

-debug

Generate a debugging version of the project. The debugging version uses the Jamaica Virtual Machine additional runtime checks and performs addition checks in compiled code. These checks enabled debugging of bugs in native code or problems in the VM or compiled code. The additional runtime checks cause a significant slowdown of the application.

-excludeLongerThan <n>

Compilation of large Java methods, especially in the combination of aggressive inlining can cause large C routines in the intermediate code. Some C compilers have difficulties with the compilation of large routines. To enable the use of Jamaica with such C compilers, the compilation of large methods can be disabled using this option.

The argument specified to -excludeLongerThan gives the minimum number of bytecode instructions a method must have to be excluded from compilation.

-Xcc <cc>

Specify the C compiler to be used to compile intermediate C code that is generated by the Jamaica Builder. Recommended is gcc 2.95.3.

-XO <optflag>

Specify the optimization level to be used when compiling the intermediate C code to native code.

-XCFLAGS <cflags>

Specify the cflags for the invocation of the C compiler. Note that the optimisation flag should be provided separately via the -XO option.

-XLDFLAGS <ldflags>

Specify the ldflags for the invocation of the C linker.

-Xld <linker>

Specify the linker to be used to create a binary file from the object file generated by the C compiler.

-Xlib <lib>

Specify the libraries that must be linked to the destination binary. The libraries must include the option that is passed to the linker. Several libraries must be separated using commas. E.g, "-Xlib -lm,-lpthread" causes linking against libm and libpthread.

-Xstaticlib <lib>

Specify the libraries that must be static linked to the destination binary. The libraries must include the option that is passed to the linker. Static linking causes larger executables, but is necessary if the target system (possibly) doesn't provide the library. Several libraries must be separated using commas. E.g, "-Xlib -lm,-lpthread" causes static linking against libm and libpthread.

-Xlinkerprefix <prefix>

Specify the libraries that must be static linked to the destination binary. The libraries must include the option that is passed to the linker. Static linking causes larger executables, but is necessary if the target system (possibly) doesn't provide the library. Several libraries must be separated using commas. E.g, "-Xlib -lm,-lpthread" causes static linking against libm and libpthread.

-Xlibpath <libpath>

Add the directories in the specified paths to the library path. Multiple libraries should be separated by by the path separator character (e.g., ":"). E.g., to use the directories /usr/local/lib and /usr/lib as library path, the option "-Xlibpath -L/usr/local/lib:-L/usr/lib" must be specified.

-target <platform>

Specify a target platform. For a list of all available platforms of your JamaicaVM Distribution, use -XavailableTargets.

-XavailableTargets

List all available target platforms of this Jamaica distribution.

-XnewTarget <original_target>

Introduce a new (virtual) target which uses the compiled jamaica libraries of <original_target>. E.g., "-XnewTarget.dilnetpc linux-gnu-i486" introduces the new target dilnetpc which can have its own settings in the property file, but shares its libraries with the linux-gnu-i486 target.

-Xstrip <strip>

Use the specified tool to remove debug information from the generated binary. This permits to reduce the size of the binary file by removing information that is not needed at runtime.

A.6 Memory and threads settings

Configuring the heap memory and threads has important impact not only on the amount of memory required by the application but as well on the runtime performance and the realtime characteristics of the code. The Jamaica Builder hence provides a number of options to configure the memory and threads of the application.

-heapSize <n>

Set the heap size to the specified size given in bytes. The heap is allocated at startup of the application. It is used for static global information (such as the internal state of the Jamaica Virtual Machine) and for the garbage collected Java heap.

The heap size can be succeeded by the letter "k" or "m" to specify a size in KBytes (1024 bytes) or MBytes (1048576 bytes. The minimum heap size is 85k.

-stackSize <n>

Set the stack size to be used for the runtime stacks of all Java threads in the destination application. The stacksize is used several times for different runtime stacks and for every Java thread. The stack size consequently has an important impact on the heap memory required by an application. In systems with tight memory, A small stack size should be selected.

The stack size can be succeeded by the letter "k" or "m" to specify a size in KBytes (1024 bytes) or MBytes (1048576 bytes. The minimum stack size is 1k.

-immortalMemorySize <n>

Set the size of the immortal memory area given in bytes. The immortal memory area is allocated form the heap the first time it is used. The immortal memory can be accessed through the class javax.realtime.ImmortalMemory.

The immortal memory area is guaranteed never to be freed by the garbage collector. Objects allocated in this area will survive the whole application run.

-numThreads <n>

Specify the maximum number of Java threads supported by the destination application. These threads and their runtime stacks are generated at startup of the application. A large number of threads consequently may require a significant amount of memory.

The minimum number of threads is two, one thread is the main Java thread and one thread is needed as finalizer thread.

-heapSizeFromEnv <var>

Specifying this options causes the creation an application that reads its heap size from the environment variable specified using this option. If this variable is not set, the heap size specified using -heapSize <n> will be used.

-stackSizeFromEnv <var>

Specifying this options causes the creation an application that reads its stack size from the environment variable specified using this option. If this variable is not set, the stack size specified using -stackSize <n> will be used.

-immortalMemorySizeFromEnv <var>

Specifying this options causes the creation an application that reads its immortal memory size from the environment variable specified using this option. If this variable is not set, the immortal memory size specified using -immortalMemorySize <n> will be used.

-numThreadsFromEnv <var>

Specifying this options causes the creation an application that reads the number of threads from the environment variable specified using this option. If this variable is not set, the number of threads specified using -numThreads <n> will be used.

-analyse <acc>

Enable memory analyse mode with accuracy given in percent. During memory analyse mode, the memory required by the application during execution will be determined. The result is an upper bound for the actual memory required during a test run of the application. This bound is at most the specified accuracy larger than the actual amount of memory used during runtime.

The result of a test run of an application built using -analyse can then be used to configure the heap size of an application such the the garbage collection work that is performed on an allocation never exceeds the amount allowed to ensure timely execution of the application's realtime code.

Using -analyse can cause a significant slowdown of the application. The slowdown is higher for a higher accuracy of the analysis, i.e., for a lower value specified as an argument to -analyse.

To configure the heap of an application, a version of the application using the option -analyse and, apart form this, the exactly same list of arguments for the final versions must be build. The heap size determined in a test run can then be used to build a final version using the preferred heap size with desired garbage collection overhead. Again, it must be ensured that the arguments list provided to the builder for this final version must be the same as the argument list provided for the version used to analyse the memory requirements. Only the -heapSize option of the final version must be set accordingly and the final version must of course be build without -analyse.

-analyseFromEnv <var>

Specifying this options causes the creation an application that reads the amount of analyse accuracy of the garbage collector from the environment variable specified using this option. If this variable is not set, the number accuracy specified using -analyse <n> will be used. Setting the environment variable to "0" will disable the analysis and cause the garbage collector to us dynamic garbage collection mode.

-staticGC <gc-work>

Run the garbage collector in static mode. In static mode, for every unit of allocation, a constant number of units of garbage collection work is performed. Compared to dynamic mode, this causes a lower worst case execution times for the garbage collection work at an allocation and a more predictable behaviour since the amount of garbage collection work is the same for any allocation. However, static mode causes higher average garbage collection overhead compared to dynamic mode.

The value specified is the number for units of garbage collection work to be performed for a unit of memory that is allocated. This value can be determined using a test run that was build with -analyse set.

The default setting is that static garbage collection is disabled and the amount of garbage collection work on an allocation is determined dynamically depending on the amount of free memory.

-staticGCFromEnv <var>

Specifying this options causes the creation an application that reads the amount of static garbage collection work on an allocation from the environment variable specified using this option. If this variable is not set, the value specified using -staticGC <n> will be used.

-blockSize <n>

Jamaica looks at the heap as a large array of single memory units, so called blocks. All allocated object are constructed out of these blocks. The size to be used for these blocks can be specified using this option.

Applications that generate a large number of very small or very large Java objects or applications for which a particular object size strongly dominates can profit from selecting the most suitable block size. This has an effect on the memory requirements of the application and on its runtime performance.

The size is specified in number for bytes per block. It must be at least 16 (when -smart is used) or 20 (otherwise). The maximum size allowed is 128. 32 Bytes is a good standard block size for most applications. For alignment reasons, a power of two should in most cases be preferred.

-threadPreemption <n>

Compiled code contains special instructions that permit thread preemption. These instructions have to be executed often enough to allow a thread preemption time that is sufficient for the destination application. Since the instructions cause an overhead in code size and runtime performance one would want to generate this code as seldom as possible.

This option enables setting of the maximum number of intermediate instructions that are permitted between the execution of thread preemption code. This directly affects the maximum thread preemption time of the application. One intermediate instructions typically corresponds to 1-2 machine instructions, while there are some intermediate instructions (calls, array accesses) that can be more expensive (20-50 machine instructions)

The thread preemption must be at least 10 intermediate instructions.

-timeSliceNanos <n>

For threads of equal priority, round robin scheduling is used when several threads are running simultaneously. Through this option, the maximum size of such a time slice can be given in nanoseconds. A special synchronization thread is used that waits for the length of a time slice and permits thread switching after every time slice.

If no round robin scheduling is needed for threads of equal priority, the size of the time slice can be set to zero. In this case, the synchronization thread is not required, such that fewer system resources are needed and the highest priority threads will not be interrupted by the synchronization thread.

-finalizerPri <pri>

Set the Java priority of the finalizer thread to <pri>. The finalizer thread is a daemon thread that runs in the background and executed the finalize() method of objects that are about to be freed by the garbage collector. The memory of objects that have a finalize() method cannot be freed before this method was executed.

A priority of zero for the finalizer thread indicates that no finalizer thread shall be created. In this case, the memory of objects that are found to be unreachable by the garbage collector cannot be freed

before their finalizers are executed explicitly through a call of java.lang.Runtime.runFinalization(). Not using a finalizer thread can be useful on very small systems to reduce the use of system resources.

-priMap <jp=sp{,jp=sp}>

Java threads are mapped directly to threads of the operating system used on the target system. The Java priorities are mapped to system-level priorities for this purpose. This options permit one to replace the default mapping used for a target system by a specific priority mapping.

The Java thread priorities are integer values in the range 1 through 255, where 1 corresponds to the lowest priority and 255 to the highest priority. The Java priorities 1 through 10 correspond to the ten priority levels of java.lang.Thread threads, while priorities staring at 11 represent the priority levels of javax.realtime.RealtimeThread threads. The highest priority is not available for Java threads, it is used for the synchronization thread that permits round robin scheduling of threads of equal priorities.

Each single Java priority can and has to be mapped to a system priority. The mapping has to contain an entry of the form <java-priority>=<system-priority>. To simplify the description of a mapping a range of priorities can be described using <from>..<to>.

Example 1: "-priMap 1..11=50,12..39=51..78,40=85" will cause all java.lang.Thread threads to use system priority 50, while the realtime threads will be mapped to priorities 51 through 78 and the synchronization thread will use priority 85. There will be 28 priority levels available for javax.realtime.RealtimeThread threads.

Example 2: "-priMap 1..52=22..104" will cause the use of system priorities 2, 4, 6, through 102 for the Java priorities 1 through 51. The synchronization thread will use priority 104. There will be 40 priority levels available for javax.realtime.RealtimeThread threads.

-strictRTSJ

The Real-Time Specification for Java (RTSJ) defines a number of classes in the package javax.realtime. These classes can be used to create realtime threads with stricter semantics than normal Java threads. In particular, these threads can run in their own memory areas (scoped memory) that are not part of the Java heap, such that memory allocation is independent of garbage collector intervention. It is even possible to create threads of class javax.realtime.NoHeapRealtimeThread that may not access any objects stored on the Java heap.

In JamaicaVM, normal Java Threads do not suffer from these restrictions, thread priorities of normal threads can be in the range permitted for RealtimeThreads (see option -priMap). Furthermore, any thread can access objects allocated on the heap without having to fear being delayed by the garbage collector. Any thread is safe from being interrupted or delayed by garbage collector activity, only higher priority threads can interrupt lower priority threads.

When using JamaicaVM, it is hence not required to use non-heap memory areas for realtime tasks and it is possible for any thread to access objects on the heap. Furthermore, scoped memory provided by the classes defined in the RTSJ are available to normal threads as well.

The strict semantics of the RTSJ require a significant runtime overhead to check that an access to an object is legal. Since these checks are not needed by JamaicaVM, they are disabled by default. However, setting the option -strictRTSJ forces JamaicaVM to perform this checks.

If -strictRTSJ is set, the following checks are performed and the corresponding exceptions are thrown: MemoryAccessError: If a NoHeapRealtimeThread attempts to access an object stored in the normal Java heap, a MemoryAccessError is thrown

IllegalStateException: If a non-RealtimeThread attempts to enter a javax.realtime.MemoryArea or tries to access the scope stack through the methods getCurrentMemoryArea, getMemoryAreaStackDepth, getOuterMemoryArea or getInitialMemoryAreaIndex defined in class javax.realtime.RealtimeThread.

A.7 Profiling

Profiling can be used to guide the compilation process and to find a good trade-off between fast compiled code and smaller interpreted byte code. This is particularly important for systems with tight memory and CPU resources.

-profile

Build an application that collects information on the amount of run time spend for the execution of different methods. This information is printed to standard output after a test run of the application has been performed.

Profiling information can only be collected when using the JamaicaVM interpreter, compiled code cannot be profiled. Consequently, "-profile" does not work in combination with "-compile"

The information collected in one of several profiling runs can then be used as an input for the option -useProfile to guide the compilation process.

-useProfile <p>

Use profiling information collected using -profile to restrict compilation to those methods that were most frequently executed during the profiling run. The default percentage of methods that are to be compiled is 10, if -percentageCompiled is not set to different value.

-percentageCompiled <n>

Use profiling information collected using -profile to restrict compilation to those methods that were most frequently executed during the profiling run. The percentage of methods that are to be compiled is given as an argument to -percentageCompiled. It must be between 1 and 100. Selecting 100 causes compilation of all methods executed during the profiling run, i.e. all methods that were not called during profiling will not be compiled.

A.8 Native code

Native code is code written in a different programming language than Java (typically C or C++). This code can be called from within Java code using the Java Native Interface (JNI). Jamaica additionally offers the Jamaica Binary Interface (JBI) to enable a more efficient access to native code

-object <files>

Specify object files that contain native code that has to be linked to the destination executable. Unlike other Java implementation, Jamaica does not access native code through shared libraries. Instead, the object files that contain native code that is referenced from within Java code are linked into the destination application file.

This option expects a list of object files that are separated using the platform dependent path separator character (e.g., ":").

-include <dirs>

Add the specified directories to the include path. This path should contain the include files generated by jamaicah for the native code referenced from Java code. The include files are used to determine whether the Java Native Interface (JNI) or Jamaica Binary Interface (JBI) is used to access the native code.

This option expects a list of paths that are separated using the platform dependent path separator character (e.g., ":").

-XTRACTPROJECT

Do nothing but print the project name

-Xstats

Show compact classfile statistics. (default: not used.)

-XcountCalls

Count calls and print statistics after execution. (default: not used.)

-XcountArrays

Count array accesses and allocations and print statistics after execution. (default: not used.)

-XcountAllocs

Count allocations for arrays and objects. (default: not used.)

-XcountFields

Count accesses to fields. (default: not used.)

-Xpg

Create GCC-profiling version. (default: not used.)

Appendix B: The Real-Time Specification for Java APIs

This appendix lists all the classes and their public or protected methods that are defined in the Real-Time Specification for Java [RTSJ] and that were implemented for the AERO JVM (with minor exceptions such as Physical Memory areas that are not required by the AERO JVM specification). Not supported are the classes

LTPhysicalMemory
VTPhyiscalMemory
PhyiscalMemoryManager
PhysicalMemoryTypeFilter

since these classes are not required by the AERO JVM specification.

B.1 High Precision Timers and Clocks

New classes provide for more accurate timers and clocks than those available in the standard Java APIs.

```
public class AbsoluteTime extends HighResolutionTime {
    public AbsoluteTime();
    public AbsoluteTime(AbsoluteTime);
    public AbsoluteTime(java.util.Date);
    public AbsoluteTime(long,int);
    public AbsoluteTime absolute(Clock, AbsoluteTime);
    public AbsoluteTime absolute(Clock);
    public AbsoluteTime add(long, int);
    public AbsoluteTime add(long, int, AbsoluteTime);
    public final AbsoluteTime add(RelativeTime);
    public AbsoluteTime add(RelativeTime, AbsoluteTime);
    public java.util.Date getDate();
    public void set(java.util.Date);
    public final RelativeTime subtract(AbsoluteTime);
    public final RelativeTime subtract(AbsoluteTime, RelativeTime);
    public final AbsoluteTime subtract(RelativeTime);
    public AbsoluteTime subtract(RelativeTime, AbsoluteTime);
    public String toString();
    public RelativeTime relative(Clock);
    public RelativeTime relative(Clock, HighResolutionTime);
}

public abstract class Clock extends Object {
    public Clock();
    public static Clock getRealtimeClock();
    public abstract RelativeTime getResolution();
    public abstract AbsoluteTime getTime();
    public abstract void getTime(AbsoluteTime);
    public abstract void setResolution(RelativeTime);
}

public abstract class HighResolutionTime extends Object implements Comparable {
    protected long milli;
    protected int nano;
    public HighResolutionTime();
    public abstract AbsoluteTime absolute(Clock);
    public abstract AbsoluteTime absolute(Clock, AbsoluteTime);
    public int compareTo(HighResolutionTime);
    public int compareTo(Object);
    public boolean equals(HighResolutionTime);
    public boolean equals(Object);
    public final long getMilliseconds();
    public final int getNanoseconds();
    public int hashCode();
    public void set(HighResolutionTime);
    public void set(long);
    public void set(long, int);
    public static void waitForObject(Object, HighResolutionTime) throws InterruptedException;
    public abstract RelativeTime relative(Clock);
    public abstract RelativeTime relative(Clock, HighResolutionTime);
}

public class OneShotTimer extends Timer {
```

```
    public OneShotTimer(HighResolutionTime,AsyncEventHandler);
    public OneShotTimer(HighResolutionTime,Clock,AsyncEventHandler);
}


public class PeriodicTimer extends Timer {
    public PeriodicTimer(HighResolutionTime,RelativeTime,AsyncEventHandler);
    public PeriodicTimer(HighResolutionTime,RelativeTime,Clock,AsyncEventHandler);
    public ReleaseParameters createReleaseParameters();
    public AbsoluteTime getFireTime();
    public RelativeTime getInterval();
    public void setInterval(RelativeTime);
}


public class RationalTime extends RelativeTime {
    public RationalTime(int);
    public RationalTime(int,long,int) throws IllegalArgumentException;
    public RationalTime(int,RelativeTime) throws IllegalArgumentException;
    public AbsoluteTime absolute(Clock, AbsoluteTime);
    public int getFrequency();
    public RelativeTime getInterarrivalTime(RelativeTime);
    public RelativeTime getInterarrivalTime();
    public void set(long, int) throws IllegalArgumentException;
    public void setFrequency(int) throws ArithmeticException;
}


public class RelativeTime extends HighResolutionTime {
    public RelativeTime();
    public RelativeTime(long,int);
    public RelativeTime(RelativeTime);
    public AbsoluteTime absolute(Clock, AbsoluteTime);
    public AbsoluteTime absolute(Clock);
    public RelativeTime relative(Clock);
    public RelativeTime relative(Clock, HighResolutionTime);
    public RelativeTime add(long, int);
    public RelativeTime add(long, int, RelativeTime);
    public final RelativeTime add(RelativeTime);
    public RelativeTime add(RelativeTime, RelativeTime);
    public final RelativeTime subtract(RelativeTime);
    public RelativeTime subtract(RelativeTime, RelativeTime);
    public String toString();
}


public abstract class Timer extends AsyncEvent {
    protected AbsoluteTime FireAt;
    protected AbsoluteTime startTime;
    protected boolean isEnabled;
    protected Timer.WaitThread WaitT;
    protected Clock TheClock;
    protected boolean isPeriodic;
    protected RelativeTime interval;
    protected RelativeTime soLong;
    protected AbsoluteTime Now;
    protected boolean NoWaiting;
    protected Timer(HighResolutionTime,Clock,AsyncEventHandler);
    public boolean isRunning();
    public boolean stop();
    public ReleaseParameters createReleaseParameters();
    public void disable();
    public void enable();
    public void destroy();
    public Clock getClock();
    public AbsoluteTime getFireTime();
    public void reschedule(HighResolutionTime);
    public void start();
}
```

B.2 Realtime Threads

New thread classes permit the use of a wider range of priorites and provide stricter semantics such as priority preemptive scheduling and no interruptions by the garbage collector.

```
public class NoHeapRealtimeThread extends RealtimeThread {
    public NoHeapRealtimeThread(SchedulingParameters,MemoryArea) throws
       IllegalArgumentException;
    public NoHeapRealtimeThread(SchedulingParameters,ReleaseParameters,MemoryArea) throws
       IllegalArgumentException;
    public NoHeapRealtimeThread(SchedulingParameters,ReleaseParameters,MemoryParameters,
       MemoryArea,ProcessingGroupParameters,Runnable) throws IllegalArgumentException;
    public void start();
}


public class RealtimeThread extends Thread implements Schedulable {
    public RealtimeThread();
```

```
    public RealtimeThread(SchedulingParameters);
    public RealtimeThread(SchedulingParameters,ReleaseParameters);
    public RealtimeThread(SchedulingParameters,ReleaseParameters,MemoryParameters,
        MemoryArea,ProcessingGroupParameters,Runnable);
    public boolean addToFeasibility();
    public static RealtimeThread currentRealtimeThread() throws ClassCastException;
    public void deschedulePeriodic();
    public MemoryParameters getMemoryParameters();
    public ProcessingGroupParameters getProcessingGroupParameters();
    public ReleaseParameters getReleaseParameters();
    public Scheduler getScheduler();
    public SchedulingParameters getSchedulingParameters();
    public void interrupt();
    public boolean removeFromFeasibility();
    public void schedulePeriodic();
    public void setMemoryParameters(MemoryParameters) throws IllegalThreadStateException;
    public void setProcessingGroupParameters(ProcessingGroupParameters);
    public void setReleaseParameters(ReleaseParameters) throws IllegalThreadStateException;
    public void setScheduler(Scheduler, SchedulingParameters, ReleaseParameters,
        MemoryParameters, ProcessingGroupParameters) throws IllegalThreadStateException;
    public void setScheduler(Scheduler) throws IllegalThreadStateException;
    public void setSchedulingParameters(SchedulingParameters) throws
        IllegalThreadStateException;
    public static void sleep(Clock, HighResolutionTime) throws InterruptedException;
    public static void sleep(HighResolutionTime) throws InterruptedException;
    public boolean setIfFeasible(ReleaseParameters, MemoryParameters,
        ProcessingGroupParameters);
    public boolean setIfFeasible(ReleaseParameters, ProcessingGroupParameters);
    public boolean setIfFeasible(ReleaseParameters, MemoryParameters);
    public boolean addIfFeasible();
    public boolean waitForNextPeriod() throws IllegalThreadStateException;
    public static MemoryArea getCurrentMemoryArea();
    public void start();
    public boolean setProcessingGroupParametersIfFeasible(ProcessingGroupParameters);
    public boolean setSchedulingParametersIfFeasible(SchedulingParameters);
    public boolean setReleaseParametersIfFeasible(ReleaseParameters);
    public boolean setMemoryParametersIfFeasible(MemoryParameters);
    public static int getMemoryAreaStackDepth();
    public static MemoryArea getOuterMemoryArea(int);
    public static int getInitialMemoryAreaIndex();
}
```

B.3 Asynchronous Events

As an alternative execution environment for realtime code, asynchronous events are provided in the RTSJ APIs.

```
public class AsyncEvent extends Object {
    protected AsyncEventHandler h;
    public AsyncEvent();
    public void addHandler(AsyncEventHandler);
    public void removeHandler(AsyncEventHandler);
    public boolean handledBy(AsyncEventHandler);
    public void setHandler(AsyncEventHandler);
    public ReleaseParameters createReleaseParameters();
    public void bindTo(String) throws UnknownHappeningException;
    public void unbindTo(String) throws UnknownHappeningException;
    public void fire();
}
```

```
public class AsyncEventHandler extends Object implements Schedulable, Runnable {
    public AsyncEventHandler NextHandler;
    public AsyncEventHandler();
    public AsyncEventHandler(Runnable);
    public AsyncEventHandler(SchedulingParameters,ReleaseParameters,MemoryParameters,
        MemoryArea,ProcessingGroupParameters,Runnable);
    public AsyncEventHandler(boolean);
    public AsyncEventHandler(boolean,Runnable);
    public AsyncEventHandler(SchedulingParameters,ReleaseParameters,MemoryParameters,
        MemoryArea,ProcessingGroupParameters,boolean);
    public AsyncEventHandler(SchedulingParameters,ReleaseParameters,MemoryParameters,
        MemoryArea,ProcessingGroupParameters,boolean,Runnable);
    public void handleAsyncEvent();
    protected final int getPendingFireCount();
    protected final int getAndClearPendingFireCount();
    protected int getAndDecrementPendingFireCount();
    protected int getAndIncrementPendingFireCount();
    public final void run();
    public MemoryArea getMemoryArea();
    public boolean addToFeasibility();
    public boolean addIfFeasible();
    public boolean setIfFeasible(ReleaseParameters, MemoryParameters);
    public boolean setIfFeasible(ReleaseParameters, MemoryParameters,
        ProcessingGroupParameters);
    public boolean setReleaseParametersIfFeasible(ReleaseParameters);
    public boolean setProcessingGroupParametersIfFeasible(ProcessingGroupParameters);
```

```
    public boolean setIfFeasible(ReleaseParameters, ProcessingGroupParameters);
    public boolean setMemoryParametersIfFeasible(MemoryParameters);
    public MemoryParameters getMemoryParameters();
    public ReleaseParameters getReleaseParameters();
    public Scheduler getScheduler();
    public SchedulingParameters getSchedulingParameters();
    public ProcessingGroupParameters getProcessingGroupParameters();
    public boolean removeFromFeasibility();
    public void setMemoryParameters(MemoryParameters);
    public void setReleaseParameters(ReleaseParameters);
    public void setScheduler(Scheduler) throws IllegalThreadStateException;
    public void setScheduler(Scheduler, SchedulingParameters, ReleaseParameters,
       MemoryParameters, ProcessingGroupParameters) throws IllegalThreadStateException;
    public void setSchedulingParameters(SchedulingParameters);
    public void setProcessingGroupParameters(ProcessingGroupParameters);
    public boolean setSchedulingParametersIfFeasible(SchedulingParameters);
}


public abstract class BoundAsyncEventHandler extends AsyncEventHandler {
    public BoundAsyncEventHandler();
    public BoundAsyncEventHandler(SchedulingParameters,ReleaseParameters,MemoryParameters,
       MemoryArea,ProcessingGroupParameters,boolean,Runnable);
}


public final class POSIXSignalHandler extends Object {
    public static final int MAX_SIGNAL_NR;
    public static final int SIGHUP;
    public static final int SIGINT;
    public static final int SIGQUIT;
    public static final int SIGILL;
    public static final int SIGTRAP;
    public static final int SIGIOT;
    public static final int SIGABRT;
    public static final int SIGEMT;
    public static final int SIGFPE;
    public static final int SIGKILL;
    public static final int SIGBUS;
    public static final int SIGSEGV;
    public static final int SIGSYS;
    public static final int SIGPIPE;
    public static final int SIGALRM;
    public static final int SIGTERM;
    public static final int SIGUSR1;
    public static final int SIGUSR2;
    public static final int SIGCLD;
    public static final int SIGCHLD;
    public static final int SIGPWR;
    public static final int SIGWINCH;
    public static final int SIGURG;
    public static final int SIGPOLL;
    public static final int SIGIO;
    public static final int SIGSTOP;
    public static final int SIGTSTP;
    public static final int SIGCONT;
    public static final int SIGTTIN;
    public static final int SIGTTOU;
    public static final int SIGVTALRM;
    public static final int SIGPROF;
    public static final int SIGXCPU;
    public static final int SIGXFSZ;
    public static final int SIGWAITING;
    public static final int SIGLWP;
    public static final int SIGFREEZE;
    public static final int SIGTHAW;
    public static final int SIGCANCEL;
    public static final int SIGLOST;
    public POSIXSignalHandler();
    protected void finalize() throws Throwable;
    public static void addHandler(int, AsyncEventHandler);
    public static void removeHandler(int, AsyncEventHandler);
    public static void setHandler(int, AsyncEventHandler);
    public static void setEvent(int, AsyncEvent);
}
```

B.4 Scheduler and Schedulability

New APIs provide a framework to provide scheduling parameters, to develop generic schedulers and analyse the schedulability of an application.

```
public class AperiodicParameters extends ReleaseParameters {
    public AperiodicParameters(RelativeTime,RelativeTime,AsyncEventHandler,AsyncEventHandler);
    public boolean setIfFeasible(RelativeTime, RelativeTime);
}


public class ImportanceParameters extends PriorityParameters {
```

```
    public ImportanceParameters(int,int);
    public int getImportance();
    public void setImportance(int);
    public String toString();
}


public class MemoryParameters extends Object {
    public static final long NO_MAX;
    public MemoryParameters(long,long) throws IllegalArgumentException;
    public MemoryParameters(long,long,long) throws IllegalArgumentException;
    public long getAllocationRate();
    public long getMaxImmortal();
    public long getMaxMemoryArea();
    public void setAllocationRate(long);
    public boolean setMaxImmortalIfFeasible(long);
    public boolean setMaxMemoryAreaIfFeasible(long);
    public boolean setAllocationRateIfFeasible(int);
}


public class PeriodicParameters extends ReleaseParameters {
    public PeriodicParameters(HighResolutionTime,RelativeTime,RelativeTime,RelativeTime,
       AsyncEventHandler, AsyncEventHandler);
    public boolean setIfFeasible(RelativeTime, RelativeTime, RelativeTime);
    public HighResolutionTime getStart();
    public void setStart(HighResolutionTime);
    public RelativeTime getPeriod();
    public void setPeriod(RelativeTime);
}


public class PriorityParameters extends SchedulingParameters {
    public PriorityParameters(int);
    public int getPriority();
    public void setPriority(int) throws IllegalArgumentException;
    public String toString();
}


public class PriorityScheduler extends Scheduler {
    public static final int MAX_PRIORITY;
    public static final int MIN_PRIORITY;
    protected PriorityScheduler();
    public boolean isFeasible();
    public static PriorityScheduler instance();
    protected synchronized boolean addToFeasibility(Schedulable);
    protected synchronized boolean removeFromFeasibility(Schedulable);
    public boolean setIfFeasible(Schedulable, ReleaseParameters, MemoryParameters);
    public synchronized boolean setIfFeasible(Schedulable, ReleaseParameters, MemoryParameters,
       ProcessingGroupParameters);
    public void fireSchedulable(Schedulable);
    public native int getMaxPriority();
    public static int getMaxPriority(Thread);
    public int getMinPriority();
    public static int getMinPriority(Thread);
    public int getNormPriority();
    public static int getNormPriority(Thread);
    public String getPolicyName();
}


public class ProcessingGroupParameters extends Object {
    public ProcessingGroupParameters(HighResolutionTime,RelativeTime,RelativeTime,RelativeTime,
       AsyncEventHandler,AsyncEventHandler);
    public boolean setIfFeasible(RelativeTime, RelativeTime, RelativeTime);
    public RelativeTime getCost();
    public AsyncEventHandler getCostOverrunHandler();
    public RelativeTime getDeadline();
    public AsyncEventHandler getDeadlineMissHandler();
    public RelativeTime getPeriod();
    public HighResolutionTime getStart();
    public void setCost(RelativeTime);
    public void setCostOverrunHandler(AsyncEventHandler);
    public void setDeadline(RelativeTime);
    public void setDeadlineMissHandler(AsyncEventHandler);
    public void setPeriod(RelativeTime);
    public void setStart(HighResolutionTime);
}


public abstract class ReleaseParameters extends Object {
    protected ReleaseParameters();
    public boolean setIfFeasible(RelativeTime, RelativeTime, AsyncEventHandler,
       AsyncEventHandler);
    public boolean setIfFeasible(RelativeTime, RelativeTime);
    public RelativeTime getCost();
    public AsyncEventHandler getCostOverrunHandler();
    public RelativeTime getDeadline();
    public AsyncEventHandler getDeadlineMissHandler();
    public void setCost(RelativeTime);
    public void setCostOverrunHandler(AsyncEventHandler);
    public void setDeadline(RelativeTime);
    public void setDeadlineMissHandler(AsyncEventHandler);
}
```

```
public interface Schedulable extends Runnable
{
    public abstract boolean addToFeasibility();
    public abstract MemoryParameters getMemoryParameters();
    public abstract ReleaseParameters getReleaseParameters();
    public abstract Scheduler getScheduler();
    public abstract SchedulingParameters getSchedulingParameters();
    public abstract ProcessingGroupParameters getProcessingGroupParameters();
    public abstract boolean removeFromFeasibility();
    public abstract void setMemoryParameters(MemoryParameters);
    public abstract void setReleaseParameters(ReleaseParameters);
    public abstract void setScheduler(Scheduler) throws IllegalThreadStateException;
    public abstract void setScheduler(Scheduler, SchedulingParameters, ReleaseParameters,
       MemoryParameters, ProcessingGroupParameters) throws IllegalThreadStateException;
    public abstract void setSchedulingParameters(SchedulingParameters);
    public abstract void setProcessingGroupParameters(ProcessingGroupParameters);
    public abstract boolean setProcessingGroupParametersIfFeasible(ProcessingGroupParameters);
    public abstract boolean setSchedulingParametersIfFeasible(SchedulingParameters);
    public abstract boolean setReleaseParametersIfFeasible(ReleaseParameters);
    public abstract boolean setMemoryParametersIfFeasible(MemoryParameters);
}


public abstract class Scheduler extends Object {
    protected Scheduler();
    protected abstract boolean addToFeasibility(Schedulable);
    public boolean setIfFeasible(Schedulable, ReleaseParameters, MemoryParameters);
    public boolean setIfFeasible(Schedulable, ReleaseParameters, MemoryParameters,
       ProcessingGroupParameters);
    public static Scheduler getDefaultScheduler();
    public abstract String getPolicyName();
    public abstract boolean isFeasible();
    protected abstract boolean removeFromFeasibility(Schedulable);
    public static void setDefaultScheduler(Scheduler);
    public abstract void fireSchedulable(Schedulable);
}


public abstract class SchedulingParameters extends Object {
    public SchedulingParameters();
}


public class SporadicParameters extends AperiodicParameters {
    public static final String arrivalTimeQueueOverflowExcept;
    public static final String arrivalTimeQueueOverflowIgnore;
    public static final String arrivalTimeQueueOverflowReplace;
    public static final String arrivalTimeQueueOverflowSave;
    public static final String mitViolationExcept;
    public static final String mitViolationIgnore;
    public static final String mitViolationReplace;
    public static final String mitViolationSave;
    public SporadicParameters(RelativeTime,RelativeTime,RelativeTime,AsyncEventHandler,
       AsyncEventHandler);
    public boolean setIfFeasible(RelativeTime, RelativeTime, RelativeTime);
    public RelativeTime getMinimumInterarrival();
    public void setMinimumInterarrival(RelativeTime);
    public String getArrivalTimeQueueOverflowBehavior();
    public int getInitialArrivalTimeQueueLength();
    public String getMitViolationBehavior();
    public void setArrivalTimeQueueOverflowBehavior(String);
    public void setInitialArrivalTimeQueueLength(int);
    public void setMitViolationBehavior(String);
}
```

B.5 Asynchronous Transfer of Control and Thread Termination

The Asynchronous transfer of control mechanisms provide a safe way to throw exceptions from one thread into a different thread and to provide a safe means for thread termination that is not available in the standard Java APIs.

```
public class AsynchronouslyInterruptedException extends InterruptedException {
    public AsynchronouslyInterruptedException();
    public static AsynchronouslyInterruptedException getGeneric();
    public boolean enable();
    public boolean disable();
    public boolean isEnabled();
    public boolean fire();
    public boolean doInterruptible(Interruptible);
    public boolean happened(boolean);
    public static void propagate();
}


public interface Interruptible
    /* ACC_SUPER bit NOT set */
```

```
{
    public abstract void run(AsynchronouslyInterruptedException) throws
AsynchronouslyInterruptedException;
    public abstract void interruptAction(AsynchronouslyInterruptedException);
}


public class Timed extends AsynchronouslyInterruptedException {
    public Timed(HighResolutionTime) throws IllegalArgumentException;
    public boolean doInterruptible(Interruptible);
    public void resetTime(HighResolutionTime);
}
```

B.6 Control over Java Monitor Behaviour

Specific APIs permit an implementation to support changing the behaviour of Java monitors to different monitor control mechanisms such as the priority ceiling protocol. The default monitor behaviour in the RTSJ is to use priority inheritance to avoid unbounded priority inversion situations.

```
public abstract class MonitorControl extends Object {
    public MonitorControl();
    public static MonitorControl getMonitorControl(Object);
    public static MonitorControl getMonitorControl();
    public static void setMonitorControl(MonitorControl);
    public static void setMonitorControl(Object, MonitorControl);
}


public class PriorityCeilingEmulation extends MonitorControl {
    public PriorityCeilingEmulation(int);
    public int getDefaultCeiling();
}


public class PriorityInheritance extends MonitorControl {
    public PriorityInheritance();
    public static PriorityInheritance instance();
}
```

B.7 Memory Management Related APIs

New classes provide access to information on the garbage collector and specific memory areas that are not under the direct control of the garbage collector.

```
public abstract class GarbageCollector extends Object {
    public GarbageCollector();
    public abstract RelativeTime getPreemptionLatency();
}


public final class HeapMemory extends MemoryArea {
    public static synchronized HeapMemory instance();
    public long memoryRemaining();
    public long memoryConsumed();
}


public final class ImmortalMemory extends MemoryArea {
    public static synchronized ImmortalMemory instance();
}


public class ImmortalPhysicalMemory extends MemoryArea {
    public ImmortalPhysicalMemory(Object,long) throws SecurityException,
       SizeOutOfBoundsException, UnsupportedPhysicalMemoryException,
       MemoryTypeConflictException;
    public ImmortalPhysicalMemory(Object,long,long) throws SecurityException,
       SizeOutOfBoundsException, OffsetOutOfBoundsException,
       UnsupportedPhysicalMemoryException, MemoryTypeConflictException, MemoryInUseException;
    public ImmortalPhysicalMemory(Object,SizeEstimator) throws SecurityException,
       SizeOutOfBoundsException, UnsupportedPhysicalMemoryException,
       MemoryTypeConflictException;
    public ImmortalPhysicalMemory(Object,long,SizeEstimator) throws SecurityException,
       SizeOutOfBoundsException, OffsetOutOfBoundsException,
       UnsupportedPhysicalMemoryException, MemoryTypeConflictException, MemoryInUseException;
    public ImmortalPhysicalMemory(Object,long,Runnable) throws SecurityException,
       SizeOutOfBoundsException, UnsupportedPhysicalMemoryException,
       MemoryTypeConflictException;
```

```
    public ImmortalPhysicalMemory(Object,long,long,Runnable) throws SecurityException,
        SizeOutOfBoundsException, OffsetOutOfBoundsException,
        UnsupportedPhysicalMemoryException, MemoryTypeConflictException, MemoryInUseException;
    public ImmortalPhysicalMemory(Object,SizeEstimator,Runnable) throws SecurityException,
        SizeOutOfBoundsException, UnsupportedPhysicalMemoryException,
        MemoryTypeConflictException;
    public ImmortalPhysicalMemory(Object,long,SizeEstimator,Runnable) throws SecurityException,

        SizeOutOfBoundsException, OffsetOutOfBoundsException,
        UnsupportedPhysicalMemoryException, MemoryTypeConflictException, MemoryInUseException;
}


public class LTMemory extends ScopedMemory {
    public LTMemory(long,long);
    public LTMemory(long,long,Runnable);
    public LTMemory(SizeEstimator,SizeEstimator);
    public LTMemory(SizeEstimator,SizeEstimator,Runnable);
    public long getMaximumSize();
    public String toString();
}


public class LTPhysicalMemory extends ScopedMemory {
    public LTPhysicalMemory(Object,long) throws SecurityException, SizeOutOfBoundsException,
        UnsupportedPhysicalMemoryException, MemoryTypeConflictException;
    public LTPhysicalMemory(Object,long,long) throws SecurityException,
        SizeOutOfBoundsException, OffsetOutOfBoundsException,
        UnsupportedPhysicalMemoryException, MemoryTypeConflictException, MemoryInUseException;
    public LTPhysicalMemory(Object,SizeEstimator) throws SecurityException,
        SizeOutOfBoundsException, UnsupportedPhysicalMemoryException,
        MemoryTypeConflictException;
    public LTPhysicalMemory(Object,long,SizeEstimator) throws SecurityException,
        SizeOutOfBoundsException, OffsetOutOfBoundsException,
        UnsupportedPhysicalMemoryException, MemoryTypeConflictException, MemoryInUseException;
    public LTPhysicalMemory(Object,long,Runnable) throws SecurityException,
        SizeOutOfBoundsException, UnsupportedPhysicalMemoryException,
        MemoryTypeConflictException;
    public LTPhysicalMemory(Object,long,long,Runnable) throws SecurityException,
        SizeOutOfBoundsException, OffsetOutOfBoundsException,
        UnsupportedPhysicalMemoryException, MemoryTypeConflictException, MemoryInUseException;
    public LTPhysicalMemory(Object,SizeEstimator,Runnable) throws SecurityException,
        SizeOutOfBoundsException, UnsupportedPhysicalMemoryException,
        MemoryTypeConflictException;
    public LTPhysicalMemory(Object,long,SizeEstimator,Runnable) throws SecurityException,
        SizeOutOfBoundsException, OffsetOutOfBoundsException,
        UnsupportedPhysicalMemoryException, MemoryTypeConflictException, MemoryInUseException;
    public String toString();
}


public abstract class MemoryArea extends Object {
    protected MemoryArea(long);
    protected MemoryArea(SizeEstimator);
    protected MemoryArea(long,Runnable);
    protected MemoryArea(SizeEstimator,Runnable);
    protected void finalize() throws Throwable;
    public void enter() throws ScopedCycleException;
    public void enter(Runnable) throws ScopedCycleException;
    public static MemoryArea getMemoryArea(Object);
    public long memoryConsumed();
    public long memoryRemaining();
    public Object newArray(Class, int) throws IllegalAccessException, InstantiationException;
    public Object newInstance(Class) throws IllegalAccessException, InstantiationException;
    public Object newInstance(reflect.Constructor, Object[]) throws IllegalAccessException,
        InstantiationException;
    public long size();
    public void executeInArea(Runnable) throws InaccessibleAreaException;
}


public final class PhysicalMemoryManager extends Object {
    public static final String DMA;
    public static final String SHARED;
    public static final String ALIGNED;
    public static final String BYTESWAP;
    public PhysicalMemoryManager();
    public static final void registerFilter(Object, PhysicalMemoryTypeFilter) throws
        DuplicateFilterException, IllegalArgumentException;
    public static final void removeFilter(Object);
    public static boolean isRemovable(long, long);
    public static boolean isRemoved(long, long);
    public static void onRemoval(long, long, AsyncEventHandler);
    public static void onInsertion(long, long, AsyncEventHandler);
}


public interface PhysicalMemoryTypeFilter
    /* ACC_SUPER bit NOT set */
{
    public abstract long find(long, long);
    public abstract long vFind(long, long);
    public abstract void initialize(long, long, long);
    public abstract boolean isRemovable();
    public abstract boolean contains(long, long);
```

```
    public abstract void onRemoval(long, long, AsyncEventHandler);
    public abstract void onInsertion(long, long, AsyncEventHandler);
    public abstract boolean isPresent(long, long);
    public abstract int getVMAttributes();
    public abstract int getVMFlags();
}


public class RawMemoryAccess extends Object {
    protected long base;
    protected long size;
    public RawMemoryAccess(Object,long,long) throws SecurityException,
        OffsetOutOfBoundsException, SizeOutOfBoundsException,
        UnsupportedPhysicalMemoryException, MemoryTypeConflictException;
    public RawMemoryAccess(Object,long) throws SecurityException, OffsetOutOfBoundsException,
        SizeOutOfBoundsException, UnsupportedPhysicalMemoryException,
        MemoryTypeConflictException;
    public byte getByte(long) throws OffsetOutOfBoundsException, SizeOutOfBoundsException;
    public void getBytes(long, byte[], int, int) throws OffsetOutOfBoundsException,
        SizeOutOfBoundsException;
    public int getInt(long) throws OffsetOutOfBoundsException, SizeOutOfBoundsException;
    public void getInts(long, int[], int, int) throws OffsetOutOfBoundsException,
        SizeOutOfBoundsException;
    public long getLong(long) throws OffsetOutOfBoundsException, SizeOutOfBoundsException;
    public void getLongs(long, long[], int, int) throws OffsetOutOfBoundsException,
        SizeOutOfBoundsException;
    public short getShort(long) throws OffsetOutOfBoundsException, SizeOutOfBoundsException;
    public void getShorts(long, short[], int, int) throws OffsetOutOfBoundsException,
        SizeOutOfBoundsException;
    public void setByte(long, byte) throws OffsetOutOfBoundsException,
        SizeOutOfBoundsException;
    public void setBytes(long, byte[], int, int) throws OffsetOutOfBoundsException,
        SizeOutOfBoundsException;
    public void setInt(long, int) throws OffsetOutOfBoundsException, SizeOutOfBoundsException;
    public void setInts(long, int[], int, int) throws OffsetOutOfBoundsException,
        SizeOutOfBoundsException;
    public void setLong(long, long) throws OffsetOutOfBoundsException,
        SizeOutOfBoundsException;
    public void setLongs(long, long[], int, int) throws OffsetOutOfBoundsException,
        SizeOutOfBoundsException;
    public void setShort(long, short) throws OffsetOutOfBoundsException,
        SizeOutOfBoundsException;
    public void setShorts(long, short[], int, int) throws OffsetOutOfBoundsException,
        SizeOutOfBoundsException;
    public long getMappedAddress();
    public long map();
    public long map(long);
    public long map(long, long);
    public void unmap();
}


public class RawMemoryFloatAccess extends RawMemoryAccess {
    public RawMemoryFloatAccess(Object,long) throws SecurityException,
        OffsetOutOfBoundsException, SizeOutOfBoundsException,
        UnsupportedPhysicalMemoryException, MemoryTypeConflictException;
    public RawMemoryFloatAccess(Object,long,long) throws SecurityException,
        OffsetOutOfBoundsException, SizeOutOfBoundsException,
        UnsupportedPhysicalMemoryException, MemoryTypeConflictException;
    public double getDouble(long) throws OffsetOutOfBoundsException, SizeOutOfBoundsException;
    public void getDoubles(long, double[], int, int) throws OffsetOutOfBoundsException,
        SizeOutOfBoundsException;
    public float getFloat(long) throws OffsetOutOfBoundsException, SizeOutOfBoundsException;
    public void getFloats(long, float[], int, int) throws OffsetOutOfBoundsException,
        SizeOutOfBoundsException;
    public void setFloat(long, float) throws OffsetOutOfBoundsException,
        SizeOutOfBoundsException;
    public void setFloats(long, float[], int, int) throws OffsetOutOfBoundsException,
        SizeOutOfBoundsException;
    public void setDouble(long, double) throws OffsetOutOfBoundsException,
        SizeOutOfBoundsException;
    public void setDoubles(long, double[], int, int) throws OffsetOutOfBoundsException,
        SizeOutOfBoundsException;
}


public abstract class ScopedMemory extends MemoryArea {
    public static FinalizeNode lastFinalizeNode;
    public ScopedMemory(long);
    public ScopedMemory(long,Runnable);
    public ScopedMemory(SizeEstimator);
    public ScopedMemory(SizeEstimator,Runnable);
    public void enter() throws ScopedCycleException;
    public void enter(Runnable) throws ScopedCycleException;
    public long getMaximumSize();
    public Object getPortal();
    public void setPortal(Object);
    public int getReferenceCount();
    public String toString();
    public void join(HighResolutionTime) throws InterruptedException;
    public void join() throws InterruptedException;
    public void joinAndEnter() throws InterruptedException, ScopedCycleException;
    public void joinAndEnter(HighResolutionTime) throws InterruptedException,
        ScopedCycleException;
    public void joinAndEnter(Runnable) throws InterruptedException, ScopedCycleException;
```

```
    public void joinAndEnter(Runnable, HighResolutionTime) throws InterruptedException,
        ScopedCycleException;
}


public final class SizeEstimator extends Object {
    public SizeEstimator();
    public long getEstimate();
    public void reserve(Class, int);
    public void reserve(SizeEstimator);
    public void reserve(SizeEstimator, int);
}


public class VTMemory extends ScopedMemory {
    public VTMemory(long,long);
    public VTMemory(long,long,Runnable);
    public VTMemory(SizeEstimator,SizeEstimator);
    public VTMemory(SizeEstimator,SizeEstimator,Runnable);
    public long getMaximumSize();
    public String toString();
}

public class VTPhysicalMemory extends ScopedMemory {
    public VTPhysicalMemory(Object,long) throws SecurityException, SizeOutOfBoundsException,
        UnsupportedPhysicalMemoryException, MemoryTypeConflictException;
    public VTPhysicalMemory(Object,long,long) throws SecurityException,
        SizeOutOfBoundsException, OffsetOutOfBoundsException,
        UnsupportedPhysicalMemoryException, MemoryTypeConflictException, MemoryInUseException;
    public VTPhysicalMemory(Object,SizeEstimator) throws SecurityException,
        SizeOutOfBoundsException, UnsupportedPhysicalMemoryException,
        MemoryTypeConflictException;
    public VTPhysicalMemory(Object,long,SizeEstimator) throws SecurityException,
        SizeOutOfBoundsException, OffsetOutOfBoundsException,
        UnsupportedPhysicalMemoryException, MemoryTypeConflictException, MemoryInUseException;
    public VTPhysicalMemory(Object,long,Runnable) throws SecurityException,
        SizeOutOfBoundsException, UnsupportedPhysicalMemoryException,
        MemoryTypeConflictException;
    public VTPhysicalMemory(Object,long,long,Runnable) throws SecurityException,
        SizeOutOfBoundsException, OffsetOutOfBoundsException,
        UnsupportedPhysicalMemoryException, MemoryTypeConflictException, MemoryInUseException;
    public VTPhysicalMemory(Object,SizeEstimator,Runnable) throws SecurityException,
        SizeOutOfBoundsException, UnsupportedPhysicalMemoryException,
        MemoryTypeConflictException;
    public VTPhysicalMemory(Object,long,SizeEstimator,Runnable) throws SecurityException,
        SizeOutOfBoundsException, OffsetOutOfBoundsException,
        UnsupportedPhysicalMemoryException, MemoryTypeConflictException, MemoryInUseException;
    public String toString();
}
```

B.8 Additional Runtime Errors and Exceptions

The new features require some additional runtime checks. New exceptions and errors are introduced for this purpose:

```
public class ArrivalTimeQueueOverflowException extends Exception implements
        java.io.Serializable {
    public ArrivalTimeQueueOverflowException();
    public ArrivalTimeQueueOverflowException(String);
}


public class DuplicateFilterException extends Exception implements java.io.Serializable {
    public DuplicateFilterException();
    public DuplicateFilterException(String);
}


public class IllegalAssignmentError extends Error implements java.io.Serializable {
    public IllegalAssignmentError();
    public IllegalAssignmentError(String);
}


public class InaccessibleAreaException extends Exception implements java.io.Serializable {
    public InaccessibleAreaException();
    public InaccessibleAreaException(String);
}


public class MITViolationException extends Exception {
    public MITViolationException();
    public MITViolationException(String);
}
```

```
public class MemoryAccessError extends Error implements java.io.Serializable {
    public MemoryAccessError();
    public MemoryAccessError(String);
}


public class MemoryInUseException extends RuntimeException implements java.io.Serializable {
    public MemoryInUseException();
    public MemoryInUseException(String);
}


public class MemoryScopeException extends Exception implements java.io.Serializable {
    public MemoryScopeException();
    public MemoryScopeException(String);
}


public class MemoryTypeConflictException extends Exception implements java.io.Serializable {
    public MemoryTypeConflictException();
    public MemoryTypeConflictException(String);
}


public class OffsetOutOfBoundsException extends Exception implements java.io.Serializable {
    public OffsetOutOfBoundsException();
    public OffsetOutOfBoundsException(String);
}


public class ResourceLimitError extends Error implements java.io.Serializable {
    public ResourceLimitError();
    public ResourceLimitError(String);
}


public class ScopedCycleException extends RuntimeException implements java.io.Serializable {
    public ScopedCycleException();
    public ScopedCycleException(String);
}


public class SizeOutOfBoundsException extends Exception implements java.io.Serializable {
    public SizeOutOfBoundsException();
    public SizeOutOfBoundsException(String);
}


public class ThrowBoundaryError extends Error implements java.io.Serializable {
    public ThrowBoundaryError();
    public ThrowBoundaryError(String);
}


public class UnknownHappeningException extends RuntimeException implements java.io.Serializable
        {
    public UnknownHappeningException();
    public UnknownHappeningException(String);
}


public class UnsupportedPhysicalMemoryException extends Exception implements
        java.io.Serializable {
    public UnsupportedPhysicalMemoryException();
    public UnsupportedPhysicalMemoryException(String);
}
```

B.9 Other APIs

Further APIs include security features, basis realtime system settings and wait-free queues for communication between NoHeapRealtimeThreads and other threads.

```
public class RealtimeSecurity extends Object {
    public RealtimeSecurity();
    public void checkAccessPhysical() throws SecurityException;
    public void checkAccessPhysicalRange(long, long) throws SecurityException;
    public void checkSetFilter() throws SecurityException;
    public void checkSetScheduler() throws SecurityException;
}


public final class RealtimeSystem extends Object {
    public static final byte BIG_ENDIAN;
    public static final byte BYTE_ORDER;
    public static final byte LITTLE_ENDIAN;
    public RealtimeSystem();
    public static GarbageCollector currentGC();
```

```
    public static int getConcurrentLocksUsed();
    public static int getMaximumConcurrentLocks();
    public static RealtimeSecurity getSecurityManager();
    public static void setMaximumConcurrentLocks(int);
    public static void setMaximumConcurrentLocks(int, boolean);
    public static void setSecurityManager(RealtimeSecurity);
}


public class WaitFreeDequeue extends Object {
    public WaitFreeDequeue(Thread,Thread,int,MemoryArea) throws IllegalArgumentException,
       IllegalAccessException, ClassNotFoundException, InstantiationException;
    public Object nonBlockingRead();
    public boolean blockingWrite(Object) throws MemoryScopeException;
    public boolean nonBlockingWrite(Object);
    public Object blockingRead();
    public boolean force(Object);
}

public class WaitFreeReadQueue extends Object {
    public WaitFreeReadQueue(Thread,Thread,int,MemoryArea,boolean) throws
       IllegalArgumentException, InstantiationException, ClassNotFoundException,
       IllegalAccessException;
    public WaitFreeReadQueue(Thread,Thread,int,MemoryArea) throws IllegalArgumentException,
       InstantiationException, ClassNotFoundException, IllegalAccessException;
    public void clear();
    public boolean isEmpty();
    public boolean isFull();
    public Object read();
    public int size();
    public void waitForData();
    public boolean write(Object) throws MemoryScopeException;
}

public class WaitFreeWriteQueue extends Object {
    public WaitFreeWriteQueue(Thread,Thread,int,MemoryArea) throws IllegalArgumentException,
       InstantiationException, ClassNotFoundException, IllegalAccessException;
    public void clear();
    public boolean isEmpty();
    public boolean isFull();
    public Object read();
    public int size();
    public boolean force(Object) throws MemoryScopeException;
    public boolean write(Object) throws MemoryScopeException;
}
```