INTERNATIONAL J CONSORTIUM SPECIFICATION
Real-Time Core Extensions

P.O. Box 1565

Cupertino, CA 95015-1565

USA

www.j-consortium.org

Copyright J Consortium 1999, 2000

Permission is granted by the J Consortium to reproduce this International Specification for the purpose of review and comment, provided this notice is included. All other rights are reserved.

THIS SPECIFICATION IS BEING OFFERED WITHOUT ANY WARRANTY WHATSOEVER, AND IN PARTICULAR, ANY WARRANTY OF NON-INFRINGEMENT IS EXPRESSLY DISCLAIMED. ANY IMPLEMENTATION OF THIS SPECIFICATION SHALL BE MADE ENTIRELY AT THE IMPLEMENTER'S OWN RISK, AND NEITHER THE J CONSORTIUM, NOR ANY OF ITS MEMBERS OR SUBMITTERS, SHALL HAVE ANY LIABILITY WHATSOEVER TO ANY IMPLEMENTER OR THIRD PARTY FOR ANY DAMAGES OF ANY NATURE WHATSOEVER DIRECTLY OR INDIRECTLY ARISING FROM THE IMPLEMENTATION OF THIS SPECIFICATION.

1.0 Scope 1
2.0 Terminology Conventions 1
2.1 Normative Terms 1
Shall 1
Shall not 1
May 1
May not 1
Should 1
Should not 1
Can 2
Implementation-defined 2
Unspecified behavior 2
Undefined behavior 2
2.2 Program Language and Technical Terminology 2
Static properties 2
Dynamic properties 2
Java 2
Baseline 2
Core 3
Extended Baseline Virtual Machine 3
Core Components 3
Core Methods 3
Core-Baseline Methods 3
Allocation Context 3
Asynchronous Transfer of Control (ATC) 3
Green Threads 3
RTOS 3
Base Priority 4
Active Priority 4
Never-Scheduled Priority 4
I/O Channel, Memory-Mapped Access, and I/O-Space Access 4
2.3 Architectural Components 4
2.4 Notational Shorthand 5
3.0 The Specification 6
3.1 Conformity Assessment 6
3.1.1 A Conforming Core Class File 6
3.1.2 A Conforming Static Core Application 6
3.1.3 A Conforming Dynamic Core Application 6
3.1.4 A Conforming Core Verifier 6
3.1.5 A Conforming Static Core Development Environment 6
3.1.6 A Conforming Static Core Executable Load Image 7

Copyright 1999, 2000 J Consortium, All Rights Reserved

3.1.7 A Conforming Dynamic Core Development Environment 7
3.1.8 A Conforming Static Core Linker 7
3.1.9 A Conforming Core Native Interface Compiler 8
3.2 Core Objects 8
3.3 Partitioning of Memory 9
3.3.1 Partitioning Protocol from Core programmer’s perspective 11
3.3.2 Partitioning Protocol from the Baseline programmer’s perspective 12
3.4 Architectural Overview of the Core Development Architecture 12
Core Source Files 12
Stylized Core Source Files 12
Core Class Files 12
Baseline Compiler 13
Figure 1: Overview of Real-Time Core Development Architecture 13
Core Verifier 14
Core Native Compiler 14
Native-Targeted Core Class Files 14
Core Class Libraries 14
Core Execution Environment 14
Static Core Execution Environment 14
Static Core Executable Load Image 14
Core Static Linker 15
Dynamic Core Execution Environment 15
Core Class Loader 15
3.5 Core Class Files 15
3.5.1 The Core Verifier 18
3.5.2 The Core Class Loader 18
3.6 Special Notations for Stylized Core Source Code 20
3.7 Core Priorities 21
3.8 Synchronization Issues 21
3.9 Task Execution Model for Execution of Core-Baseline Methods 24
3.10 The Core Memory Model 24
3.11 Abort Mechanism and Asynchronous Transfer of Control in General 25
3.11.1 Asynchronous Transfer of Control 26
3.12 Stack Allocation of Dynamic Objects 27
3.13 Initialization and Class Loading 29
3.14 Execution-Time Analyzable Code 29
Figure 2: Analyzable Conditional Control Flow 30
Figure 3: Analyzable Multi-Way Conditional Control Flow 31
3.14.1 Analyzability of Core Source Code 33
3.14.2 Predictability of the Core Execution Environment 34
Table 1: Predictability Requirements for Core API Libraries 35
Table 2: Predictability Requirements for the C/Native API 56

3.15 Core Class Loading API Overview 57
3.16 C/Native API 57
3.16.1 Obtaining Access to Core Objects 57
coreRegistryLookup() 57
3.16.2 Understanding Core Resource Needs and Contention 57
maxCorePriority() 57
minCorePriority() 58
corePriorityMap() 58
maxBaselinePriority() 58
minBaselinePriority() 58
coreInterruptLevels() 58
3.16.3 Synchronizing and Coordinating with the Baseline Domain 58
semaphoreP() 58
semaphoreV() 59
semaphoreVall() 59
enterSynchronized() 59
exitSynchronized() 60
3.17 The Core API 60
3.17.1 The CoreObject Class 60
CoreObject Constructor 61
CoreObject.clone() 61
CoreObject.equals() 61
CoreObject.getClass() 61
CoreObject.hashCode() 61
CoreObject.notify() 61
CoreObject.notifyAll() 62
CoreObject.toString() 62
CoreObject.wait() 62
CoreObject.arrayAddress() 62
CoreObject.sizeof() 62
3.17.2 The CoreThrowable Class 63
CoreThrowable Constructors 63
CoreThrowable.getMessage() 63
3.17.3 The CoreRuntimeException Class 63
CoreRuntimeException Constructors 63
3.17.4 The CoreException Class 63
CoreException Constructors 64
3.17.5 The ScopedException Class 64
ScopedException Constructors 64
ScopedException.enable() 64
ScopedException.disable() 64
3.17.6 The CoreClass Class 65

Copyright 1999, 2000 J Consortium, All Rights Reserved

CoreClass.forName() 65
CoreClass.getComponentType() 65
CoreClass.isArray() 65
CoreClass.isAssignableFrom() 65
CoreClass.isInstance() 65
CoreClass.isInterface() 65
CoreClass.isPrimitive() 65
CoreClass.newInstance() 65
CoreClass.toString() 66
CoreClass.verification() 66
CoreClass.loadClass() 66
CoreClass.unloadClass() 66
3.17.7 The CoreArray Class 66
Table 3: Core Array Representation Within Baseline Domain 67
length() 68
atGet() 68
atPut() 68
3.17.8 The AllocationContext Class 68
AllocationContext Constructors 69
AllocationContext.available() 70
AllocationContext.allocated() 70
AllocationContext.release() 70
3.17.9 The SpecialAllocation Class 70
SpecialAllocation.context() 70
SpecialAllocation.run() 70
SpecialAllocation.execute() 71
3.17.10 The PCP Interface 71
PCP.ceilingPriority() 72
3.17.11 The Atomic Interface 72
3.17.12 The CoreString Class 73
CoreString Constructors 73
CoreString.charAt() 73
CoreString.hashCode() 73
CoreString.equals() 73
CoreString.length() 74
3.17.13 The DynamicCoreString Class 74
DynamicCoreString Constructors 74
DynamicCoreString.concat() 74
DynamicCoreString.getChars() 74
DynamicCoreString.substring() 75
DynamicCoreString.toCharArray() 75
DynamicCoreString.toLowerCase() 75

	DynamicCoreString.toUpperCase()	75
3.17.14	The ATCEventHandler class	75
	ATCEventHandler Constructor	76
	ATCEventHandler.handleATCEvent()	76
3.17.15	The ATCEvent class	76
	ATCEvent Constructor	76
	ATCEvent.defaultAction()	76
3.17.16	The CoreRegistry class	76
	CoreRegistry.stackAllocation()	76
	CoreRegistry.registerStackable()	77
	CoreRegistry.registerBaseline()	77
	CoreRegistry.registerCoreClass()	77
	CoreRegistry.coerce()	77
	CoreRegistry.profiles()	78
	CoreRegistry.publish()	78
	CoreRegistry.unpublish()	78
3.17.17	The SignalingSemaphore Class	79
	SignalingSemaphore.P()	79
	SignalingSemaphore.V()	79
	SignalingSemaphore.Vall()	79
	SignalingSemaphore.numWaiters()	79
3.17.18	The CountingSemaphore Class	80
	CountingSemaphore.P()	80
	CountingSemaphore.V()	80
	CountingSemaphore.numWaiters()	80
	CountingSemaphore.count()	80
3.17.19	The Mutex Class	81
	Mutex Constructors	81
	Mutex.lock()	81
	Mutex.unlock()	81
3.17.20	The Configuration Class	81
	Configuration.tick_duration	82
	Configuration.ticks_per_slice	82
	Configuration.uptime_precision	82
	Configuration.default_stack_size	82
	Configuration.stack_overflow_checking	82
	Configuration.min_core_priority	82
	Configuration.system_priority_map	82
	Configuration.little_endian	83
3.17.21	The Time Class	83
	Time.tickDuration()	83
	Time.uptimePrecision()	83

 Copyright 1999, 2000 J Consortium, All Rights Reserved

	Time.day()	83
	Time.h()	83
	Time.hertz()	84
	Time.m()	84
	Time.ms()	84
	Time.ns()	84
	Time.s()	84
	Time.toString()	84
	Time.uptime()	85
	Time.us()	85
3.17.22	The Unsigned class	85
	Unsigned.compare()	85
	Unsigned.ge()	85
	Unsigned.gt()	85
	Unsigned.le()	86
	Unsigned.lt()	86
	Unsigned.eq()	86
	Unsigned.neq()	86
	Unsigned.toByte()	87
	Unsigned.toShort()	87
	Unsigned.toInt()	87
	Unsigned.toLong()	87
	Unsigned.toString()	87
	Unsigned.toHexString()	88
3.17.23	The CoreTask Class	88
	CoreTask Constructor	88
	Static Methods	89
	CoreTask.currentTask()	89
	CoreTask.defaultStackSize()	89
	CoreTask.maxBaselinePriority()	89
	CoreTask.maxCorePriority()	89
	CoreTask.maxSystemPriority()	90
	CoreTask.minBaselinePriority()	90
	CoreTask.minCorePriority()	90
	CoreTask.minSystemPriority()	90
	CoreTask.numInterruptPriorities()	90
	CoreTask.stackOverflowChecking()	90
	CoreTask.systemPriorityMap()	90
	CoreTask.ticksPerSlice()	91
	Instance Methods	91
	CoreTask.abort()	91
	CoreTask.abortWorkException()	91

	CoreTask.asyncHandler()	91
	CoreTask.join()	91
	CoreTask.resume()	92
	CoreTask.setPriority()	92
	CoreTask.signalAsync()	92
	CoreTask.sleep()	93
	CoreTask.sleepUntil()	93
	CoreTask.stackDepth()	93
	CoreTask.stackSize()	93
	CoreTask.start()	93
	CoreTask._start()	93
	CoreTask.stop()	93
	CoreTask.suspend()	94
	CoreTask.systemPriority()	94
	CoreTask.work()	94
	CoreTask.yield()	94
3.17.24	The ISR_Task Class	94
	ISR_Task Constructor	95
	ISR_Task.serviced()	95
	ISR_Task.trigger()	96
	ISR_Task.work()	96
	ISR_Task.ceilingPriority()	96
	ISR_Task.arm()	96
	ISR_Task.disarm()	97
3.17.25	The SporadicTask Class	97
	SporadicTask Constructor	97
	SporadicTask.trigger()	98
	SporadicTask.work()	98
	SporadicTask.pendingCount()	98
	SporadicTask.clearPending()	98
3.17.26	The IOPort class	98
	IOPort.createIOPort()	99
	IOPort.readByte()	99
	IOPort.writeByte()	99
	IOPort.readShort()	99
	IOPort.writeShort()	99
	IOPort.readInt()	100
	IOPort.writeInt()	100
	IOPort.readLong()	100
	IOPort.writeLong()	100
3.17.27	Core Throwable Types	100
	Table 4: Core CoreThrowable Classes	102

Copyright 1999, 2000 J Consortium, All Rights Reserved

4.0 Baseline API 103
Semaphore Operations 104
CoreTask Operations 104
Core Execution Profiles 104
Starting Up a Core Execution Environment 104
4.1 The BaselineCoreClassLoader Class 104
BaselineCoreClassLoader Constructors 104
BaselineCoreClassLoader semantics 104
4.2 The CoreDomain Class 105
CoreDomain.lookup() 105
CoreDomain.defineClass() 105
CoreDomain.loadClass() 105
CoreDomain.instantiate() 106
CoreDomain.profiles() 106
4.3 The ObjectNotFoundException Class 106
4.4 The CoreBaselineThrowable Class 106
CoreBaselineThrowable Constructors 106
CoreBaselineThrowable.getCoreThrowable() 106
4.5 The CoreBaselineRuntimeException Class 107
CoreBaselineRuntimeException Constructor 107
CoreBaselineRuntimeException.getCoreException() 107
4.6 The CoreBaselineException Class 107
CoreBaselineException Constructors 107
CoreBaselineException.getCoreException() 107

5.0 Acknowledgments. 108

6.0 Informative References 108

Annex A History. 109
A.1 Revision 1.0.14 109
A.2 Revision 1.0.13 110
A.3 Revision 1.0.12 111
A.4 Revision 1.0.11 112
A.5 Revision 1.0.10 112
A.6 Revision 1.0.9 112
A.7 Revision 1.0.8 113
A.8 Revision 1.0.7 114
A.9 Revision 1.0.6 116
A.10 Revision 1.0.5 120
A.11 Revision 1.0.4 122
A.12 Revision 1.0.3 122
A.13 Revision 1.0.2 125

Annex B Requirements for the Core Specification 126
B.1 The Working Principles of the Real-Time Java Working Group 126
B.2 Additional Requirements 127

Annex C Background and Rationale 130
C.1 Historical Background 130
C.1.1 NIST Requirements for the Real-Time Core 130
C.2 NCITS Principles for Real-Time Core 130
C.3 Rationale for Partitioning of Memory 137
C.4 Comments Regarding the Core Verifier 138
C.5 Comments on Syntactic Core Extensions 138
Figure 4: Overview of Real-Time Core Development Architecture 139
Special Notations for Syntactic Core Source Code 140
C.6 Clarification and Rationale re: Stack Allocation 141
C.7 Motivation for Special Class Loading Semantics 143
C.8 Clarifications re: Execution Time Analyzability 143
C.9 Rationale for Core Class Loading Requirements 143
C.10 Comments on Run-Time Differentiation between Core and Baseline Tasks 143
C.11 Comments re: the PCP Interface 144
C.12 Rationale for the CoreString and DynamicCoreString Specifications 145
C.13 Rationale for Semaphores to Complement Built-In Java Primitives 145
C.14 Rationale for the Mutex Class 145
C.15 Comments on Loading and Starting Core Tasks from Baseline Domain 145
C.16 Comments on Explicit Memory Management 146
C.17 Rationale and Discussion Regarding Asynchronous Transfer of Control 146
Abortion of a task 149
Timing out a sequence of code 149
Nested timeouts 150
Software interrupts 151
System mode changes 151
C.18 Comments re: low-level I/O Services 151

Annex D Implementation Suggestions 153
D.1 Comments on the Implementation of Partitioned Heaps (Section 3.3) 153
D.2 Comments on Implementation of Multiple Method Tables (Section 3.3) 153
D.3 Comments on Implementation of Stack Allocation 154
Dynamic stack allocation 154
Static stack allocation 154
Figure 5: Method Tables for Core Objects 154

Copyright 1999, 2000 J Consortium, All Rights Reserved

Real-Time Core Extensions

This document represents a draft revision to the specification for Real-Time Core Extensions for the Java[1] Platform, based on the collective work of the members of the J Consortium's Real-Time Java Working Group. Included in this document is discussion of requirements, historical perspectives and rationale, and suggestions for implementation of the specification.

Send comments to rtcore@j-consortium.org.

1. Java is a registered trademark of Sun Microsystems, Inc. in the United States and other countries.

1.0 Scope

This International Specification describes the form and meaning of programs written to make use of Real-Time Core Extensions, known throughout this document as simply "Core", on single-processor computers. The document's purpose is to promote the portability of Core application software and to ensure compatibility between implementations of Core development tools and run-time environments.

2.0 Terminology Conventions

2.1 Normative Terms

Throughout this document, the following terms shall have the meanings defined herein:

Shall. This identifies a conformance requirement.

Shall not. This identifies a prohibited feature or behavior.

May. This identifies an optional feature or behavior.

May not. This means the same as "need not".

Should. This identifies a recommended practice, but is not required.

Should not. This identifies a practice that is not recommended, but is not prohibited.

Copyright 1999, 2000 J Consortium, All Rights Reserved

Can. This identifies features or behavior that are available to an application. Implementations shall support such features and behaviors as conformance requirements.

Implementation-defined. This identifies behavior for a correct program construct and correct data that depends on the characteristics of the implementation, and shall be documented for each implementation. Example: the content of a required diagnostic message.

Unspecified behavior. This identifies behavior for a correct program construct and correct data, for which the specification explicitly imposes no requirements. Example: the order in which the arguments to a function are evaluated.

Undefined behavior. This identifies behavior upon the use of a non-portable or erroneous program construct, erroneous data, or indeterminately valued objects, for which this specification imposes no requirements.

2.2 Program Language and Technical Terminology

Static properties. With regard to computer programming languages, a static property is an attribute of a computer program that is determined at compile or link time rather than run time. Attributes that cannot be determined at compile time are called dynamic properties. Static linking describes the process of linking software components together prior to run time. Static memory management describes a mechanism in which the compiler determines that particular memory cells are required for execution of the program (or program component) and sets that memory aside at the moment the program begins to execute and doesn't reclaim that memory until the program (or program component) finishes its execution.

Dynamic properties. With regard to computer programming languages, a dynamic property is an attribute of a computer program that cannot be determined at compile time, but must instead be determined at run time. Attributes that can be determined at compile time are called static properties. Dynamic linking describes the process of linking software components together on the fly, while programs are running. Dynamic memory management describes a mechanism in which the program issues requests for allocation of new memory while it is running, and in which particular previously allocated objects are released by the application and reclaimed while the program continues to run.

Java. Throughout this document, the word "Java" is a trademark of Sun Microsystems in the U.S. and other countries. In this document, "Java" is used to describe the Java programming language and programming platforms as these were originally described by Sun Microsystems in references 5 and 8, including the many variants that have come into existence since the original specifications were published.

Baseline. A Core Execution Environment may, but need not, be combined with a traditional Java virtual machine, as illustrated in Figure 1 on page 13. When combined in this manner, the traditional (non-real-time) Java virtual machine and the programming language that is used to develop applications for execution within the traditional Java virtual machine are known as Baseline components. This specification imposes no constraints on which version of Java is implemented by the Baseline component except to

Copyright 1999, 2000 J Consortium, All Rights Reserved

require that the Baseline component implement the Java services described in Section 4.0.

Core. To emphasize the distinction between non-real-time Java technologies, and the special real-time variant of the Java programming language that is described in this document, we use the word "Core" to describe software components designed to run in the Core Execution Environment, as it is described in this document.

Extended Baseline Virtual Machine. A Core Execution Environment may, but need not, be combined with a Baseline virtual machine. The combination of a Core Execution Environment with a Baseline virtual machine is known as an Extended Baseline Virtual Machine.

Core Components. A Core Component is a software component written as a Core Source File, designed to run within the Core Execution Environment.

Core Methods. A Core Method is a method of a Core Component which is only visible to other Core Components. Contrast this with Core-Baseline methods.

Core-Baseline Methods. A Core-Baseline method is a method of a Core Component that is only visible to Baseline components and from within other Core-Baseline methods. Contrast this with Core Methods.

Allocation Context. An Allocation Context is an abstraction that serves to logically group a number of allocated Core objects. When an Allocation Context is released, all of the memory required to represent the Core objects that were allocated within that Allocation Context immediately becomes eligible to be reclaimed and recycled by an appropriate garbage collector.

Asynchronous Transfer of Control (ATC). The normal flow of control within a Core program is sequential execution of statements. The normal sequential control flow is modified by branching statements, including while loops, for loops, switch statements, and if-else statements. These forms of program-controlled control flow are known as *synchronous transfer of control*. When control flow is modified by some event that is not under the control of the currently executing thread, this is known as *asynchronous transfer of control*.

Green Threads. Both Baseline and Core programming languages provide built-in support for multiple threads (tasks). In both cases, certain aspects of the implementation of multiple tasks are not constrained by the specification. In particular, the Baseline and Core specifications do not require that programming language threads be mapped one-to-one to operating system tasks. One way to implement the programming language run-time is to dedicate one operating system task to the run-time, and to implement multiple tasks and task dispatching using a small kernel that is part of the implementation of the programming language run-time. We use the term "green threads", a phrase apparently coined by Sun Microsystems, to describe such an implementation.

RTOS. RTOS is an acronym representing "Real-Time Operating System". Though each real-time operating systems has been designed to satisfy a particular audience's special needs, most real-time operating systems share the objectives of enabling the creation of

small, highly efficient, highly predictable applications. For real-time applications, predictability refers to the ability to predict when the application will perform certain actions. Highly predictable real-time systems allow prediction of activities within tolerances measured in tens of microseconds with very high degrees of confidence. For less predictable real-time systems, the tolerances are much higher (measured, for example, in tens of milliseconds) and the degree of confidence may be much lower.

Base Priority. The Base Priority of a Core task is the priority initially specified when the task is constructed and possibly modified by invocation of the task's setPriority() method. See Section 3.7 (starting on page 21).

Active Priority. The Active Priority of a Core task is the priority at which the task is currently being dispatched. Note that the Active Priority shall be higher than the Base Priority if the task has inherited priority from a higher priority task while it is accessing a particular shared data structure. The Active Priority shall also be higher than the Base Priority if the task is executing a synchronized method of a Core component that implements the PCP interface, and the object's ceiling priority is higher than this task's Base Priority. The Active Priority shall be lower than the Base Priority if the task has been suspended and has not otherwise inherited priority higher than the Never-Scheduled Priority level. See Section 3.7 (starting on page 21).

Never-Scheduled Priority. The Never-Scheduled Priority is a special priority level that is used to identify tasks that shall not be dispatched for execution unless execution is required by a priority inversion avoidance mechanism. See Section 3.7 (starting on page 21).

I/O Channel, Memory-Mapped Access, and I/O-Space Access. Throughout this document, we use the phrase "memory-mapped access" to describe access to memory-mapped I/O channels, and the phrase "I/O-space access" to describe access to I/O ports residing in I/O space. We use the term "I/O channel" to represent either or both.

2.3 Architectural Components

A number of additional terms are defined in Section 3.4 as part of the Architectural Overview of the Core Implementation. Included among the terms described there are:

- Core Source Files (See page 12)
- Stylized Core Source Files (See page 12)
- Core Class Files (See page 12)
- Baseline Compiler (See page 13)
- Core Verifier (See page 14)
- Core Native Compiler (See page 14)
- Native-Targeted Core Class Files (See page 14)
- Core Class Libraries (See page 12)
- Core Execution Environment (See page 14)
- Static Core Execution Environment (See page 14)
- Static Core Executable Load Image (See page 14)

Copyright 1999, 2000 J Consortium, All Rights Reserved

- Core Static Linker (See page 15)
- Dynamic Core Execution Environment (See page 15)
- Core Class Loader (See page 15)

2.4 Notational Shorthand

Throughout this document, we use two shorthand notations to describe the signatures of particular Core API methods. These shorthands consist of the baseline keyword used to identify a method as Core-Baseline methods (see “Core-Baseline Methods” on page 3) and the stackable keyword to identify method arguments that are designed to refer to stack-allocated objects (see “Stack Allocation of Dynamic Objects” on page 27).

In particular, we use the baseline keyword in the list of attributes that comprises the signature of each Core-Baseline method, as in the following:

```
public baseline void foo(int i, float x) {
    ...
}
```

This notation is short-hand for the representative invocation of CoreRegistry.registerBaseline(“foo(IF)V”), as described in “CoreRegistry.registerBaseline()” on page 77. If multiple methods for a given class are declared with the baseline attribute in their signatures, this notation is equivalent to a single invocation of CoreRegistry.registerBaseline() in the class’s static initializer with the string argument created by catenating together each Core-Baseline method’s name and signature, each separated from each of its neigbhors by a semicolon.

Similarly, we use the stackable keyword as an attribute of a parameter which is declared to honor all of the protocols required for reference variables that may refer to stack-allocated objects. To use the stackable attribute in the declaration of a method signature is short-hand for the equivalent invocation of CoreRegistry.registerStackable() (see “CoreRegistry.registerStackable()” on page 77). For example, the method signature:

```
public stackable org.rtjwg.CoreObject foo(int i, stackable org.rtjwg.CoreObject x);
```

is shorthand for the Core method whose implementation begins with a registerStackable() invocation which identifies this and x as references to stack-allocatable objects, as represented by the following example implementation.

```
public java.lang.Object foo(int i, java.lang.Object x) {
    CoreRegistry.registerStackable("x;this");
    . . .
```

3.0 The Specification

3.1 Conformity Assessment

Real-time core extensions comprise development tools, run-time environments, required libraries, and specific constraints on the way a Core application is represented. This section describes what it means to conform to the specification for Real-Time Core Extensions for the Java platform.

3.1.1 A Conforming Core Class File

1. Uses the same format as Java 1.1 class files, as described in reference 3.
2. Adheres to a more stringent set of programming constraints, as described in Section 3.5 (starting on page 15).

3.1.2 A Conforming Static Core Application

1. Is represented as one or more Java Virtual Machine class files, according to the class file format that is described in reference 3.
2. Adheres to all of the special restrictions identified in section 3.5 of this document.
3. Does not contain any invocations of the CoreClass.loadClass() or CoreClass.unloadClass() methods (see "CoreClass.loadClass()" on page 66 and "CoreClass.unloadClass()" on page 66)

3.1.3 A Conforming Dynamic Core Application

1. Is represented as one or more Java Virtual Machine class files, according to the class file format that is described in reference 3.
2. Adheres to all of the special restrictions identified in section 3.5 of this document.
3. Contains at least one invocation of the CoreClass.loadClass() or CoreClass.unloadClass() methods (see "CoreClass.loadClass()" on page 66 and "CoreClass.unloadClass()" on page 66)

3.1.4 A Conforming Core Verifier

1. Accepts as input a Core class file and verifies that the Core class file is of the proper format by enforcing all of the byte-code verification requirements described in reference 3 as supplemented by the additional rules described in "Core Class Files" on page 15 of this document).
2. The Core verifier may be packaged either as part of the Core Execution Environment or as a dedicated tool that verifies that class files contain code that conforms with the constraints of the Core specification.

3.1.5 A Conforming Static Core Development Environment

1. Includes Core class file implementations of all of the class libraries described in Section 3.17 (starting on page 60) (the Core API) of this document. All of the class file implementations shall conform to the descriptions and requirements provided in Section 3.17 except that the CoreClass.loadClass() and CoreClass.unloadClass() methods need not be implemented.
2. Includes a conforming Core Verifier.

Copyright 1999, 2000 J Consortium, All Rights Reserved

3. Includes a conforming Static Core Linker and whatever native components (also known as Static Core Execution Environment) are required to be linked together with the Core class file implementations of the Core API and with the Core class file representations of any conforming static Core application in order to produce a conforming static Core Executable Load Image. The native components represented by the Static Core Execution Environment shall include the C/Native API as described in Section 3.16 (starting on page 57),
4. Does not necessarily implement support for stack allocation of local variables, but does implement the CoreRegistry.registerStackable() method.
5. May, but need not, include a Core Native Compiler.
6. May, but need not, include support for integration of native methods within Core applications. Native method support, if provided, shall be implementation-defined.

3.1.6 A Conforming Static Core Executable Load Image

1. Is an executable program comprised of a Core application bound to the subset of Core API libraries required for execution of that particular Core application and bound to whatever native components are required for execution of that Core application.

3.1.7 A Conforming Dynamic Core Development Environment

1. Includes Core class file implementations of all of the class libraries described in Section 3.17 (starting on page 60) (the Core API) of this document. All of the class file implementations shall conform to the descriptions and requirements provided in Section 3.17 except that the CoreClass.loadClass() and CoreClass.unloadClass() methods need not be implemented.
2. Includes a conforming Core Verifier.
3. Includes a Dynamic Core Java Execution Environment which includes implementations of the Baseline API as described in Section 4.0 (starting on page 103), the C/Native API as described in Section 3.16 (starting on page 57), and whatever additional native components are required to enable the Dynamic Core Java Execution Environment to dynamically load and execute any conforming dynamic Core application.
4. Does not necessarily implement support for stack allocation of local variables, but does implement the CoreRegistry.registerStackable() method.
5. May, but need not, include a Core Native Compiler.
6. May, but need not, include support for integration of native methods within Core applications. Native method support, if provided, shall be implementation-defined.

3.1.8 A Conforming Static Core Linker

1. Must be able to process any collection of conforming Core class files, producing as output an executable image that implements the semantics of those core class files linked together.
2. May, but need not, provide the capability of linking native method implementations into the resulting Static Core Executable Load Image. It is implementation-defined whether native method programming is supported by an implementation of the

Core extensions. If native method programming is supported, it is implementation-defined how to link native methods into the Static Core Executable Load Image.

3. May, but need not, include support for integration of native methods within Core applications. Native method support, if provided, shall be implementation-defined.

3.1.9 A Conforming Core Native Interface Compiler

1. Shall process any conforming Core class file and produce as output a C header file which identifies the internal organization, providing at minimum, the ability to access all Core-declared fields in the corresponding real-time Core objects. The form of the information provided in the C header file is implementation-defined.

3.2 Core Objects

Objects allocated within the Core Execution Environment shall exhibit special characteristics that are (or may be) different than objects allocated within a Baseline virtual machine. In particular:

1. Core objects shall not be relocated. Once the location of a Core object has been determined, that object's location in memory shall not change.
2. There are two ways for software developers to author Core class files. Either they use a traditional Baseline Compiler and a special Core Verifier, or they use a specially designed Core Compiler which integrates the functionality of a traditional Baseline Compiler with the Core Verifier. This is illustrated in Figure 1 on page 13. Depending on which set of development tools they prefer to use, Core programmers use different syntaxes to describe their intent.
   a. If they use a traditional Baseline Compiler and a Core Verifier, they express core concepts using notations that we characterize in this document as *Stylized Core* source. This is described more completely in Section 3.6 (starting on page 20).
   b. If they use a Core Compiler, they express concepts using notations that we characterize in this document as *Syntactic Core* source. This is described more completely in Section C.5 (starting on page 138).

   In either case, the contents of the Core class file is the same. The Core Compiler translates Syntactic Core source code into a Core class file that looks as if it had been translated by a Baseline Compiler from the equivalent Stylized Core source code.
3. When a Core task does a new memory allocation, this never blocks or causes garbage collection to run. If memory is not available, `new()` immediately throws a previously allocated instance of `CoreOutOfMemoryException`. A memory allocation request may fail either because there is not sufficient free memory available, or because whatever free memory is available has become fragmented.
4. Core tasks are only allowed to allocate instances of `org.rtjwg.CoreObject` and its subclasses.
5. Except for the special Core-Baseline methods described in paragraph 3 of Section 3.3, only Core tasks are allowed to execute the methods of Core objects. We call these methods which are only executable by Core tasks "Core methods".
6. In the Core methods, programmers shall not perform string catenation except for catenation of string literals (compile-time constants) for which the source-language

Copyright 1999, 2000 J Consortium, All Rights Reserved

compiler replaces the string-catenate expression with a single string literal. This restriction shall be enforced by the Core Verifier.

7. Every Core object is allocated within a particular Allocation Context. Each Core task has a default Allocation Context. Within particular dynamic scopes, Core objects are allocated from programmer specified Allocation Contexts.
8. A Core application may invoke the release() method of any Allocation Context to cause the Core Execution Environment to reclaim the memory used to represent all of the objects allocated within that Allocation Context. Assuming that the Core Execution Environment is not bound to a Baseline virtual machine, the Core Execution Environment shall simply reclaim the memory without performing any checks to verify that the memory objects to be reclaimed are no longer in use. However, if the Core Execution Environment is bound to a Baseline virtual machine as part of an Extended Baseline Virtual Machine, the semantics of the Allocation Context's release() method are different, as described in paragraph 11 of Section 3.3.

3.3 Partitioning of Memory

System integrators have the option of combining the Core Execution Environment with a Baseline virtual machine. The combination of these two components is known as an Extended Baseline Virtual Machine. An Extended Baseline Virtual Machine shall support two logical heaps. One heap holds Core objects. The other holds Baseline objects. The idea is that objects within the Baseline heap are managed by way of automatic garbage collection. The memory for objects residing within the Core heap is managed under explicit programmer control.

Key differentiating characteristics of the Core objects are listed below:

1. Core classes are identified by the way they are loaded. There is no syntax to distinguish Core classes. Instead, a special Baseline service allows Baseline components to cause particular classes to be loaded and executed within the Core Execution Environment. This service is described in Section 4.0. Alternatively, system integrators can identify certain Java class files as Core classes by requesting that they be linked into a Core Executable Image by identifying those classes as inputs to the Core Static Linker. All classes dynamically loaded into a Dynamic Core Execution Environment or statically linked into a Static Core Executable Image are known as Core classes. All instances of these Core classes are known as Core objects. All Core objects reside in the Core heap.
2. Core methods shall not invoke methods of Baseline objects. Further, Core-Baseline methods shall not invoke methods of Baseline objects. Baseline threads shall not invoke Core methods.
3. A special protocol is available to allow developers of Core components to identify the set of methods that are visible to the Baseline world. We call these methods Core-Baseline methods. Core tasks shall not invoke Core-Baseline methods. Further, Core-Baseline methods shall not invoke Baseline methods. Core programmers identify the Core-Baseline methods of a Core class by concatenating the method names and signatures of the Core-Baseline methods together, separated by semicolons, into a single Core string and passes this string to the static CoreRegistry.registerBaseline() method, as described in "CoreRegistry.registerBaseline()" on page 77.

4. Note in Figure 1 on page 13 that there are several paths for deploying Core programs. Either the Core class file can be loaded dynamically into a Dynamic Core Execution Environment, or the Core class file can be compiled to native machine language by a Core Native Compiler and then dynamically loaded into a Dynamic Core Execution Environment, or the Core class can be statically linked by a Core Static Linker, either in byte code or native code form, with an appropriate collection of run-time services known as the Static Core Execution Environment.

5. A special registry shall allow Core Components to publish particular core objects so they may be seen by Baseline components. To make a Core object visible to the Baseline domain, the Core component invokes:

   ```
   CoreRegistry.publish(CoreString, CoreObject);
   ```

   passing as a first argument the CoreString representation of the symbolic name by which the Core object is to be known within the CoreRegistry dictionary, and passing a reference to the Core object as its second argument.

   At some later time, the Core component may decide to remove the object from the CoreRegistry dictionary. It does so by invoking:

   ```
   CoreRegistry.unpublish(CoreString);
   ```

   passing as its sole argument a CoreString object which matches the name (same sequence of characters) by which the particular object was originally published.

   Note that removing a particular object from the CoreRegistry dictionary does not necessarily cause that object's memory to be reclaimed, even if the Core domain has already released the object's Allocation Context. This is because the Core object may still be reachable from the Baseline domain, either directly or indirectly.

   A more detailed description of the CoreRegistry class is provided in Section 3.17.16.

   Given that a Core object has been installed into the CoreRegistry dictionary, a Baseline component can obtain a reference to the object by invoking:

   ```
   core_object_reference = CoreDomain.lookup(String);
   ```

   passing as the argument to the lookup() method a java.lang.String() object that has the same sequence of characters as the symbolic name by which the object is identified in the CoreRegistry dictionary.

   A more detailed description of the CoreDomain class is provided in Section 4.2.

6. Since Core objects may become visible to the Baseline world (through the publish() service of the CoreRegistry class), each Core object needs to support two APIs. In particular, the Core API derives from org.rtjwg.CoreObject and includes the Core methods of all classes on the inheritance hierarchy between org.rtjwg.CoreObject and the class. The Baseline API derives from java.lang.Object, and includes the Baseline methods of java.lang.Object, plus the Core-Baseline methods on the inheritance hierarchy from org.rtjwg.Object to the class. Note that within the Baseline world, org.rtjwg.Object extends java.lang.Object.

   To reduce the memory required to implement certain Core objects, an optimizing Core Execution Environment need not support the Baseline API for objects for which it can demonstrate through program analysis that they are not visible to the Baseline domain.

7. Style guidelines prohibit Baseline threads from direct access to the instance and class variables of Core objects. The Core Verifier shall enforce this restriction.

Copyright 1999, 2000 J Consortium, All Rights Reserved

8. Style guidelines prohibit the Core-Baseline methods from modifying the pointer instance and pointer class variables of Core objects. The Core Verifier shall enforce this restriction.
9. No code within Core-Baseline methods is allowed to make any reference to Baseline objects. Note that this restriction prohibits the passing of arguments to Core-Baseline methods which are references to Baseline objects. This restriction shall be enforced by the Core Verifier.
10. Baseline threads are not allowed to allocate instances of org.rtjwg.CoreObject and its subclasses. Any attempt by a Baseline thread to allocate a new instance of org.rtjwg.CoreObject or one of its derivatives shall fail by throwing an UnsatisfiedLinkError exception.
11. When the Core Execution Environment is bound to a Baseline virtual machine as part of an Extended Baseline Virtual Machine, a Core application may invoke the release() method of any Allocation Context to release the Core Execution Environment's claim to the memory used to represent all of the objects contained within that Allocation Context. The Core Execution Environment shall reclaim the memory of these objects only after it has verified that the objects are not reachable from the Baseline virtual machine.

    Reachability of Core objects is defined in the traditional garbage collection sense. If there exists some chain of Baseline-visible pointers starting with a live variable residing within the Baseline domain which terminates with a pointer to Core object X, we say that object X is reachable. Therefore, the memory for object X cannot be reclaimed.

    A reference field contained within a Baseline object is a Baseline-visible pointer. If the Baseline virtual machine has a reference to Core object U, a reference field contained within object U is Baseline-visible if object U has a Core-Baseline method which returns the value of this reference field. If it is possible for the Baseline virtual machine to obtain a reference to Core object V (by, for example, invoking a Core-Baseline method on a Core object that is already referenced from the Baseline virtual machine), a reference contained within object V is Baseline-visible if object V has a Core-Baseline method which returns the value of this reference field.

3.3.1 Partitioning Protocol from Core programmer's perspective

When developing applications that involve cooperation between Core components and Baseline components, it is necessary for the developers of each component to honor an appropriate sharing protocol. The developer of Core components sees the object partitioning protocol as follows:

1. It is my responsibility to make sure I'm done with object X before I release the Allocation Context to which object X belongs. Once I've released the Allocation Context, it is an error for the Core tasks to access any of the objects belonging to that Allocation Context or to assign any of those the objects' addresses to any field of a Core object. By deferring the release of an Allocation Context until after all of the objects allocated within that Allocation Context are no longer in use, the Core programmer prevents premature reclamation of the Core objects.
2. It is my responsibility to make sure I release the Allocation Context for object X when I am certain that I am done using object X and all other objects that were allocated within that Allocation Context. By taking responsibility to release each Allo-

cation Context as soon as it is known that the objects allocated within that Allocation Context are no longer in use, the Core programmer prevents memory leaks.

3. Once I've released the Allocation Context for object X, I have no need to worry about object X becoming visible to me again by any means. (In other words, I can be assured that references to object X will not "hide out" in the Baseline world and then at some later time find their way back into the domain of the Core components.)
4. I realize that object X may be useful to other components in the system, and I have no assurance of how long it will be before those components allow the memory dedicated to object X to be reallocated to other purposes (unless I've entered into some sort of "contract" with those other components that governs the sharing of information between our two worlds).

3.3.2 Partitioning Protocol from the Baseline programmer's perspective

When developing applications that involve cooperation between Core components and Baseline components, it is necessary for the developers of each component to honor an appropriate sharing protocol. The developer of Baseline components sees the object partitioning protocol as follows:

1. From my perspective, Core objects are garbage collected just the same as other objects.
2. I can only access or modify Core objects by way of Core-Baseline methods.
3. I am not allowed to modify the pointer (reference) fields of Core objects.

3.4 Architectural Overview of the Core Development Architecture

The Core specification comprises development tools, run-time environments, officially defined libraries, and application code. This section provides an overview of how the various components fit together. Figure 1 on page 13 illustrates the relationship between the various components.

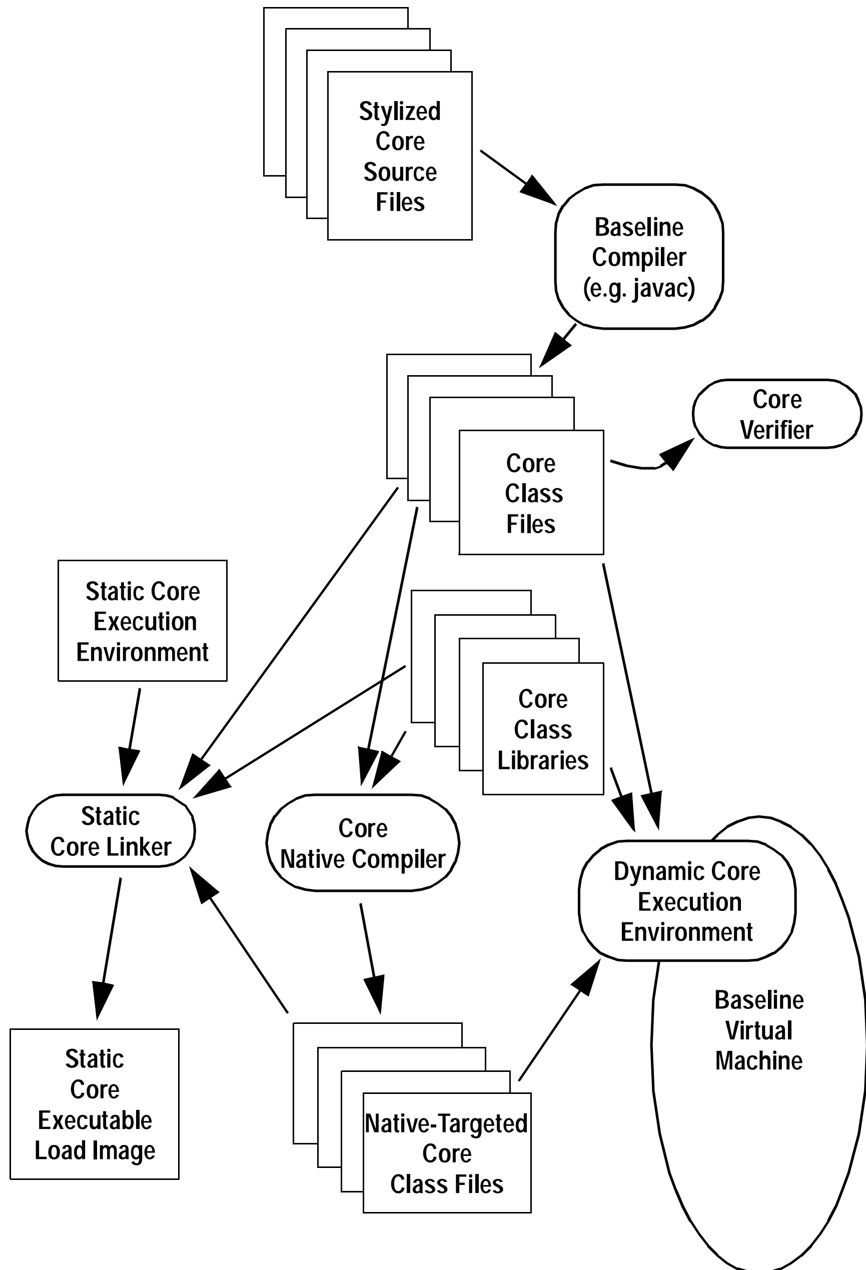
Core Source Files. Core source files are authored by Core application developers and system integrators. There are two distinct conventions for representing Core Source Files, known as Syntactic Core Source Files and Stylized Core Source Files. Throughout this document, we use the phrase "Core Source Files" to indicate that our comments apply to both conventions.

Stylized Core Source Files. Stylized Core Source Files are Core Source Files written to use Baseline syntax without any special Core syntaxes. Rather than use special syntaxes, the Core programmer adheres to specific style conventions and invokes particular Core API methods to describe special real-time behaviors.

Core Class Files. Core class files use the same format as Java 1.1 class files, as described in reference 3, except the code represented in Core class files must adhere to a more stringent set of programming constraints, as described in Section 3.5 (starting on page 15).

Copyright 1999, 2000 J Consortium, All Rights Reserved

Figure 1. Overview of Real-Time Core Development Architecture



Baseline Compiler. Core source files can be compiled using a Baseline Compiler as long as the author of the source code follows the particular style guidelines that are identified in Section 3.6.

Core Verifier. The Core Verifier examines the contents of Core Class Files and ensures that the code contained therein has adhered to the stringent Core programming guidelines. Figure 1 on page 13 shows the Core verifier running as a distinct development pass. Alternatively, the Core verifier may be integrated into the Core Static Linker, the dynamic Core class loader, and/or the Core Compiler.

Core Native Compiler. A Core Native Compiler processes the contents of a Core Class File in order to provide a dynamically loadable native translation of the contents of the Core Class File. This specification for Real-Time Core Extensions for the Java Platform does not specify the behavior of the Core Native Compiler. Nor does it specify the internal organization of Native-Targeted Core Class Files. The significance of including these components in the architecture overview is to emphasize that the Core specification shall enable the creation of products that play the roles identified in this architectural overview, without constraining how the products function.

Native-Targeted Core Class Files. A Native-Targeted Core Class File includes an Implementation Defined representation of a Core Class File's translation to a particular computer's native machine language.

Core Class Libraries. The Core Class Libraries comprise all of the class libraries described in this document, all of which descend from org.rtjwg.CoreObject. The Core Class Libraries are examples of Core Class Files.

Core Execution Environment. A Core Execution Environment is a run-time environment within which Core programs are executed. There are two kinds of Core Execution Environments: Dynamic and Static. Throughout this document, we use the phrase "Core Execution Environment" when our comments apply equally to both dynamic and static systems.

Static Core Execution Environment. The Static Core Execution Environment consists of object-file executables to be linked by the Core Static Linker with the Core Class Libraries and application programs in the form of Core class files. The Static Core Execution Environment takes responsibility for task dispatching, maintenance of priority queues, implementation of priority inheritance and priority ceiling protocols, and interface to interrupt handling hardware. Depending on a vendor's implementation, the static Core Execution Environment may also include a byte-code interpreter and an interface to the target's operating system. (However, if the vendor chooses to use the Core Static Linker to translate all byte codes to native code, then the Static Core Execution Environment need not include a byte code interpreter.) Included within the Static Core Execution Environment is a porting/integration layer that glues the run-time environment to the host operating system.

Static Core Executable Load Image. A Static Core Executable Load Image is a completely linked executable program which includes the following components:

1. That subset of the Core Class Files that the Core Static Linker determines to be necessary for execution of the selected Core Components, from which certain methods and variables may have been pruned because the Core Static Linker determined through analysis of the application that those methods and variables are not useful to the application.

Copyright 1999, 2000 J Consortium, All Rights Reserved

2. That subset of the Core Class Libraries that the Core Static Linker determines to be necessary for execution of the selected Core Components, from which certain methods and variables may have been pruned because the Core Static Linker determined through analysis of the application that those methods and variables are not useful to the application.
3. That subset of the Static Core Execution Environment that the Core Static Linker determines to be necessary for execution of the selected Core Components.

Within the Static Core Executable Load Image, it is implementation-defined which, if any, of the Core Class Files and Core Class Libraries have been translated to native machine language.

Core Static Linker. The Core Static Linker takes responsibility for linking together the various components of the Core application into an executable load image. Optionally, the Core Static Linker may verify that all Core class files adhere to appropriate style guidelines. Another option is for the Core Static Linker to translate byte codes to native machine language.

Dynamic Core Execution Environment. The Dynamic Core Execution Environment provides all of the same services as the Static Core Execution Environment. Additionally, the Dynamic Core Execution Environment includes support for dynamic class loading. It is illustrated in Figure 1 on page 13 in combination with a Baseline virtual machine to emphasize that the Core dynamic class loader depends on support from certain components that run only in the Baseline virtual machine environment. (Static Core applications may also be deployed in concert with Baseline components. Since the Baseline support is optional for static Core applications, the executable load image produced by the Core Static Linker is not shown to be bound to a Baseline virtual machine.)

Core Class Loader. Within the Dynamic Core Execution Environment, the Core Class Loader is responsible for dynamically loading Core Class Files and Native-Targeted Core Class Files into the Dynamic Core Execution Environment. Within the Core Static Linker, the Core Class Loader is responsible for finding Core Class Files and Native-Targeted Core Class Files and processing their contents in order to link the various Core Components into a single Static Core Executable Load Image.

3.5 Core Class Files

The requirements that characterize valid Core Class Files are different from the requirements that are imposed upon Baseline class files. For purposes of this discussion, a Core object is an instance of a class that is loaded by the Core Class Loader. The key differences between Core Class Files and Baseline class files are as follows:

1. Every Core class must extend from `org.rtjwg.CoreObject`.
2. Every Core class must include a static initializer which contains, as its first line of executable code, an invocation of `CoreRegistry.registerCoreClass()`. This indicates to the class loader that this class is intended for execution in the Core Execution Environment.
3. If the class contains any Core-Baseline methods, the next line of the class static initializer following the invocation of `CoreRegistry.registerCoreClass()` must be an invo-

cation of CoreRegistry.registerBaseline(). The argument to this method invocation is a CoreString object identifying the names and signatures of each Core-Baseline method, as described in "CoreRegistry.registerBaseline()" on page 77.

4. For each Core method (excluding the Core-Baseline methods) of the class that contains reference variables for which the programmer intends that the referenced objects be stack allocatable, the first line of the method must be an invocation of CoreRegistry.registerStackable(), with a CoreString argument which identifies the list of variables. Additional information on stack allocation of objects is provided in Section 3.12.
5. For each reference to java.lang.Object from within a Core Class File, it is understood that java.lang.Object is a placeholder which really represents org.rtjwg.CoreObject. The Core Class Loader shall replace the reference with a reference to org.rtjwg.CoreObject when the class is loaded.
6. For each reference to java.lang.Throwable, it is understood that java.lang.Throwable is a placeholder which really represents org.rtjwg.CoreThrowable (See Section 3.17.2). The Core Class Loader shall replace the reference with a reference to org.rtjwg.CoreThrowable when the class is loaded.
7. For each reference to java.lang.Exception, it is understood that java.lang.Exception is a placeholder which really represents org.rtjwg.CoreException (See Section 3.17.4). The Core Class Loader shall replace the reference with a reference to org.rtjwg.CoreException when the class is loaded.
8. For each reference to java.lang.Error, it is understood that java.lang.Error is a placeholder which really represents org.rtjwg.CoreError (See Section 3.17.3). The Core Class Loader shall replace the reference with a reference to org.rtjwg.CoreError when the class is loaded.
9. For each occurrence of the anewarray and multianewarray byte-code instructions, it is understood that the type of the object pushed onto the Core run-time stack by execution of this byte-code instruction is CoreArray (See Section 3.17.7), which extends from CoreObject. For each variable declared in the Core Class File to be of Array type, it is understood that the type represented by the variable is really CoreArray. The Core class loader shall replace every reference to an array type with an appropriate subclass of CoreArray. Within the Core Execution Environment, CoreArray objects behave the same as Baseline arrays behave within the Baseline virtual machine environment (with respect to subscripting operations, testing for equality, inquiring as to length, etc.).
10. If a particular Core Class File defines the org.rtjwg.CoreObject class, that class definition shall provide implementations of the following method signatures:

    ```
    public final CoreClass _getClass();
    public final void _wait();
    public final void _notify();
    public final void _notifyAll();
    ```

    It is understood that these methods represent getClass(), wait(), notify(), and notifyAll() respectively. The Core Class Loader shall overwrite the names of each of these method definitions when the class is loaded.
11. Within the class file's constant pool, any constant of type CONSTANT_String is understood to be a placeholder for an equivalent CoreString object (See Section

Copyright 1999, 2000 J Consortium, All Rights Reserved

3.17.12). The Core Class Loader shall make an appropriate substitution when the class is loaded.

12. After performing the substitutions described in Paragraphs 5 through 11 above, the Core Verifier shall enforce type consistency of byte-code instructions as described in Reference 8 under the heading “Verification of Class Files”. Type consistency checking includes checking of method invocations to make sure that the invoked methods are available with appropriate signatures in the corresponding objects and/or classes.
13. For all methods of Core objects except for the Core-Baseline methods, these methods shall not invoke any method of an object that is not a Core object, and shall not invoke any Core-Baseline methods of Core objects.
14. For all Core-Baseline methods of Core objects, these methods shall not invoke any method of an object that is not a Core object, and shall not invoke any Core methods of Core objects. Core-Baseline methods of Core objects are allowed only to invoke other Core-Baseline methods of other Core objects.
15. For all Core-Baseline methods of Core objects, the arguments to these methods shall be either of primitive type or shall be of type org.rtjwg.CoreObject (or descendants thereof). Reference arguments to Core-Baseline methods shall not refer to Baseline objects. The value returned from a Core-Baseline method may be a reference to a Core object.
16. Except for the Core-Baseline methods that have been defined for a particular Core class, the fields and methods of Core objects shall not be visible to the Baseline domain.
17. The code contained within the Core-Baseline methods of Core objects shall not write to any Core object’s instance or class reference variables.
18. The code contained within the methods of Core objects shall not include any string catenation operations. Note that any catenation of string literals that was present in the Core source code must have been replaced within the Core Class File by the Baseline Compiler or Core Compiler with the string literal that represents the concatentation of the individual string literals.
19. For each synchronized context that occurs within a Core class that is declared to implement the Atomic interface (See Section 3.17.11), the body of code contained within the synchronized context must be execution-time analyzable (See Section 3.14).
20. The Core Class File shall only include byte-code representations of source code statements of the form:

    synchronized (object) statement

    if object is this.
21. The code contained within finally statements of Core methods (this restriction does not apply to Core-Baseline methods) shall not terminate abruptly, and shall not execute throw. Abrupt termination means the control jumps out of the finally statement because of a break, continue, or return statement.
22. For each local and argument variable identified as stackable (see Section 3.12), the variable usage shall conform to the special constraints described in Section 3.12.
23. For each class that extends org.rtjwg.ISR_Task, the implementation of the work() method shall be declared to be synchronized.

3.5.1 The Core Verifier

The Core Verifier is a tool to assist with the development of Core application code. This is a required component of a conforming Core implementation. For Core programs deployed as part of a Static Core Execution Environment, the Core Verifier shall be applied to the Core class files prior to building of the static executable load image. For Core programs designed for deployment within a Dynamic Core Execution Environment, the supplier of a conforming Core class file shall apply the Core Verifier to the class file before deploying the application.

The Core Verifier is responsible for verifying that a particular Core Class File adheres to the various restrictions that characterize valid Core Class Files. The Core Verifier can be packaged either as a stand-alone tool, or bundled within the Core Class Loader or within the Core Static Linker. The user interface is Implementation Defined.

1. The Core Verifier shall perform all of the standard checking that is described as "Class File Verification" for the Java Virtual Machine (See Reference 8), subject to the conceptual replacement substitutions that are described in Section 3.5.
2. The Core Verifier shall enforce all of the special constraints described in Section 3.5 of this specification.

3.5.2 The Core Class Loader

The Core Class Loader performs a number of special transformations to the class file as it is loaded. Both before and after making these transformations, the Core Class Loader performs a number of special checks designed to improve the likelihood that the class being loaded is properly formatted. The checks done by the Core Class Loader are much less comprehensive than the checks performed by the Core Verifier.

If invoked from within a Dynamic Core Execution Environment, CoreClass.loadClass() throws a previously allocated CoreClassFormatError exception if any of the checks described below fail. If invoked from within the Baseline virtual machine environment, CoreDomain.loadClass() throws a ClassFormatError exception if any of the checks described below fail. If the Core Class Loader is running as part of the Core Static Linker and one of the checks described below fails, the Core Static Linker shall not produce a Static Core Executable Load Image. The format and nature of any diagnostic reporting is implementation-defined.

The Core Class Loader shall perform the following transformations and checks as it is loading a new class, in the specified order:

1. Check to make sure that this class has a static initializer that contains as its first executable code an invocation of CoreRegistry.registerCoreClass(). After verifying the presence of this invocation, remove the invocation from the loaded class.
2. For each reference to java.lang.Object within this class, replace it with a reference to org.rtjwg.CoreObject.
3. For each reference to java.lang.Throwable within this class, replace it with a reference to org.rtjwg.CoreThrowable.
4. For each reference to java.lang.Exception, replace it with a reference to org.rtjwg.CoreException.

Copyright 1999, 2000 J Consortium, All Rights Reserved

5. For each reference to java.lang.RuntimeException, replace it with a reference to org.rtjwg.CoreRuntimeException.
6. If the name of the class being loaded is org.rtjwg.CoreObject, check to make sure the class provides implementations of the following method signatures:

```
public final CoreClass _getClass();
public final void _wait();
public final void _notify();
public final void _notifyAll();
```

   Overwrite the names of these methods with getClass, wait, notify, and notifyAll respectively.
7. For each CONSTANT_String object contained within the constant pool of this class, replace it with an appropriate instance of a CoreString constant (representing the same sequence of characters).
8. Check to see if the next executable code within the static initializer for this class is an invocation of CoreRegistry.registerBaseline(). If so, examine the CoreString argument of the registerBaseline() invocation and make sure that this class provides implementations of each of the named methods. Mark each of the named methods as a Core-Baseline method. Then remove the invocation of CoreRegistry.registerBaseline() from the static initializer for this class.
9. For each method of this class except those methods that were marked in step 8 above as Core-Baseline methods, check to see if the first executable code within the method's implementation is an invocation of CoreRegistry.registerStackable(). If so, examine the CoreString argument of the registerStackable() invocation to determine which local variables are stackable. If this Core Execution Environment claims to support stack allocation of dynamic objects (by returning true from CoreRegistry.stackAllocation()), then the Core Class Loader shall perform whatever implementation-defined transformations are necessary in order to ensure that all new memory allocations which assign their result to a stackable variable shall be allocated on the run-time stack. Otherwise, the Core Class Loader shall not perform any special processing for the stackable variables. In either case, the Core Class Loader shall remove the invocation of CoreRegistry.registerStackable() from the method's implementation in the loaded class.
10. For each invocation of CoreRegistry.coerce() that is found within this class, the Core Class Loader shall check that the argument derives from org.rtjwg.CoreObject. After performing this check, the Core Class Loader shall remove the invocation of CoreRegistry.coerce(), replacing this invocation with the method's original argument and a run-time type checking instruction if the surrounding context requires this run-time check.
11. If the class is to be loaded into an Extended Baseline Virtual Machine, for which it is necessary for Baseline and Core components to coexist and cooperate, the Core Class Loader shall build an appropriate Baseline API for each of the Core classes that might become visible to the Baseline domain. The Baseline API describes the collection of Core-Baseline methods to which instances of this class respond. Furthermore, the Core Class Loader shall cause a Baseline Class representing the Baseline API of this Core class to be loaded into the corresponding Baseline virtual machine environment. Whenever the Baseline domain gains access to an instance of this Core class, the Baseline virtual machine sees this Core object as an instance

of the Baseline class that represents this object's Baseline API. In creating the Baseline API for this class, the Core Class Loader performs the following additional transformations:

a. If a particular Core-Baseline method's argument makes reference to a Core array type, the signature of this argument within the Baseline API is CoreArray or an appropriate derivative (See Section 3.17.7).

b. If a particular Core-Baseline method throws CoreException, the signature of this method within the Baseline API indicates that the method throws CoreBaselineException (See Section 4.6).

c. If a particular Core-Baseline method throws CoreRuntimeException, the signature of this method within the Baseline API indicates that the method throws CoreBaselineRuntimeException (See Section 4.5).

d. If a particular Core-Baseline method throws CoreThrowable or some derivative of CoreThrowable other than CoreException or CoreRuntimeException or their descendants, the signature of this method within the Baseline API shall indicate that this method throws CoreBaselineThrowable (See Section 4.4).

3.6 Special Notations for Stylized Core Source Code

Stylized Core source code is code written for execution in the Core Execution Environment, which is designed to be compiled by a Baseline Compiler. A special Core Verifier analyzes the class file to make sure that the class-file translation produced by the Baseline Compiler adheres to the special constraints that characterize valid Core Class Files.

Following is a list of special notations for the use of developers in creating Core software components using Stylized Core programming conventions.

1. If a Core programmer declares a variable to be of type array (or makes any reference to an array type), it is understood that this means CoreArray. CoreArray extends from CoreObject.
2. If a Core programmer declares a class to extend from java.lang.Throwable, it is understood that the class really extends from CoreThrowable (in place of java.lang.Throwable).
3. If a Core programmer uses a string constant, it is understood that this is really a constant of type CoreString. CoreString extends from CoreObject.
4. If a Core programmer fails to indicate the type from which a class extends, it is understood that the class extends from CoreObject. All references to java.lang.Object within a Core program are understood to be references to CoreObject.
5. Given that the Core programmer may be dealing with objects that extend from CoreObject but which look to the Baseline Compiler like they extend from java.lang.Object, the Core programmer may coerce such objects to CoreObject by invoking the static coerce() method of org.rtjwg.CoreRegistry.

   Typical usage is to further coerce the result returned from the coerce() method to the type that you really expect this object to be. Consider, as an example, the following code fragment:

```
try {
  doSomething();
```


Copyright 1999, 2000 J Consortium, All Rights Reserved

```
} catch (java.lang.Exception x) {
   MyCoreException cx;
   cx = (MyCoreException) CoreRegistry.coerce(x);
   cx.handleException();
}
```

The Core Class Loader gives special treatment to this particular method, in most cases, replacing dynamic type coercion and checking code with a static check.

3.7 Core Priorities

Core priorities are numbered from 1 to 128, with 128 being the most urgent priority. All of the 128 core priorities are higher than the ten Baseline priorities.

Each Core task shall be represented by the combination of a Base Priority and an Active Priority. The Base Priority is the priority initially specified when the task is constructed and possibly modified by invocation of the task's setPriority() method. The Active Priority is the priority at which the task is currently being dispatched. Note that the Active Priority shall be higher than the Base Priority if the task has inherited priority from a higher priority task while it is accessing a particular shared data structure. The Active Priority shall also be higher than the Base Priority if the task is executing a synchronized method of a Core component that implements the PCP interface, and the object's ceiling priority is higher than this task's Base Priority. The Active Priority shall be lower than the Base Priority if the task has been suspended and has not otherwise inherited priority higher than the Never-Scheduled Priority level. The Never-Scheduled Priority is a special priority level that is used to identify tasks that shall not be dispatched for execution unless execution is required by a priority inversion avoidance mechanism.

3.8 Synchronization Issues

This section describes specific requirements for the implementation of synchronization and blocking within the Core Execution Environment.

1. The Core Execution Environment shall run only on single-processor computers. A future version of the Core specification may address the special issues that are relevant to running the Core Execution Environment on multiprocessor computers.
2. The implementation of synchronized locks within the Core Execution Environment shall not allocate memory upon entry into or departure from a synchronized context. Similarly, no memory shall be allocated by execution of the lock() and unlock() methods of the Mutex class.
3. An attempt to obtain a synchronized lock using a source-level construct such as the following:

   ```
   synchronized (<object>) statement;
   ```

   shall abort by throwing CoreIllegalMonitorStateException if <object> does not represent this.
4. Queues for wait/notify monitors, Mutex locks, SignalingSemaphore and CountingSemaphore implementations, and for the implementation of synchronized statements in classes that do not implement the PCP interface shall conform to the following:

a. Each queue shall be maintained in priority order, with multiple entries of the same priority maintained in sequential order according to insertion time (FIFO).

b. If a task's priority drops due to loss of inherited priority, and consequently some other higher priority task becomes ready to run, this task shall be placed onto the ready queue at the leading position of that portion of the queue that represents tasks of this task's new priority.

c. When a running task becomes preempted by a higher-priority task, the preempted task shall be placed onto the ready queue at the leading position of that portion of the queue that represents tasks of the preempted task's priority.

d. When a running task's time slice expires, the preempted task shall be placed onto the ready queue at the trailing position of that portion of the queue that represents tasks of the preempted task's priority.

e. When a blocked task becomes runnable, that previously blocked task shall be placed on the ready queue at the trailing position of that portion of the queue that represents tasks of this task's priority.

f. A running task can explicitly change its own priority or the priority of another task. If the currently running task's priority is explicitly increased, the task shall continue to run. If the currently running task's priority is explicitly decreased and it continues to be the highest priority task that is ready to run, the task shall continue to run. Otherwise, if the priority of some task that is currently ready to run (but is not running) is explicitly raised such that it becomes the highest priority ready task, that task shall preempt the currently running task. In all other cases in which a task's priority is explicitly changed, the changed task shall be placed on the appropriate queue (the ready queue if the task is ready to run, or the appropriate block queue if the task is waiting for a particular event) at the trailing position of that portion of the queue that represents tasks of this task's new priority.

g. When a running task yields by executing the CoreTask.yield() method, the task shall be placed on the ready queue at the trailing position of that portion of the queue that represents tasks of this task's priority.

h. At no other time shall the position of a task within a task priority queue be affected.

Note that in the context of the Core Execution Environment, requirement (e) above says that if a task T is blocked on a org.rtjwg.CoreObject.wait() operation and becomes runnable either because:

i. a task that was blocked (e.g. in org.rtjwg.CoreObject.wait(), org.rtjwg.SignalingSemaphore.P(), org.rtjwg.CountingSemaphore.P(), org.rtjwg.Mutex.lock(), or org.rtjwg.CoreTask.join()) is awakened by asynchronous event handling, or

ii. because the task was sleeping, and has slept the designated amount of time, or

iii. because some other task awakens this task by invoking org.rtjwg.CoreObject.notify() or

iv. because some other task awakens this task by invoking org.rtjwg.CoreObject.notifyAll(),

Copyright 1999, 2000 J Consortium, All Rights Reserved

the task T shall be placed at the end of that portion of the ready queue that represents tasks of this task's priority. If several tasks having the same priority are awakened by invocation of org.rtjwg.CoreObject.notifyAll() then all of these awakened tasks shall be placed at the end of that portion of the ready queue that corresponds to their respective priorities. If multiple tasks of equal priority are awakened by the notifyAll() invocation, these tasks shall be queued in FIFO order.

5. There shall be no blocking and consequently no queue of waiting tasks in the implementation of synchronized contexts for classes that implement the PCP interface. Synchronization of PCP objects shall be implemented using a priority ceiling protocol as defined here:

   a. On entry into a PCP-synchronized context, the Core Execution Environment checks to make sure that the priority of the current task is less than or equal to the ceiling priority associated with this PCP context. Otherwise, entry into the synchronized context is denied and the attempt to enter terminates by throwing a CorePCPError object.

   b. Assuming that entry into the PCP-synchronized context is not prohibited by the check performed in step a, the priority of the task is immediately raised to the level that is identified as the ceiling priority associated with this synchronization context.

   c. As long as this task continues to execute within the PCP-synchronized context, this task shall be prohibited from performing any operation that might block the task. If this task attempts to enter a synchronized context belonging to some other object except for PCP-synchronized contexts with higher ceiling priority than the currently locked PCP-synchronized context, or if it invokes CoreObject.wait(), or if it invokes SignalingSemaphore.P() or CountingSemaphore.P(), or if it invokes Mutex.lock(), the Core Execution Environment shall abort the offending operation by throwing a CorePCPError object.

   d. The Core Execution Environment shall assure that only one Core task at a time executes within any of the special contexts identified as PCP-synchronized regions. A sufficient, but not necessary, implementation consists of elevating the task's priority to the ceiling level and then suspending time slicing while the currently executing task is running within a priority ceiling context. The key required behaviors are that a task that is executing within a priority ceiling context runs uninterrupted until either:

      i. it is preempted by a higher priority task (a task with priority higher than the PCP ceiling priority), or

      ii. it completes execution of the body of code that comprises the PCP-synchronized context.

   e. Upon exit from the PCP-synchronized context, the Core Execution Environment shall:

      i. Restore this task's priority to its original value, queuing this task on the ready queue and dispatching the new highest priority ready task if it is no longer the highest priority ready task.

      ii. If there are no other Core tasks executing within PCP-synchronized contexts, the Core Execution Environment shall enable time slicing. The amount of time allotted to the first time slice shall be implementation-defined.

iii. If this CoreTask has received a stop() request, the Core Execution Environment shall begin processing the request by aborting the code that is currently executing and shall now give each suspended try statement an opportunity to execute its finally code.

3.9 Task Execution Model for Execution of Core-Baseline Methods

The Core programmer may identify certain methods of any Core class to be Core-Baseline methods. A Core-Baseline method is one that shall be invoked only from the Baseline domain. The type checking performed by the Core Verifier prevents a CoreTask from invoking a Core-Baseline method.

A Baseline thread which invokes a Core-Baseline method shall transfigure itself into the equivalent of a CoreTask for the duration of time that it is executing the Core-Baseline method. Upon return from the Core-Baseline method, the thread shall restore itself to have normal Baseline thread behavior. The key significance of this semantics is as follows:

1. All Core-Baseline methods shall execute with Base Priority equal to one, which is the lowest priority within the Core Execution Environment.
2. When a Core-Baseline method enters a synchronized context, all of which are governed either by the Core's priority inheritance or priority ceiling protocols, the priority of the running thread is automatically adjusted as required to implement the appropriate priority inversion avoidance protocol.
3. When a Core-Baseline method acquires a Mutex lock, its priority is automatically adjusted as required to implement priority inheritance protocols associated with the Mutex lock as long as the thread's control remains within the Core-Baseline method.
4. If a Core-Baseline method acquires a Mutex lock and then returns without releasing the lock, other core tasks which attempt to access the same lock shall experience priority inversion until such time as the Mutex lock is released. This results because the Core Execution Environment is unable to inherit priority to Baseline threads.
5. If a Baseline thread uses the Core-Baseline Mutex._lock() method to acquire a mutual exclusion lock, that particular lock is likely to exhibit priority inversion because the priority inheritance mechanism is not able to inherit Core task priorities to Baseline threads.

Note that it is generally inadvisable for Core programmers to write Core-Baseline methods that return without releasing all of the Mutex locks they might have acquired.

3.10 The Core Memory Model

A number of important issues have been raised regarding ambiguities, lack of conformance, and undesirable consequences associated with the Java memory model as it has been defined in reference 2. These issues are discussed in 12, 13, and 14. It is important for the Real-Time Java Working Group to take a stance on these issues by defining the Core Memory Model. At this time, we have permission from the authors to use reference 12 as a normative reference.

Copyright 1999, 2000 J Consortium, All Rights Reserved

3.11 Abort Mechanism and Asynchronous Transfer of Control in General

Invoking the stop() or abort() methods of org.rtjwg.CoreTask, or throwing CoreTask.abort-WorkException() shall cause the corresponding task to be aborted. When a task is aborted, all finally statements associated with currently executing contexts of the task shall be executed in reverse order of entry (the finally statement for the last try statement entered shall execute before all of the others). Further, the synchronization locks associated with currently executing synchronized contexts shall be unlocked, also in reverse order of entry into the corresponding synchronized contexts.

In order to improve the likelihood that stop() requests will be serviced quickly, the Core specification imposes a number of restrictions on the contents of finally statements. The purpose of these restrictions is to ensure that if control reaches a finally statement as part of the cleanup associated with abortion of a CoreTask, control will next flow to the surrounding finally statement following completion of the code contained within this finally statement. The restrictions described here constrain program control to stay within the finally statements associated with currently executing try contexts. After all finally statements have been executed, abortion of the CoreTask is complete. Even though these restrictions prevent control from flowing outside the finally statements, these restrictions are not sufficient to guarantee that finally statements complete their execution in a timely manner. For example, a finally statement may contain an infinite loop, or it may attempt to enter a synchronized context associated with an object that some other task has already synchronized indefinitely, or its attempt to coordinate with other tasks might result in a deadlock situation. In the spirit of supporting friendly cooperation between Core tasks, it is the Core programmer's responsibility, as a "trusted expert", to structure finally statements so that they run to completion in small bounded time. Otherwise, when some other task requests to abort this task, it will not abort in a timely manner.

The special Core requirements are as follows:

1. Except for Core-Baseline methods, finally statements within Core methods shall not contain break, continue, or return statements.
2. Except for Core-Baseline Methods, finally statements shall not include throw statements.
3. The Core Verifier and the Core Compiler shall enforce the above restrictions.
4. If a CoreTask is executing finally statements as part of the cleanup associated with responding to a stop() or abort() invocation, or as part of the handling for a thrown ScopedThrowable exception, and a CoreThrowable object is thrown from within the body of one of the finally statements (or from a method that was invoked from within the body of one of the finally statements), the Core Execution Environment shall catch and mask the thrown CoreThrowable object, shall consider the finally statement that threw the CoreThrowable object to have completed its execution, and shall resume cleanup activities by starting up execution of the next outer-nested finally statement if there is one, or shall consider cleanup activities to have been completed if there are no outer-nested finally statements to execute.
5. If a CoreTask is executing within a synchronized region of code that corresponds to an object that implements the Atomic interface when the CoreTask's stop() or abort() or signalAsync() method is invoked, handling of the asynchronous event handling request is deferred until after the CoreTask completes execution of the body of code that comprises the Atomic-synchronized context.

3.11.1 Asynchronous Transfer of Control

The CoreTask.abort() and CoreTask.stop() methods shall be implemented using a general purpose asynchronous transfer of control mechanism. Throughout the remainder of this section, we assume that the stop() method invokes the abort() method. Therefore, all discussion describing constraints imposed on implementation of the abort() method shall apply equally to implementation of the stop() method.

Asynchronous transfer of control is triggered by the CoreTask.signalAsync() and CoreTask.abort() methods (See Section 3.17.23 (starting on page 88)). When either of these methods is invoked, the following shall be performed for the target task:

1. If the task is constructed to ignore asynchronous events and this transfer-of-control request was triggered by invocation of the signalAsync() method (rather than by invocation of the CoreTask.abort() or CoreTask.stop() methods), ignore the request (throwing a CoreATCEventsIgnoredException in response to the signalAsync() invocation). Note that CoreTask.abort() and CoreTask.stop() always have the effect of aborting the CoreTask.work() method, even if the task was constructed to ignore asynchronous events.
2. If the task is currently executing within a deferral region, the task is allowed to continue executing until control leaves the deferral region. There are two kinds of deferral regions:
   a. The body of a synchronized statement contained within a class that implements the Atomic interface is a deferral region.
   b. The body of a finally statement is a deferral region.

   Once control has left the body of the deferral region, proceed to step 3.
3. If this control-transfer request was triggered by an abort() invocation, go to step 8.
4. Create a new activation frame on the task's run-time stack for execution of its event handling code. Establish the appropriate context on the run-time stack to arrange that if the event handling routine returns, the task's control resumes with the next instruction in sequence following the last instruction that was executed before the asynchronous control transfer took place.
5. The event handler for the task would have been set by a prior action of one of the following forms:
   a. At the time the task was constructed, one of the constructor arguments provides a reference to the initial event handler for the task.
   b. Subsequently, the event handler may have been replaced by invoking the task's asyncHandler() method.

   The Core Execution Environment shall invoke the handleATCEvent() method of the task's current event handler, using the task's run-time stack for the activation frame.
6. If the invoked handleATCEvent() method returns, control resumes within the interrupted method at the point where execution was originally preempted.
7. Otherwise, if the invoked handleATCEvent() method throws an exception, this exception is propagated up the call chain starting with the context that was originally preempted by the asynchronous event handler.
8. This control-transfer request was triggered by invocation of the task's abort() invocation. The Core Execution Environment shall invoke an appropriate implementa-

Copyright 1999, 2000 J Consortium, All Rights Reserved

tion-defined method to trigger abortion of the task. This implementation-defined method shall be declared with reduced visibility so as to not be accessible to Core application code. The Core Execution Environment shall provide dedicated temporary memory for this method's activation frame (rather than building the activation frame on the task's run-time stack) so as to avoid the risk of overflowing the task's run-time stack.

9. The implementation-defined method that is invoked to handle the abort() request shall throw the special ScopedThrowable object that is represented by CoreTask.currentTask().abortWorkException().

3.12 Stack Allocation of Dynamic Objects

The Core system shall support stack allocation according to the following protocols:

1. Within each Core method (excluding Core-Baseline methods), the programmer identifies which local reference variables are stackable. To say that a particular reference variable is stackable is to say that any new operation that assigns its result directly to this local variable shall be satisfied by allocating the new object from the run-time stack. The notational conventions depend on the programmer's choice of developer tools:
   a. Stylized Core source programmers concatenate the names of the variables into a string constant, separated by semicolons, and pass this string constant to the final public static method CoreRegistry.registerStackable(String) as the first executable code in the method's implementation. The following example shows the declaration of a method for an object which itself might reside on the stack (because this is stackable), which takes as an argument a reference to a CoreObject which might reside on the stack, and which presumably allocates an array of integers which would reside on the stack.

```
public java.lang.Object foo(int i, java.lang.Object x) {
   int [] ia;

   CoreRegistry.registerStackable("x;ia;this");

   // Body of method's implementation goes here
}
```

   Note that this example uses java.lang.Object as a placeholder representing org.rtjwg.CoreObject. This is the convention followed by Stylized Core source code developers.

   b. Syntactic Core source programmers use the stackable keyword in the declarations of each variable that is considered to reference a stack-allocatable object. For example, the program above might be represented by the alternative notation:

```
public stackable org.rtjwg.CoreObject foo(int i, stackable org.rtjwg.CoreObject x) {
   stackable int [] ia;

   // Body of method's implementation goes here
}
```

Note the use of the stackable keyword as an attribute of the foo() method. This signifies that the this argument is also stackable.

Throughout this document, we use the latter shorthand notation to identify stackable arguments in our descriptions of the officially defined Core API services.

The objective is that the class loader shall have an easy way to determine which variables are stackable, without impacting run-time overhead. After examining the arguments to the CoreRegistry.registerStackable() method, the Core Class Loader shall discard the invocation of registerStackable().

2. If a particular method has parameters (including this) which are declared to be stackable, then any class inheriting from this class must declare the same parameters (at least) to be stackable.

   This restriction is required in order to support polymorphism. If a particular method is known to accept stackable arguments, then all subclass implementations of the same method must accept the same stackable arguments. Otherwise, supporting stack allocation requires that interprocedural analysis of stackable arguments be performed each time new classes are loaded into the Core Execution Environment.

3. Additional restrictions are of the form described below. Throughout this discussion, the word “variable” refers to both local variables and to incoming arguments.

   a. Each Core Execution Environment shall identify through the CoreRegister.stackAllocation() API whether it supports stack allocation, returning true from this method if and only if all objects that this Core specification identifies as stack allocatable shall be allocated on the run-time stack.

   b. For each variable that is declared as stackable, a new object request that assigns its result to this variable shall be satisfied from the run-time stack if the Core Execution Environment claims to support stack allocation. If a stackable variable is declared to refer to a multi-dimensional array, all dimensions of any newly allocated array assigned to this variable shall be stack allocated.

   c. In order to allow the Core Execution Environment to blindly stack allocate each new object that is assigned to a stackable variable (including argument variables), the Core Verifier and Core Compiler shall enforce the following:

      i. There shall be no data path within the method that allows the value of any stackable variable to be copied to a local variable that is not identified as stackable.

      ii. There shall be no data path within the method that allows the stackable variable’s value to be copied into a field of a Core object (as an instance or static variable).

      iii. There shall be no data path within the method that allows the value of the stackable variable to be returned from this method as a return value.

      iv. There shall be no data path within the method that allows the stackable variable’s value to be copied to an outgoing argument list for invocation of another method unless the invoked method declares the corresponding formal argument to be of type stackable.

      v. For each new operation that assigns its result to a stackable variable, the constructor shall declare its this argument to be stackable.

      vi. Any new operation that assigns its result to a stackable variable shall not appear within a loop of the method.

Copyright 1999, 2000 J Consortium, All Rights Reserved

vii. If a given Core Execution Environment does not implement stack allocation, any allocated objects that would otherwise have been stack allocated are allocated instead within the currently active AllocationContext. The memory for these objects shall be reclaimed when the corresponding AllocationContext is released. See Section 3.17.8 for additional discussion on the topic of allocation contexts.

Note that all of these restrictions are described and enforced in terms of Core Class Files rather than source code. There are certain source-level notations, such as creation of inner classes that make reference to local final objects created in an outer class context, that appear to conform with the above-described restrictions even though the byte-code translation of these notations does not.

3.13 Initialization and Class Loading

The Core Execution Environment shall perform all class resolution and initialization at "load time". For dynamically loaded classes, load time is defined as the time when the class is dynamically loaded. For statically loaded classes, it is implementation-defined whether load time means the time when the Core Static Linker builds the memory image or the bootup time of the Core Execution Environment. This enables classes to be initialized prior to burning ROM, or to initialize themselves out of ROM at power-up time. Either approach offers superior performance to that of the Baseline language as it is currently specified.

3.14 Execution-Time Analyzable Code

The execution model for the Java language assumes that Java byte codes are validated by a byte-code analyzer prior to execution. In the Baseline environment, the purpose of this byte-code analyzer is to ensure that the byte codes are type-consistent. Besides making sure that byte codes will not introduce type mismatch errors, the Core Verifier has the additional responsibility of determining through analysis that particular bodies of Core code can be analyzed to determine their worst-case execution times (*WCET*).

The large majority of code comprising a Core program is not intended to be execution-time analyzable. However, there are certain contexts in which reliable compliance with stringent time constraints requires that the maximum time for execution of particular code segments be known prior to run time.

The following describes the properties that characterize byte-code representations of Core program segments that are considered to be execution-time analyzable.

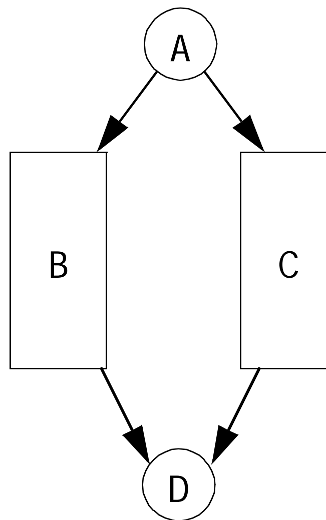
1. A straight-line sequence (without conditional or unconditional branching and without method invocations and without throw statements) of Java virtual machine instructions is execution-time analyzable as long as the sequence of instructions does not include new, newarray, anewarray, multianewarray, aastore, checkcast, or instanceof byte-code instructions.
2. The athrow instruction shall be execution-time analyzable. Note that the Core Execution Environment does not capture the stack backtrace in the representation of a thrown object. Note also that the time required to execute the athrow instruction includes the time required to find the appropriate catch clause. Though this time is context specific, the total cost can be calculated for any given context by summing

the appropriate contributions associated with searching each nested method activation frame as part of the cost associated with that method's invocation.

3. The code represented by an invokestatic or invokespecial instruction is execution-time analyzable if the body of the static or final method to be invoked is execution-time analyzable.

4. Given a program control flow consisting of a conditional branch and two alternative code flows that reunite at a common instruction, this complete control flow is execution-time analyzable if each of the alternative arms of the control flow is execution-time analyzable. In terms of the symbols diagrammed in Figure 2 on page 30, we say that the code path from point A to point D is execution-time analyzable if and only if the body of code (which need not be consecutive instructions) represented by B is execution-time analyzable and the body of code represented by C is also execution-time analyzable.

Figure 2. Analyzable Conditional Control Flow

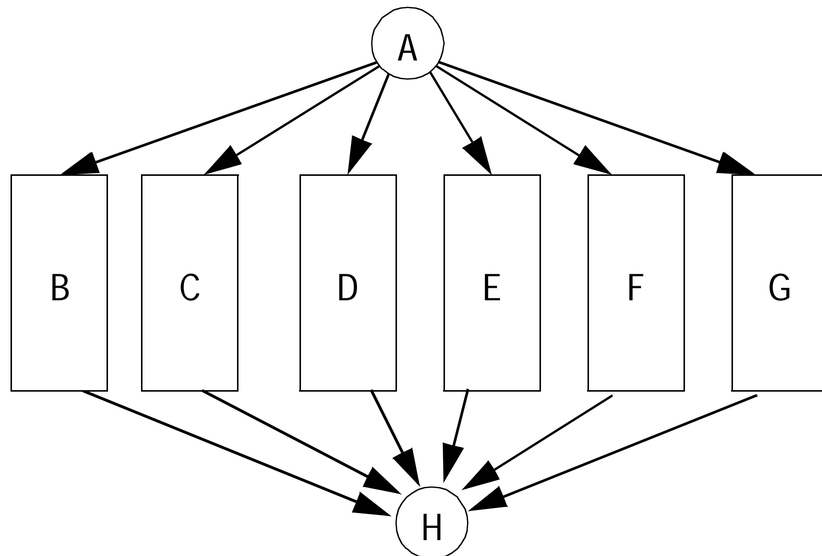


5. In Java byte code, both the lookupswitch and tableswitch instructions represent multiway conditional branches. A program control flow that starts with either of these byte-code instructions and ends at a common execution point reached by all paths originating from the starting point is execution-time analyzable if all of the paths between the starting point and ending point are execution-time analyzable. In terms of the symbols diagrammed in Figure 3 on page 31, we say that the code path from point A to point H is execution-time analyzable if and only if the bodies of code (which need not be consecutive instructions) represented by B, C, D, E, F, and G are each execution-time analyzable.

6. Within a class file's method representation, try clauses are identified by the exception_table array data structure (See Reference 8). Each entry in this table specifies the range of virtual machine instructions that is handled by each catch clause associated with the try statement. If a finally statement is associated with a particular

 Copyright 1999, 2000 J Consortium, All Rights Reserved

Figure 3. Analyzable Multi-Way Conditional Control Flow



try statement, the Baseline or Core Compiler inserts into the method's class-file representation an additional exception handler which handles all of the virtual machine instructions ranging from the body of the try statement to and including all bodies of all programmer-declared exception handlers corresponding to that try statement. This special exception handler is invoked if one of the programmer-defined exception handlers throws an exception during its execution.

To identify the sequence of code representing the body of a try statement, look at the range of instructions spanned by the exception handlers identified in the exception_table data structure. Some of the identified ranges represent the bodies of try statements. The others represent the combined bodies of a try statement and all of its programmer-defined exception handlers. Identify the finally statement entries by separating out the entries whose ranges are supersets of other ranges. All of the remaining entries identify ranges that correspond to the bodies of try statements.

To identify the sequence of code representing each programmer-defined exception handler associated with a particular try statement, look at each of the exception handler entries for that try statement in the exception_table data structure. (Don't include the special finally statement exception handler.) Each of these entries identifies the first instruction of each exception handler. The body of code for the exception handler starts with this first instruction and ends with a goto byte-code instruction that jumps to the code following the try statement's body. The destination address of the goto instruction is the first point of convergence between the control-flow subgraph starting at the try statement first instruction and the control flow subgraph starting at the exception handler's first instruction.

To identify the sequence of code representing the body of the finally statement associated with a particular try statement, look at the finally statement entry (described above) within the exception_table data structure and extract from this entry the target

address. (Note that some try statements don't have finally statements.) This represents the address of the first virtual machine instruction in the special finally-statement exception handler. The instruction at this address is a jsr instruction, which jumps to the subroutine representing the body of this finally statement. For each identified range, the body of the try statement comprises the code starting with the instruction at the jsr target address and includes all code up to and including the ret instruction that marks the end of the finally statement subroutine. In some cases, the finally statement may have multiple ret instructions. The body of the finally statement subroutine is execution-time analyzable if every path from the entry point to any of the ret instructions that represents completion of this subroutine is execution-time analyzable.

A try statement, including its catch and finally clauses, is execution-time analyzable if and only if (1) the body of the try statement itself is execution-time analyzable, (2) the body of each catch clause, if any, associated with this try statement is execution-time analyzable, and (3) the body of the finally clause, if any, associated with this try statement is execution-time analyzable.

7. As described in Reference 9, a *natural loop* is defined as follows:

a. A *basic block* is a sequence of consecutive byte-code instructions into which control enters at the first instruction and from which control leaves following execution of the last instruction, without any possibility of halting or branching except after the last instruction.

b. A *flow graph* is a collection of nodes representing basic blocks of a computer program which are connected by directed edges representing possible control flow between basic blocks. In particular, the flow graph has a directed edge from node B_1 to node B_2 if:

i. There is a conditional or unconditional jump from the last instruction in the basic block represented by node B_1 to the first instruction in the basic block represented by B_2, or if

ii. The basic block represented by node B_2 immediately follows the basic block represented by node B_1 in the program sequence and the last instruction in block B_1 is not an unconditional jump instruction.

c. We say that node d of a flow graph *dominates* node n if every path from the initial node of the flow graph to node n passes through node d. Note that every node dominates itself.

d. A *back edge* is a directed edge of a flow graph whose head dominates its tail. (Given a directed edge pointing from node B_1 to node B_2, we call B_1 the tail of the directed edge and B_2 the head of the directed edge.) Each back edge in the flow graph corresponds to a loop.

e. Given a back edge $n \rightarrow d$, the *natural loop* of that edge is the node d plus all nodes that can reach node n without passing through node d. We call node d the *header* of the loop.

Algorithms to identify dominator relationships and natural loops within an arbitrary flow graph are available in Reference 9. Given a natural loop, we define the following two additional terms for purposes of facilitating discussion regarding the analysis of loop execution time:

Copyright 1999, 2000 J Consortium, All Rights Reserved

a. We characterize a *departure edge* of natural loop N to be a directed edge for which the head is a node not contained within the loop and the tail is a node contained within the loop.

b. For each departure edge, we call the node that represents the departure edge's tail a *departure node*.

A natural loop is considered to be execution-time analyzable if and only if all of the following conditions are satisfied:

a. Every path within the flow graph from the loop header back to the loop header is execution-time analyzable.

b. There exists at least one departure node for the loop that exhibits the following properties:

i. The departure node dominates each node within the loop that has a back edge to the loop's header. Note that the header dominates itself, and the node containing the back edge also dominates itself. Note further that the departure node must terminate with a conditional branch. Otherwise, there would be no way for the departure node to be contained within the loop and yet have an edge directed to a node that is outside the loop.

ii. The condition upon which the departure node decides whether to depart from the loop is a simple integer magnitude comparison involving a local variable (call the variable *j*) and an integer constant value with no additional arithmetic.

iii. Within the loop, there is only one assignment to the variable *j*. This assignment must be contained within a basic block whose node dominates all other nodes within this loop that have back edges directed to this loop's header. Furthermore, this basic block shall not be contained within any inner nested loop. An inner nested loop is a natural loop whose header is contained within this loop and is distinct from this loop's header. In summary, the variable *j* shall be incremented or decremented exactly once on each iteration of this loop. Furthermore, the value assigned to variable *j* must be obtained by adding or subtracting a non-zero integer constant to the previous value of the variable *j*.

iv. There is only one definition of the variable *j* which reaches the header of the loop (See *reaching definitions* in Reference 9) from outside of the loop and the value assigned to *j* by this definition must be a simple integer constant.

3.14.1 Analyzability of Core Source Code

The characterization of execution-time analyzable code presented above is described in terms of class-file byte-code representations. Most Core programmers prefer to think in terms of Core source code conventions rather than in terms of their byte-code representations. To facilitate development of reliable Core source code components, the Core specifications requires that a Core Compiler shall translate all of the following constructs into execution-time analyzable byte-code program segments:

1. A straight-line body of Core Source Code shall be translated by the Core Compiler into execution-time analyzable byte code provided that this body of code does not

include any new memory allocation requests, reference type coercions, instanceof operators, or assignments to an element of a reference array.

2. A throw statement shall be translated by the Core Compiler into execution-time analyzable byte code if the expression that defines the value to be thrown meets the constraints of paragraph 1 immediately above.
3. An invocation of a static or final method shall be translated by the Core Compiler into execution-time analyzable byte code provided that the implementation of the invoked static or final method is execution-time analyzable.
4. The Core Compiler shall translate if statements and if-else statements to execution-time analyzable byte code if the conditional expression, the body of the if-clause, and the body of the else-clause, if present, are all execution-time analyzable.
5. The Core Compiler shall translate switch statements to execution-time analyzable byte code if the controlling expression and the bodies of code representing each case are each independently execution-time analyzable.
6. The Core Compiler shall translate for statements to execution-time analyzable byte code if the iteration variable is an integer that is initialized to a constant prior to the loop and incremented or decremented by a constant value exactly once on each iteration of the loop as part of the for statement's control clause, and the body of the for loop is itself execution-time analyzable. The Core Compiler shall allow break and continue statements in execution-time analyzable loops.

3.14.2 Predictability of the Core Execution Environment

In order to enable deployment of execution-time predictable Core real-time components, the Core specification imposes the following constraints on implementations of the Core virtual machine:

1. The time required to execute all virtual machine instructions is constant, except for the following special instructions:
   a. The time required to execute new, newarray, anewarray, and multianewarray instructions is implementation-defined and need not be constant or predictable.
   b. The maximum time required to execute the aastore, checkcast, and instanceof instructions shall be proportional to the depth of the loaded class hierarchy.
   c. The maximum time required to execute an athrow instruction is proportional to the depth of the current thread's run-time stack, measured in stack frames.
   d. The time required to execute an invokeinterface instruction is implementation-defined and need not be constant or predictable.
2. The CPU time and dynamic memory impact of each of the official Core API libraries, including Core-Baseline methods, shall be as detailed in Table 1 on page 35. Within this table, saying that CPU requirements are implementation-defined means that the supplier of a conforming Core Execution Environment shall either provide documentation that details the CPU requirements for the particular implementation running on a particular platform, or shall provide tools and appropriate documentation to allow users to measure the implementation-defined CPU requirements for each method. Providing statistically significant measurement-based characterizations of CPU requirements shall be an acceptable replacement for analytical guarantees.

Copyright 1999, 2000 J Consortium, All Rights Reserved

TABLE 1. Predictability Requirements for Core API Libraries

Class Name		
Method Name	**CPU Requirements**	**Memory Impact**
CoreObject		
constructors	Bounded by an implementation-defined constant.	The new object shall be allocated within the current AllocationContext. No other memory shall be allocated.
clone()	Bounded by an implementation-defined function which is linear in the size of the object being cloned.	The new object shall be allocated within the current AllocationContext. No other memory shall be allocated.
equals()	Bounded by an implementation-defined constant.	No memory allocation.
getClass()	Bounded by an implementation-defined constant.	No memory allocation.
hashCode()	Bounded by an implementation-defined constant.	No memory allocation.
notify()	Bounded by an implementation-defined constant.	No memory allocation.
notifyAll()	Bounded by an implementation-defined constant. The work of notifying multiple waiting tasks shall be distributed amongst the wait() invocations of the waiting tasks.	No memory allocation.
toString()	Bounded by an implementation-defined constant.	The returned CoreString object and the corresponding character buffer, if any, shall be allocated within the current AllocationContext. How much memory is required to represent a CoreString object of the specific length shall be implementation-defined. No other memory shall be allocated.
wait()	No CPU-time bound required on this method.	No memory allocation.
arrayAddress()	Bounded by an implementation-defined constant.	No memory allocation.
sizeof()	Bounded by an implementation-defined constant.	No memory allocation.

TABLE 1. Predictability Requirements for Core API Libraries

Class Name			
	Method Name	**CPU Requirements**	**Memory Impact**
CoreThrowable			
	constructors	Bounded by an implementation-defined constant.	The new CoreThrowable object shall be allocated within the current AllocationContext. The CoreString message argument shall not be copied. Instead, the constructed CoreThrowable object shall simply maintain a reference to the supplied message argument. No other memory shall be allocated.
	getMessage()	Bounded by an implementation-defined constant.	No memory allocation.
CoreRuntimeException[a]			
	constructors	Bounded by an implementation-defined constant.	The new CoreThrowable object shall be allocated within the current AllocationContext. The CoreString message argument shall not be copied. Instead, the constructed CoreThrowable object shall simply maintain a reference to the supplied message argument. No other memory shall be allocated.
CoreException[b]			
	constructors	Bounded by an implementation-defined constant.	The new CoreException object shall be allocated within the current AllocationContext. The CoreString message argument shall not be copied. Instead, the constructed CoreException object shall simply maintain a reference to the supplied message argument. No other memory shall be allocated.

Copyright 1999, 2000 J Consortium, All Rights Reserved

TABLE 1. Predictability Requirements for Core API Libraries

Class Name			
	Method Name	**CPU Requirements**	**Memory Impact**
ScopedException			
	constructors	Bounded by an implementation-defined constant.	The new ScopedException object shall be allocated within the current AllocationContext. The CoreString message argument shall not be copied. Instead, the constructed ScopedException object shall simply maintain a reference to the supplied message argument. No other memory shall be allocated.
	enable()	Bounded by an implementation-defined constant.	No memory allocation.
	disable()	Bounded by an implementation-defined constant.	No memory allocation.
CoreClass			
	forName()	No CPU-time bound required on this method.	No memory allocation.
	getComponentType()	Bounded by an implementation-defined constant.	No memory allocation.
	isArray()	Bounded by an implementation-defined constant.	No memory allocation.
	isAssignableFrom()	No CPU-time bound required on this method.	No memory allocation.
	isInstance()	No CPU-time bound required on this method.	No memory allocation.
	isInterface()	Bounded by an implementation-defined constant.	No memory allocation.
	isPrimitive()	Bounded by an implementation-defined constant.	No memory allocation.

TABLE 1. Predictability Requirements for Core API Libraries

Class Name		
Method Name	**CPU Requirements**	**Memory Impact**
CoreClass		
newInstance()	No CPU-time bound required on this method.	The new object shall be allocated within the current AllocationContext. No other memory shall be allocated, except for whatever memory is allocated by execution of the new object's no-argument constructor.
toString()	No CPU-time bound required on this method.	The returned CoreString object and the corresponding character buffer, if any, shall be allocated within the current AllocationContext. How much memory is required to represent a CoreString object of the specific length shall be implementation-defined. No other memory shall be allocated.
verification()	Bounded by an implementation-defined constant.	No memory allocation.
loadClass()	No CPU-time bound required on this method.	No bound on the number of objects allocated by this method. May allocate multiple temporary objects within the current AllocationContext. None of these objects is used following return from this method. Additionally, a small implementation-defined quantity of more permanent objects shall be allocated within a special implementation-defined AllocationContext for the purpose of representing the newly loaded class within the Core Execution Environment. When (if) this class is subsequently unloaded, the unloadClass() method shall release the special AllocationContext.
unloadClass()	No CPU-time bound required on this method.	As a side effect of unloading this class, the special implementation-defined AllocationContext that was created for the purpose of representing this class shall be released. No memory shall be allocated.

Copyright 1999, 2000 J Consortium, All Rights Reserved

TABLE 1. Predictability Requirements for Core API Libraries

Class Name			
	Method Name	**CPU Requirements**	**Memory Impact**
CoreArray[c]			
	constructors	Bounded by an implementation-defined function that is linear in the number of slots in the array.	The new array object is allocated within the current AllocationContext. No other memory is allocated.
	length()	Bounded by an implementation-defined constant.	No memory allocation.
	atGet()	Bounded by an implementation-defined constant.	No memory allocation.
	atPut()	Bounded by an implementation-defined constant.	No memory allocation.
AllocationContext			
	constructors	No CPU-time bound required for this method.	The new AllocationContext object is allocated within the current AllocationContext. No other memory is allocated.
	available()	Bounded by an implementation-defined constant.	No memory allocation.
	allocated()	Bounded by an implementation-defined constant.	No memory allocation.
	release()	No CPU-time bound required for this method.	No memory allocation.
SpecialAllocation			
	context()	No constraint. This is an abstract method which must be implemented by the application developer.	No constraint. This is an abstract method which must be implemented by the application developer.
	run()	No constraint. This is an abstract method which must be implemented by the application developer.	No constraint. This is an abstract method which must be implemented by the application developer.
	execute()	The work performed by this method, excluding the work performed by this.run() which is invoked from within this method, shall be bounded by an implementation-defined constant.	No memory allocation shall be performed by this method. However, there is no bound on the amount of memory that might be allocated from within the run() method which is invoked by this method.

TABLE 1. Predictability Requirements for Core API Libraries

Class Name			
	Method Name	**CPU Requirements**	**Memory Impact**
CoreString			
	constructors	Bounded by an implementation-defined function that depends on the length of the CoreString to be constructed.	The newly constructed CoreString object and the corresponding character buffer, if any, shall be allocated within the current AllocationContext. How much memory is required to represent a CoreString object of the specific length shall be implementation-defined. No other memory shall be allocated.
	charAt()	Bounded by an implementation-defined constant.	No memory allocation.
	_charAt()	Bounded by an implementation-defined constant.	No memory allocation.
	hashCode()	Bounded by an implementation-defined function that depends on the length of this CoreString object.	No memory allocation.
	_hashCode()	Bounded by an implementation-defined function that depends on the length of this CoreString object.	No memory allocation.
	equals()	Bounded by an implementation-defined function that depends on the length of this CoreString object.	No memory allocation.
	length()	Bounded by an implementation-defined constant.	No memory allocation.
	_length()	Bounded by an implementation-defined constant.	No memory allocation.

Copyright 1999, 2000 J Consortium, All Rights Reserved

TABLE 1. Predictability Requirements for Core API Libraries

Class Name			
	Method Name	**CPU Requirements**	**Memory Impact**
DynamicCoreString			
	constructors	Bounded by an implementation-defined function that depends on the length of the DynamicCoreString to be constructed.	The newly constructed DynamicCoreString object and the corresponding character buffer, if any, shall be allocated within the current AllocationContext. How much memory is required to represent a DynamicCoreString object of the specific length shall be implementation-defined. No other memory shall be allocated.
	concat()	Bounded by an implementation-defined function that depends on the sum of the lengths of the two strings that are being concatenated.	The returned DynamicCoreString object and the corresponding character buffer, if any, shall be allocated within the current AllocationContext. How much memory is required to represent a DynamicCoreString object of the specific length shall be implementation-defined. No other memory shall be allocated.
	getChars()	Bounded by an implementation-defined function that depends on the length of this DynamicCoreString object.	No memory allocation.
	length()	Bounded by an implementation-defined constant.	No memory allocation.
	_length()	Bounded by an implementation-defined constant.	No memory allocation.

TABLE 1. Predictability Requirements for Core API Libraries

Class Name		
Method Name	**CPU Requirements**	**Memory Impact**
DynamicCoreString		
substring()	Bounded by an implementation-defined function that depends on the length of the requested substring.	The returned DynamicCoreString object and the corresponding character buffer, if any, shall be allocated within the current AllocationContext. If the current AllocationContext is different from the AllocationContext within which this DynamicCoreString resides, the substring() method shall make a new copy of the substring characters which shall reside within an object belonging to the current AllocationContext. How much memory is required to represent a DynamicCoreString object of the specific length shall be implementation-defined. No other memory shall be allocated.
toCharArray()	Bounded by an implementation-defined function that depends on the length of the requested character array.	The returned array of characters shall be allocated within the current AllocationContext. No other memory shall be allocated.
toLowerCase()	Bounded by an implementation-defined function that depends on the length of this DynamicCoreString object.	The returned DynamicCoreString object and the corresponding character buffer, if any, shall be allocated within the current AllocationContext. The returned DynamicCoreString shall not make reference to any character buffer object residing in an AllocationContext that is not the current AllocationContext. How much memory is required to represent a DynamicCoreString object of the specific length shall be implementation-defined. No other memory shall be allocated.

Copyright 1999, 2000 J Consortium, All Rights Reserved

TABLE 1. Predictability Requirements for Core API Libraries

Class Name / Method Name	CPU Requirements	Memory Impact
DynamicCoreString		
toUpperCase()	Bounded by an implementation-defined function that depends on the length of this DynamicCoreString object.	The returned DynamicCoreString object and the corresponding character buffer, if any, shall be allocated within the current AllocationContext. The returned DynamicCoreString shall not make reference to any character buffer object residing in an AllocationContext that is not the current AllocationContext. How much memory is required to represent a DynamicCoreString object of the specific length shall be implementation-defined. No other memory shall be allocated.
ATCEventHandler		
constructor	Bounded by an implementation-defined constant.	The new ATCEventHandler object is allocated within the current AllocationContext. No other memory is allocated.
handleATCEvent()	Bounded by an implementation-defined constant.	No memory allocation.
ATCEvent		
constructor	Bounded by an implementation-defined constant.	The new ATCEvent object is allocated within the current AllocationContext. No other memory is allocated.
defaultAction()	Bounded by an implementation-defined constant.	No memory allocation.

TABLE 1. Predictability Requirements for Core API Libraries

Class Name		
Method Name	**CPU Requirements**	**Memory Impact**
CoreRegistry		
stackAllocation()	Bounded by an implementation-defined constant.	No memory allocation.
registerStackable()	The registerStackable() method shall be removed from the executable code by the Core Class Loader. Thus, the implementation of registerStackable() shall require no CPU time.	No memory allocation.
registerBaseline()	The registerBaseline() method shall be removed from the executable code by the Core Class Loader. Thus, the implementation of registerBaseline() shall require no CPU time.	No memory allocation.
registerCoreClass()	The registerCoreClass() method shall be removed from the executable code by the Core Class Loader. Thus, the implementation of registerCoreClass() shall require no CPU time.	No memory allocation.
coerce()	Bounded by an implementation-defined constant.	No memory allocation.
profiles()	No CPU-time bound required for this method.	The array returned from this method shall be allocated in the current AllocationContext. The CoreString objects referenced from the array shall not be allocated by invocation of this method. Instead, these CoreString objects shall be pre-allocated from within an implementation-defined AllocationContext and reused for each invocation of the profiles() method.

Copyright 1999, 2000 J Consortium, All Rights Reserved

TABLE 1. Predictability Requirements for Core API Libraries

Class Name			
	Method Name	**CPU Requirements**	**Memory Impact**
CoreRegistry			
	publish()	No CPU-time bound required for this method.	A small implementation-defined number of objects shall be allocated within a special implementation-defined AllocationContext for the purpose of representing the published information within the Core Execution Environment. When (if) this entry is subsequently unpublished, this special AllocationContext shall be released. No other memory shall be allocated.
	unpublish()	No CPU-time bound required for this method.	The implementation-defined AllocationContext that was created by the corresponding invocation of the publish() method shall be released.

TABLE 1. Predictability Requirements for Core API Libraries

Class Name		
Method Name	**CPU Requirements**	**Memory Impact**
SignalingSemaphore		
constructor	Bounded by an implementation-defined constant.	The new SignalingSemaphore object is allocated within the current AllocationContext. No other memory is allocated.
P()	Bounded by an implementation-defined function that depends only on the number of other tasks that are concurrently performing P() or _P() operations on this semaphore.	No memory allocation.
_P()	Bounded by an implementation-defined function that depends only on the number of other tasks that are concurrently performing P() or _P() operations on this semaphore.	No memory allocation.
V()	Bounded by an implementation-defined constant.	No memory allocation.
_V()	Bounded by an implementation-defined constant.	No memory allocation.
Vall()	Bounded by an implementation-defined constant. Note that the effort required to signal multiple blocked waiters shall be distributed between the various tasks' P() invocations.	No memory allocation.
_Vall()	Bounded by an implementation-defined constant. Note that the effort required to signal multiple blocked waiters shall be distributed between the various tasks' P() invocations.	No memory allocation.
numWaiters()	Bounded by an implementation-defined constant.	No memory allocation.
_numWaiters()	Bounded by an implementation-defined constant.	No memory allocation.

Copyright 1999, 2000 J Consortium, All Rights Reserved

TABLE 1. Predictability Requirements for Core API Libraries

Class Name		
Method Name	**CPU Requirements**	**Memory Impact**
CountingSemaphore		
constructor	Bounded by an implementation-defined constant.	The new CountingSemaphore object is allocated within the current AllocationContext. No other memory is allocated.
P()	Bounded by an implementation-defined function that depends only on the number of other tasks that are concurrently performing P() or _P() operations on this semaphore.	No memory allocation.
_P()	Bounded by an implementation-defined function that depends only on the number of other tasks that are concurrently performing P() or _P() operations on this semaphore.	No memory allocation.
V()	Bounded by an implementation-defined constant.	No memory allocation.
_V()	Bounded by an implementation-defined constant.	No memory allocation.
numWaiters()	Bounded by an implementation-defined constant.	No memory allocation.
_numWaiters()	Bounded by an implementation-defined constant.	No memory allocation.
count()	Bounded by an implementation-defined constant.	No memory allocation.
_count()	Bounded by an implementation-defined constant.	No memory allocation.

TABLE 1. Predictability Requirements for Core API Libraries

Class Name			
	Method Name	**CPU Requirements**	**Memory Impact**
Mutex			
	constructor	Bounded by an implementation-defined constant.	The new Mutex object is allocated within the current AllocationContext. No other memory is allocated.
	lock()	Bounded by an implementation-defined function that depends only on the number of other tasks that are performing lock() or _lock() operations on this Mutex object.	No memory allocation.
	_lock()	Bounded by an implementation-defined function that depends only on the number of other tasks that are performing lock() or _lock() operations on this Mutex object.	No memory allocation.
	unlock()	Bounded by an implementation-defined constant.	No memory allocation.
	_unlock()	Bounded by an implementation-defined constant.	No memory allocation.

Copyright 1999, 2000 J Consortium, All Rights Reserved

TABLE 1. Predictability Requirements for Core API Libraries

Class Name		
Method Name	**CPU Requirements**	**Memory Impact**
Time		
tickDuration()	Bounded by an implementation-defined constant.	No memory allocation.
uptimePrecision()	Bounded by an implementation-defined constant.	No memory allocation.
day()	Bounded by an implementation-defined constant.	No memory allocation.
h()	Bounded by an implementation-defined constant.	No memory allocation.
hertz()	Bounded by an implementation-defined constant.	No memory allocation.
m()	Bounded by an implementation-defined constant.	No memory allocation.
ms()	Bounded by an implementation-defined constant.	No memory allocation.
ns()	Bounded by an implementation-defined constant.	No memory allocation.
s()	Bounded by an implementation-defined constant.	No memory allocation.
toString()	Bounded by an implementation-defined constant.	The returned CoreString object and the corresponding character buffer, if any, shall be allocated within the current AllocationContext. How much memory is required to represent a CoreString object of the specific length shall be implementation-defined. No other memory shall be allocated.
uptime()	Bounded by an implementation-defined constant.	No memory allocation.
us()	Bounded by an implementation-defined constant.	No memory allocation.

TABLE 1. Predictability Requirements for Core API Libraries

Class Name		
Method Name	**CPU Requirements**	**Memory Impact**
CoreTask		
constructor	No CPU-time bound required for this method.	The CoreTask object shall be allocated in the current AllocationContext. Certain additional implementation-defined objects shall be allocated, as required to implement the services associated with this CoreTask object. These additional objects shall be allocated within the default AllocationContext for this CoreTask. When this CoreTask's AllocationContext is released, the Core Execution Environment shall overwrite all automatically constructed references to these implementation-defined objects with null.
currentTask()	Bounded by an implementation-defined constant.	No memory allocation.
defaultStackSize()	Bounded by an implementation-defined constant.	No memory allocation.
maxBaselinePriority()	Bounded by an implementation-defined constant.	No memory allocation.
maxCorePriority()	Bounded by an implementation-defined constant.	No memory allocation.
maxSystemPriority()	Bounded by an implementation-defined constant.	No memory allocation.
minBaselinePriority()	Bounded by an implementation-defined constant.	No memory allocation.
minCorePriority()	Bounded by an implementation-defined constant.	No memory allocation.
minSystemPriority()	Bounded by an implementation-defined constant.	No memory allocation.
numInterruptPriorities()	Bounded by an implementation-defined constant.	No memory allocation.
stackOverflowChecking()	Bounded by an implementation-defined constant.	No memory allocation.
systemPriorityMap()	Bounded by an implementation-defined constant.	The returned integer array shall be allocated from the current AllocationContext. No other memory shall be allocated.

Copyright 1999, 2000 J Consortium, All Rights Reserved

TABLE 1. Predictability Requirements for Core API Libraries

Class Name		
Method Name	**CPU Requirements**	**Memory Impact**
CoreTask		
ticksPerSlice()	Bounded by an implementation-defined constant.	No memory allocation.
abort()	No CPU-time bound required for this method.	No memory allocation.
abortWorkException()	Bounded by an implementation-defined constant.	No memory allocation.
asyncHandler()	Bounded by an implementation-defined constant.	No memory allocation.
join()	No CPU-time bound required for this method.	No memory allocation.
resume()	No CPU-time bound required for this method.	No memory allocation.
setPriority()	No CPU-time bound required for this method.	No memory allocation.
signalAsync()	No CPU-time bound required for this method.	No memory allocation.
sleep()	No CPU-time bound required for this method.	No memory allocation.
sleepUntil()	No CPU-time bound required for this method.	No memory allocation.
stackDepth()	Bounded by an implementation-defined constant.	No memory allocation.
stackSize()	Bounded by an implementation-defined constant.	No memory allocation.
start()	No CPU-time bound required for this method.	Bounded by an implementation-defined constant. All of the new memory shall be allocated in the default AllocationContext of this CoreTask.
_start()	No CPU-time bound required for this method.	Bounded by an implementation-defined constant. All of the new memory shall be allocated in the default AllocationContext of this CoreTask.
stop()	No CPU-time bound required for this method.	No memory allocation.
suspend()	No CPU-time bound required for this method.	No memory allocation.

TABLE 1. Predictability Requirements for Core API Libraries

Class Name			
	Method Name	**CPU Requirements**	**Memory Impact**
CoreTask			
	systemPriority()	Bounded by an implementation-defined constant.	No memory allocation.
	work()	Bounded by an implementation-defined constant.	No memory allocation.
	yield()	No CPU-time bound required for this method.	No memory allocation.
ISR_Task			
	constructor	No CPU-time bound required for this constructor.	The ISR_Task object itself shall be allocated in the current AllocationContext. Certain additional implementation-defined objects (e.g. the run-time stack) shall also be allocated, as required to implement all of the services associated with this ISR_Task object. These additional objects shall be allocated within the default AllocationContext for this ISR_Task. When it is time to release this ISR_Task's AllocationContext, the Core Execution Environment shall overwrite all of the automatically constructed references to these implementation-defined objects with null pointers.
	serviced()	Bounded by an implementation-defined constant.	No memory allocation.
	work()	Bounded by an implementation-defined constant.	No memory allocation.
	ceilingPriority()	Bounded by an implementation-defined constant.	No memory allocation.
	trigger()	No CPU-time bound required for this method.	No memory allocation.
	arm()	Bounded by an implementation-defined constant.	No memory allocation.
	disarm()	Bounded by an implementation-defined constant.	No memory allocation.

Copyright 1999, 2000 J Consortium, All Rights Reserved

TABLE 1. Predictability Requirements for Core API Libraries

Class Name		
Method Name	**CPU Requirements**	**Memory Impact**
SporadicTask		
constructor	No CPU-time bound required for this constructor.	The SporadicTask object itself shall be allocated in the current AllocationContext. Certain additional implementation-defined objects (e.g. the run-time stack) shall also be allocated, as required to implement all of the services associated with this SporadicTask object. These additional objects shall be allocated within the default AllocationContext for this SporadicTask. When it is time to release this SporadicTask's AllocationContext, the Core Execution Environment shall overwrite all of the automatically constructed references to these implementation-defined objects with null.
trigger()	Bounded by an implementation-defined constant.	No memory allocation.
work()	Bounded by an implementation-defined constant.	No memory allocation.
pendingCount()	Bounded by an implementation-defined constant.	No memory allocation.
clearPending()	Bounded by an implementation-defined constant.	No memory allocation.

TABLE 1. Predictability Requirements for Core API Libraries

Class Name		
Method Name	**CPU Requirements**	**Memory Impact**
IOPort[d]		
createIOPort()	No CPU-time bound required for this method.	The returned IOPort subclass shall be allocated in the current AllocationContext. No other memory shall be allocated.
readByte()	Bounded by an implementation-defined constant.	No memory allocation.
writeByte()	Bounded by an implementation-defined constant.	No memory allocation.
readShort()	Bounded by an implementation-defined constant.	No memory allocation.
writeShort()	Bounded by an implementation-defined constant.	No memory allocation.
readInt()	Bounded by an implementation-defined constant.	No memory allocation.
writeInt()	Bounded by an implementation-defined constant.	No memory allocation.
readLong()	Bounded by an implementation-defined constant.	No memory allocation.
writeLong()	Bounded by an implementation-defined constant.	No memory allocation.

Copyright 1999, 2000 J Consortium, All Rights Reserved

TABLE 1. Predictability Requirements for Core API Libraries

Class Name		
Method Name	**CPU Requirements**	**Memory Impact**
Unsigned		
compare()	Bounded by an implementation-defined constant.	No memory allocation.
ge()	Bounded by an implementation-defined constant.	No memory allocation.
gt()	Bounded by an implementation-defined constant.	No memory allocation.
le()	Bounded by an implementation-defined constant.	No memory allocation.
lt()	Bounded by an implementation-defined constant.	No memory allocation.
eq()	Bounded by an implementation-defined constant.	No memory allocation.
neq()	Bounded by an implementation-defined constant.	No memory allocation.
toByte()	Bounded by an implementation-defined constant.	No memory allocation.
toShort()	Bounded by an implementation-defined constant.	No memory allocation.
toInt()	Bounded by an implementation-defined constant.	No memory allocation.
toLong()	Bounded by an implementation-defined constant.	No memory allocation.
toString()	Bounded by an implementation-defined constant.	The returned CoreString object and the corresponding character buffer, if any, shall be allocated within the current AllocationContext. How much memory is required to represent a CoreString object of the specific length shall be implementation-defined. No other memory shall be allocated.
toHexString()	Bounded by an implementation-defined constant.	The returned CoreString object and the corresponding character buffer, if any, shall be allocated within the current AllocationContext. How much memory is required to represent a CoreString object of the specific length shall be implementation-defined. No other memory shall be allocated.

a. The characterization of the constructor for CoreRuntimeException applies to CoreIllegalMonitorStateException, CoreOutOfMemoryException, CoreArrayIndexOutOfBoundsException, CoreClassFormatError.

b. The characterization of the constructor for CoreException applies also to the constructors for CoreOperationNotPermittedException, CoreSecurityException, CoreBadPriorityException, CoreEmbeddedConflictException, CoreATCEventsIgnoredException, CoreBadArgumentException, CoreUnsignedCoercionException, CoreClassInUseException, CoreClassNotFoundException, CoreArithmeticOverflowException, and CoreObjectNotAddressableException.

c. Within this table, all of the comments relevant to CoreArray apply equally to CoreBoolArray, CoreByteArray, CoreShortArray, CoreCharArray, CoreIntArray, CoreLongArray, CoreFloatArray, CoreDoubleArray, and CoreRefArray.

d. Within this table, all comments pertaining to IOPort apply equally to each of its officially defined subclasses, including IOPort8I, IOPort8O, IOPort8IO, IOPort16I, IOPort16O, IOPort16IO, IOPort32I, IOPort32O, IOPort32IO, IOPort64I, IOPort64O, and IOPort64IO.

3. The CPU time and dynamic memory impact of the C/Native API libraries described in Section 3.16 (starting on page 57) shall be as detailed in Table 2 on page 56.

TABLE 2. Predictability Requirements for the C/Native API

C Function Name	CPU Requirements	Memory Impact
coreRegistryLookup()	No CPU-time bound requirement for this function.	No memory allocation.
maxCorePriority()	Bounded by an implementation-defined constant.	No memory allocation.
minCorePriority()	Bounded by an implementation-defined constant.	No memory allocation.
corePriorityMap()	Bounded by an implementation-defined constant.	No memory allocation.
maxBaselinePriority()	Bounded by an implementation-defined constant.	No memory allocation.
minBaselinePriority()	Bounded by an implementation-defined constant.	No memory allocation.
coreInterruptLevels()	Bounded by an implementation-defined constant.	No memory allocation.
semaphoreP()	No CPU-time bound requirement for this function.	No memory allocation.
semaphoreV()	Bounded by an implementation-defined constant.	No memory allocation.
semaphoreVall()	Bounded by an implementation-defined constant.	No memory allocation.
enterSynchronized()	No CPU-time bound requirement for this function.	No memory allocation.
exitSynchronized()	Bounded by an implementation-defined constant.	No memory allocation.

Copyright 1999, 2000 J Consortium, All Rights Reserved

4. The CPU time and dynamic memory impact of the Baseline API libraries described in Section 4.0 (starting on page 103) are not constrained by this specification.

3.15 Core Class Loading API Overview

Note that the Core Execution Environment supports dynamic class loading only if it is combined with a Baseline Virtual Machine as part of an Extended Baseline Virtual Machine. The system integrator's API for customizing the core class loader shall consist of the following class declaration:

```
public class CoreClassLoader extends java.lang.Object {
   byte[ ] findClassBytes(java.lang.String name) throws ClassNotFoundException;
}
```

Note that CoreClassLoader is a Baseline component. The responsibility of the findClassBytes() method is to find the byte-code representation of the class named by its string argument and return this representation as an array of bytes. The default implementation of findClassBytes() searches the local file system for the requested class, using the CoreClassPath environment variable to guide its search order. To implement a different search or load strategy, the system integrator implements a class that extends CoreClassLoader and overrides findClassBytes() to provide whatever alternative behavior is desired. Whenever the core class loader needs to load a class, it locates the bytes that represent the class to be loaded by invoking the system integrator's findClassBytes() method.

See Section 4.0 for additional discussion on configuration of the core class loader.

3.16 C/Native API

3.16.1 Obtaining Access to Core Objects

coreRegistryLookup(). The coreRegistryLookup() function shall look up the core object that is stored in the core registry and identified by the specified name. The name argument to this function is a null-terminated array of bytes, according to the standard string conventions for the C programming language (See Reference 10). Since C characters are only 8 bits wide, and Java characters are 16 bits wide, the C string argument to this function is not able to describe all names that might be present in the CoreRegistry dictionary. When converting this string argument to a Java string for purposes of comparing with existing entries in the CoreRegistry dictionary, the coreRegistryLookup() function fills the eight high-order bits of each Java character with zeros. coreRegistryLookup() returns null if no such object is found in the registry. The internal organization of core objects shall be available through static tools, the capabilities of which are not constrained by this specification because they are implementation-defined. An example of such a tool is javah, by Sun Microsystems. The C prototype is shown below:

```
CoreObject *coreRegistryLookup(char name[]);
```

3.16.2 Understanding Core Resource Needs and Contention

maxCorePriority(). The maxCorePriority() function shall return the maximum system-level priority used by the real-time core tasks. The C prototype is shown below:

```
int maxCorePriority();
```

minCorePriority(). The minCorePriority() function shall return the minimum system-level priority used by the Core tasks. Note that maxCorePriority() - minCorePriority() might not equal 127, in case, for example, the core dispatcher uses green threads.The C prototype is shown below:

```
int minCorePriority();
```

corePriorityMap(). The corePriorityMap() method shall fill in the elements of the 128-entry integer array whose address is passed as its argument with values representing the system priorities to which each of the Core priority levels correspond. The first entry in this array is the system priority level at which Core priority-1 tasks execute. The second entry in this array is the system priority level at which Core priority-2 tasks execute, and so on. The C prototype is shown below:

```
void corePriorityMap(int map[ ]);
```

maxBaselinePriority(). The maxBaselinePriority() function shall return the maximum system-level priority used by the Baseline threads.The C prototype is shown below:

```
int maxBaselinePriority();
```

minBaselinePriority(). The minBaselinePriority() function shall return the minimum system-level priority used by the Baseline threads. Note that maxBaselinePriority() - minBaselinePriority() might not equal 9, in case, for example, the Baseline dispatcher uses an internal task dispatcher (green threads) rather than the dispatcher of the underlying real-time operating system. The C prototype is shown below:

```
int minBaselinePriority();
```

coreInterruptLevels(). The coreInterruptLevels() function shall return the number of interrupt priority levels that might be masked by Core tasks. The interrupt priority levels are assumed to begin with the lowest interrupt priority level. It may be the case that higher priority interrupts cannot be handled by Core tasks, as limited by the system configuration. Suppose, for example, that a particular target supports 16 interrupt priority levels, of which the highest 8 interrupt priority levels must be implemented in C (not the real-time core). In this case, coreInterruptLevels() shall return 8. The C prototype is shown below:

```
int coreInterruptLevels();
```

3.16.3 Synchronizing and Coordinating with the Baseline Domain

Note that the core API provides more semaphore operations than are provided to the C/Native programmer. It is intentional that the interface between the core and native worlds is small and simple.

semaphoreP(). The semaphoreP() function shall perform a semaphore P() operation on the Core object whose reference is passed as its argument. That Core object should be either a CountingSemaphore or a SignalingSemaphore. The semantics of this function depends on the type of its argument. If semaphore represents a SignalingSemaphore, then semaphoreP() represents a SignalingSemaphore.P() operation. If semaphore represents a

Copyright 1999, 2000 J Consortium, All Rights Reserved

CountingSemaphore, then semaphoreP() represents a CountingSemaphore.P() operation. If semaphore is neither, semaphoreP() shall return an error code (-1). Otherwise, semaphoreP() shall indicate normal termination by returning a status code of 0. The C prototype is shown below:

```
int semaphoreP(CoreObject *semaphore);
```

semaphoreV(). The semaphoreV() function performs a semaphore V() operation on the Core object whose reference is passed as its argument. That Core object should be either a CountingSemaphore or a SignalingSemaphore. The semantics of this function depends on the type of its argument. If semaphore represents a SignalingSemaphore, then semaphoreV() represents a SignalingSemaphore.V() operation. If semaphore represents a CountingSemaphore, then semaphoreV() represents a CountingSemaphore.V() operation. If semaphore is neither, semaphoreV() shall return an error code (-1). Otherwise, semaphoreV() shall indicate normal termination by returning a status code of 0. The C prototype is shown below:

```
int semaphoreV(CoreObject *semaphore);
```

semaphoreVall(). The semaphoreVall() function performs a semaphore Vall() operation on the core object passed as its argument. If the semaphore argument represents a SignalingSemaphore, semaphoreVall() shall perform a SignalingSemaphore.Vall() operation. If the semaphore argument does not represent a SignalingSemaphore, semaphoreVall() shall return an error code (-1). Otherwise, semaphoreVall() shall indicate normal termination by returning a status code of 0.

The implementation of semaphoreVall() shall be constant-time, allowing its use from within a time-constrained interrupt handler or other Atomic-Synchronized context. The work of waking up the various waiting tasks shall be distributed between the various P() operations that are waiting to be signaled.

The C prototype is shown below:

```
int semaphoreVall(CoreObject *semaphore);
```

enterSynchronized(). The enterSynchronized() function shall perform the equivalent of entering a synchronized context associated with its any_object argument. If any_object does not implement the PCP interface, this function shall block the current task until all other threads and tasks have released their locks on this object. If any_object implements the PCP interface, this function shall adjust the active priority of the current task according to implementation-defined conventions consistent with this Core Execution Environment. If any_object implements the Atomic interface, the C programmer should take care to ensure that the code that is executed following return from enterSynchronized() and preceding execution of the corresponding exitSynchronized() function is execution-time analyzable. This recommendation is not enforced. Failure to adhere to this recommendation may compromise the real-time integrity of the Core Execution Environment.

If the native execution environment supports the ability to abort or to otherwise interrupt the execution of native tasks, the implementation of enterSynchronized() shall be robust to this possibility. In other words, if a task becomes blocked during execution of enterSynchronized(), and that task is aborted before access to the requested region has

been granted, the Core Execution Environment's internal data structures shall be left in a coherent and consistent state.

Note that nesting of PCP-synchronized contexts is only allowed if the ceiling priorities associated with inner-nested contexts are strictly greater than the ceiling priorities of the outer-nested contexts. enterSynchronized() shall return an error code (-1) if the requested service cannot be provided because of illegal nesting of PCP-synchronized contexts. Otherwise, enterSynchronized() shall return a success code, represented by 0.

The C prototype is shown below:

```
int enterSynchronized(CoreObject *any_object);
```

exitSynchronized(). The exitSynchronized() function shall perform the equivalent of exiting a synchronized context associated with its any_object argument. Note that synchronization contexts may nest, and particular contexts may be entered multiple times. If a particular context has been entered multiple times, it must be exited the same number of times before this task releases exclusive access to the context. The Core Execution Environment shall maintain an internal counter recording how many times each synchronized context is entered, incrementing this counter for each execution of the context's enterSynchronized() function and decrementing this counter for each execution of the context's exitSynchronized() function.

If this execution of exitSynchronized() decrements the synchronized context entry count to zero, exitSynchronized() shall release exclusive access to this context. If any_object implements the PCP interface, releasing exclusive access consists of lowering the active priority of the current task. Otherwise, releasing exclusive access consists of releasing the lock associated with the context's controlling object.

enterSynchronized() shall return an error code (-1) if the requested service cannot be performed because the current task does not own exclusive access to the the context represented by any_object. Otherwise, exitSynchronized() shall return a success code, represented by 0.

The C prototype is shown below:

```
int exitSynchronized(CoreObject *any_object);
```

3.17 The Core API

This section describes the APIs that are used by developers of Core components. Unless specifically identified as Core-Baseline methods, all methods are presumed to be Core methods. Core methods are visible only to other Core components.

3.17.1 The CoreObject Class

CoreObject is the root of the core object hierarchy. CoreObject serves a purpose similar to java.lang.Object in the Baseline domain.

Note that the Baseline compiler sees org.rtjwg.CoreObject as extending from java.lang.Object. However, it is the responsibility of the Core programmer to avoid

Copyright 1999, 2000 J Consortium, All Rights Reserved

invoking any of the methods inherited from java.lang.Object that are not specifically identified in the Core specification as being supported by org.rtjwg.CoreObject. The Core Verifier shall reject as invalid any Core class file that makes reference to non-supported methods.

Though the typical Core programmer does not have to worry about such details, it is important to note that special tricks must be applied in order to author the implementation of CoreObject. In particular, certain methods of java.lang.Object are defined to be final, meaning that subclasses are not allowed to override their implementations. The implementation of CoreObject must override the getClass(), wait(), notify(), and notifyAll() methods, all of which are defined in java.lang.Object to be final. To work around this restriction, the Core programmer who implements CoreObject names methods _getClass(), _wait(), _notify(), and _notifyAll() methods, respectively. The Core class loader shall overwrite the implementations of getClass(), wait(), notify() and notifyAll() with the specially named replacements.

CoreObject Constructor. There shall be one constructor for the CoreObject class. The Core signature follows:

```
public CoreObject();
```

The following methods are supported for the CoreObject class.

CoreObject.clone(). The clone() method shall make a copy of this object, copied one level deep, and shall return a reference to the new copy. The Core signature is shown below:

final public protected *stackable* Object clone();

CoreObject.equals(). The equals() method shall return true if and only if object o is the same object as this object. (Note that subclasses can redefine the "meaning" of equals().) The Core signature is shown below:

public boolean *stackable* equals(*stackable* Object o);

CoreObject.getClass(). The getClass() method shall return a reference to the CoreClass object that represents this object's class information. The Core signature is shown below:

final public CoreClass *stackable* getClass();

CoreObject.hashCode(). The hashCode() method shall return an integer that represents the hash code associated with this object. The Core signature is shown below:

public int *stackable* hashCode();

CoreObject.notify(). The notify() method shall wake up the CoreTask task that has the highest priority among tasks waiting on this object's condition and has been waiting the longest amount of time if multiple tasks of the same highest priority are associated with this same monitor. If no objects are waiting on this condition, the notify() method shall have no effect on the state of this object's monitor. If the object for which the notify() method is invoked implements the PCP interface or if the currently executing task does not own exclusive access to the corresponding object's monitor, this method throws a

previously allocated CoreIllegalMonitorStateException exception. (Since CoreIllegalMonitorStateException is a subclass of CoreRuntimeException, this exception does not appear in the method's signature.) The Core signature is shown below:

```
final public void stackable notify();
```

CoreObject.notifyAll(). The notifyAll() method wakes up all CoreTask objects that are waiting for the condition associated with this monitor to be signaled. If the object for which the notifyAll() method is invoked implements the PCP interface or if the currently executing task does not own exclusive access to the corresponding object's monitor, this method throws a previously allocated CoreIllegalMonitorStateException exception. (Since CoreIllegalMonitorStateException is a subclass of CoreRuntimeException, this exception does not appear in the method's signature.) The Core signature is shown below:

```
final public void stackable notifyAll();
```

CoreObject.toString(). The toString() method shall return a reference to a CoreString object, allocated in the currently active allocation context, that provides an abstract implementation-defined textual representation of this object. The Core signature is shown below:

```
public CoreString stackable toString();
```

CoreObject.wait(). The wait() method shall cause the currently executing core task to be put to sleep until this task is the highest priority task on the monitor queue and some other Core task invokes this object's notify() method or until some other Core task invokes the notifyAll() method. If the object for which the wait() method is invoked implements the PCP interface or if the currently executing task already owns exclusive access to some PCP object's monitor, this method shall throw a previously allocated CoreIllegalMonitorStateException exception. (Since CoreIllegalMonitorStateException is a subclass of CoreRuntimeException, this exception does not appear in the method's signature.) The Core signature is shown below:

```
final public void wait();
```

CoreObject.arrayAddress(). The arrayAddress() method shall return the address of this primitive array if this object is a Core array of primitive type. Otherwise, this method shall throw a previously allocated instance of CoreObjectNotAddressableException. Note that this method shall return the address of the first element of the array rather than the start address of the object that contains the array elements. The Core Execution Environment shall represent arrays of primitive elements using whatever convention is followed by the dominant C compilers supporting the given architecture. The Core signature is shown below:

```
final public long stackable arrayAddress() throws CoreObjectNotAddressableException;
```

CoreObject.sizeof(). The sizeof() method shall return the number of bytes used to represent this object, including any alignment padding and bookkeeping fields inserted for the benefit of garbage collection. The Core signature is shown below:

```
final public int stackable sizeof();
```


Copyright 1999, 2000 J Consortium, All Rights Reserved

3.17.2 The CoreThrowable Class

The org.rtjwg.CoreThrowable class is the Core Execution Environment's analog of java.lang.Throwable. Every reference to java.lang.Throwable shall be replaced with a reference to CoreThrowable by the Core Class Loader. Within the Core Execution Environment, all exceptions thrown and caught must extend from org.rtjwg.CoreThrowable.

Unlike its java.lang.Throwable analog, the CoreThrowable class shall not maintain a representation of the run-time stack backtrace.

CoreThrowable Constructors. There shall be two constructors for the CoreThrowable class. The first takes no arguments and shall create a CoreThrowable object with no particular message. The second shall take a single CoreString argument, which represents the message to be associated with this CoreThrowable object, and shall create a CoreThrowable object which maintains a reference to its message argument. The Core signatures are as follows:

```
public CoreThrowable();
public CoreThrowable(CoreString message);
```

CoreThrowable.getMessage(). The getMessage() method returns a reference to the CoreString object that was passed as an argument to the CoreThrowable constructor, or returns null if this CoreThrowable object was constructed with no message.

3.17.3 The CoreRuntimeException Class

The org.rtjwg.CoreRuntimeException class, which extends org.rtjwg.CoreThrowable, is the Core analog of java.lang.RuntimeException. Every reference to java.lang.RuntimeException shall be replaced with a reference to CoreRuntimeException by the Core Class Loader. Within the Core Execution Environment, CoreRuntimeException represents exceptional events that are not expected to occur. A method that throws a CoreRuntimeException object shall not be required by the Core Compiler (or by the Java Compiler) to declare in its signature that it throws CoreRuntimeException. A context that invokes a method that might throw a CoreRuntimeException object shall not be required to catch the CoreRuntimeException object or to declare that the context might throw the CoreRuntimeException object. In the common vernacular, the CoreRuntimeException class represents "unchecked" exceptions.

CoreRuntimeException Constructors. There are two constructors for the CoreRuntimeException class. The first shall take no arguments and shall create a CoreRuntimeException object with no particular message. The second shall take a single CoreString argument, which represents the message to be associated with this CoreRuntimeException object and shall create a CoreRuntimeException object that maintains a reference to its message argument. The Core signatures are as follows:

```
public CoreRuntimeException();
public CoreRuntimeException(CoreString message);
```

3.17.4 The CoreException Class

The org.rtjwg.CoreException class, which extends org.rtjwg.CoreThrowable, is the Core analog of java.lang.Exception. Every reference to java.lang.Exception shall be replaced with a reference to CoreException by the Core Class Loader. Within the Core Execution Envi-

ronment, CoreException represents exceptional events that are not expected to occur. A method that throws a CoreException object shall be required by the Core Compiler (and by the Baseline Compiler) to declare in its signature that it throws CoreException. A context that invokes a method that might throw a CoreException object shall be required either to catch the CoreException object or to declare that the context might throw the CoreException object. In the common vernacular, the CoreException class represents a "checked" exception.

CoreException Constructors. There shall be two constructors for the CoreException class. The first shall takes no arguments and shall create a CoreException object with no particular message. The second shall take a single CoreString argument, which represents the message to be associated with this CoreException object, and shall create a CoreException object that maintains a reference to its message argument. The Core signatures are as follows:

```
public CoreException();
public CoreException(CoreString message);
```

3.17.5 The ScopedException Class

The org.rtjwg.ScopedException class extends org.rtjwg.CoreThrowable. A ScopedException object is special in that when thrown, it is only catchable by catch clauses belong to the method within which the ScopedThrowable object was most recently enabled. When the object is constructed, it is automatically enabled in the context that invoked the constructor.

ScopedException Constructors. There are two constructors for the ScopedException class. The first shall take no arguments and shall create a ScopedException object with no particular message. The second shall take a single CoreString argument, which represents the message to be associated with this ScopedException object and shall create a ScopedException object that maintains a reference to its message argument. The Core signatures are as follows:

```
public ScopedException();
public ScopedException(CoreString message);
```

ScopedException.enable(). The enable() method establishes the context of the calling method as the only method that can catch this exception. If a ScopedException is enabled multiple times, the most recent enable() invocation is the one that establishes the catching context. The Core signature follows:

```
public final void enable();
```

ScopedException.disable(). The disable() method disables this ScopedException. If an ATCEventHandler attempts to throw a disabled ScopedException, the effect is to simply return from the ATCEventHandler, causing the asynchronously signaled CoreTask to resume execution as if it had never been signaled. If a disabled ScopedException is thrown from a normal CoreTask execution context (rather than from within an ATCEventHandler), the exception shall not be caught and shall cause the CoreTask's work() method to abort execution. The Core signature follows:

```
public final void disable();
```


Copyright 1999, 2000 J Consortium, All Rights Reserved

3.17.6 The CoreClass Class

The CoreClass class extends CoreObject. Its role is similar to java.lang.Class.

CoreClass.forName(). The forName() method shall return the CoreClass object associated with the class or interface known by the string name supplied as its argument if the named class was previously loaded. The forName() method shall not cause the class to be loaded. If the class is not currently loaded, this method shall throw a previously allocated instance of CoreClassNotFoundException. The Core signature is shown below:

```
public static CoreClass forName(CoreString className)
                throws CoreClassNotFoundException;
```

CoreClass.getComponentType(). The getComponentType() method shall return the CoreClass object that represents the component type of this object, which is presumed to be an array. If this object is not an array, getComponentType() shall return null. The Core signature is shown below:

```
final public CoreClass getComponentType();
```

CoreClass.isArray(). The isArray() method shall return true if and only if this CoreClass object represents an array class. The Core signature is shown below:

```
final public boolean isArray();
```

CoreClass.isAssignableFrom(). The isAssignableFrom() method shall return true if and only if the class or interface represented by this CoreClass object is either the same as, or is a superclass or superinterface of, the class or interface represented by the supplied CoreClass parameter. The Core signature is shown below:

```
final public boolean isAssignableFrom(CoreClass cls);
```

CoreClass.isInstance(). The isInstance() method shall return true if and only if its obj argument represents an instance of the class or interface represented by this CoreClass. If this CoreClass represents an array type, isInstance() shall return true if and only if obj is or can be coerced to be of the array's type. If this CoreClass represents a primitive type, isInstance() shall return false. The Core signature is shown below:

```
final public boolean isInstance(CoreObject obj);
```

CoreClass.isInterface(). The isInterface() method shall return true if and only if this CoreClass object represents an interface type. The Core signature is shown below:

```
final public boolean isInterface();
```

CoreClass.isPrimitive(). The isPrimitive() method shall return true if and only if this CoreClass object represents a primitive type. The Core signature is shown below:

```
final public boolean isPrimitive();
```

CoreClass.newInstance(). The newInstance() method shall create a new instance of the class represented by this CoreClass object. The Core signature is shown below:

```
final public CoreObject newInstance();
```

CoreClass.toString(). The toString() method shall return an implementation-defined CoreString textual representation of this CoreClass object. The CoreString object returned from the toString() method shall be allocated in the current allocation context. The Core signature is shown below:

```
final public CoreString toString();
```

CoreClass.verification(). The verification() method shall return true if and only if this particular Core Execution Environment performs verification of loaded class files. If this method returns true, the verification performed by the Core class loader shall conform to the specification of the Core Verifier (See Section 3.5.1). The Core signature is shown below:

```
final public static boolean verification();
```

CoreClass.loadClass(). The loadClass() method shall load and fully resolve the class named by its CoreString argument, throwing a previously allocated instance of CoreClassNotFoundException if this class, or any of the classes it makes reference to cannot be found. This method is omitted from the Static Core Execution Environment and the Core Static Linker issues an appropriate error message if any of the Core application code that it is linking attempts to invoke this method. The loadClass() method shall throw a previously allocated instance of CoreClassFormatError if this particular Core Execution Environment claims to perform verification of newly loaded classes (See "CoreClass.verification()" on page 66) and the requested class, or any of the classes it makes reference to, fails byte-code verification as performed by the Core Verifier. The Core signature is shown below:

```
final public static CoreClass loadClass(CoreString class_name)
        throws CoreClassNotFoundException, CoreClassFormatError;
```

CoreClass.unloadClass(). The unloadClass() method shall remove this class from the set of loaded classes and shall reclaim the memory used to represent this class, throwing a previously allocated instance of CoreClassInUseException if there exist instances of this class, or if other loaded classes make reference to this class. This method is omitted from the Static Core Execution Environment and the Core Static Linker issues an appropriate error message if any of the Core application code that it is linking attempts to invoke this method. The Core signature is shown below:

```
final public unloadClass() throws CoreClassInUseException;
```

3.17.7 The CoreArray Class

The CoreArray class, which represents arrays within the Core Execution Environment, extends CoreObject. All uses of special array syntax within Core source code shall be treated within the Core Execution Environment as special CoreArray (or derivative) objects. This means that the Core Execution Environment allows the subscript operation to be performed on an object of type CoreArray. Further, it means that a new operation that allocates an array within the Core Execution Environment produces a CoreArray object. All of CoreBoolArray, CoreByteArray, CoreShortArray, CoreCharArray, CoreIntArray, CoreLongArray, CoreFloatArray, CoreDoubleArray, and CoreRefArray extend CoreArray.

Copyright 1999, 2000 J Consortium, All Rights Reserved

If the Baseline environment obtains a reference to a core array object, the Baseline envi-

TABLE 3. Core Array Representation Within Baseline Domain

Core Type	Baseline Type	Core-Baseline Methods
Array of boolean	CoreBoolArray	*baseline* public final int length(); *baseline* public final boolean atGet(int index) throws CoreArrayIndexOutOfBoundsException; *baseline* public final void atPut(int index, boolean b) throws CoreArrayIndexOutOfBoundsException;
Array of byte	CoreByteArray	*baseline* public final int length(); *baseline* public final byte atGet(int index) throws CoreArrayIndexOutOfBoundsException; *baseline* public final void atPut(int index, byte b) throws CoreArrayIndexOutOfBoundsException;
Array of short	CoreShortArray	*baseline* public final int length(); *baseline* public final short atGet(int index) throws CoreArrayIndexOutOfBoundsException; *baseline* public final void atPut(int index, short s) throws CoreArrayIndexOutOfBoundsException;
Array of char	CoreCharArray	*baseline* public final int length(); *baseline* public final char atGet(int index) throws CoreArrayIndexOutOfBoundsException *baseline* public final void atPut(int index, char c) throws CoreArrayIndexOutOfBoundsException;
Array of int	CoreIntArray	*baseline* public final int length(); *baseline* public final int atGet(int index); throws CoreArrayIndexOutOfBoundsException *baseline* public final void atPut(int index, int i) throws CoreArrayIndexOutOfBoundsException;
Array of long	CoreLongArray	*baseline* public final int length(); *baseline* public final long atGet(int index) throws CoreArrayIndexOutOfBoundsException; *baseline* public final void atPut(int index, long x) throws CoreArrayIndexOutOfBoundsException;
Array of float	CoreFloatArray	*baseline* public final int length(); *baseline* public final float atGet(int index) throws CoreArrayIndexOutOfBoundsException; *baseline* public final void atPut(int index, float f) throws CoreArrayIndexOutOfBoundsException;

TABLE 3. Core Array Representation Within Baseline Domain

Core Type	Baseline Type	Core-Baseline Methods
Array of double	CoreDoubleArray	*baseline* public final int length(); *baseline* public final double atGet(int index) throws CoreArrayIndexOutOfBoundsException; *baseline* public final void atPut(int index, double d) throws CoreArrayIndexOutOfBoundsException;
Array of any core reference type	CoreRefArray	*baseline* public final int length(); *baseline* public final CoreObject atGet(int index) throws CoreArrayIndexOutOfBoundsException;

ronment sees this Core array as one of the nine types identified in the second column of Table 3 on page 67. From within the Baseline domain, this Core object does not look like a Baseline array. In other words, the Baseline domain is not allowed to access the data contained within this object using Baseline subscripting operations. Instead, the Baseline domain is required to access the data contained within the Core array by invoking the Core-Baseline methods described in the third column of this table. The significance of these methods is described below.

length(). This method shall return the number of elements in the corresponding Core array object.

atGet(). This method shall return the array element at the specified index position from within the corresponding core array object, or shall throw a previously allocated instance of CoreArraySubscriptOutOfBoundsException if the requested index position is out of range for the given array.

atPut(). This method shall overwrite the array element at the specified index position within the corresponding Core array object with the value supplied as the method's second argument, or shall throw a previously allocated instance of CoreArraySubscriptOutOfBoundsException if the requested index position is out of range for the given array. Note that CoreRefArray does not implement the atPut() method. This is intentional. The reason for this omission is that the Baseline domain is not allowed to overwrite reference fields of Core objects.

3.17.8 The AllocationContext Class

The AllocationContext class extends CoreObject. Every Core object is allocated within a particular allocation context, represented abstractly by the AllocationContext class. Associated with every Core task is a dedicated AllocationContext object which serves as the tasks' default allocation context. This means that by default, every new object is allocated within the allocation context that represents the task's default allocation context.

There are no public interfaces to allow Core components to directly manipulate the allocation context of a core task. When a Core task completes its execution, the allocation context is automatically released, making all of the objects allocated by that Core task eligible for garbage collection. The precise moment at which a Core task is considered to have completed its execution depends on what type of task it is:

Copyright 1999, 2000 J Consortium, All Rights Reserved

1. If this is an ISR_Task or SporadicTask, the task is not considered to have “completed” execution until after its stop() method has been invoked and that method has executed to completion. This is the only way for one of these kinds of tasks to complete execution.
2. Otherwise, this must be a CoreTask. There are several ways for a CoreTask (which is not one of the above three subclasses) to complete execution:
   a. It may return from its work() method.
   b. It may throw an uncaught exception, including the special exception returned from its abortWorkException() method.
   c. The task’s stop() method may be invoked, in which case the task is considered to have completed execution upon return from the stop() method invocation.

AllocationContext Constructors. There are three constructors for AllocationContext.

1. The first shall take no arguments and shall create an AllocationContext object that is configured with no particular bound on how much memory might be allocated from within that context. The location of the allocation region within memory shall be determined by the Core Execution Environment in an implementation-defined manner. The Core signature for this constructor follows:

   ```
   public AllocationContext();
   ```
2. The second constructor shall take an argument identifying the maximum total number of bytes authorized to be allocated within the corresponding allocation region and shall create an AllocationContext object that is configured to allocate no more than the specified number of bytes. When this form of constructor is used, the allocation region is required to be contiguous memory, and the Core Execution Environment shall use a constant-time allocation algorithm which simply increments or decrements a region-specific allocation pointer by the size of each allocation request. The location of the allocation region within memory shall be determined by the Core Execution Environment in an implementation-defined manner. This constructor shall throw a previously allocated instance of CoreOutOfMemoryException if there is not a large enough region of contiguous memory to satisfy the request. The Core signature for this constructor follows:

   ```
   public AllocationContext(long maximum_bytes)
       throws CoreOutOfMemoryException;
   ```
3. The third constructor shall take an argument identifying the maximum total number of bytes authorized to be allocated within the corresponding allocation region and a second CoreString argument identifying the name of the special memory block within which the allocation region is to be allocated. This constructor shall create an AllocationContext object that is configured to allocate no more than the specified number of bytes from within the specified memory block. When this form of constructor is used, the allocation region is required to be contiguous memory, and the Core Execution Environment shall use a constant-time allocation algorithm which simply increments or decrements a region-specific allocation pointer by the size of each allocation request. The idea is that in particular configurations, special names might be given to memory blocks representing fast static memory, dual-ported memory, or non-volatile battery powered RAM. The naming conventions for individual memory blocks shall be implementation-defined. This constructor shall throw a previously allocated instance of CoreOutOfMemoryException if there is not a

large enough region of contiguous memory within the requested memory block to satisfy the request. The Core signature for this constructor follows:

```
public AllocationContext(long maximum_bytes, CoreString block_name)
    throws CoreOutOfMemoryException;
```

AllocationContext.available(). If this AllocationContext was constructed with an argument specifying the maximum number of bytes to be allocated, the available() method shall return the number of bytes that are currently available to be allocated within this AllocationContext. If no limit was specified when the AllocationContext was constructed, the available() method shall return the special code of -1. The Core signature is shown below:

```
final public long available();
```

AllocationContext.allocated(). The allocated() method shall return the total number of bytes, including alignment padding and bookkeeping header information associated with allocated objects, that have been allocated within this AllocationContext. The Core signature is shown below:

```
final public long allocated();
```

AllocationContext.release(). Core components invoke an AllocationContext's release() method to indicate that all of the objects allocated within that context, including the AllocationContext object itself, are now eligible for garbage collection. The memory dedicated to these objects shall not be reclaimed until after the Core Execution Environment verifies that the respective objects are no longer visible to the Baseline domain. The Core signature for the release() method is shown below:

```
final public void release();
```

3.17.9 The SpecialAllocation Class

The SpecialAllocation class extends CoreObject. By default, all new objects shall be allocated within the default allocation context of the currently executing CoreTask. To allocate objects within some other allocation context, Core programmers extend the abstract SpecialAllocation class by implementing the run() and context() methods. Core tasks invoke the SpecialAllocation.execute() method to establish a new allocation context. Since SpecialAllocation is an abstract class, there are no constructors.

SpecialAllocation.context(). Implementations of the abstract context() method shall return a reference to the AllocationContext object that represents the special allocation context established to keep track of all objects allocated during execution of this SpecialAllocation object's execute() method, excluding any objects that might be allocated during execution of other SpecialAllocation object's inner-nested execute() methods. To use special allocation contexts, Core programmers must implement the context() method to return a reference to the appropriate AllocationContext object. The Core signature is:

```
public abstract AllocationContext context();
```

SpecialAllocation.run(). This is an abstract method, which is invoked during execution of the execute() method. To use special allocation contexts, Core programmers must imple-

Copyright 1999, 2000 J Consortium, All Rights Reserved

ment the run() method, providing the body of code that is to execute within the new allocation context. The Core signature is shown below:

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

SpecialAllocation.execute(). The execute() method is invoked to enter into the special allocation context. The effect of calling execute() shall be to (1) establish the new allocation context to be the AllocationContext whose reference is returned from the context() method, (2) invoke this object's run() method, and (3) restore the original allocation context upon return (or thrown exception) from the run() method. The Core signature follows:

```
final public void execute();
```

3.17.10 The PCP Interface

The PCP interface represents the intent to use the priority ceiling protocol for synchronization. If a core object implements this interface, the Core Execution Environment shall use the modified priority ceiling protocol defined here for all synchronization associated with that object. In particular:

1. If some task running at a priority higher than a particular PCP object's ceiling priority attempts to synchronize on that object, the synchronization attempt shall fail by throwing a previously allocated instance of CoreIllegalMonitorStateException.
2. For any class that implements the PCP interface, it is improper to invoke the wait(), notify(), or notifyAll() methods of that class's instances. Any attempt to invoke these methods shall fail by throwing a previously allocated instance of CoreIllegalMonitorStateException.
3. Obtaining a synchronization lock (whether it is a PCP object or a priority inheritance object) for a Core object shall not require allocation of memory.
4. When a task is executing with possession of a PCP object's synchronization lock, the Core task shall run at the corresponding PCP object's ceiling priority.
5. No queues shall be used in the implementation of a priority ceiling lock.
6. PCP synchronization shall not cause the currently running task to block.
7. No time slicing of tasks at equal or lower priority shall be allowed while the running task holds a priority ceiling lock.
8. Blocking I/O and synchronizing operations shall not be permitted while the current task holds a PCP synchronization lock. Any core service invoked from within a PCP-synchronized context that might block shall not perform the requested operation and shall instead throw a previously allocated instance of CoreIllegalMonitorStateException exception. Examples of methods that shall automatically throw CoreIllegalMonitorStateException if invoked from within a PCP-locked context include CoreTask.sleep(), CoreTask.sleepUntil(), CoreTask.join(), Mutex.lock(), SignalingSemaphore.P(), CountingSemaphore.P(), CoreObject.wait(), and entry into a synchronized context that is not identified as PCP.
9. Static and dynamic nesting of priority ceiling locks shall be permitted. However, entry into an inner-nested PCP-locked context shall only be allowed if the priority ceiling associated with the inner context is greater than the active priority of the currently executing task. Otherwise, entry into the inner-nested PCP-locked context

shall be denied by throwing a previously allocated instance of CoreIllegalMonitorStateException.

10. For PCP objects, third-party synchronization shall be prohibited. In other words, the code fragment:

```
synchronized (o) {
  doSomething();
}
```

represents an inappropriate request within the Core Execution Environment unless object o happens to equal this. If object o does not equal this, attempted execution of the above statement results in throwing of a previously allocated instance of CoreIllegalMonitorStateException.

11. The Core Execution Environment shall give special handling to the construction of objects that implement the PCP interface. Whenever a PCP object is constructed, the Core Execution Environment shall invoke the object's ceilingPriority() method to determine the intended ceiling priority for the object. If ceilingPriority() returns an interrupt-level priority but the corresponding object does not implement Atomic (See Section 3.17.11), the constructor shall fail by throwing a previously allocated instance of CoreIllegalMonitorStateException.

The methods supported by the PCP class follow:

PCP.ceilingPriority(). The ceilingPriority() method is the only method defined in the PCP interface. It shall return the priority which is the intended ceiling priority for this Core object. The Core Execution Environment shall invoke this method only once for each instantiated object that implements the PCP interface. If the return value is -1, this indicates that the corresponding object is never used for locking and therefore does not require memory to be allocated to represent a locking mechanism. If a particular PCP object identifies itself as not implementing a lock, and subsequently some Core component attempts to synchronize on that object, the synchronization attempt shall fail by throwing a previously allocated instance of CoreIllegalMonitorStateException. The Core signature is shown below:

```
abstract int stackable ceilingPriority();
```

3.17.11 The Atomic Interface

The Atomic interface is used to distinguish PCP objects that adhere to special restrictions and provide special semantics. The Atomic interface shall extend the PCP interface. There shall be no public variables or methods defined for this interface. Rather, use of this interface is simply an indication to the Core class loader that certain objects deserve special treatment. The special treatment given to Atomic objects shall be as follows:

1. Only objects that implement the Atomic interface shall be allowed to set their priority ceiling to an interrupt-level priority. This has the effect of assuring that system interrupts shall not be disabled for arbitrarily long periods of time.
2. Each of the bodies of code that comprise the synchronized statements associated with an Atomic object shall be execution-time analyzable. The definition of execution-time analyzable code is provided in Section 3.14.

Copyright 1999, 2000 J Consortium, All Rights Reserved

3. If a task is executing synchronized code of an Atomic object ("Atomic synchronized code") when a request to abort the task is delivered to the Core Execution Environment, the Core Execution Environment shall defer abortion of the task until after the synchronized code completes its execution.

3.17.12 The CoreString Class

The CoreString class shall extend org.rtjwg.CoreObject. The CoreString class shall be used to represent string literal constants within Core components.

CoreString Constructors. There shall be two constructors for CoreString. The first shall accept as its argument an array of characters and shall produce a CoreString object representing that sequence of characters. The second shall accept as its arguments an array of characters (value), an integer offset within the array (offset), and an integer length field (length). It shall produce a CoreString object containing length characters copied from the value array starting with the character at position offset. This second constructor shall throw a previously allocated instance of CoreArrayIndexOutOfBoundsException if offset is negative or if the sum of the offset and length parameters exceeds the length of the value character array. The Core signatures for the two constructors are shown below:

```
public CoreString(char[] value);
public CoreString(char[] value, int offset, int length) throws
        CoreArrayIndexOutOfBoundsException;
```

CoreString.charAt(). The charAt() method shall return the character found at the specified position (index) within this CoreString object. An index value of zero shall correspond to the first character in the string. If the requested position is negative, or if it exceeds the length of the string, the charAt() method shall throw a previously allocated instance of CoreArrayIndexOutOfBoundsException. The Core-Baseline _charAt() method shall behave the same as the core charAt() method, except that it is intended to be invoked from a Baseline thread. The Core signatures are shown below:

```
final public char charAt(int index) throws CoreArrayIndexOutOfBoundsException;
final public baseline char _charAt(int index) throws CoreArrayIndexOutOfBoundsException;
```

CoreString.hashCode(). The hashcode() method shall return an integer value that corresponds to the sequence of characters represented by this CoreString object. If two CoreString objects represent the same sequence of characters, their respective hash codes shall be the same. The Core-Baseline _hashCode() method shall behave the same as the hashCode() method, except it is intended to be invoked from a Baseline thread. The Core signatures are shown below:

```
final public int hashCode();
final public baseline int _hashCode();
```

CoreString.equals(). The equals() method shall return true if and only if its CoreString argument represents the exact same sequence of characters as this string. Its Core signature is shown below:

```
final public boolean equals(CoreString s);
```

CoreString.length(). The length() method shall return the number of characters in this CoreString object. The Core-Baseline _length() method shall behave the same as the length() method, except it is intended to be invoked from a Baseline thread. The Core signatures are shown below:

```
final public int length();
final public baseline int _length();
```

3.17.13 The DynamicCoreString Class

The DynamicCoreString class shall extend CoreString. This class has considerably more functionality than CoreString.

DynamicCoreString Constructors. There are five constructors, with signatures as shown below, for DynamicCoreString. The first takes no arguments and shall construct a DynamicCoreString object of length zero. The second takes as its argument an array of bytes and shall construct a DynamicCoreString object with as many characters as the length of the byte array, with each byte converted into the appropriate Unicode character in sequence within the resulting DynamicCoreString object. The meaning of the bytes stored in the byte array shall be interpreted according to ASCII conventions. The third constructor is like the second, except the character sequence for the DynamicCoreString is taken from the byte array starting with the byte at index position offset and ending with the byte at index position (offset + length - 1). This constructor shall throw a previously allocated instance of CoreArrayIndexOutOfBoundsException if its offset or length arguments are negative or if (offset + length) is greater than the length of the array. The fourth and fifth constructors are like the second and third constructors respectively, except the input arrays shall hold Unicode characters instead of ASCII bytes.

The Core signatures for the five constructors are shown below:

```
public DynamicCoreString();
public DynamicCoreString(byte[] bytes);
public DynamicCoreString(byte[] bytes, int offset, int length)
            throws CoreArrayIndexOutOfBoundsException;
public DynamicCoreString(char[] chars);
public DynamicCoreString(char[] value, int offset, int length)
          throws CoreArrayIndexOutOfBoundsException;
```

DynamicCoreString.concat(). The concat() method shall create and return a new DynamicCoreString object that represents the concatenation of this string with the string supplied as its str argument. The Core signature is shown below:

```
final public DynamicCoreString concat(CoreString str);
```

DynamicCoreString.getChars(). The getChars() method shall copy the sequence of characters found within this string starting at index position source_begin and ending at index position source_end into the character array named destination starting at index position destination_begin. This method shall throw a previously allocated instance of CoreArrayIndexOutOfBoundsException if source_begin is less than 0, if source_end is greater than the length of this string, if source_end is less than source_begin, if destination_begin is less than zero, or if the destination array is not long enough to represent all of the characters

Copyright 1999, 2000 J Consortium, All Rights Reserved

to be copied into the array starting from index position destination_begin. The Core signature is shown below:

```
final public void getChars(int source_begin, int source_end,
                           char[] destination, int destination_begin)
        throws CoreArrayIndexOutOfBoundsException;
```

DynamicCoreString.substring(). The substring() method shall create a new DynamicCoreString representing the sequence of characters from this DynamicCoreString starting at index position begin_index and ending at index position end_index. This method shall throw a previously allocated instance of CoreArrayIndexOutOfBoundsException if begin_index is less than zero, end_index is less than begin_index, or end_index is greater than the length of this DynamicCoreString. The Core signature is shown below:

```
final public DynamicCoreString substring(int begin_index, int end_index)
        throws CoreArrayIndexOutOfBoundsException;
```

DynamicCoreString.toCharArray(). The toCharArray() method shall create a new character array of the same length as this DynamicCoreString object and initialize the elements of the character array by copying the characters from this DynamicCoreString object in sequential order. The Core signature is shown below:

```
final public char[] toCharArray();
```

DynamicCoreString.toLowerCase(). The toLowerCase() method shall create a new DynamicCoreString object of the same length as this DynamicCoreString and shall initialize the characters of the new DynamicCoreString by copying the characters of this DynamicCoreString object in sequential order, replacing each upper case character with the corresponding lower case character during the copying process. The definition of which character encodings are considered to be upper case and which are lower case, and the mapping between the two is defined by Unicode conventions. The Core signature is shown below:

```
final public DynamicCoreString toLowerCase();
```

DynamicCoreString.toUpperCase(). The toUpperCase() method shall create a new DynamicCoreString object of the same length as this DynamicCoreString and initialize the characters of the new DynamicCoreString by copying the characters of this DynamicCoreString object in sequential order, replacing each lower case character with the corresponding upper case character during the copying process. The definition of which character encodings are considered to be upper case and which are lower case, and the mapping between the two is defined by Unicode conventions. The Core signature is shown below:

```
final public DynamicCoreString toUpperCase();
```

3.17.14 The ATCEventHandler class

The ATCEventHandler class shall extend org.rtjwg.CoreObject. This class represents the main entry point for asynchronous transfer of control event handlers. Each CoreTask for which asynchronous event handling is enabled shall have an associated ATCEventHandler object. When an asynchronous event is signaled to that task, the Core Execution Environment shall invoke the corresponding ATCEventHandler's handleATCEvent() method.

ATCEventHandler Constructor. The constructor for ATCEventHandler shall take no arguments. The Core signature follows:

```
public ATCEventHandler();
```

ATCEventHandler.handleATCEvent(). The handleATCEvent() method shall invoke the defaultAction() method of its ATCEvent argument e and then return. The handleATCEvent() method is declared to throw a CoreThrowable object because in many cases, the desired result of asynchronous event handling is to abort a particular section of code by throwing an exception from within the asynchronous event handler. The Core signature follows:

```
public void handleATCEvent(ATCEvent e) throws CoreThrowable;
```

Note that application developers may override this method to implement different semantics for the asynchronous event handlers associated with particular Core tasks.

3.17.15 The ATCEvent class

The ATCEvent class shall extend org.rtjwg.CoreObject. This class represents an asynchronous event. To signal an asynchronous event to a Core task t, construct an ATCEvent object e and pass this ATCEvent e as the sole argument to t's signalAsync() method.

ATCEvent Constructor. The constructor for ATCEvent shall take no arguments. The Core signature follows:

```
public ATCEvent();
```

ATCEvent.defaultAction(). The defaultAction() method shall perform no side effects and shall simply return. The defaultAction() method is declared to throw a CoreThrowable object because in many cases, the desired result of asynchronous event handling is to abort a particular section of code by throwing an exception from within the asynchronous event handler. The Core signature is shown below:

```
public void defaultAction() throws CoreThrowable();
```

Note that application developers may override this method to implement different semantics for particular asynchronous event objects.

3.17.16 The CoreRegistry class

The CoreRegistry class shall extend org.rtjwg.CoreObject. The role of this class is to provide a repository for configuration information and for information that is shared between the core domain and the native and Baseline domains. There are no public constructors, since all methods are static and there are no instance variables.

CoreRegistry.stackAllocation(). The stackAllocation() method shall return true if and only if this Core Execution Environment supports stack allocation of objects. Otherwise, it shall return false. All Core Execution Environments that claim to support stack allocation shall behave the same with regards to which objects are stack allocated. The Core signature for this method follows:

```
public static boolean stackAllocation();
```


Copyright 1999, 2000 J Consortium, All Rights Reserved

CoreRegistry.registerStackable(). For each Core class, the registerStackable() method shall be invoked as the first executable code within any method that desires to identify any of its local variables (including incoming arguments and this) as potentially stack allocatable. The string argument to registerStackable() is a list of the names of the arguments and local variables whose referents shall be allocated on the stack if the Core Execution Environment supports stack allocation of locals and arguments. The variable names are separated by semicolons. In the case that a constructor has stackable arguments or local variables, and the constructor invokes its super-class constructor, the invocation of registerStackable() shall come immediately following the invocation of the super-class constructor. In order to identify local variables and arguments by original source code name in the class file representation, the Core class file for any class that contains invocations of the CoreRegistry.registerStackable() method shall contain the symbolic information that is produced by common Baseline compilers when the debug flags are enabled. The Core signature is shown below:

```
public static void registerStackable(stackable CoreString s);
```

CoreRegistry.registerBaseline(). The registerBaseline() method shall be invoked as the second line of executable code within the static initializer associated with a CoreClass if the CoreClass has any methods to identify as Core-Baseline methods (meaning the methods can be invoked from the Baseline domain). The first executable line of the static initializer shall be the invocation of registerCoreClass(). The Core signature follows:

```
public static void registerBaseline(CoreString methods);
```

The methods argument identifies the Core-Baseline methods by listing the name and signature of each method, separating each method's description from the others with a semicolon. For notational convenience, method signatures can be wildcarded using the asterisk character (*). For example, the method represented by the signature "foo(IF)V" can be abbreviated as "foo*". Note that "*" represents only the signature. It does not stand in place of any text from the method's name.

CoreRegistry.registerCoreClass(). The registerCoreClass() method shall be invoked as the first executable code in the static initializer for a class that intends to be loaded as a Core Class File. The presence or absence of this method's invocation within the static initializer of the class is the key indicator of whether this class is intended for the Baseline domain or for the Core domain. The Core signature is as follows:

```
public static void registerCoreClass();
```

CoreRegistry.coerce(). Given that the Core programmer might be dealing with objects that extend from CoreObject but which look to the Baseline compiler like they extend from java.lang.Object, the Core programmer can coerce such objects to CoreObject by invoking the static coerce() method of org.rtjwg.CoreRegistry. The Core signature follows:

```
public static CoreObject coerce(java.lang.Object o);
```

Typical usage is to further coerce the result returned from the coerce() method to the type that the Core programmer really expects this object to be. Consider, as an example, the following code fragment:

```
try {
   doSomething();
} catch (java.lang.Exception x) {
   MyCoreException cx;
   cx = (MyCoreException) CoreRegistry.coerce(x);
   cx.handleException();
}
```

The Core class loader gives special treatment to this particular method, in most cases, removing dynamic type coercion and checking code in favor of a static check.

CoreRegistry.profiles(). The profiles() method shall return an array of CoreString representing the collection of all real-time profiles that are present within this Core Execution Environment. Profile naming conventions serve to differentiate key features of the profiles, as follows:

1. A profile whose name begins with the substring "org.j-consortium" is considered to be an official J Consortium profile. The specification for the profile shall have been formalized by the J Consortium. The J Consortium maintains the official definition of the profile and may provide mechanisms to assess conformance of implementations.
2. All other profiles are considered to be proprietary, defined by particular individuals or industry organizations. The specification and conformance assessment for these profiles is handled external to the J Consortium.
3. Any profile whose name ends with the special character "-" shall disable certain capabilities that would normally be present in the Core Execution Environment. Any profile whose name does not end with the special character "-" shall not disable any capabilities that would normally be present in the Core Execution Environment. To ensure that a particular Core Execution Environment supports all of the features of the Core specification, a Core component could verify through examination of the names of the system's profiles that none of the installed profiles removes any core functionality.

The Core signature for the profiles() method follows:

```
public static CoreString [ ] profiles();
```

CoreRegistry.publish(). The publish() method shall publish core_object for access by Baseline and/or native components. The publish() method shall allocate and initialize memory for a private copy of the name CoreString argument and for additional implementation-defined objects for use in representing this entry within the CoreRegistry's private data tables. This private copy of the name argument shall be allocated within a dedicated implementation-defined AllocationContext. The Core signature is shown below:

```
public static void publish(CoreString name, CoreObject core_object);
```

CoreRegistry.unpublish(). The unpublish() method shall remove the previously published core object that is identified by its name argument from the CoreRegistry tables and shall release the AllocationContext that was previously dedicated to representing this entry within the CoreRegistry database. After the entry has been unpublished, subsequent

Copyright 1999, 2000 J Consortium, All Rights Reserved

attempts by the Baseline and/or native domains to lookup the CoreObject known by this name shall fail. The Core signature is shown below:

```
public static void unpublish(CoreString name);
```

3.17.17 The SignalingSemaphore Class

The SignalingSemaphore class extends org.rtjwg.CoreObject. The key difference between CountingSemaphore and SignalingSemaphore is that SignalingSemaphore shall not buffer V() operations. The default (only) constructor shall take no arguments.

SignalingSemaphore.P(). The P() method shall implement a semaphore P operation. Care shall be taken in the implementation of P() to avoid race conditions between multiple threads invoking P(), V(), and/or Vall() methods on the same semaphore. The Core-Baseline _P() method serves the same purpose for the Baseline environment. The Core signatures are shown below:

final public void *stackable* P();
final public *baseline* void _P();

SignalingSemaphore.V(). The V() method shall implement a semaphore V operation, releasing the highest priority longest waiting Core task or Baseline thread that is blocked on this semaphore. If no tasks or threads are currently blocked on this semaphore, the V() method has no effect. Care shall be taken in the implementation of V() to avoid race conditions between multiple threads invoking P(), V(), and/or Vall() methods on the same semaphore. The Core-Baseline _V() method serves the same purpose.

The Core signature is shown below:

final public void *stackable* V();
final public *baseline* void _V();

SignalingSemaphore.Vall(). The Vall() method shall awaken all Core tasks and Baseline threads that are blocked on this semaphore. If no tasks or threads are currently blocked on this semaphore, the Vall() method has no effect. Care shall be taken in the implementation of Vall() to avoid race conditions between multiple threads invoking P(), V(), and/or Vall() methods on the same semaphore. The Core-Baseline _Vall() method shall serve the same purpose.

The implementation of Vall() shall be constant-time, allowing its use from within a time-constrained interrupt handler. The work of waking up the various waiting tasks shall be distributed between the various P() operations that are waiting to be signaled.

The Core signatures are shown below:

final public void *stackable* Vall();
final public *baseline* void _Vall();

SignalingSemaphore.numWaiters(). The numWaiters() method shall report how many tasks or threads are waiting or blocked on this semaphore. The Core-Baseline _numWaiters() method serves the same purpose.

The Core signature is shown below:

```
final public int stackable numWaiters();
final public baseline int _numWaiters();
```

3.17.18 The CountingSemaphore Class

The CountingSemaphore class shall extend org.rtjwg.CoreObject. The key difference between CountingSemaphore and SignalingSemaphore is that CountingSemaphore buffers V operations. The default (only) constructor shall take no arguments.

CountingSemaphore.P(). The P() method is a semaphore P operation. If the value of the semaphore's count field is greater than zero, the P operation shall decrement the count. If the value of the semaphore's count field equals zero, the currently executing task or thread shall block until some other task or thread performs a V operation on this same counting semaphore. Care shall be taken in the implementation of P() to avoid race conditions between multiple threads invoking P() and/or V() methods on the same semaphore. The Core-Baseline _P() method serves the same purpose. The Core signatures are shown below:

```
final public void stackable P();
final public baseline void _P();
```

CountingSemaphore.V(). The V() method represents a semaphore V operation. If a core task or Baseline thread is currently waiting to lock this semaphore, this method shall awaken the highest priority, longest waiting task that is blocked on this semaphore. Otherwise, this operation shall increment the value of the count field associated with this semaphore. Care shall be taken in the implementation of V() to avoid race conditions between multiple threads invoking P() and V() methods on the same semaphore. The Core-Baseline _V() operation serves the same purpose. The Core signatures are shown below:

```
final public void stackable V();
final public baseline void _V();
```

CountingSemaphore.numWaiters(). The numWaiters() method shall report how many tasks or threads are blocked waiting on this semaphore. The Core-Baseline _numWaiters() method shall serve the same purpose. The Core signatures are shown below:

```
final public int stackable numWaiters();
final public baseline int _numWaiters();
```

CountingSemaphore.count(). The count() method shall report the current value of this semaphore's internal count field. The Core-Baseline _count() method shall serve the same purpose. The Core signature is shown below:

```
final public int stackable count();
final public baseline int _count();
```


Copyright 1999, 2000 J Consortium, All Rights Reserved

3.17.19 The Mutex Class

The Mutex class is used like a semaphore to enforce mutual exclusion. The key distinction between semaphores and a Mutex object is that the Mutex class shall implement priority inheritance. The task or thread that locks a Mutex object shall continue to own mutual exclusion until that same task or thread unlocks the Mutex object. If some higher priority task or thread attempts to lock the same Mutex object while it is already locked by a lower priority task or thread, the priority inheritance mechanism shall automatically elevate the priority of the original lock holder to the level of the higher priority task or thread that is requesting access to the lock. The implementation of priority inheritance shall be transitive, meaning that active priority of the task holding the lock is always at least as high as the highest priority of any task that is waiting for entry into the locked resource. If a CoreTask aborts or stops while it is holding a Mutex lock, the Core Execution Environment shall automatically release the lock.

Mutex Constructors. The default and only constructor for Mutex takes no arguments.

Mutex.lock(). The lock() method shall obtain the lock associated with this Mutex object, blocking the current task until other tasks release their lock if necessary. The Core-Baseline _lock() method serves the same purpose. The Core signatures are shown below:

final public void *stackable* lock();
final public baseline void _lock();

Mutex.unlock(). The unlock() method shall release a previously obtained lock. If the lock is not currently held by the current task or thread, this method throws a previously allocated instance of CoreIllegalMonitorStateException. The Core-Baseline _unlock() method serves the same purpose. The Core signatures are shown below:

final public void *stackable* unlock() throws CoreIllegalMonitorStateException;
final public *baseline* void _unlock() throws CoreIllegalMonitorStateException;

3.17.20 The Configuration Class

The Core Execution Environment can be configured in multiple distinct forms. The system integrator shall set configuration preferences by modifying the implementation of the Configuration class.

The Configuration class extends org.rtjwg.CoreObject. To configure the Core Execution Environment, the system integrator edits the constants defined in this class. When defining these variables, the system integrator must take care to ensure that the requested configuration is consistent with the capabilities of the underlying hardware.

It is implementation-defined which combinations of configuration parameters are supported by each Core Execution Environment. The constant numbers programmed into the Configuration class are suggestions to the Core Execution Environment. Programmers should never assume that the suggested parameter values have been honored. In all cases, APIs are provided to allow Core components to query the Core Execution Environment to discover how it is actually configured.

Configuration.tick_duration. tick_duration is the requested number of nanoseconds between timer ticks. The Core Execution Environment shall round up all timeouts and time slice requests to the nearest timer tick. The Core declaration for this variable is:

```
public static final int tick_duration;
```

Configuration.ticks_per_slice. ticks_per_slice represents the desired number of timer ticks in each time slice. If tick_duration equals 1,000 and ticks_per_slice equals 10, the system integrator is asking for 10 microseconds per time slice. Special significance is given to a value of zero. If ticks_per_slice() is set to zero, this represents a desire to disable all time slicing for this configuration of the Core Execution Environment. The Core declaration for this variable is:

```
public final static int ticks_per_slice;
```

Configuration.uptime_precision. uptime_precision represents the desired resolution of the result returned from the upTime() method. If uptime_precision has value 100, this means that the result returned from uptime() shall be accurate to within plus or minus 100 nanoseconds. The Core declaration for uptime_precision is shown below:

```
public static final int uptime_precision;
```

Configuration.default_stack_size. default_stack_size represents the default size, measured in 32-bit words, of CoreTask. If default_stack_size has value 1,024, this means that unless specified to the contrary, each CoreTask is started up with a stack size of 1,024 words. The Core declaration for default_stack_size is shown below:

```
public static final int default_stack_size;
```

Configuration.stack_overflow_checking. stack_overflow_checking represents whether or not this Core Execution Environment is configured to perform stack overflow checking. If this variable's value is true, stack overflow checking shall be enabled. Otherwise, stack overflow checking should, but need not, be disabled. A conforming implementation of the Core Execution Environment must support the option of performing stack overflow checking. A conforming implementation of the Core specification need not honor the request to disable stack overflow checking. The Core declaration for stack_overflow_checking is shown below:

```
public static final boolean stack_overflow_checking;
```

Configuration.min_core_priority. min_core_priority represents the intended system-level priority that corresponds to the Core task priority level 0. The Core declaration for this variable is shown below:

```
public static final int min_core_priority;
```

Configuration.system_priority_map. system_priority_map represents the desired mapping from Core priorities to underlying operating system priorities. This array has 128 entries. The first entry in this array is the system priority level at which Core priority-1 tasks should execute. The second entry in this array is the system priority level at which Core priority-2 tasks should execute, and so on. Note that a conforming implementation of the Core Execution Environment need not honor a configuration request to define the system priority map. The Core declaration for this variable is shown below:

Copyright 1999, 2000 J Consortium, All Rights Reserved

```
public static final int [] system_priority_map;
```

Configuration.little_endian. If little_endian is true, this represents a request to treat all IOPort classes as little-endian channels. To say that the channel is little endian means that the byte whose address is the same as the address of a multi-byte value stored at the same location represents the least-significant byte of that larger value. If little_endian is false, this represents a request to treat all IOPort classes as big-endian channels. To say that the channel is big endian means that the byte whose address is the same as the address of a multi-byte value stored at the same location represents the most-significant byte of that larger value. The Core declaration for this variable is shown below:

```
public static final boolean little_endian;
```

3.17.21 The Time Class

Mainly as an aid to enhance source code readability, the Time class provides unit conversions between common units of time measurement. The standard representation of time is a 64-bit long integer, representing nanoseconds.

The Time class extends CoreObject. This class is not designed to be instantiated. Rather, Time provides a variety of services in the form of static methods.

Time.tickDuration(). This method shall return the number of nanoseconds between consecutive ticks of the Core Execution Environment's timer. Note that the value returned from this method might not equal Configuration.tick_duration in cases that the system integrator's request could not be satisfied. The Core signature follows:

```
public static int tickDuration();
```

Time.uptimePrecision(). The uptimePrecision() method shall return the precision of the uptime() method, measured in nanoseconds. For example, if uptimePrecision() returns 100, this means that the result returned from uptime() is accurate to within plus or minus 100 nanoseconds. Note that the value returned from this method might not equal Configuration.uptime_precision in cases that the system integrator's request could not be satisfied. The Core signature follows:

```
public static int uptimePrecision();
```

Time.day(). The day() method shall return the number of nanoseconds in day days, throwing a previously allocated instance of CoreArithmeticOverflowException if the number of nanoseconds is too large to be represented in a 64-bit integer. The Core signature is shown below:

```
public static long day(int day) throws CoreArithmeticOverflowException;
```

Time.h(). The h() method shall return the total number of nanoseconds in h hours, throwing a previously allocated instance of CoreArithmeticOverflowException if the number of nanoseconds is too large to be represented in a 64-bit integer. The Core signature is shown below:

```
public static long h(int h) throws CoreArithmeticOverflowException;
```

Time.hertz(). The hertz() method shall return the number of nanoseconds in a period corresponding to freq Hertz, throwing a previously allocated instance of CoreArithmeticOverflowException if the number of nanoseconds is too large to be represented in a 64-bit integer. The period is rounded down. The Core signature is shown below:

```
public static long hertz(int freq) throws CoreArithmeticOverflowException;
```

Time.m(). The m() method shall return the number of nanoseconds in m minutes, throwing a previously allocated instance of CoreArithmeticOverflowException if the number of nanoseconds is too large to be represented in a 64-bit integer. The Core signature is shown below:

```
public static long m(int m) throws CoreArithmeticOverflowException;
```

Time.ms(). The ms() method shall return the number of nanoseconds in ms milliseconds, throwing a previously allocated instance of CoreArithmeticOverflowException if the number of nanoseconds is too large to be represented in a 64-bit integer. There are two versions of this method. The Core signatures are shown below:

```
public static long ms(int ms) throws CoreArithmeticOverflowException;
public static long ms(long ms) throws CoreArithmeticOverflowException;
```

Time.ns(). The ns() method shall return its ns argument. This method serves no real purpose other than facilitating the creation of self-documenting code. The Core signature is shown below:

```
public static long ns(long ns);
public static long ns(int ns);
```

Time.s(). The s() method shall return the number of nanoseconds in s seconds, throwing a previously allocated instance of CoreArithmeticOverflowException if the number of nanoseconds is too large to be represented in a 64-bit integer. The Core signature is shown below:

```
public static long s(long s) throws CoreArithmeticOverflowException;
public static long s(int s) throws CoreArithmeticOverflowException;
```

Time.toString(). The toString() method shall return a string representation of its ns argument according to the template: "ddddd hh:mm:ss.decimal" where d represents days, h represents the total number of whole hours, m represents the total number of whole minutes, s represents the total number of whole seconds, and decimal represents the fractional number of seconds. All numbers are represented in English decimal notation. To facilitate report formatting, the various fields are fixed width. In particular:

d: 5 characters (right justified)
h: 2 characters (right justified, 0 filled)
m: 2 characters (right justified, 0 filled)
s: 2 characters (right justified, 0 filled)

decimal: 9 characters (left justified, 0 filled, rounded in the last digit to an even number if the tenth digit equals 5 and all remaining digits equal 0)

A side effect of invoking toString() is to create a new CoreString object in the current allocation context. The memory for this CoreString object shall become eligible for recycling

Copyright 1999, 2000 J Consortium, All Rights Reserved

when the corresponding allocation context is released. The Core signature is shown below:

```
public static CoreString toString(long ns);
```

Time.uptime(). The uptime() method shall return the number of nanoseconds since the system was last restarted. The Core signature is shown below:

```
public static long uptime();
```

Time.us(). The us() method shall return the number of nanoseconds represented by its us argument, which is expressed in terms of microseconds, throwing a previously allocated instance of CoreArithmeticOverflowException if the number of nanoseconds is too large to be represented in a 64-bit integer. The Core signature is shown below:

```
public static long us(long us) throws CoreArithmeticOverflowException;
public static long us(int us) throws CoreArithmeticOverflowException;
```

3.17.22 The Unsigned class

One shortcoming in the Java language is its lack of built-in support for unsigned integers. This section describes the API for the Unsigned class, a Core library that provides support for traditional unsigned arithmetic. This class is not intended to be instantiated. Instead, the services are provided in the form of static methods.

Unsigned.compare(). There are four variants of the compare() method, targeted to the four different integer sizes that might be used to represent unsigned integers. In all cases, the compare() method shall return -1 if its first argument is smaller than the second, 0 if the two arguments are equal, and 1 if its first argument is larger than the second. All comparisons treat their arguments as if they are encoded according to unsigned integer conventions.

The Core signatures for the four variants of compare() follow:

```
static int compare(byte b1, byte b2);
static int compare(short s1, short s2);
static int compare(int i1, int i2);
static int compare(long l1, long l2);
```

Unsigned.ge(). The ge() method shall return true if its first argument is greater than or equal to its second argument, and false otherwise. The magnitude comparison assumes both arguments are encoded according to unsigned integer conventions. There are four variants of ge(), with signatures as shown below, to address each of the four different integer sizes that might be used to represent unsigned integer quantities.

```
static boolean ge(byte b1, byte b2);
static boolean ge(short s1, short s2);
static boolean ge(int i1, int i2);
static boolean ge(long l1, long l2);
```

Unsigned.gt(). The gt() method shall return true if its first argument is greater than its second argument, and false otherwise. The magnitude comparison assumes both arguments are encoded according to unsigned integer conventions. There are four variants of gt(),

with signatures as shown below, to address each of the four different integer sizes that might be used to represent unsigned integer quantities.

```
static boolean gt(byte b1, byte b2);
static boolean gt(short s1, short s2);
static boolean gt(int i1, int i2);
static boolean gt(long l1, long l2);
```

Unsigned.le(). The le() method shall return true if its first argument is less than or equal to its second argument, and false otherwise. The magnitude comparison assumes both arguments are encoded according to unsigned integer conventions. There are four variants of le(), with signatures as shown below, to address each of the four different integer sizes that might be used to represent unsigned integer quantities.

```
static boolean le(byte b1, byte b2);
static boolean le(short s1, short s2);
static boolean le(int i1, int i2);
static boolean le(long l1, long l2);
```

Unsigned.lt(). The lt() method shall return true if its first argument is less than its second argument, and false otherwise. The magnitude comparison assumes both arguments are encoded according to unsigned integer conventions. There are four variants of lt(), with signatures as shown below, to address each of the four different integer sizes that might be used to represent unsigned integer quantities.

```
static boolean lt(byte b1, byte b2);
static boolean lt(short s1, short s2);
static boolean lt(int i1, int i2);
static boolean lt(long l1, long l2);
```

Unsigned.eq(). The eq() method shall return true if its first argument is equal to its second argument, and false otherwise. The magnitude comparison assumes both arguments are encoded according to unsigned integer conventions. There are four variants of eq(), with signatures as shown below, to address each of the four different integer sizes that might be used to represent unsigned integer quantities.

```
static boolean eq(byte b1, byte b2);
static boolean eq(short s1, short s2);
static boolean eq(int i1, int i2);
static boolean eq(long l1, long l2);
```

Unsigned.neq(). The neq() method shall return true if its first argument is not equal to its second argument, and false otherwise. The magnitude comparison assumes both arguments are encoded according to unsigned integer conventions. There are four variants of neq(), with signatures as shown below, to address each of the four different integer sizes that might be used to represent unsigned integer quantities.

```
static boolean neq(byte b1, byte b2);
static boolean neq(short s1, short s2);
static boolean neq(int i1, int i2);
static boolean neq(long l1, long l2);
```


Copyright 1999, 2000 J Consortium, All Rights Reserved

Unsigned.toByte(). The toByte() method shall coerce its unsigned integer argument to an unsigned 8-bit quantity, throwing a previously allocated instance of CoreUnsignedCoercionException if the number to be coerced is greater than 255 (2^8 - 1). There are four variants of toByte(), with signatures as shown below, to address each of the four different integer sizes that might be used to represent unsigned integer quantities.

```
static byte toByte(byte b);
static byte toByte(short s) throws CoreUnsignedCoercionException;
static byte toByte(int i) throws CoreUnsignedCoercionException;
static byte toByte(long l) throws CoreUnsignedCoercionException;
```

Unsigned.toShort(). The toShort() method shall coerce its unsigned integer argument to a 16-bit quantity, throwing a previously allocated instance of CoreUnsignedCoercionException if the number to be coerced is greater than 65,535 (2^{16} - 1). There are four variants of toShort(), with signatures as shown below, to address each of the four different integer sizes that might be used to represent unsigned integer quantities.

```
static short toShort(byte b);
static short toShort(short s);
static short toShort(int i) throws CoreUnsignedCoercionException;
static short toShort(long l) throws CoreUnsignedCoercionException;
```

Unsigned.toInt(). The toInt() method shall coerce its unsigned integer argument to a 32-bit quantity, throwing a previously allocated instance of CoreUnsignedCoercionException if the number to be coerced is greater than 4,294,967,295 (2^{32} - 1). There are four variants of toInt(), with signatures as shown below, to address each of the four different integer sizes that might be used to represent unsigned integer quantities.

```
static int toInt(byte b);
static int toInt(short s);
static int toInt(int i);
static int toInt(long l) throws CoreUnsignedCoercionException;
```

Unsigned.toLong(). The toLong() method shall coerce its unsigned integer argument to a 64-bit quantity. Note that there is no possibility of overflow when coercing to 64-bit unsigned quantities. There are four variants of toLong(), with signatures as shown below, to address each of the four different integer sizes that might be used to represent unsigned integer quantities.

```
static long toLong(byte b);
static long toLong(short s);
static long toLong(int i);
static long toLong(long l);
```

Unsigned.toString(). The toString() method shall take an unsigned integer as its argument and return a CoreString object representing the value of its supplied argument in unsigned decimal representation. There are four variants of toString(), with signatures as shown below, to address each of the four different integer sizes that might be used to represent unsigned integer quantities. The CoreString object returned from toString() is allocated within the current task's allocation context.

```
static CoreString toString(byte b);
static CoreString toString(short s);
static CoreString toString(int i);
static CoreString toString(long l);
```

Unsigned.toHexString(). The toHexString() method shall take an unsigned integer as its argument and return a CoreString object representing the value of its supplied argument in hexadecimal representation. The length of the resulting CoreString object shall depend on the type (not the value) of the argument, padding with zero as appropriate. Alphabetic characters in the resulting string shall be lower case. There are four variants of toHexString(), with signatures as shown below, to address each of the four different integer sizes that might be used to represent unsigned integer quantities. The CoreString object returned from toHexString() is allocated within the current task's allocation context.

```
static final CoreString toHexString(byte b);     // Returns 1-character string
static final CoreString toHexString(short s);    // Returns 2-character string
static final CoreString toHexString(int i);      // Returns 4-character string
static final CoreString toHexString(long l);     // Returns 8-character string
```

3.17.23 The CoreTask Class

The CoreTask class shall extend CoreObject. This class represents the analog of java.lang.Thread within the Core domain.

To create a Core task, the Core programmer extends CoreTask, providing an implementation of the work() method. To start the task's execution as an independently executing thread, the application invokes the CoreTask object's start() or _start() methods.

Upon invoking the start() or _start() methods of a newly constructed CoreTask object, the Core Execution Environment shall initiate execution of the task. For a CoreTask object, this causes the work() method to be invoked. Once the work() method terminates, the CoreTask has completed its execution. As long as the CoreTask.work() method continues to execute, additional increments of CPU time are dedicated toward execution of this method according to the fixed priority round-robin scheduling system that is part of the Core Execution Environment.

There are two subclasses of CoreTask, named ISR_Task and SporadicTask, which represent special forms of real-time tasks. For SporadicTask and ISR_Task, invocation of the start() or _start() methods makes the task eligible to be triggered for execution by the corresponding asynchronous event.

CoreTask Constructor. When a CoreTask is created, it is necessary to identify several characteristics of the task, as listed below:

1. Whether or not asynchronous event handling other than abort() and stop() is enabled for this core task.
2. The size of this task's run-time stack.
3. The size and type of the default allocation context for this CoreTask.
4. The task's Base Priority.

Copyright 1999, 2000 J Consortium, All Rights Reserved

The signature of the CoreTask constructor is shown here:

```
public CoreTask(ATCEventHandler handler, long stack_size,
        long allocation_size, CoreString allocation_block_name, int priority)
            throws CoreBadPriorityException, CoreEmbeddedConflictException;
```

If handler is null, this CoreTask shall ignore asynchronous event signaling other than abort() and stop() requests. Otherwise, the initial event handler for this task is represented by handler. The stack_size argument specifies the number of 32-bit words on the task's run-time stack. If stack_size equals zero, the task's stack will be the default stack size. The allocation_size argument specifies the number of bytes in the task's default AllocationContext. If allocation_size equals zero, the default AllocationContext for this task is variable, growing at run time based on demand and availability of memory. The allocation_block_name argument specifies the name of the block of memory within which the AllocationContext's memory shall be located. If this argument equals null, the Core Execution Environment shall place the AllocationContext's memory region in the host computer system's main memory. If allocation_block_name specifies an allocation region that does not exist within this Core Execution Environment, or if the Core Execution Environment chooses (for implementation-defined reasons) to not permit this task to use the named memory region as its default allocation region, the constructor shall throw a previously allocated instance of CoreEmbeddedConflictException. The priority argument specifies the Base Priority of the CoreTask object. The constructor throws a previously allocated instance of CoreBadPriorityException if the priority argument is outside the range of valid Core task priorities.

Static Methods.

CoreTask.currentTask(). The currentTask() method shall return a reference to the task that is currently executing in the Core Execution Environment. The Core signature is shown below:

```
public static CoreTask currentTask();
```

CoreTask.defaultStackSize(). The defaultStackSize() method shall return the default stack size, specified in terms of 32-bit words. Note that the value returned from this method might not equal Configuration.default_stack_size in cases that the system did not honor the system integrator's request. The Core signature is shown below:

```
static long defaultStackSize();
```

CoreTask.maxBaselinePriority(). The maxBaselinePriority() method shall return the system-level priority that corresponds to the top Baseline thread priority. For example, if java.lang.Thread priority number 10 corresponds to the host operating system's priority number 12, this method shall return 12. The Core signature is shown below:

```
public static int maxBaselinePriority();
```

CoreTask.maxCorePriority(). The maxCorePriority() method shall return the system-level priority that corresponds to the top Core priority. For example, if core priority number 128 corresponds to system priority level 140, this method shall return 140. The Core signature is shown below:

```
public static int maxCorePriority();
```

CoreTask.maxSystemPriority(). The maxSystemPriority() method shall return the maximum priority number for identifying the system-level priorities supported by the host operating system. For example, if the host operating system supports priorities numbered from 0 to 255, this method returns 255. The Core signature is shown below:

```
public static int maxSystemPriority();
```

CoreTask.minBaselinePriority(). The minBaselinePriority() method shall return the system-level priority that corresponds to the bottom Baseline thread priority. For example, if java.lang.Thread priority number 1 corresponds to the host operating system's priority number 3, this method shall return 3. The Core signature is shown below:

```
public static int minBaselinePriority();
```

CoreTask.minCorePriority(). The minCorePriority() method shall return the system-level priority that corresponds to the bottom Core priority. For example, if core priority number 0 corresponds to system priority level 13, this method shall return 13. The Core signature is shown below:

```
public static int minCorePriority();
```

CoreTask.minSystemPriority(). The minSystemPriority() method shall return the minimum priority number for identifying the system-level priorities supported by the host operating system. For example, if the host operating system supports priorities numbered from 0 to 255, this method shall return 0. The Core signature is shown below:

```
public static int minSystemPriority();
```

CoreTask.numInterruptPriorities(). The numInterruptPriorities() method shall return the number of priorities that are dedicated to interrupt handling. The interrupt-level priorities are always the highest priorities in the system. If numInterruptPriorities() returns 12, for example, Core priorities 117 through 128 are known to represent interrupt-level priorities.

```
public static final int numInterruptPriorities();
```

CoreTask.stackOverflowChecking(). The stackOverflowChecking() method shall return true if and only if the Core Execution Environment is configured to perform stack overflow checking. Note that conforming Core Execution Environments might run with stack overflow checking enabled even if Configuration.stack_overflow_checking is false. The Core signature is shown below:

```
public static boolean stackOverflowChecking();
```

CoreTask.systemPriorityMap(). The systemPriorityMap() method shall return an integer array with 128 entries in it, representing the system priorities to which each of the Core priority levels correspond. The first entry in this array is the system priority level at which Core priority-1 tasks execute. The second entry in this array is the system priority level at which Core priority-2 tasks execute, and so on. The returned array is a private copy of this information, allocated in the currently active AllocationContext. The Core signature is shown below:

Copyright 1999, 2000 J Consortium, All Rights Reserved

```
public static int [ ] systemPriorityMap();
```

CoreTask.ticksPerSlice(). The ticksPerSlice() method shall return the number of timer ticks that the system is dedicating to each time slice of a CoreTask. If ticksPerSlice() returns zero, this indicates that this Core Execution Environment is configured to perform no time slicing. Note that the result of this method might not equal Configuration.ticks_per_slice in case the Core Execution Environment does not honor the system integrator's request. The Core signature is shown below:

```
static public int ticksPerSlice();
```

Instance Methods.

CoreTask.abort(). The abort() method shall cause this Core task to abort execution of the most recent invocation of its work() method. The implementation of abort() shall support the same semantics as the implementation of asynchronous transfer of control. This means that abort() requests are deferred during execution of code that defers asynchronous event handling. Even Core tasks that are constructed with asynchronous event handling disabled shall abort themselves in response to invocation of the task's abort() method.

```
public final void abort();
```

CoreTask.abortWorkException(). The abortWorkException() method shall return a reference to a previously allocated ScopedThrowable object that is provided for the purpose of aborting the work() method associated with this CoreTask object. The scope for this exception shall belong to that part of the Core Execution Environment that invokes this task's work() method. The Core signature for this method is shown below:

```
public final ScopedThrowable abortWorkException()
```

To abort the currently executing task's work() method, the Core programmer might execute the following:

```
throw CoreTask.currentTask().abortWorkException();
```

CoreTask.asyncHandler(). The asyncHandler() method shall atomically set the asynchronous event handler for this CoreTask, returning a reference to the previously established asynchronous event handler. If this task was constructed without an asynchronous event handler, asynchronous event handling is permanently disabled for this task. In that case, asyncHandler() throws a previously allocated instance of CoreATCEventsIgnoredException instead of changing the asynchronous event handler. The Core signature is shown below:

```
public final ATCEventHandler asyncHandler(ATCEventHandler new_handler)
        throws CoreATCEventsIgnoredException;
```

CoreTask.join(). The join() method causes the current task to block until this task terminates execution. For ISR_Task and SporadicTask, termination means that the task's stop() method has been invoked and completely processed. For CoreTask, termination means either that the task's stop() method has been invoked and completely processed, or that the task has returned from its work() invocation. The Core signature is shown below:

```
public final void join();
```

CoreTask.resume(). If this task is currently in a suspended state (because of a prior invocation of the suspend() method), the resume() method shall cause this task's Base Priority to be restored to the value it held at the moment the task's suspend() method was invoked, or to the new value specified by the most recent invocation of the task's setPriority() method during the time this task was suspended. If this task is not currently in a suspended state, the resume() method shall have no effect. The Core signature is shown below:

```
public final void resume();
```

CoreTask.setPriority(). The setPriority() method shall set the Base Priority for the given task, performing a security check to see if the current thread is allowed to modify the priority of this CoreTask. This method shall throw a previously allocated instance of CoreSecurityException if the current thread is not allowed to modify the priority of the specified task and throws a previously allocated instance of CoreBadPriorityException if the requested priority is not in the range of acceptable core priorities. If this task is currently executing within a priority ceiling context for which the ceiling priority is lower than the value of this method invocation's new_priority argument, the effect of the setPriority() method shall be deferred until after this task leaves the priority ceiling context. The Core signature is shown below:

```
public final void setPriority(int new_priority) throws
    CoreBadPriorityException, CoreSecurityException, CoreIllegalMonitorStateException;
```

CoreTask.signalAsync(). The signalAsync() method shall cause this task to invoke its current event handler, passing ATCEvent e as an argument. If the event handler returns, control resumes within this task at the instruction that follows the point at which the asynchronous event handling began. If the event handler throws an exception, it is as if the exception was thrown by whatever code was executing within this CoreTask when control was interrupted by asynchronous event handling. If this task was configured to ignore asynchronous events, signalAsync() throws a previously allocated instance of CoreATCEventsIgnoredException. The Core signature is shown below:

```
public final void signalAsync(ATCEvent e) throws CoreATCEventsIgnoredException;
```

If this task is executing code contained within a finally statement, or is executing code contained within a synchronized block of any object that implements Atomic at the moment signalAsync() is invoked, handling of the asynchronous event is deferred until control leaves that context.

We say an ATCEvent object is pending on a particular thread if that object has been passed as an argument to a completed invocation of that thread's signalAsync() method but the thread has not yet begun to execute its ATCEventHandler.handleATCEvent() method. In the case that this thread has received a previous invocation of signalAsync() and is still waiting to process that previous request because the thread is still executing within a deferral region (a finally statement or synchronized statement associated with an Atomic object), the ATCEvent argument of signalAsync() will be placed on a queue of pending asynchronous transfer of control events associated with this thread unless this same event is already pending for this or some other thread. If this event is already

Copyright 1999, 2000 J Consortium, All Rights Reserved

pending on some thread, the new signalAsync() invocation is simply ignored. (To assure that no asynchronously signaled events go ignored, application programmers should structure their software so that each event corresponds to a different thread, and that the event is not signaled a second time to that thread until after the thread has processed the first signaling of the event.) Pending events are processed in FIFO order as soon as this thread leaves its deferral region. This results in nesting of the asynchronous transfer of control event handlers, with the handler for the most recently signaled event nested within the others.

If this task is currently blocked (on entry to a Mutex.lock(), or entry to a synchronized context, or a semaphore P() operation), or is suspended in a CoreTask.sleep(), CoreTask.sleepUntil(), CoreTask.join(), or CoreObject.wait() invocation, the suspending operation is interrupted by the signalAsync() invocation. If the event handler returns, the blocking operation is restarted. This means that this CoreTask loses its place in any FIFO queue associated with the blocking operation.

CoreTask.sleep(). The sleep() method shall cause the current task to sleep a minimum of sleep_time nanoseconds. The Core signature is shown below:

```
public static sleep(long sleep_time);
```

CoreTask.sleepUntil(). The sleepUntil() method shall cause the current task to sleep until the specified time arrives, where time is measured according to Time.uptime(). The Core signature is shown below:

```
public static sleepUntil(long alarm_time);
```

CoreTask.stackDepth(). The stackDepth() method returns the number of words currently in use on this task's run-time stack. The Core signature is shown below:

```
public final int stackDepth();
```

CoreTask.stackSize(). The stackSize() method returns the total number of words allocated to this task's run-time stack. The Core signature is shown below:

```
public final int stackSize();
```

CoreTask.start(). The start() method shall start the Core task, making it ready for execution. Note that certain subclasses of CoreTask (e.g. SporadicTask) do not begin to run immediately following invocation of the start() method. Rather, these subclasses of CoreTask begin execution at some point following invocation of the start() method, in response to an asynchronous trigger invocation. The Core signature is shown below:

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

CoreTask._start(). The _start() method shall start up a Core task (in the same way that the CoreTask.start() method starts up a CoreTask) from the Baseline domain. The Core signature is shown below:

final public *baseline* void _start();

CoreTask.stop(). The stop() method shall render the CoreTask inoperable, making it no longer eligible for dispatching by the Core Execution Environment. If this task is execut-

ing its work() method when stop() is invoked, this method implements the equivalent of CoreTask.abort() followed by whatever additional work is required by the semantics of the stop() method. Following invocation of the stop() method, subsequent invocations of start() have no effect. If the CoreTask is running (or suspended) when the stop() method is invoked, all finally clauses associated with nested execution of try statements by this CoreTask are executed, enabling release of all synchronization locks held by the task. If the core task is executing within an “Atomic Synchronized” region (See Section 3.17.11), abortion of the CoreTask is deferred until after the Atomic Synchronized region completes its execution. Similarly, if the core task is executing the body of a finally statement, abortion of the core task is deferred until after the finally statement has executed to completion. The Core signature is shown below:

```
public void stop();
```

CoreTask.suspend(). The suspend() method shall temporarily set the Base Priority of this task to the special Never Scheduled Priority level. Assuming that this task is not currently inheriting a higher priority level, this causes this task to be put to sleep until it is subsequently awakened by some other task’s invocation of the resume(). If, however, the task holds a synchronization lock that is required by some other task, this task will continue to run at its active priority, as determined by the corresponding lock’s priority inversion avoidance mechanism (either priority inheritance or immediate priority ceiling). The Core signature is shown below:

```
public final void suspend();
```

CoreTask.systemPriority(). The systemPriority() method shall return the system-level priority that corresponds to this Core task’s priority level. For example, if this real-time core task is running at host operating system priority 23, regardless of what core priority this might correspond to, this method shall return 23. The Core signature is shown below:

```
public final int systemPriority();
```

CoreTask.work(). The work() method shall be invoked by the Core Execution Environment to do the work of this task. The default implementation of the work() method simply returns void. Normally, the Core programmer will override this default implementation with an appropriate replacement. The Core signature is shown below:

```
public void work();
```

CoreTask.yield(). The yield() method shall cause the currently executing Core task to yield the remainder of its time slice to another Core task of equal priority. If no other Core tasks of equal priority are ready to run, the yield() method shall have no effect. The Core signature is shown below:

```
final public void yield();
```

3.17.24 The ISR_Task Class

The ISR_Task class extends CoreTask and implements Atomic. This class is used to implement interrupt service routines. With ISR_Task objects, the associated work is triggered by physical or software interrupts. The work of an ISR_Task is executed as part of an interrupt service routine rather than an operating system thread. Multiple ISR_Task objects may be registered to service the same interrupt event. Each time the shared inter-

Copyright 1999, 2000 J Consortium, All Rights Reserved

rupt is triggered, the Core Execution Environment shall invoke the work() methods associated with each of the interrupt handlers in sequence, ordered according to the priority of the ISR_Task objects that represent the respective interrupt handlers with higher priority ISR_Task objects serviced before lower priority ISR_Task objects. Following completion of each ISR_Task's work() method, the Core Execution Environment shall invoke the ISR_Task's serviced() method to determine whether the interrupt has been completely serviced. If ISR_Task.serviced() returns true, the Core Execution Environment shall consider interrupt processing done for this particular trigger, and shall not invoke the remaining lower priority ISR_Task objects' work() methods for this particular trigger.

ISR_Task Constructor. When an ISR_Task is created, it is necessary to identify several characteristics of the task, as listed below:

1. The size of this task's run-time stack.
2. The size and type of the default allocation context for this CoreTask.
3. The task's Base Priority.
4. The number of the interrupt that is to trigger execution of this ISR_Task.

The signature of the ISR_Task constructor is shown here:

```
public ISR_Task(long stack_size, long allocation_size,
                CoreString allocation_block_name, int priority, int interrupt_no)
            throws CoreBadPriorityException, CoreEmbeddedConflictException;
```

The stack_size argument specifies the number of words on the task's run-time stack. If stack_size equals zero, the task's stack will be the default stack size. The allocation_size argument specifies the number of bytes in the task's default AllocationContext. If allocation_size equals zero, the default AllocationContext for this task is variable, growing at run time based on demand and availability of memory. The allocation_block_name argument specifies the name of the block of memory within which the AllocationContext's memory shall be located. If this argument equals null, the Core Execution Environment shall place the AllocationContext's memory region in the host computer system's main memory. If allocation_block_name specifies an allocation region that does not exist within this Core Execution Environment, or if the Core Execution Environment chooses (for implementation-defined reasons) to not permit this task to use the named memory region as its default allocation region, the constructor shall throw a previously allocated instance of CoreEmbeddedConflictException. The priority argument specifies the Base Priority at which the ISR_Task's work() method executes each time the corresponding interrupt is triggered. The interrupt_no argument specifies the number of the interrupt that is to trigger execution of this ISR_Task's work() method. If interrupt_no equals -1, this ISR_Task object is not bound to a hardware interrupt, and can only be triggered by software. The constructor throws a previously allocated instance of CoreBadPriorityException if the priority argument is outside the range of valid Core task priorities or is lower than the interrupt priority of the interrupt that is to trigger execution of this interrupt service routine. There is no lower bound on this task's priority if the interrupt_no argument equals -1. The constructor throws CoreEmbeddedConflictException if the Core Execution Environment cannot bind this ISR_Task to the requested interrupt number.

ISR_Task.serviced(). If multiple ISR_Task objects share a single interrupt, the Core Execution Environment shall invoke the work() methods for these tasks in order of decreas-

ing priority. If multiple ISR_Task objects of the same priority are bound to the same interrupt number, their respective work() methods shall be executed in the order that these ISR_Task objects were bound to the corresponding interrupt (by invocation of the ISR_Task's arm() method). Following completion of each work() method, the Core Execution Environment shall invoke that same task's serviced() method to determine if the interrupt is considered to have been completely serviced. If a given task returns true from its serviced() method, this indicates that the interrupt has been completely serviced and the Core Execution Environment shall not invoke any additional ISR_Task.work() methods for this particular interrupt trigger. The default implementation of ISR_Task.serviced() shall return false. The Core signature follows:

```
public boolean serviced();
```

ISR_Task.trigger(). The trigger() method allows software to trigger execution of this interrupt service routine. Invoking trigger() has the effect of causing this ISR_Task alone to run its work() method at the ISR_Task's interrupt priority level. Note that this ISR_Task's work() method will be invoked even if this ISR_Task is currently disarmed. Also note that invoking the trigger() method for this ISR_Task does not cause whatever other ISR_Task objects are bound to the same interrupt to have their work() methods executed.

Trigger requests (whether by hardware or software) shall not be queued. For each trigger, the Core Execution Environment shall defer execution of the corresponding work() method as long as other tasks are running at higher priority, and as long as other interrupt service routine tasks are running at equal priority. If the same ISR_Task is triggered again while it is still deferring execution of its work() method from a previous trigger, the new trigger shall have no effect.

The Core signature is shown below:

```
public final void trigger();
```

ISR_Task.work(). The Core Execution Environment shall invoke the work() method each time the interrupt is triggered. This method is "Atomic Synchronized", meaning that the work() method must be execution-time analyzable. During execution of this method, interrupts at this object's priority ceiling level and below are disabled. The default implementation of the work() method simply returns void. The Core signature is shown below:

```
public synchronized void work();
```

All implementations of the work() method in subclasses of ISR_Task shall declare the method to be synchronized. The Core Verifier shall enforce this restriction.

ISR_Task.ceilingPriority(). The ceilingPriority() method shall return 129 minus CoreTask.numInterruptPriorities(). Note that subclasses of ISR_Task may override this method to return a different priority. The Core signature is shown below:

```
public short ceilingPriority();
```

ISR_Task.arm(). The arm() method shall cause this ISR_Task to become armed. When first constructed, ISR_Task objects are not armed. This means that the ISR_Task's work() method is not invoked by the Core Execution Environment in response to signaling of

Copyright 1999, 2000 J Consortium, All Rights Reserved

the corresponding hardware interrupt. To install an interrupt handler, the Core programmer must first construct the ISR_Task, following which he must invoke the start() or _start() methods, following which he must invoke the arm() method. The Core signature is shown below:

```
public final void ISR_Task.arm();
```

ISR_Task.disarm(). The disarm() method shall cause this ISR_Task to become disarmed. When first constructed, ISR_Task objects are not armed. This means that the ISR_Task's work() method is not invoked by the Core Execution Environment in response to signaling of the corresponding hardware interrupt. To install an interrupt handler, the Core programmer must first construct the ISR_Task, following which he must invoke the start() or _start() methods, following which he must invoke the arm() method. To return the ISR_Task to disarmed state after arming it, the Core programmer invokes the disarm() method. The Core signature is shown below:

```
public final void ISR_Task.arm();
```

3.17.25 The SporadicTask Class

The SporadicTask class extends CoreTask. Use this class to implement responses to sporadic (asynchronous) events. To trigger a SporadicTask to respond to an asynchronous event, invoke the task's trigger() method. This causes the task's work() method to be executed by this task running at the designated priority. If the task is still executing a previous invocation of its work() method when a new execution is triggered, the new request is queued so that this task can perform the requested invocation of the work() method following completion of previously triggered executions of the work() method.

SporadicTask Constructor. When a SporadicTask is created, it is necessary to identify several characteristics of the task, as listed below:

1. Whether or not asynchronous event handling other than abort() and stop() is enabled for this core task.
2. The size of this task's run-time stack.
3. The size and type of the default allocation context for this CoreTask.
4. The task's Base Priority.

The signature of the SporadicTask constructor is shown here:

```
public SporadicTask(ATCEventHandler handle, long stack_size, long allocation_size,
            CoreString allocation_block_name, int priority)
        throws CoreBadPriorityException, CoreEmbeddedConflictException;
```

If handler is null, this SporadicTask shall ignore asynchronous event signaling. Otherwise, the initial event handler for this task is represented by handler. The stack_size argument specifies the number of words on the task's run-time stack. If stack_size equals zero, the task's stack will be the default stack size. The allocation_size argument specifies the number of bytes in the task's default AllocationContext. If allocation_size equals zero, the default AllocationContext for this task is variable, growing at run time based on demand and availability of memory. The allocation_block_name argument specifies the name of the block of memory within which the AllocationContext's memory shall be located. If

this argument equals null, the Core Execution Environment shall place the AllocationContext's memory region in the host computer system's main memory. If allocation_block_name specifies an allocation region that does not exist within this Core Execution Environment, or if the Core Execution Environment chooses (for implementation-defined reasons) to not permit this task to use the named memory region as its default allocation region, the constructor shall throw a previously allocated instance of CoreEmbeddedConflictException. The priority argument specifies the Base Priority at which the SporadicTask's work() method executes each time the corresponding interrupt is triggered.

SporadicTask.trigger(). The trigger() method allows software to trigger execution of this sporadic task. Each invocation of the trigger() method is queued. The SporadicTask object remembers the number of pending work() invocations and decrements this count each time it completes an execution of work(). If no work() invocations are pending, this task suspends itself awaiting a subsequent invocation of trigger(). The Core signature for the trigger() method is shown below:

```
public final void trigger();
```

SporadicTask.work(). The Core Execution Environment shall invoke the work() method each time the sporadic task is triggered. The default implementation of the work() method simply returns void. The Core signature is shown below:

```
public synchronized void work();
```

SporadicTask.pendingCount(). The pendingCount() method returns the difference between the number of times this task has been triggered (by invoking its trigger() method) and the number of times this task has completed execution of its work() method in response to previous trigger() invocations. Note that pendingCount() treats a triggered invocation as still pending until the triggered work() invocation completes. The Core signature is shown below:

```
public final int pendingCount();
```

SporadicTask.clearPending(). The clearPending() method clears all pending invocations of this task's work() method except for the currently executing work() invocation, if any. Immediately following execution of clearPending(), pendingCount() returns zero if this task is not currently executing its work() method and one otherwise. The Core signature is shown below:

```
public final void clearPending();
```

3.17.26 The IOPort class

A frequent need of embedded and real-time programmers is to be able to transfer data into and out of physical device ports that are seen by the embedded processor as I/O ports or memory-mapped I/O channels. This class, and its subclasses, provide the ability to perform these actions.

There are many subclasses of IOPort, each one named according to the following template:

IOPort*<port-width><permissions>I*

Copyright 1999, 2000 J Consortium, All Rights Reserved

Within this template, *<port-width>* is replaced with 8, 16, 32, or 64 representing 8-bit, 16-bit, 32-bit, and 64-bit ports respectively. *<permissions>* is replaced with I, O, or IO, representing permission to read only, write only, or both read and write. For example, the class IOPort8O represents an 8-bit output-only port. All methods of the IOPort subclasses are final.

There is no constructor for IOPort or for any of its subclasses. Instead, IOPort provides a static factory method named createIOPort(). Given that the arguments to createIOPort() specify the port width and permissions, createIOPort() returns an instance of the IOPort subclass which represents the requested port width and I/O permissions.

IOPort.createIOPort(). Use this method to create instances of an IOPort subclass class. Each instance of an IOPort subclass is configured with permissions to perform a restricted subset of the full IOPort API. For example, instances of IOPort8O only permit 8-bit output operations. For instances of IOPort8O, all other I/O services (input operations, and operations that attempt to transfer 16, 32, or 64 bits) terminate by throwing CoreOperationNotPermittedException. The Core signature for createIOPort() is shown below:

```
public static IOPort createIOPort(long address, boolean memory_mapped, int port_width,
                    boolean read_permission, boolean write_permission)
                    throws CoreEmbeddedConflictException;
```

Instead of returning the requested IOPort object, createIOPort() throws CoreEmbeddedConflictException if the Core Execution Environment cannot grant the requested I/O access. The conditions under which createIOPort() might throw this exception are implementation-defined.

IOPort.readByte(). The readByte() method fetches an 8-bit value from the corresponding port, assuming this is an instance of IOPort8I or IOPort8IO. In all other cases, this method terminates by throwing a previously allocated instance of CoreOperationNotPermittedException. The Core signature is shown below:

```
public byte readByte() throws CoreOperationNotPermittedException;
```

IOPort.writeByte(). The writeByte() method stores an 8-bit value to the corresponding port, assuming this is an instance of IOPort8O or IOPort8IO. This method returns the value of its single argument. In all other cases, this method terminates by throwing a previously allocated instance of CoreOperationNotPermittedException. The Core signature is shown below:

```
public byte writeByte(byte b) throws CoreOperationNotPermittedException;
```

IOPort.readShort(). The readShort() method fetches a 16-bit value from the corresponding port, assuming this is an instance of IOPort16I or IOPort16IO. In all other cases, this method terminates by throwing a previously allocated instance of CoreOperationNotPermittedException. The Core signature is shown below:

```
public short readShort() throws CoreOperationNotPermittedException;
```

IOPort.writeShort(). The writeShort() method stores a 16-bit value to the corresponding port, assuming this is an instance of IOPort16O or IOPort16IO. This method returns the

value of its single argument. In all other cases, this method terminates by throwing a previously allocated instance of CoreOperationNotPermittedException. The Core signature is shown below:

```
public short writeShort() throws CoreOperationNotPermittedException;
```

IOPort.readInt(). The readInt() method fetches a 32-bit value from the corresponding port, assuming this is an instance of IOPort32I or IOPort32IO. In all other cases, this method terminates by throwing a previously allocated instance of CoreOperationNotPermittedException. The Core signature is shown below:

```
public int readInt() throws CoreOperationNotPermittedException;
```

IOPort.writeInt(). The writeInt() method stores a 32-bit value to the corresponding port, assuming this is an instance of IOPort32O or IOPort32IO. This method returns the value of its single argument. In all other cases, this method terminates by throwing a previously allocated instance of CoreOperationNotPermittedException. The Core signature is shown below:

```
public int writeInt() throws CoreOperationNotPermittedException;
```

IOPort.readLong(). The readLong() method fetches a 64-bit value from the corresponding port, assuming this is an instance of IOPort64I or IOPort64IO. In all other cases, this method terminates by throwing a previously allocated instance of CoreOperationNotPermittedException. The Core signature is shown below:

```
public long readLong() throws CoreOperationNotPermittedException;
```

IOPort.writeLong(). The writeLong() method stores a 64-bit value to the corresponding port, assuming this is an instance of IOPort64O or IOPort64IO. This method returns the value of its single argument. In all other cases, this method terminates by throwing a previously allocated instance of CoreOperationNotPermittedException. The Core signature is shown below:

```
public long writeLong() throws CoreOperationNotPermittedException;
```

3.17.27 Core Throwable Types

The Core specification distinguishes four broad classes of throwable types.

1. CoreThrowable: org.rtjwg.CoreThrowable extends org.rtjwg.CoreObject. Within the Core Execution Environment, only CoreThrowable objects shall be thrown and caught. If a method declares itself to throw objects, the type of the thrown object shall be CoreThrowable or one of its derivatives.
2. CoreException: org.rtjwg.CoreException extends org.rtjwg.CoreThrowable. Within the Core Execution Environment, CoreException is used to indicate a throwable object that typical Core components would want to catch. The Core Verifier shall enforce that any context that might throw a CoreException object declares in its signature that it does so. Further, the Core Verifier shall enforce that any context that invokes a method that might throw a CoreException object either catches the CoreException object or declares that it propagates the thrown CoreException object. In the common vernacular, a CoreException is a “checked” exception.

Copyright 1999, 2000 J Consortium, All Rights Reserved

3. CoreRuntimeException: org.rtjwg.CoreRuntimeException extends org.rtjwg.CoreThrowable. Within the Core Execution Environment, CoreRuntimeException is used to indicate a throwable object that typical Core components would probably not want to catch. The Core Verifier shall not require that contexts that might throw a CoreRuntimeException object declare that they do so. Further, the Core Verifier shall not require contexts that invoke methods that might throw CoreRuntimeException objects to catch the object or to declare that the context propagates the thrown CoreRuntimeException object. In the common vernacular, a CoreRuntimeException is an "unchecked" exception.
4. ScopedException: org.rtjwg.ScopedException extends org.rtjwg.CoreException. A ScopedException object is special in that when thrown, it is only "catchable" by catch clauses belonging to the method within which the ScopedThrowable object was enabled. The intended uses of ScopedException objects are as follows:
   - a. A routine that anticipates the need to establish a special asynchronous event handler which will cause abortion of a particular scoped region of code constructs a ScopedException object, establishes a scope-specific ATCEventHandler to throw this ScopedException object, and initiates execution of the scoped context.
   - b. When the asynchronous ATCEventHandler is signaled, its handleATCEvent() method throws the previously created ScopedException object.
   - c. In processing this thrown exception, the Core Execution Environment does not allow any intervening scopes to "see" the thrown exception. Even a catch clause that is declared to catch any CoreException object does not match this thrown exception. The only catch clause that are allowed by the Core Execution Environment to see the thrown exception are the catch clauses found within the method that constructed the exception.

   ScopedException supports two methods that are not supported by CoreException: enable() and disable(). The Core signatures are as follows:

   ```
   public final void enable();          // enable this ScopedException
   public final void disable();         // disable this ScopedException
   ```

   The special semantics of these two methods are as described here:

   - d. If an ATCEventHandler attempts to throw a ScopedException that has been disabled, the effect is to simply return from the ATCEventHandler (returning to the code that had been executing so it can resume execution, as if the event had never been signaled).
   - e. The activation frame from within which a ScopedException is enabled represents the only scope that can catch the exception. A catch clause contained within any other invoked method's activation frame is unable to see this scoped exception. If an object is enabled multiple times, the most recent enabling is the one that establishes its context. Enabling and disabling of ScopedException objects does not nest.
   - f. When a ScopedException is instantiated, it is automatically enabled in the context from within which the object was constructed.
   - g. Whenever a method's activation frame is removed from the run-time stack, all of the ScopedException objects that are enabled for that specific activation frame are automatically disabled. This is done atomically with respect to handling of nested asynchronous transfer of control events.

Note that the Core Execution Environment does not support the analog of java.lang.RuntimeException, which extends java.lang.Exception but is unchecked.

Table 4 on page 102 details the various CoreThrowable classes that are part of the official Core specification. In all cases, each of these CoreThrowable classes supports two constructors, one taking no arguments and the other taking a single CoreString object which represents the message to which this CoreThrowable object shall maintain a reference.

TABLE 4. Core CoreThrowable Classes

Class Name	Super Class	Usage
CoreIllegalMonitorStateException	CoreRuntimeException	Thrown if a wait() or synchronization request violates the rules for priority inheritance or priority ceiling synchronization.
CoreOutOfMemoryException	CoreRuntimeException	Thrown if a request to allocate memory cannot be immediately satisfied because of insufficient memory.
CoreArrayIndexOutOfBoundsException	CoreRuntimeException	Thrown if a response to an array (or CoreString) subscripting operation reaches beyond the length of the array.
CoreClassFormatError	CoreRuntimeException	Thrown if an attempt to dynamically load a class cannot be satisfied because the class, or one of the class it refers to, is of improper format or because it fails byte-code verification.
CoreOperationNotPermittedException	CoreException	Thrown if a particular operation is not permitted (or supported) for a particular combination of parameters and/or system state.
CoreSecurityException	CoreException	Thrown if request to create a task or to change the priority or period of a task is not permitted for the requesting component.
CoreBadPriorityException	CoreException	Thrown if a request to set the priority of a Core task is outside the range of valid Core priorities.
CoreBadArgumentException	CoreException	Thrown if an argument to a method has an unacceptable value.
CoreEmbeddedConflictException	CoreException	Thrown if a request to obtain access to a particular I/O resource (such as an interrupt vector) conflicts with some other software component's access to the same resource.
CoreATCEventsIgnoredException	CoreException	Thrown if a component requests to signal an asynchronous event or to change the asynchronous event handler for a Core task that does not support asynchronous event handling.
CoreUnsignedCoercionException	CoreException	Thrown if a request to coerce an unsigned integer to a smaller size unsigned integer overflows the capacity of the smaller integer.

Copyright 1999, 2000 J Consortium, All Rights Reserved

TABLE 4. Core CoreThrowable Classes

Class Name	Super Class	Usage
CoreClassInUseException	Core-Exception	Thrown if a request to unload a class cannot be satisfied because the class is currently in use.
CoreClassNotFound-Exception	Core-Exception	Thrown if a request to dynamically load a class cannot be satisfied because the class cannot be found.
CoreArithmeticOverflow-Exception	Core-Exception	Thrown by certain contexts (such as unit conversion operations of the Time class) if arithmetic operations result in overflow.
CoreObjectNot-AddressableException	Core-Exception	Thrown if the CoreObject whose address is requested is not an array of primitive type.

Throughout the Core API description provided in this specification, it is stated that Core methods which throw exceptions do so by throwing "previously allocated instances" of the exceptions. The point in emphasizing this detail is that the act of throwing an exception does not require allocation of any new objects and that the thrown exception object need not be explicitly released. All of the previously allocated exceptions described in the official Core API descriptions shall be allocated once during startup of the Core virtual machine.

4.0 Baseline API

The real-time core has been designed to facilitate cooperation between components written for execution on the Baseline virtual machine and components written for execution within the Core Execution Environment. Core Execution Environments need not support the optional connection to the Baseline virtual machine. When the Core Execution Environment is combined with a Baseline virtual machine, the Baseline API shall support the services described in this section.

As discussed in Section D.2 (starting on page 153), every Core object has two application programmer interfaces, one for the core domain and the other for the Baseline domain. Conceptually, these two interfaces are represented by distinct class representations. Consider, for example, representation of org.rtjwg.CoreObject. Though this class and instances of it reside in the Core domain, this same class is also visible to the Baseline domain. However, the Baseline domain does not know about the variables or the core methods associated with this object.

This section discusses the Baseline domain's view of particular Core objects. In this section, when we speak of CoreObject and CoreClass, we are specifically referring to the Baseline view of those classes. Another subtle issue to emphasize is that even though the Baseline domain cannot see everything that is inside of a core object, if the Baseline domain passes a reference to a core object back into the core domain (by supplying the reference as an argument to a Core-Baseline method), the core domain can see the private information that had been invisible to the Baseline domain.

Semaphore Operations. Baseline components can synchronize with Core components by performing appropriate semaphore operations. See Section 3.17.17 for a description of the Baseline SignalingSemaphore operations and Section 3.17.18 for a description of the Baseline CountingSemaphore operations.

CoreTask Operations. A special service is provided to allow Baseline components to start up core tasks. See "CoreTask._start()" on page 93 for more information on this topic.

Core Execution Profiles. To determine the set of optional profiles that have been installed into a particular Core Execution Environment, the Baseline component invokes the CoreDomain.profiles() method, described in the section titled "CoreDomain.profiles()" on page 106.

Starting Up a Core Execution Environment. To start up a Dynamic Core Execution Environment, the Baseline programmer must load and instantiate the special BaselineCoreClassLoader class, described in Section 4.1.

4.1 The BaselineCoreClassLoader Class

BaselineCoreClassLoader Constructors. There are two constructors for BaselineCoreClassLoader. The first of these takes no arguments. Its Core signature is shown here:

```
public BaselineCoreClassLoader();
```

The first instantiation of the BaselineCoreClassLoader class shall define the mechanism for all Core class loading to be performed within this Core Execution Environment. If the first BaselineCoreClassLoader instance is constructed using the no-argument constructor, the Core Execution Environment shall use the default implementation of CoreClassLoader to search for class files that need to be loaded. As described below, the default CoreClassLoader implementation searches the local file system as directed by the CoreClassPath environment variable for class files to be loaded.

To override the default behavior, a system integrator extends CoreClassLoader, overriding the implementation of findClassBytes(). See Section 3.15 for a more complete description of the CoreClassLoader class. This new implementation of findClassBytes() might search a compressed ROM image, or probe a network class file server, or email a special request to have the class file transmitted over digital wireless carrier. To connect the special implementation of findClassBytes() to the BaselineCoreClassLoader, the system integrator shall supply a reference to the specialized CoreClassLoader object as an argument to the BaselineCoreClassLoader constructor. The signature for the second form of the BaselineCoreClassLoader constructor is shown below:

```
public BaselineCoreClassLoader(CoreClassLoader specialized_ccl);
```

BaselineCoreClassLoader semantics. BaselineCoreClassLoader extends java.lang.ClassLoader. In this regard, it behaves like other Java class loaders. However, there are a number of ways in which BaselineCoreClassLoader is distinct. The special attributes of this class loader are as follows:

Copyright 1999, 2000 J Consortium, All Rights Reserved

1. Insofar as the Baseline domain is concerned, the Baseline interface to all Core classes is represented as if those classes had been loaded by the BaselineCoreClass-Loader class loader.
2. The BaselineCoreClassLoader class loader shall only load classes that correspond to Core objects and selected infrastructure routines (such as the Baseline CoreDomain class). Any attempt to load any other class with the BaselineCoreClassLoader shall abort by throwing a ClassNotFoundException exception.
3. BaselineCoreClassLoader shall be a final class, meaning that it cannot be extended.
4. The constructors for BaselineCoreClassLoader shall load org.rtjwg.CoreDomain, shown below, and all of the other key Core classes that are required for implementation of the Baseline interface to the Core domain, including org.rtjwg.CoreObject, org.rtjwg.CoreThrowable, and org.rtjwgCoreIntArray.
5. The BaselineCoreClassLoader class loader shall perform security manager checks on all class load requests, making sure that the requests originate from within CoreDomain.defineClass() or CoreDomain.loadClass(), or indirectly from within the Core Execution Environment's CoreClassLoader.defineClass() or CoreClassLoader.loadClass() methods. Any other requests to load Core classes shall be refused by throwing a java.lang.SecurityException.

4.2 The CoreDomain Class

The CoreDomain class extends java.lang.Object. The static initializer for the CoreDomain class creates the primordial instance of the CoreDomain class. The CoreDomain class publishes a static variable named core which represents the primordial instance of this class.

```
public final static CoreDomain core;
```

CoreDomain.lookup(). The lookup() method returns the core object that was previously published in the core registry with the specified name, throwing ObjectNotFoundException if no such object exists in the core registry. The Core signature is shown below:

```
public final CoreObject lookup(String name) throws ObjectNotFoundException;
```

CoreDomain.defineClass(). The defineClass() method converts the sub-sequence ranging from position offset to position (offset + len - 1) within the array of bytes b into an instance of class Class. The Class object returned from defineClass() represents the Baseline interface to the newly loaded class. The Core methods of the newly loaded class are not visible to the Baseline domain, so they are omitted from the Baseline class representation. A side effect of loading a Baseline class in this way is that the Core version of the same class is loaded into the Core Execution Environment. The defineClass() method shall throw ClassNotFoundException if any referenced class was not previously loaded. The Core signature is shown below:

```
public final CoreClass defineClass(String name, byte[] b, int off, int len)
            throws ClassNotFoundException;
```

CoreDomain.loadClass(). The loadClass() method shall load the class specified by its name argument, searching for the class file representation according to the strategy represented by this Core Execution Environment's CoreClassLoader implementation. This method shall resolve all referenced classes. The Core signature is shown below:

```
public final CoreClass loadClass(String name) throws ClassNotFoundException;
```

CoreDomain.instantiate(). The instantiate() method instantiates a Core object within the Core heap to implement the CoreClass c. The no-argument constructor for the newly allocated Core object runs as if it were invoked from a Core task. In other words, the constructor is not a Core-Baseline method. The Core signature is shown below:

```
public final CoreObject instantiate(CoreClass c);
```

CoreDomain.profiles(). The profiles() method shall return an array of java.lang.String objects representing the collection of all real-time profiles that are present within this Core Execution Environment. Profile naming conventions serve to differentiate key features of the profiles, as described in the section titled “CoreRegistry.profiles()” on page 78.

The Core signature for the profiles() method follows:

```
public static String [ ] profiles();
```

4.3 The ObjectNotFoundException Class

The org.rtjwg.ObjectNotFoundException class extends java.lang.Exception. This class has a two constructors, one taking no arguments and the other taking a single java.lang.String argument to represent the message associated with this exception. There are no other methods defined for this exception class.

4.4 The CoreBaselineThrowable Class

The org.rtjwg.CoreBaselineThrowable class, which extends java.lang.Throwable, is a Baseline class. If a Core-Baseline method is declared to throw a CoreThrowable object which does not derive from either the CoreRuntimeException or CoreException classes, the Core Class Loader shall represent the Baseline API of this method as throwing CoreBaselineThrowable. At run time, if this method terminates by throwing a CoreThrowable object, the Core Execution Environment shall wrap a CoreBaselineThrowable object around the thrown CoreThrowable object by constructing the CoreBaselineThrowable object, passing a reference to the thrown CoreThrowable object as the sole argument to the CoreBaselineThrowable constructor. The CoreBaselineThrowable object shall be constructed in the Baseline context, and its stack backtrace shall begin at the point of the Core-Baseline method invocation whose execution terminated by throwing the CoreThrowable object.

CoreBaselineThrowable Constructors. There shall be only one constructor for the CoreBaselineThrowable class. This constructor shall require a reference to an org.rtjwg.CoreThrowable object as its sole argument. The Core signature is shown below:

```
public CoreBaselineThrowable(org.rtjwg.CoreThrowable throwable_core_exception);
```

CoreBaselineThrowable.getCoreThrowable(). The getCoreThrowable() method shall return a reference to the org.rtjwg.CoreThrowable object that was supplied as the sole argument to the CoreBaselineThrowable constructor. The Core signature is shown below:

```
final public org.rtjwg.CoreBaselineThrowable getCoreThrowable();
```


Copyright 1999, 2000 J Consortium, All Rights Reserved

4.5 The CoreBaselineRuntimeException Class

The org.rtjwg.CoreBaselineRuntimeException class, which extends java.lang.RuntimeException, is a Baseline class. When a Core-Baseline method terminates by throwing a CoreRuntimeException object, the Core Execution Environment shall wrap a CoreBaselineRuntimeException object around the thrown CoreRuntimeException object by constructing the CoreBaselineRuntimeException object, passing a reference to the thrown CoreRuntimeException object as the sole argument to the constructor. The CoreBaselineRuntimeException object shall be constructed in the Baseline context, and its stack backtrace shall begin at the point of the Core-Baseline method invocation whose execution terminated by throwing the CoreRuntimeException object.

CoreBaselineRuntimeException Constructor. There shall be only one constructor for the CoreBaselineError class. This constructor shall require a reference to an org.rtjwg.CoreRuntimeException object as its sole argument. The Core signature is shown below:

```
public CoreBaselineRuntimeException(
                org.rtjwg.CoreRuntimeException throwable_core_exception);
```

CoreBaselineRuntimeException.getCoreException(). The getCoreException() method shall return a reference to the org.rtjwg.CoreException object that was supplied as the sole argument to the CoreBaselineError constructor. The Core signature is shown below:

```
final public org.rtjwg.CoreBaselineRuntimeException getCoreException();
```

4.6 The CoreBaselineException Class

The org.rtjwg.CoreBaselineException class, which extends java.lang.Exception, is a Baseline class. When a Core-Baseline method terminates by throwing a CoreException object, the Core Execution Environment shall wrap a CoreBaselineException object around the thrown CoreException object by constructing the CoreBaselineException object, passing a reference to the thrown CoreException object as the sole argument to the CoreBaselineException constructor. The CoreBaselineException object shall be constructed in the Baseline context, and its stack backtrace shall begin at the point of the Core-Baseline method invocation whose execution terminated by throwing the CoreException object.

CoreBaselineException Constructors. There shall be only one constructor for the CoreBaselineException class. This constructor shall require a reference to a org.rtjwg.CoreException object as its sole argument. The Core signature is shown below:

```
public CoreBaselineException(org.rtjwg.CoreException throwable_core_exception);
```

CoreBaselineException.getCoreException(). The getCoreException() method shall return a reference to the org.rtjwg.CoreException object that was supplied as the sole argument to the CoreBaselineException constructor. The Core signature is shown below:

```
final public org.rtjwg.CoreBaselineException getCoreException();
```

5.0 Acknowledgments

This work represents the results of many people's efforts, including the various participants in the J Consortium's Real-Time Java Working Group, NewMonics real-time development team, and NewMonics administrative support staff. We thank all for their contributions to this specification.

6.0 Informative References

1. *Requirements For Real-time Extensions For the Java ™ Platform*, edited by Lisa Carnahan and Marcus Ruark, National Institute of Standards and Technologies. April 1999.
2. Java Language Reference, 2nd Edition, by Mark Grand, O'Reilly Publications, July 1997, ISBN 1-56592-326-X.
3. Java Virtual Machine, by Jon Meyer and Troy Downing, March 1997, ISBN 1-56592-194-1.
4. *The Java™ Programming Language, Second Edition*, by Ken Arnold and James Gosling, Addison-Wesley, 1998, 464 pages, ISBN 0-201-31006-6.
5. *The Java Language Specification*, by James Gosling, Bill Joy and Guy Steele, Addison-Wesley, September 1996, ISBN 0-201-63451-1.
6. *The Java Class Libraries Volume 1, Second Edition*, by Patrick Chan, Rosanna Lee and Douglas Kramer, Addison-Wesley, July 1998, ISBN 0-201-31002-3.
7. *The Java Class Libraries Volume 2, Second Edition*, by Patrick Chan and Rosanna Lee, Addison-Wesley, April 1998, ISBN 0-201-31003-1.
8. *The Java Virtual Machine Specification*, by Tim Lindholm, Frank Yellin, Bill Joy, and Kathy Walrath, Addison-Wesley, 1998, 256 pages, ISBN 0-201-63452-X.
9. *Compilers: Principles, Techniques, and Tools*, by Alfred Aho, Ravi Sethi, and Jeffrey Ullman, 1986, 796 pages, ISBN 0-201-10088-6.
10. C ISO/IEC 9899:1990
11. C++ ISO/IEC/14882: 1998
12. Improving the Java Memory Model Using CRF, by Jan-Willem Maessen, Arvind, and Xiaowei Shen, in *Proceedings of the Conference on Object Oriented Programming, Systems, Languages, and Applications*, Minneapolis, Minnesota, 2000.
13. Fixing the Java Memory Model, by William Pugh, in *Proceedings of the ACM Java Grande Conference,* June 1999.
14. The Java Memory Model, Issues and Discussions hosted at http://www.cs.umd.edu/~pugh/java/memoryModel/.

Copyright 1999, 2000 J Consortium, All Rights Reserved

Annex A History

A.1 Revision 1.0.14

This revision represents changes motivated by the possible opportunity to present the Core Specification to ISO under ISO's PAS program. Specific changes are listed below:

1. Removed the word Java from the title and from many of the notational terms used throughout the document. Concern was raised that using Java in the title of an international standard might violate Sun Microsystems trademark guidelines.
2. Various small changes to correct misspellings, cut-and-paste errors, and to improve clarity. These are scattered throughout the document.
3. Reordered the document to move the edit history, requirements, rationale, and implementation suggestions into annexes, removing them from the body of the specification.
4. Removed the notion of Syntactic Core extensions from the Core specification. The use of baseline and stackable keywords is no longer supported as conforming syntax. These words are used only as a notational convenience in presenting Core library signatures.
5. Revised the discussion of conformity assessment (Section 3.1 (starting on page 6)) to make conformity requirements more clear and precise. Removed all syntax dependencies from conformity requirements. Conformity is now defined in terms of class file representations rather than source code syntax.
6. Removed the entire "I/O Subsystem" section from the Core specification. This material was redundant with the specification being developed concurrently by the Real-Time Data Access Working Group. Keeping the efforts of the two groups synchronized was difficult and time consuming. In its place, the Core specification has new simplified definitions of ISR_Task, SporadicTask, and IOPort. We expect that the Real-Time Data Access Working Group will eventually supplement these classes by defining a variant of the Real-Time Access Profile which is designed for integration within a Core Execution Environment.
7. Replaced CoreError with CoreRuntimeException and CoreBaselineError with CoreBaselineRuntimeException. It was felt this represents a better match to the existing experience of current Baseline programmers.
8. Added Section 3.9 (starting on page 24), which clarifies the required scheduling behavior for Baseline threads executing within the Core Execution Environment.
9. Added Section 3.10 (starting on page 24), which discusses briefly the need to clarify the Core Memory Model. This section needs further work.
10. Added discussion of predictability requirements for the C/Native API and for the Baseline API. See Section 3.14.2 (starting on page 34). Updated Table 1 on page 35 to reflect changes to the API and to correct several errors from the previous revision.
11. Replaced the C library function corePriorityInterleave() with corePriorityMap() in Section 3.16 (starting on page 57). Added enterSynchronized() and exitSynchronized() methods to that same section.

A.2 Revision 1.0.13

In preparation for submission to ISO through the PAS program, the Real-Time Data Access Group and Real-Time Java Working Group identified two general areas that could be improved in order to achieve significant quality improvement to the specification. These are:

1. To remove unnecessary reference and dependency on the Baseline specification, and
2. To further unify the Core specification with the evolving specification for the Real-Time Data Access Profile.

This draft represents proposed changes intended to address both of these concerns. The changes identified in this draft have not yet been approved by the Real-Time Java Working Group. The following list identifies a number of additional contemplated changes that have not yet been folded into the document.

1. Add a OneShotEvent class that is similar to PeriodicEvent class except that execution of the corresponding work() method occurs only once each time this event handler is enabled. Following the one-time execution of the event handler, the event handler automatically disables itself. The OneShotEvent class is patterned after the class by the same name which is defined in the Real-Time Data Access specification version 1.5.
    a. It shall be implementation-defined when a OneShotEvent handler's work() method is invoked relative to the timing of any fixed-period timer ticks that might be part of the system. In particular, the work() invocation may either precede or trail the periodic delay by up to one full period.
2. For all kinds of events (PeriodicEvent, OneShotEvent, SporadicEvent, and InterruptEvent), any such events that are triggered while that event is disabled shall be ignored. This represents a change in the specified behavior for SporadicEvent.
3. There are a number of contemplated changes regarding queueing and buffer overrun:
    a. For all kinds of events (PeriodicEvent, OneShotEvent, SporadicEvent, and InterruptEvent), add enableQueue() and disableQueue() methods. While queueing is enabled, each event maintains a count of how many times it has been triggered and not serviced. Note that each event has a single event handler.
    b. The meaning of Event.disable() is to prevent new events from being queued.
    c. For all kinds of events, the enable() and disable() methods may involve interaction with the operating system, which may result in error conditions being signaled by the operating system. For this reason, the enable() and disable() methods are now declared to throw CoreOperationFailedException.
    d. The meaning of Event.disableQueue() is also to prevent new events from being queued. Additionally, Event.disableQueue() wipes the event queue clean.
    e. Add an onError() method to each of the EventHandler classes.
    f. Add an error() method to each kind of event. This method returns an integer code representing the reason that the onError() method was invoked. A special error code named OverrunError is defined to equal one (1) in IOEventHandlerInterface. Other error codes remain to be specified. The intent is that we will specify additional error codes for publication in this specification.

Copyright 1999, 2000 J Consortium, All Rights Reserved

g. If the work() method of an event handler is still running and the one of the Events that this event handler handles is enabled, and queueing is disabled for that event when a new event arrives, we invoke the onError() method for the corresponding EventHandler object.

h. Eliminate the numberOverruns() method from PeriodicTask.

4. Add an IODescription class patterned after the class by the same name in the Real-Time Data Access profile specification, version 1.5.
5. Add an enumerate() method to IONodeLeaf, which returns an array of IODescription objects representing all of the proxies and events that have been created by this IONodeLeaf object.
6. Add an event handler to the IOChannel proxy objects. Specify this as an argument to the createIO() invocation.
7. Add a Version class, which has the following final fields:

```
CoreString spec_no                  // which version of the specification does
                                    // this implement?
CoreString vendor_name;             // Which vendor supplied this class?
CoreString vendor_version;          // What is vendor's version number for
                                    // this product?
```

8. Add an IOEventHandler class from which PeriodicTask, InterruptTask, and SporadicTask derive. This is patterned after the class by the same name in the Real-Time Data Access profile specification, version 1.5.

   a. Tentatively, add a reference to SymbolTable as an argument to the various createXX() methods. This argument may be null, signifying that the object to be created is not named symbolically.

9. Move pendingCount() to EventHandler class and remove this method from Event classes.
10. Give the IONode constructor an argument named driver_name, of type CoreString, which represents the name of a system driver which shall be initialized by this constructor. The driver_name argument shall name a Java class which implements the necessary driver services in an implementation-defined manner. This is patterned after the same concept as described in the Real-Time Data Access profile specification, version 1.5.
11. A change is proposed to the range checking associated with the IONodeLeaf.createIO() method, which would read as follows: “If the mode attribute’s IOInterface.IOMemoryMapped bit is set and the sum of the I/O channel’s offset attribute with the range calculated above exceeds the value returned from the memoryRange() method invoked on this IONodeLeaf object, throw an instance of OperationNotPermittedException. Note that the memory range of this IONodeLeaf is computed by subtracting this IONodeLeaf’s base address from the memory range of this IONodeLeaf’s parent.)

A.3 Revision 1.0.12

Upon distribution of revision 1.0.11, Omron observed inconsistencies in the signatures of the value() methods of the IOChannel subclasses. In some cases, these were declared to throw an exception, and in other cases, they were declared to not throw an exception.

That was an editing oversight. In this revision, all of the value() methods are declared to not throw an exception.

A.4 Revision 1.0.11

During the 30-day strategic review period initiated following the Jan. 31, 2000 J Consortium board meeting, a number of oversights and small errors were uncovered. These are addressed in this “errata” revision.

1. In Section C.15 (starting on page 145), the sample code did not compile. We found it necessary to capitalize Class and change the initial assignment to the cd variable. Also, we added constructor arguments to the invocation of CoreClass.instantiate().
2. In Section C.17 (starting on page 146), we modified the sample timeout code and the description thereof to improve clarity.
3. In Section 3.5 (starting on page 15), paragraph 20, removed mention of the Atomic interface. This change was supposed to have been incorporated into revision 1.0.3 of this document.
4. In the PeriodicEvent class description (since removed from this document), remove the numberOverruns() method as this is redundant with PeriodicTask.numberOverruns().
5. In the IONodeLeaf class description (since removed from this document), remove the readable and writeable arguments of the createIO??() method. Add discussion of the "proxy" attribute for IONodeLeaf descriptions. In this same section, add an InterruptTask argument to the IONodeLeaf.createInterrupt() method.
6. In Section 4.2 (starting on page 105), we clarified that the constructor triggered by execution of CoreDomain.instantiate() runs as a Core task, and not as a Baseline thread executing a Core-Baseline method.
7. A number of minor typographic errors were corrected.

A.5 Revision 1.0.10

The J Consortium Board met on Jan. 31, 2000 and approved the start of the 30-day strategic review period concurrent with publication of the specification for additional public review. Prior to beginning this review, board members requested that a small number of minor errors and oversights be corrected. This is what was addressed with this Revision 1.0.10.

A.6 Revision 1.0.9

The J Consortium Technical Committee met on Jan. 27, 2000 and approved revision 1.0.8 for submission to the J Consortium board to begin the 30-day strategic review. In preparation for that review, a small number of minor typographic errors and editing oversights were corrected, resulting in revision 1.0.9. Additionally, the following changes, each of which had been discussed previously by participants of the Real-Time Java Working Group but which were accidentally omitted from subsequent drafts of the specification, were incorporated:

1. Allow nesting of PCP synchronization locks. This change is reflected in Section 3.17.10 (starting on page 71) and discussed in Section C.11 (starting on page 144).

Copyright 1999, 2000 J Consortium, All Rights Reserved

2. Removed the prohibition on invocation of methods from within finally clauses. This change is reflected in Section 3.5 (starting on page 15) and in Section 3.11 (starting on page 25).

A.7 Revision 1.0.8

A meeting of the Real-Time Java Working Group was held on Jan. 25, 2000. As a result of that meeting, the following additional revisions were made to this specification and the resulting specification was forwarded to the Technical Committee of the J Consortium to be advanced to its next milestone.

1. Remove “draft” from the title of Section 3.0 (starting on page 6). Also, replace a number of occurrences of the word “draft” with the word “revision”.
2. Add IOEventInterface.PeriodicEventCode to the list of special cases associated with invocations of IOEventInterface.fire(). This change is reflected in the description of the IOEventInterface interface, which was removed in a subsequent revision of this document.
3. Replace IONodeLeaf.createIOxxx() with IONodeLeaf.createIO??(). This change is reflected in the description of the IONodeLeaf class, which was removed from a subsequent revision of this document, and in assorted other locations that make use of this name.
4. Add to description of IONodeLeaf.createIO??() that if the IOChannel object is created with implicit_io argument set to true, the created IOChannel object’s read() and write() methods throw CoreOperationNotPermittedException. This change is reflected in the description of the IONodeLeaf class, which was removed from a subsequent revision of this document.
5. In the description of the IODeviceDescription class, which was removed from a subsequent revision of this document, remove redundant reference to “memory-mapped I/O addresses”. In this same section, clean up the wording of numbered paragraph 1.
6. For IODeviceDescription objects that represent I/O channels, replace the "range" attribute with an "entries" attribute. Change the meaning from size of spanned address space measured in bytes to number of entries spanned by this multi-port channel, each entry representing a scalar I/O channel of the width specified by the IODeviceDescription’s "mode" attribute. This change is reflected in the descriptions of the IODeviceDescription and IONodeLeaf classes, both of which were removed from a subsequent revision of this document.
7. In the description of IOInterface.mode() (subsequently changed to IODescription.mode()), explain that the value() methods transfer a block of data as a complete array if the IOArrayAccess bit is set for a particular IOInterface object. This change is reflected in the description of the IOInterface interface, which was removed from a subsequent revision of this document.
8. Add a “Scope” section, as Section 1.0 (starting on page 1).
9. A few typographic errors were corrected.

A.8 Revision 1.0.7

Based on a meeting of the Real-Time Java Working Group which was held on Jan. 21, 2000, the following additional revisions were made to this specification.

1. Add to IOEventHandlerInterface and to the classes that implement this interface a method named handleEvent(). This method has the effect of setting the event for this invocation of work(), triggering execution of the work() method, and then waiting for the work() method to complete its processing. The handleEvent() method is synchronized in the following sense: Once the handleEvent() method has been invoked, no other invocations of handleEvent() are allowed to overwrite the value of the set event until the corresponding invocation of work() completes. Having introduced the handleEvent() method described above, remove the setEvent() method from IOEventHandlerInterface and the classes that implement this interface. These changes are reflected in sections treating IOEventHandlerInterface, PeriodicTask, and SporadicTask, all of which were removed from a subsequent revision of this document.
2. Remove the explicit constructor from the InterruptEvent class. This change is reflected in the section treating InterruptEvent, which was removed from a subsequent revision of this document.
3. Change the constructor for IONodeLeaf to take a single integer interrupt number rather than a string that potentially represents multiple interrupt numbers. If multiple interrupt numbers need to be associated with a particular device, application developers must describe that device using multiple IODeviceDescription objects, one for each of the distinct interrupt numbers. These changes are reflected in the section describing IONodeLeaf, which was removed from a subsequent revision of this document.
4. Add an exchangeInterruptNumber() method to IONodeLeaf. This has the effect of replacing the value of the interrupt number associated with the IONodeLeaf object. The replacement is atomic with respect to invocations of the createInterrupt() method. This change is reflected in the section describing IONodeLeaf, which was removed from a subsequent revision of this document.
5. Remove the length argument of the IONodeLeaf.createIOxxx() method. Instead, compute the length based on the value of the "range" attribute of the corresponding entry within the IODeviceDescription object. This change is reflected in the section describing IONodeLeaf, which was removed from a subsequent revision of this document.
6. When creating an I/O channel using the createIOxxx() method, clarify the meaning of a special slash character ('/') within the entry_name argument. In particular, the decimal digits that follow the slash shall represent an offset relative to the base address of this I/O channel, which offset is measured in terms of the data transfer size associated with this channel (i.e. an offset of 3 for a 32-bit channel represents a byte-offset from the base memory address of 12). Further, modify the specification so that when creating an I/O channel using the special slash character entry naming convention, the created I/O channel represents only a single data value rather than an array of data values. This change is reflected in the section describing IONodeLeaf, which was removed from a subsequent revision of this document.
7. Throughout the document, use the phrase "memory-mapped access" to describe access to memory-mapped I/O channels, use the phrase "I/O-space access" to describe access to I/O ports residing in I/O space, and use the term "I/O channel" to

Copyright 1999, 2000 J Consortium, All Rights Reserved

represent either or both. These changes are reflected throughout the document. Appropriate definitions were added to Section 2.2 (starting on page 2).

8. Analogous to the "timer" attribute for IONodeLeaf.createPeriodic() and the "trigger" attribute for IONodeLeaf.createSporadic() methods, define a special "trigger" attribute for IONodeLeaf.createInterrupt(). This attribute shall either have the value "Interrupt-Event", or it shall hold the name of a class that extends from InterruptEvent. This change is reflected in the section describing IONodeLeaf, which was removed from a subsequent revision of this document.
9. Add the IOImplicit and IOExclusive symbolic constants back into the definition of the IOInterface interface. These are provided for convenience of the application developer and have no meaning insofar as built-in APIs are concerned. Add a symbolic constant named IOArrayAccess which identifies I/O channels that can be treated as I/O arrays. These changes are reflected in the sections describing IONodeLeaf and IOInterface, both of which were removed from a subsequent revision of this document.
10. Add enable() and isEnabled() methods to IOInterface and the classes that implement this interface. This change is reflected in the sections describing IOInterface and IOChannel, both of which were removed from a subsequent revision of this document.
11. Add a constructor to allow new IODeviceDescription objects to be created and added to the system dynamically. Also, add a method to allow such dynamically added IODeviceDescription objects to be removed from the system. These capabilities need not be supported by all conforming implementations. If dynamic manipulation of the IODeviceDescription database is not supported, invocations of the constructor and removal method throw CoreOperationNotPermittedException. Further, add a dynamic_devices variable to the Configuration class and a static dynamicDevices() method to the IODeviceDescription class which represent whether or not this system allows IODeviceDescription objects to be added while the system is running. These changes are reflected in the section describing IODeviceDescription, which was removed from a subsequent revision of this document, and Section 3.17.20 (starting on page 81).
12. Add clarification re: address arithmetic for IOChannel nodes. In particular, base memory and I/O addresses are expressed in terms of byte addresses. However, when using the forward slash convention to name an entry argument for a createIOxxx() invocation, the offset number is expressed in terms of the channel size. So, for example, if the IODeviceDescription entry named dma_buffer represents 512 32-bit integers, and an application invokes createIO8Bit(), with "dma_buffer/8" as its first argument, the resulting IO8Bit object refers to the ninth integer in the dma_buffer, which is found at byte offset 32 relative to the beginning of the dma_buffer address range. For another example, suppose that we create an IO8Bit object to represent the entire dma_buffer by invoking createIO8Bit() with "dma_buffer" as its first argument. Invoking read(8) on the resulting IO8Bit object fetches the ninth integer (found at byte offset 32 from the base address) of the dma_buffer memory range. These changes are reflected in the section describing IONodeLeaf, which was removed from a subsequent revision of this document..
13. Add clarification re: endian behavior of I/O operations. In particular, all multi-byte values are transmitted to and from I/O channels using the representation that is most natural for a given platform. Add a little_endian variable to the Configuration

class and a static littleEndian() method to the IODeviceDescription class, both of which are true if the natural representation on this platform is little-endian, and false otherwise. These changes are reflected in Section 3.17.20 (starting on page 81) and the section describing IODeviceDescription, which was removed from a subsequent revision of this document.

14. On the cover page, removed "draft" from the title and other cover material, and added trademark symbol and trademark attribution for the Java trademark. Removed the word "draft" from the document's abstract.
15. Throughout the document, changed the footer to say Copyright 1999, 2000 on all even-numbered pages.
16. Renamed the readDevice() method to update(). Renamed the writeDevice() method to flush().
17. Assorted typographic errors were corrected.

A.9 Revision 1.0.6

In response to comments received at the Jan. 14, 2000 meeting of the Real-Time Java Working Group, the following additional revisions were made to this specification.

1. Clarify that the IOEventHandlerInterface.setEvent() method is called automatically before calling IOEventHandlerInterface.work() each time an event triggers execution of this event handler. Add a protected method to PeriodicEvent to allow subclasses to trigger the start of each new period. These changes are reflected in the sections describing IOEventHandlerInterface and PeriodicEvent, both of which were removed from subsequent revisions of this document.
2. Do not require that 1-bit IOChannel objects be implemented using implicit reading and writing. This change is reflected in the section describing IOChannel, which was removed from a subsequent revision of this document.
3. Create new IOChannel sub-classes to represent block-transfer I/O operations, as an addition (not a replacement) to existing capabilities. For example:

   ```
   class IO8BitArray {
     public byte[] value();
     public void value(byte [] b);
   }
   ```

   These changes are reflected in sections describing the various sub-classes of IOChannel, all of which were removed from a subsequent revision of this document.
4. For the various I/O proxy classes (IOChannel and all of its descendants), rename the existing read() and write() methods to be readDevice() and writeDevice() methods. Then add the following methods to the subclasses:
   a. read(): has effect of atomically performing a readDevice() operation followed by a value() operation.
   b. read(offset): has effect of atomically performing a readDevice(offset) operation followed by a value() operation.
   c. write(value): has effect of performing a value(val) operation followed by a writeDevice() operation.
   d. write(value, offset): has effect of performing a value(val) operation followed by a writeDevice(offset) operation.

Copyright 1999, 2000 J Consortium, All Rights Reserved

These changes are reflected in the sections describing IOChannel and its subclasses, all of which were removed from a subsequent revision of this document.

5. Establish better consistency between the use of interfaces and the use of classes. Note that we have IOEventHandlerInterface which is implemented by SporadicTask, PeriodicTask, and InterruptTask; and we have IOEventInterface which is implemented by InterruptEvent, SporadicEvent, and PeriodicEvent. We should also have IOInterface which is implemented by IOChannel. These changes are reflected in the sections describing IOInterface and IOChannel, both of which were removed from a subsequent revision of this document.
6. Add a disable() method to IOInterface and IOChannel. This change is reflected in the sections describing IOInterface and IOChannel, both of which were removed from a subsequent revision of this document.
7. Use a special subclass of IONode named IONodeLeaf to represent leaf nodes within the IONode hierarchy. These are different from interior nodes in the following respects:
   a. Only leaf nodes have an associated IODeviceDescription object.
   b. Only leaf nodes keep track of which interrupt numbers are associated with the node. Since multiple interrupts may be associated with a given device, this information is represented as a string, encoded in the leaf IONode's constructor according to the conventions demonstrated in the following example:

      ```
      int-1:int-2:int-3=5:1:1
      ```

      This example shows three interrupts, named int-1, int-2, and int-3, which are associated with interrupt numbers 5, 1, and 1 respectively. The corresponding IODeviceDescription object should have entries by these same names, with each entry having an attribute named "type", for which the associated value is "Interrupt".
   c. When leaf nodes are constructed, they do not need to specify mem_range and io_range arguments. These ranges are instead represented in the corresponding IODeviceDescription object.
   d. Only leaf nodes are allowed to create IOChannel proxies (instantiate subclasses of IOChannel).

   These changes are reflected in the sections describing IONode and IONodeLeaf, both of which were removed from a subsequent revision of this document.
8. Replace the IONodeLeaf.createIO() method with multiple methods, each one returning an instance of a different IOChannel subclass. Each of these methods takes arguments indicating:
   a. Whether readDevice() and writeDevice() operations on the IOChannel object are implicit or explicit.
   b. Whether the IOChannel object represents read permission.
   c. Whether the IOChannel object represents write permission.
   d. Whether the IOChannel object represents exclusive access to the given channel.

   Further, remove implicit and exclusive mode information from the IODeviceDescription representation.

   These changes are reflected in the sections describing IONode and IOInterface, both of which were removed from a subsequent revision of this document.

9. Fix the descriptions of IONode.createIOxxx() and IONode.createInterrupt(). The current revision says createIO() instantiates an InterruptEvent and createInterrupt() instantiates a subclass of IOChannel. Reverse these. These changes are reflected in the section describing IONodeLeaf, which was removed from a subsequent revision of this document.
10. For all of the IONodeLeaf.createIOxxx() operations, use the entry-name within the corresponding IODeviceDescription to identify the I/O channel to be created. Allow a forward slash followed by a sequence of decimal digits to be appended to the end of the entry name. If present, this sequence of digits represents an offset from the base address associated with the channel range. For example, the following two code sequences are equivalent:

```
// Version 1
   IO1Bit m_proxy = IONodeLeaf_xx.createIO1Bit("entry-name", ...);
   m_proxy.readDevice(7);
// Version 2
   IO1Bit n_proxy = IONodeLeaf_xx.createIO1Bit("entry-name/7", ...);
   n_proxy.readDevice();
```

   This change is reflected in the section describing IONodeLeaf, which was removed from a subsequent revision of this document.
11. Add a new constructor for IONode and IONodeLeaf which does not include arguments to specify the io_offset and io_range arguments. If these are not specified, they default to 0 and the size of parent node's io_range respectively. For the root node, which doesn't have a parent, these default to represent the beginning and end of the range of valid I/O-space addresses for the host platform. These changes are reflected in the sections describing IONode and IONodeLeaf, both of which were removed from a subsequent revision of this document.
12. Add an IONodeLeaf.createPeriodic() method. Among its arguments is an entry name. The named entry must have an attribute named "type" with value "Periodic". Additionally, the entry must have an attribute named "timer" for which the string argument is the name of the class to be instantiated. If the class is named "PeriodicEvent", the created PeriodicEvent shall use the default system timer. Otherwise, the named class must be a subclass of PeriodicEvent, and may use a different timer than the system default. The public constructor for PeriodicEvent has been removed. This change is reflected in the section describing IONodeLeaf, which was removed from a subsequent revision of this document.
13. Add an IONode.createSporadic() method. Among its arguments is an entry name. The named entry must have an attribute named "type" with value "Sporadic". Additionally, the entry must have an attribute named "trigger" for which the string argument is the name of the class to be instantiated. If the class is not named "SporadicEvent", the named class must be a subclass of "SporadicEvent". The public constructor for SporadicEvent has been removed. This change is reflected in the section describing IONode, which was removed from a subsequent revision of this document..
14. Change the conventions for representation of information within IODeviceDescription.
    a. For entries that represent I/O proxies, there shall be no required attribute named "address". Instead, there shall be an attribute named "offset", whose value is the byte offset relative to the corresponding IONode's base address of this

Copyright 1999, 2000 J Consortium, All Rights Reserved

channel, encoded as a sequence of lower-case hexadecimal digits with a leading "0x" prefix. If this IODeviceDescription's "mode" attribute has the IOMemoryMapped bit set, the "offset" field is computed relative to the IONode's base memory address. Otherwise, the "offset" field is computed relative to the IONode's base I/O address.

b. For entries that represent I/O proxies, the "mode" attribute shall encode only the values of the IO1Bit, IO8Bit, IO16Bit, IO32Bit, IO64Bit, IOReadPermission, IOWritePermission, and IOMemoryMapped bit fields. It shall not represent the values of the IOImplicit and IOExclusive fields.

c. If a particular entry represents an interrupt vector, it must have an attribute named "type" with value equal to "Interrupt". The interrupt number associated with this interrupt shall be determined by the corresponding IONode's representation.

These changes are reflected in the sections describing IODeviceDescription and IOInterface, both of which were removed from a subsequent revision of this document.

15. IODeviceDescription should specify the range of memory and I/O addresses relative to the parent's respective base addresses. Thus, there is no need to supply range arguments when constructing a leaf node of the IONode hierarchy. These changes are reflected in the sections describing IONodeLeaf and IODeviceDescription, both of which were removed from a subsequent revision of this document.

16. Make the Core Verifier be required in any conforming implementation of the Core development environment. In particular, Core Verification must be performed on each Core program before execution of that program. These changes are reflected in Section C.4 (starting on page 138) and Section 3.5.1 (starting on page 18).

17. Change the behavior of CoreTask.setPriority(). If the task for which setPriority() is invoked is running within a priority ceiling context when setPriority() is invoked, the effect of setPriority() shall be deferred until after that task leaves its priority ceiling context. This change is reflected in Section 3.17.23 (starting on page 88).

18. The previous revision of the specification states that time slicing shall be inhibited while the currently executing task executes within a priority ceiling context. While this is a reasonable implementation, it is not the only feasible way to implement the desired semantics. The key requirement is to enforce that priority ceiling regions are executed with mutual exclusion, and leave it to the discretion of implementors to enforce this behavior. This change is reflected in Section 3.8 (starting on page 21).

19. The constructor for InterruptTask should not take an ATCEventHandler argument, since the InterruptTask's work method always runs to completion with asynchronous event handling deferred. This change is reflected in Section 3.17.24 (starting on page 94).

20. Change ScopedException to extend CoreException instead of CoreError. This change is reflected in Section 3.17.27 (starting on page 100).

21. Add enable() and disable() methods to ScopedException. These have the following semantics:

a. If an ATCEventHandler attempts to throw a ScopedException that has been disabled, the effect is to simply return from the ATCEventHandler (returning to the code that had been executing so it can resume execution, as if the event had never been signaled).

b. The activation frame from within which a ScopedException is enabled represents the only scope that can catch the exception. A catch clause contained within any other invoked method's activation frame is unable to see this scoped exception. If an object is enabled multiple times, the most recent enabling is the one that establishes its context. Enabling and disabling of ScopedException objects does not nest.

c. When a ScopedException is instantiated, it is automatically enabled in the context from within which the object was constructed.

d. Whenever a method's activation frame is removed from the run-time stack, all of the ScopedException objects that are enabled for that specific activation frame are automatically disabled. This is done atomically with respect to handling of nested ATC events.

These changes are reflected in Section 3.17.27 (starting on page 100).

22. Update Table 1 on page 35 to represent all of the methods of all classes in the Core API libraries.

23. Several typographic errors were corrected.

A.10 Revision 1.0.5

During the week of Jan. 10, 2000, members of the Real-Time Java Working Group were asked to review revision 1.0.4 and the public review comments in preparation for finalizing the specification. This revision results from observations made by participants of the Real-Time Java Working Group during this review period. This revision has not been approved by the Real-Time Java Working Group membership.

1. The core NIST requirements state that the core specification must identify the resource requirements associated with services provided within the real-time core execution environment. This has been missing from previous versions of the core specification. Add it. This change is reflected in Section 3.14.2 (starting on page 34).
2. Exchange the definitions of CoreTask.stackSize() and CoreTask.stackDepth(). This change is reflected in Section 3.17.23 (starting on page 88).
3. Add a sizeof() method to CoreObject. This change is reflected in Section 3.17.1 (starting on page 60).
4. Add an allocated() method to AllocationContext. This change is reflected in Section 3.17.8 (starting on page 68).
5. Change the signature of AllocationContext.available() to return long. This change is reflected in Section 3.17.8 (starting on page 68).
6. Clarify description of constructor for ATCEventHandler. This change is reflected in Section 3.17.14 (starting on page 75).
7. Clarify description of constructor for ATCEvent. This change is reflected in Section 3.17.15 (starting on page 76).
8. Modify behavior of CoreRegistry.publish() to assure that the memory used to represent CoreRegistry data structures is not released prematurely.
9. Make PeriodicTask implement the IOEventHandlerInterface. Remove its executionPeriod() and numberOverruns() methods. These changes are reflected in the section

Copyright 1999, 2000 J Consortium, All Rights Reserved

describing PeriodicTask, which was removed from a subsequent revision of this document.

10. Add the numberOverruns() method to PeriodicEvent. In the same class, modify the return type of the handler() method to be PeriodicTask. This change is reflected in the section describing PeriodicEvent, which was removed from a subsequent revision of this document.
11. In IOEventInterface, rename SoftwareEventCode to be SporadicEventCode. Rename TimerEventCode to be PeriodicEventCode. Replace mention of SoftwareEvent with SporadicEvent. Revise the signature of exchangeHandler() to throw CoreOperationNotPermittedException. These changes are reflected in the section describing IOEventInterface, which was removed from a subsequent revision of this document.
12. Add a getType() method to IOEventHandlerInterface. Define symbolic constants in this same class for SporadicTaskCode, InterruptTaskCode, and PeriodicTaskCode. Define the getType() method for SporadicTask, InterruptTask, and PeriodicTask. These changes are reflected in sections describing IOEventHandlerInterface, PeriodicTask, SporadicTask, and InterruptTask, all of which were removed from a subsequent revision of this document.
13. Correct the description of IOChannel.mode() to properly identify that 7 bits are required to represent the channel width. This change is reflected in the section describing IOChannel, which was removed from a subsequent revision of this document.
14. Make clear in the description of CoreTask that the start() and _start() methods do not result in immediate execution of the work() method for PeriodicTask, InterruptTask, and SporadicTask subclasses. This change is reflected in Section 3.17.23 (starting on page 88).
15. Change the signature of SporadicEvent.handler() to return SporadicTask. This change is reflected in the section describing SporadicEvent, which was removed from a subsequent revision of this document.
16. Remove the constructors for InterruptEvent and all IOChannel subclasses. These changes are reflected in sections describing InterruptEvent, IOChannel, and all of the IOChannel subclasses, all of which were removed from a subsequent revision of this document.
17. Add createIO() and createInterrupt() methods to the IONode class. These changes are reflected in the section describing IONode, which was removed from a subsequent revision of this document.
18. Add a symbolic constant named IOExclusive to the IOChannel class. This change is reflected in the section describing IOChannel, which was removed from a subsequent revision of this document.
19. Remove attributeConstants() from the IODeviceDescription class. Add entryNames() and modify the definition of attributeNames(). These changes are reflected in the section describing IODeviceDescription, which was removed from a subsequent revision of this document.
20. Remove armInterrupt() and disarmInterrupt() from the InterruptEvent class. These changes are reflected in the section describing InterruptEvent, which was removed from a subsequent revision of this document.
21. Remove the value(x) method from all read-only subclasses of IOChannel. Remove the value() method from all write-only subclasses of IOChannel. Require that 1-bit I/

O objects be configured for implicit I/O. These changes are reflected in the sections describing IOChannel and its subclasses, all of which were removed from a subsequent revision of this document.

22. Miscellaneous typographic, spelling, and punctuation fixes, along with improvements to indexing.

A.11 Revision 1.0.4

A meeting of the Real-Time Java Working Group was held on Jan 7, 2000. At this meeting, the group surveyed the changes incorporated in Revision 1.0.3. A few minor editing changes were requested, which are incorporated in Revision 1.0.4.

1. Remove all references to DeviceRegistry and DeviceCapability as these classes have been removed from the specification.
2. Add cross references to point 34 of Section A.12 (starting on page 122).
3. Fix a few typographic and formatting errors.

A.12 Revision 1.0.3

A meeting of the Real-Time Java Working Group was held on Dec. 7, 1999. The purpose of this meeting was to review comments received during the public review period for Revision 1.0.2. Revision 1.0.3 was prepared in response to the received comments. Specific issues with the 1.0.2 revision which have been addressed in the 1.0.3 revision are listed here.

1. The prohibition on string catenation in Core components is too severe. We need to allow catenation of string literals, as long as the catenation is performed by the Baseline Compiler. This change is reflected in paragraph 6 of Section 3.2 (starting on page 8) and paragraph 18 of Section 3.5 (starting on page 15).
2. The requirement that entry into and departure from a synchronized context not allocate memory needs to generalize to apply to locking and unlocking operations performed on Mutex objects as well. This change is reflected in paragraph 2 of Section 3.8 (starting on page 21).
3. The prohibition on use of synchronized statements to lock Atomic objects other than this needs to generalize to apply to all Core objects. Furthermore, the wording of this requirement needs to be edited so as to allow the synchronized statement to lock this object. This change is reflected in paragraph 3 of Section 3.8 (starting on page 21).
4. The discussion of synchronization issues must include the possibility that a blocked task becomes runnable because some other task signals an asynchronous event to this blocked task. This change is reflected in paragraph 4 of Section 3.8 (starting on page 21).
5. Introduce the notion of asynchronous transfer of control, as it has been proposed for inclusion in the Core specification. These changes are reflected in newly drafted Paragraph 5 of Section B.2 (starting on page 127), Section C.17 (starting on page 146), Section 3.17.14 (starting on page 75), and “The Event Class” (since removed); and in modifications of Section 3.17.23 (starting on page 88) and Section 3.17.27 (starting on page 100).

Copyright 1999, 2000 J Consortium, All Rights Reserved

6. Mention that a task may become runnable because some other task signals an asynchronous event. This change is reflected in Section 3.8 (starting on page 21), paragraph 4.
7. Add a way to timeout a Mutex.lock() invocation. This is handled by introduction of the asynchronous transfer of control mechanism (See paragraph 5).
8. Add a CoreTask.join() method, along with a way to time it out. This change is reflected in Section 3.17.23 (starting on page 88). The timeout capability is provided by the asynchronous transfer of control mechanism (See paragraph 5).
9. Say that when CoreObject.notifyAll() awakens multiple tasks of equal priority, they are awakened in FIFO order. This change is reflected in Section 3.8 (starting on page 21), paragraph 4.
10. The special treatment given to thrown CoreError objects during execution of a finally statement that is part of the cleanup associated with asynchronous abortion of a task needs to generalize to all exceptions thrown during execution of finally statements executing in this cleanup mode. Generalizing this behavior allows the Core specification to relax its prohibition on invoking other methods from within finally statements. Accompanying this change, we need to modify the Core specification to allow Core finally statements to invoke other methods. This change is reflected in paragraph 4 of Section 3.11 (starting on page 25).
11. Add discussion regarding asynchronous abortion that execution of finally statements is “abort deferred”. If an asynchronous abort request is received during execution of a finally statement, the executing thread does not respond to the abort request until after the finally statement has completed its execution. (This is consistent with the general notion that finally statements are always executed to termination, and are never aborted by asynchronous requests.) This change is reflected in Section 3.11 (starting on page 25).
12. Allow for the possibility that some implementations of the Core specification do not support time slicing. If ticksPerSlice() returns zero, that means time slicing is disabled. These changes are reflected in Section 3.17.20 (starting on page 81) and Section 3.17.23 (starting on page 88).
13. Make clear that if a multi-dimensional array is considered to be stackable, all dimensions are considered stackable. This change is reflected in Section 3.12 (starting on page 27).
14. Explain why the inner-class stack-allocation example presented by Aonix is not a valid Core program, and consequently why the example does not represent a loophole in the Core specification. These changes are reflected in Section C.6 (starting on page 141) and Section 3.12 (starting on page 27).
15. Delete the requirement that “support for the Core specification and all profiles be all or nothing”. This is confusing and misleading. In the same paragraph, and throughout the document, substitute “conform to” for “comply with”. To many readers, “comply with” suggests the Sun Microsystems style of conformity assessment, which depends on demonstrating compatibility with a “reference implementation”. Instead, the J Consortium defines conformance in terms of the specification, as demonstrated through execution of appropriate test suites. These changes are reflected in paragraph 3.f of Section B.2 (starting on page 127).
16. State that all run-time error exceptions that are thrown by official Core API libraries are pre-allocated. It is important to establish this in order to assure deterministic

execution of Core applications. These changes are reflected throughout the document, in the descriptions of each method that might throw an exception. Summary overview comments are provided in Section 3.17.27 (starting on page 100).

17. Specify the Core priority semantics in terms of "Base" and "Active" priorities, as suggested in comments submitted by Aonix. These changes are reflected in Section 3.7 (starting on page 21).
18. Specify exactly when a CoreTask's allocation context is released, so that its memory may be reclaimed. For PeriodicTask, InterruptTask, and SporadicTask, the task's AllocationContext is released after the task's stop() method has been executed. For CoreTask tasks which do not extend from any of the above three subclasses, the allocation context is released upon termination of the work() method, which may be triggered by several different events. These changes are reflected in Section 3.17.8 (starting on page 68).
19. Specify for AllocationContext that if the size is specified when the AllocationContext is created, the allocation region will be contiguous and allocation requests will be served in constant time. These changes are reflected in Section 3.17.8 (starting on page 68).
20. For AllocationContext, provide an option to allow programmers to specify the location, in memory, of the allocation region. For example, the application developer may desire that particular AllocationContext regions reside in fast local memory. These changes are reflected in Section 3.17.8 (starting on page 68).
21. The priority interleave stuff is too confusing and probably not sufficiently general. Replace priority interleave with an array that provides a one-way map from core priorities to operating system priorities. These changes are described in Section 3.17.20 (starting on page 81) and Section 3.17.23 (starting on page 88). See the descriptions of Configuration.system_priority_map and CoreTask.systemPriorityMap().
22. Make the Core Static Linker reject invocations of unloadClass() and loadClass(). (Developers who are using the Core Static Linker are not supposed to be using dynamic class loading and unloading.) These changes are reflected in Section 3.17.6 (starting on page 65). See the descriptions of the loadClass() and unloadClass() methods of CoreClass.
23. Throughout the document, replace uses of the word "prototype" with the word "signature" in all contexts that are speaking of Java source code. This change is reflected throughout the document.
24. Get rid of OngoingTask. Use CoreTask to implement the behavior originally intended for OngoingTask. This change is reflected in Section 3.17.23 (starting on page 88).
25. Define a SporadicTask class, which extends from CoreTask. This is like InterruptTask except it is intended to be triggered by software and it does not require execution-time analyzable implementations of the work() method. This change is reflected in Section 3.17.25 (starting on page 97).
26. Allow the _start() Core Baseline method for PeriodicTask, InterruptTask, and SporadicTask in addition to allowing this for CoreTask. This change is reflected in Section 3.17.23 (starting on page 88).
27. Remove pendingCount() and clearPendingCount() from Interrupt. Also, remove hardwareInterruptBuffer(). These are not necessarily portable across all targeted platforms. These changes are reflected in the section describing InterruptEvent, which was removed from a subsequent revision of this document.

Copyright 1999, 2000 J Consortium, All Rights Reserved

28. Explain that by default, all interrupts (which are armed at startup) are handled by interrupt handlers that provide implementation-defined behavior. This change is reflected in the section describing InterruptEvent, which was removed from a subsequent revision of this document.
29. For the Unsigned class, rename the equal() method as eq(). Add ge(), le() and neq() methods to the Unsigned class. These changes are reflected in Section 3.17.22 (starting on page 85).
30. Be more explicit in describing overflow conditions for the Unsigned class's toByte(), toShort(), and toInt() methods. These changes are reflected in Section 3.17.22 (starting on page 85).
31. For interrupt handlers, support an atomic exchangeHandler() method to allow atomic changing of the routine responsible for handling interrupts. (Atomicity is measured with respect to triggering of the interrupt. Each trigger is handled either by the original handler or the new handler. No triggers are ignored, and no trigger is handled by multiple handlers.) This change is reflected in the section describing InterruptEvent, which was removed from a subsequent revision of this document.
32. Create a new CoreTask constructor that allows the option of specifying the size of the default allocation context for the task. Be sure to define appropriate variants of this constructor for PeriodicTask, InterruptTask, and SporadicTask. These changes are reflected in Section 3.17.23 (starting on page 88), the sections describing PeriodicTask and InterruptTask, both of which were removed from a subsequent revision of this document, and Section 3.17.24 (starting on page 94).
33. For Core profiles, specify that official J Consortium profiles are named using the org.j-consortium prefix rather than the org.rtjwg prefix. This change is reflected in Section 3.17.16 (starting on page 76).
34. Refine the definition of the IOChannel system for improved compatibility with the Real-Time Access profile. These changes are reflected in a number of sections which were removed from a subsequent revision of this document.

A.13 Revision 1.0.2

Revision 1.0.2 of this document was published September 27, 1999. This was the first revision intended specifically for official public review.

Annex B Requirements for the Core Specification

B.1 The Working Principles of the Real-Time Java Working Group

The Real-Time Java Working Group's working principles follow:

1. Real-time Java programs written in Core notations must support limited cooperation with programs written in the Baseline language on the same Java virtual machine. The specification for Core extensions shall enable implementations in which execution of Core components in cooperation with Baseline components does not degrade the performance of either the Core or Baseline components.
2. Programs written for the Core extensions must support limited cooperation with programs written according to the specifications for higher level real-time Java profiles (subject to resource availability and contention issues) in environments that implement these optional real-time profiles. The specification for Core shall enable implementations in which execution of Core components in cooperation with components written for higher-level real-time profiles does not degrade the performance of either the Core components or the higher-level real-time profile components.
3. Core extensions offer "minimal latency", where latency means the least upper bound on the time (the longest time) required by a Core interrupt handler to respond to an asynchronous event. We quantify our expectation for minimal latency as follows: The semantics of the real-time core shall be sufficiently simple that interrupt handling latencies and context switching overheads for programs running in the Core Execution Environment can match the latencies and context switching overheads of today's RTOS products running programs written in C or C++. As a point of reference, we expect that commercial implementations of the Core extensions shall demonstrate that this objective has been achieved.
4. Core real-time extensions shall offer "maximal throughput". Support for maximal throughput means the Core specification shall enable implementations that offer throughputs that are essentially the same as are offered by today's optimizing C++ compilers, except for semantics differences required, for example, to check array subscripts.
5. Real-time Java programs that are written using Core extensions need not incur the run-time overhead of coordinating with a garbage collector. Among the overheads that shall not be required by the Core specification are (1) read and write barriers on access to dynamically allocated objects and stack locations, (2) garbage collection scanning of run-time stacks, and (3) pointer identification information required to support garbage collection.
6. Baseline components and components written for yet-to-be-defined higher-level real-time profiles shall be able to read and write the data fields of objects that reside in the Core "object space", where access could be restricted to accessor and setter methods. Code written for the Core Execution Environment need not be able to read or write the data fields of objects that live in the Baseline object space.
7. In the Core domain, it might not be possible for the programming language compiler or run-time environment to enforce compliance with protocols that enable reliable coordination between independent software components. Protections shall be put in place to prevent programmers who are using Baseline programming nota-

Copyright 1999, 2000 J Consortium, All Rights Reserved

tions from compromising the reliability of components written to use the Core extensions.

8. Components written for execution in the Core environment shall run on a wide variety of different operating systems, with different underlying CPUs, and integrated with different supporting Baseline virtual machine implementations. Furthermore, it is important to enable the creation of applications that are composed of a combination of Core and Baseline components. Therefore, there shall be a way for Baseline components to load and execute Core components. There shall be a documented entry point which allows Core components to be run without change on competing platforms adhering to this Core specification. (e.g. Browsers have the Applet as a code entry point, and a Browser supports more than one Applet concurrently. We need to have something like an Applet, but GUI-less.)
9. Program components written for execution in the Core Execution Environment can be dynamically loaded and unloaded within Dynamic Core Execution Environments.

B.2 Additional Requirements

Subsequent to the RTJWG Meeting in which the original nine working principles were identified, additional requirements were introduced into the group's set of constraints. These are identified here:

1. The Core specification shall support the ability to perform stack allocation of dynamic objects under programmer control. It is implementation-defined whether particular implementations of the Core Execution Environment honor programmer requests to allocate objects on the stack.
2. The Core specification shall be designed to support a small footprint, requiring no more than 100K for a typical Static Core Execution Environment.
3. The Core specification shall enable the creation of profiles which expand or subtract from the capabilities of the Core foundation.
    a. The description of each profile must clearly identify whether it resides in the Core Execution Environment (e.g. safety critical) or in the Baseline virtual machine (e.g. real-time garbage collection), or both.
    b. The Core specification shall provide support both for profiles officially supported by the J Consortium and proprietary or 3rd party profiles.
    c. Profiles shall be named using reverse domain name conventions (e.g. com.aonix.high_integrity).
    d. There shall be an API available to Baseline programmers to allow Baseline components to determine which profiles are supported by a particular Core Execution Environment.
    e. There shall be an API available to Core programmers to allow Core components to determine which profiles are supported by a particular Core Execution Environment.
    f. If a particular Core Execution Environment claims to conform to the Core specification, it shall support all features of the Core specification. If a particular Core Execution Environment claims to support a particular profile, it shall support all features of that profile's specification.

g. Each profile may add to or disable certain specified capabilities of either or both of the Core or Baseline domains. The description of the Core specification shall enumerate which of the specified capabilities might be disabled by "acceptable profiles" (e.g., AllocationContext.release(), stackable, disabling of stack overflow checking, configuration of tick and time slice duration, support for suspending and resuming tasks).

h. A cursory review (perhaps the registration authority provides a registry of which profiles are known to disable capabilities, and all profiles not identified by the registration authority must be considered "unknowns") of the profiles supported by a particular Core Execution Environment reveals whether the profiles disable particular capabilities. A correctly written Core application shall run on any Core Execution Environment for which none of the supported profiles disables any of the officially specified Core capabilities. For each profile that is known to disable particular Core capabilities, a mechanism shall be available for determining exactly what capabilities are missing from the Core Execution Environment.

4. The requirements for Core dynamic class loading facilities are as follows:

   a. Support for dynamic class loading in a Core Execution Environment shall be optional.

   b. The dynamic class loader for the Core Execution Environment shall be implemented as a Baseline component. This means that dynamic class loading shall not be available in Core Execution Environments that are not paired with a Baseline virtual machine.

   c. The Core APIs for dynamic class loading shall support flexibility regarding where and how dynamic classes are loaded. Integrators of Core Execution Environments shall be able to configure the Core Execution Environment to specify where to search for and how to obtain the class files that are to be dynamically loaded.

   d. The Core dynamic class loader need not be as sophisticated or general as the Baseline class loader. In particular, we do not anticipate the need for application-specific core class loaders. Instead, the Core dynamic class loader shall allow integrators to define an implementation-specific core class loader that serves all core class loading needs of a particular implementation of the Core Execution Environment.

   e. All Core classes shall be fully resolved and initialized at the time they are dynamically loaded.

5. Requirements for Core asynchronous transfer of control are as follows:

   a. Asynchronous transfer of control shall apply only when the affected code permits asynchronous transfer of control. Asynchronously transferring control out of code that was not designed for the possibility of asynchronous transfer of control might introduce program logic inconsistencies.

   b. There shall be a mechanism to allow Core application programmers to establish syntactic contexts within which asynchronous transfer of control shall be deferred. If some other task requests asynchronous transfer of control while this task is executing within a deferral context, the control transfer is delayed until this task completes execution of the code contained within the deferred context.

Copyright 1999, 2000 J Consortium, All Rights Reserved

c. The asynchronous transfer of control mechanism shall support common programming idioms, such as abortion of a task, timing out of a code sequence (including nested timeouts), mode change for a particular task, and software interrupt during code.

d. The asynchronous transfer of control mechanism shall prevent unintended catches of any exceptions that are used in the implementation of asynchronous transfer of control, if the asynchronous transfer of control mechanism relies upon exceptions.

e. The asynchronous transfer of control mechanism must address the question of whether nested timeouts work properly.

f. The asynchronous transfer of control mechanism shall be easy for Core programmers to use and understand.

g. The run-time implementation costs of asynchronous transfer of control shall be paid primarily by those components that make use of this mechanism. The run-time overhead imposed by the asynchronous transfer of control implementation on Core components that do not use this feature shall be minimal.

h. The asynchronous transfer of control mechanism shall provide a way to protect against stack overflow caused by asynchronous event handling by stack-limited Core tasks.

i. The asynchronous transfer of control mechanism shall provide a way for Core application programmers to establish contexts within which particular context-specific asynchronous event handlers are relevant and enabled.

j. It is required that the asynchronous transfer of control mechanism support abortion of the currently executing task. It is desirable (but not required) that asynchronous transfer of control support resumption semantics (for which the original Core component is resumed following execution of the event handler).

Annex C Background and Rationale

C.1 Historical Background

Since June of 1998, the U.S. National Institute of Standards and Technology has been hosting regular meetings of the "Requirements Group for Real-time Extensions for the Java™ Platform". This group includes representatives from 37 different companies. For additional information on the NIST requirements group, refer to its web page: http://www.nist.gov/rt-java.

Using a consensus-based approach, the NIST-sponsored group has drafted a document detailing requirements for real-time extensions for the Java platform. These requirements represent the collective input of technology suppliers, technology users, and the academic research community.

C.1.1 NIST Requirements for the Real-Time Core

Members of the NIST-sponsored group recognize that the needs of the real-time industry are varied and diverse. Satisfying all of the needs of the entire prospective user community will require monumental effort. Further, the needs of particular constituencies conflict with the needs of others. In recent meetings of the NIST group, the consensus position has been to partition real-time extensions into a real-time core and a collection of optional real-time profiles. Throughout this document, we use the term "Core" to represent the API and special syntaxes and restrictions associated with the real-time core. According to consensus positions reached at the NIST meetings, key characteristics of the real-time core are:

1. The real-time core shall provide services of the sort that are typically provided by commercially available real-time operating systems. The core shall not endeavor to "advance the state of the art" in development of real-time software.
2. The real-time core shall be simpler to implement than the full range of capabilities that are required by the NIST group's requirements document.
3. The real-time core shall provide a foundation upon which more sophisticated higher level real-time capabilities would be constructed as optional profiles.

It should be noted that the real-time core does not address all of the requirements of the NIST document. It is specifically intended to address only the above subset of the full set of requirements. The intent is that the many NIST requirements that have not been addressed in the core requirements will be addressed by higher level real-time profiles which supplement the real-time core.

The consensus positions resulting from the NIST requirements meetings are described in Reference 1.

C.2 NCITS Principles for Real-Time Core

On January 11, 1999 (before the formation of the J Consortium), a subcommittee of the Real-Time Java Working Group met to discuss the core requirements of the NIST requirements group and to begin work on a straw man specification[1]. One of the results

Copyright 1999, 2000 J Consortium, All Rights Reserved

of that meeting was a document titled "Consensus Positions of the Real-Time Java Working Group: Scarecrow (1/11/99)". That document, which describes the group's general recommendations for a specification for Core real-time extensions, was submitted to the NCITS R1 committee, and was assigned document reference number R1/99-007.

Document R1/99-007 was prepared in anticipation of NCITS approval of proposed standardization work for real-time Core. However, on January 15, 1999, NCITS announced that its members had voted to reject the proposed standards activities. Among the reasons cited by those who voted against the effort, the principal objections were as follows:

1. There was a question of whether it would be possible to create a specification for real-time Core which did not infringe on Sun Microsystems' intellectual property rights.
2. Concern was raised that if it were possible to create a real-time Core specification that does not infringe on Sun Microsystems' intellectual property, the specification would necessarily be sub-optimal in comparison with a specification that might be developed by Sun Microsystems, which would not have to work around possible intellectual property issues.
3. Concern was raised that Java standardization work carried out within NCITS might fragment the Java marketplace.

Even though NCITS rejected the proposed standardization work, members of the Real-Time Java Working Group felt it was important to continue work on refining a draft specification for Real-Time Core Extensions for the Java Language in order to address the concerns that had been raised by the NCITS voting membership.

This specification grows from the NCITS R1/99-007 document. In this document, we expand and clarify on the points of R1/99-007. Additionally, this document reflects changes to the recommendations of R1/99-007 as have been motivated by feedback collected as part of the public review process.

The Real-Time Java Working Group recognized that there was considerable flexibility in fulfilling the NIST group's core requirements. In order to narrow the breadth of opportunity, this group formulated a list of principles for real-time Core. These principles, which are described in Section B.1, are intended to supplement and clarify the NIST requirements.

In general, the Real-Time Java Working Group took the position that real-time Core would address the needs of a particular important class of real-time programs that are characterized by the following attribute:

1. At that time, the Real-Time Java Working Group was a group of companies who shared a common interest in advancing the art of real-time programming with the Java language. Most of the core members of the Real-Time Java Working Group also participated in the NIST meetings that produced the NIST requirements document and have now joined the J Consortium to continue work under its sponsorship.

> Nearly all dynamic memory is allocated during initialization of the program, and following initialization of the program, no further dynamic memory management is required.

The significance of this observation is mainly to justify the exclusion of real-time garbage collection from the Core specification. It is not to say that the Core specification should not provide any support for any form of dynamic memory management. The Core specification shall not be prohibited from providing support for dynamic memory management, and to the degree that limited forms of dynamic memory management can be supported without compromising other guiding principles, that is desirable.

Specific comments and rationale for each of the guiding principles is presented here:

1. Regarding guiding principle number 1, we emphasize that neither the semantics nor the typical implementation of the Baseline language is appropriate for real-time programming. Though it might be possible to redefine the semantics of the Baseline language to make it more appropriate for real-time programming, it is the position of the RTJWG that this would not be practical. A key obstacle is the legacy now supported by the Baseline language. This legacy already includes millions of lines of existing Java source code and hundreds of licensees of Sun's Java technologies, most of whom have little interest in the specialized niche needs of the real-time community. For this reason, we make a strong distinction between Baseline programming and Core programming, and we state the requirement that these two worlds be able to cooperate with each other.
2. Though the J Consortium has not yet defined the services to be provided by each of the higher level real-time profiles mentioned in this paragraph, the NIST Requirements document (See Reference 1) states that high-level profiles shall support deadline driven task scheduling; so-called negotiating components; and accurate, defragmenting, paced garbage collection.

   Of key importance is the observation that satisfying the first working principle is significantly easier than satisfying the second. One reason for this is that the garbage collection requirements for the Baseline platform are very lax in comparison with the likely garbage collection requirements for a high-level real-time profile. The more sophisticated garbage collection required by high-level real-time profiles generally imposes higher penalties on both latency and throughput.
3. As originally introduced to the NIST requirements group, the intent of Core extensions is to provide services equivalent to what is currently offered by commercially available off-the-shelf real-time operating systems. During the past decade, real-time operating system vendors have been pushed by their customers to compete in, among other areas, interrupt response times and context switching efficiency. In order to satisfy these same customers who drove this marketplace competition, we felt it important to address these same requirements.

   Feedback received in response to distribution of the R1/99-007 document has requested that we characterize minimal latency and maximal throughput in terms of Java overheads rather than describing the total cost resulting from the combination of RTOS services with Java overheads. In those terms, this principle essentially states that the Core extensions shall be defined such that implementations are possible in which the scheduling and context switching overhead of real-time Core tasks is zero.

Copyright 1999, 2000 J Consortium, All Rights Reserved

4. This objective, like the one that precedes it, is motivated by the intent to address the demands of current users of real-time operating systems.

   We recognize that there are certain semantic differences between the Java language and C++. One example is the behavior of array subscript operations. In the Java language, the array subscripting operation implies an array subscript bounds check. In C++, it doesn't. Thus, this operation has different semantics between the Java and C++ languages. Given that the operation has different semantics, we do not expect equivalent performance; the two languages are doing different things. However, the Core specification shall enable implementations of instance method invocation and field access for dynamically allocated objects that perform with performance equivalent to that of C++.

   Feedback received in response to distribution of the R1/99-007 document has requested that we characterize minimal latency and maximal throughput in terms of Java overheads rather than describing the total cost resulting from the combination of RTOS services with Java overheads. In those terms, this principle essentially states that Core extensions shall be defined such that implementations are possible in which the run-time overhead of coordinating real-time Core tasks with the garbage collector and with other components of the Java virtual machine is zero in comparison with the costs of comparable services on typical commercially available real-time operating systems.

5. A decision made by the Real-Time Java Working Group was that programs written using Core extensions need not incur the overhead of garbage collection. This was motivated by (1) the recognition that the target constituency for the Core extensions is programs that allocate nearly all memory during startup and have no subsequent dynamic memory allocation needs, (2) the objective that the Core extensions support maximal throughput, and (3) the objective that the Core extensions support minimal latency.

   While we recognize that garbage collection is a key benefit of the Java language, we also perceive that garbage collection imposes significant costs in terms of run-time efficiency and system complexity. There are large classes of real-time software components (e.g. typical interrupt handlers and device drivers) that derive little benefit from having automatic garbage collection and our group felt that imposing the burden of garbage collection on those components would only discourage the use of the Java language as an appropriate technology for implementation of those components.

   In comparison to the current state of the art in development of real-time software, which tends to favor the C language, we see numerous benefits in the use of the Java language beyond the benefits of garbage collection alone. In particular:

   a. Portable binary code representations
   b. Ability to leverage widely available off-the-shelf Java development environments
   c. Good object-oriented programming language features facilitate maintenance and reuse of software
   d. Strong compile- and load-time type checking
   e. Familiar syntax and development environments to the many developers who have already developed skills as Java programmers

f. Straightforward integration and access to all of the APIs of the Baseline platform (though these Baseline APIs will not necessarily promise real-time performance)

g. Support for secure dynamic loading

Note, in the statement of this objective, our choice of the words “need not”. The significance of this wording is to emphasize that there may exist implementations of the Core extensions that do incur the respective garbage collection costs. However, it is our intent to make sure the definition of the core semantics allows more efficient implementations.

In order to allow compile- and load-time enforcement of partitioning between the Core domain and the Baseline domain, the Core specification partitions all APIs and application-specific methods between those methods that are available to Baseline components and those that are only available to the Core tasks. Below, we identify a number of the reasons for this partitioning of APIs.

a. The existing Baseline API definitions and implementations are not “real-time ready”. You cannot, for example, safely abort a thread that is in the middle of executing a Baseline library function. And the synchronization semantics that we intend to carefully define for the use of real-time components are different and incompatible with large bodies of existing Java library code. Another difficulty is that existing Baseline library routines are not resource predictable in terms of memory or CPU time requirements. In summary, you cannot calculate a worst-case execution time, and you cannot abort the code if the routine runs too long.

b. Almost all of the Baseline libraries assume the presence of a garbage collector. (Though a developer may discover through inspection of the source code implementations of Baseline libraries that certain of these libraries do not allocate temporary objects, there is no general assurance that future implementations of the same libraries will not allocate memory.) In our initial discussions about the Core extensions, the consensus position was that we did not want to rely on real-time garbage collection. Instead, we had identified as our constituency the important class of problems that allocates memory during startup and thereafter simply makes use of previously allocated objects. This is the class of problems for which we “tuned” the Core specification. Given that we felt it essential to avoid the burden of a real-time garbage collector in the Core domain, the use of existing Baseline libraries from within the Core domain was viewed as inappropriate, because nearly all existing Baseline libraries depend on automatic garbage collection for reliable operation.

c. One of the requirements for the Core extensions is that the resource requirements of each “service” supported by Core extensions be precisely defined. If we want to include the full Baseline API, we need to analyze and constrain the resources required to implement each method of the complete Baseline API. That task appears impractical, especially considering the rapid rate at which Baseline libraries continue to evolve. It is much more practical to define a small set of API libraries for use by Core components, and to carefully define the resource requirements of these libraries.

d. Given that one of the objectives of the Core extensions is to provide maximal throughput, it is important that the implementation of Core methods not incur the overhead of coordinating with garbage collection. This means, for exam-

Copyright 1999, 2000 J Consortium, All Rights Reserved

ple, that a method that is invoked from a Core task does not need to incur the overhead of read or write barriers when accessing the fields of an object whose reference is passed into the method as an argument. Since the implementation of this method does not include read and write barrier overheads, it is important that this method not be invoked with a reference to a garbage-collected object as its argument. Thus, a protocol that allows clear distinction between methods that deal with garbage collected objects and methods that deal only with Core objects (so-called Core methods) is required. In order to minimize the performance impact of enforcing this protocol, it is desirable for differentiation between Core and Baseline methods to be based on static (compile-time) information rather than run-time checks.

e. Another benefit of partitioning the APIs involves the ability to shrink memory footprints of embedded real-time applications. By restricting the Core API to a small set of primitive services, we enable tremendous shrinkage of the Java footprint. The Baseline libraries are huge in comparison to typical embedded software systems. Static linking techniques have been demonstrated that prune large amounts of the standard Baseline libraries from an embedded product's load image. However, if a system must support dynamic loading, the static linking approach does not work and very little of the Baseline API can be pruned from the load image without violating Sun's specifications for the Baseline virtual machine. With a limited-library Core Execution Environment, we have the opportunity to build a very small footprint configuration without sacrificing the ability to support dynamic loading of Core components. Such systems can be designed so that the Baseline side is totally static, and can thus be reduced in size using static-link-time pruning. Only Core components would be dynamically loaded in this configuration.

6. Given the desire to support limited cooperation between Baseline components and Core components, and given that Core programs shall not support garbage collection, we felt it very important to provide mechanisms to facilitate information sharing and synchronization between components written for execution in the Core and Baseline environments respectively.

In terms of intended functionality, think of the Core components as comprising an operating system kernel, and think of the Baseline components as comprising the application space. In traditional operating system environments, the kernel is allowed access to user space, but user applications are not allowed access to kernel memory. Here, we reverse these restrictions.

The reason we do not want Core components to have direct access to Baseline objects is because those objects are subject to garbage collection. If the Core objects were to have access to garbage collected objects, then dispatching of Core tasks would have to coordinate with the Baseline garbage collector, and this would likely have a negative impact on the latency of Core tasks. An additional difficulty with allowing Core tasks to access garbage-collected objects is that this would make it more difficult for the garbage collector to know when objects are dead. Not only would the garbage collector have to examine the thread state of each Baseline thread, but it would also have to examine the thread state of each Core thread. And in order to enable the garbage collector to examine the thread state of a Core task, additional bookkeeping overhead would have to be inserted into the protocols associated with running of Core tasks. This would have a negative impact on the throughput performance of Core tasks.

We do allow Baseline threads to access Core objects. Because of the Java language's strong type checking and support for secure data encapsulation, this does not compromise the integrity of the Core components. There is no way for a Baseline application to see or modify data contained within Core objects unless the programmer of the Core components makes that data available by providing appropriate accessor or setter methods.

Note the restriction that access to the fields of Core objects be directed by way of accessor or setter methods. Though we can envision implementations that would not require this restriction, it was the sentiment of the group that imposing this restriction would offer greater flexibility to implementors of the Core Execution Environment. We expect this choice will not impose a performance penalty, given the ability to in-line methods.

7. Writing Core applications is like writing device drivers for an operating system kernel. Consistent with current practices of commercial real-time operating system users, programmers who write Core applications have access to very powerful tools, and accidental or malicious misuse of these tools could compromise integrity of the system. For this reason, only trusted expert real-time programmers should author Core components. These programmers are responsible for considering global resource contention issues and for following recommended coordination protocols.

   It is our intent that security mechanisms shall be available to help programmers avoid accidents. Wherever possible, these security mechanisms should be enforced at compile- and load-time rather than at run time. Doing so reduces the run-time overhead of the security enforcement protocol. Though we intend to build upon existing Java type-checking mechanisms to eliminate many common programming errors, we recognize that there are certain kinds of errors that cannot be prevented by these mechanisms. Thus, we acknowledge in stating this objective that insofar as Core programming is concerned, we would prefer to allow the use of these "dangerous tools" in spite of the risks they engender, rather than prohibit all such tools in order to assure elimination of all such risks.

   It is our expectation that the amount of code written in the Core notations is typically only a small fraction of any particular real-time software system. The great majority of code in a typical system would be written either as Baseline threads, or using the higher level real-time profiles. Core extensions are intended for implementation of components that require extreme efficiency, either in throughput or response latency, or both.

   Though we allow sharing of objects between Core and Baseline components, we require that the specification for the Core extensions provide protection mechanisms to ensure that Baseline components do not compromise the integrity of Core components. This is because developers of Baseline applications are not necessarily "trusted experts".

8. The intent is that there shall be a documented way for Baseline software components to cause Core components to be loaded and executed. Further, this implies that the Core API definitions are precise enough to allow creation of portable Core components (which will run in a wide variety of different Java virtual machines, each produced by a different vendor).

Copyright 1999, 2000 J Consortium, All Rights Reserved

9. Note that the security requirements of Core components may be different than the security requirements of the Baseline language, and may be context specific. Security checking for Core components, if any, is implementation-defined.

C.3 Rationale for Partitioning of Memory

In order to provide high reliability and allocation efficiency, certain garbage collectors relocate objects as part of a memory defragmenting effort. We specify that Core objects shall not be relocated to emphasize that this is part of the semantics of Core objects. This is an important behavioral constraint because it means that Core objects may be shared with non-Java tasks (if memory sharing is supported by the host operating system), with non-Java interrupt handlers, and with hardware DMA devices.

We impose the restriction that Core methods shall not in general be invoked by Baseline components because the implementation of Core methods may not include synchronization code for coordination with garbage collection. If Baseline threads were to invoke these Core methods, passing as arguments references to Baseline objects, this would lead to the possibility of the following sorts of problems: (1) the garbage collector might reclaim an object while the Core method is trying to access it, or (2) the Core method copies the Baseline reference into a Core data structure, introducing the likelihood that the Baseline garbage collector will reclaim the object at some future time while the object is still visible to the Core domain.

Conceptually, each Core object shall have two method tables. One of the method tables is used exclusively by Core components. The other method table is used exclusively by Baseline components. Baseline components do not need to understand the internal organization of Core objects because they are not allowed to directly access any data fields. They are only allowed to invoke methods.

Instead of allowing Baseline direct access to the instance variables of Core objects, the object partitioning protocol requires that all such access be made by way of accessor and setter methods (the so-called Core-Baseline methods).

Note that we prohibit Core-Baseline methods from modifying the pointer fields of Core objects, even indirectly through invocation of a setter method. If we were to allow Core-Baseline methods to modify the pointer (reference) fields of Core objects, we would introduce the possibility that the reachability of particular Core objects could be modified by execution of Core-Baseline methods. That in turn would require a more sophisticated garbage collection interaction protocol between the Core and Baseline domains. In the interest of simplicity and run-time efficiency (avoiding write barriers in the implementation of Core-Baseline methods), we chose to prohibit Core-Baseline methods from modifying pointer instance and heap variables.

The Core Verifier shall reject as invalid any classes that make reference to `StringBuffer` objects. This is because `StringBuffer` objects create scratch memory that must be reclaimed by a garbage collector, and the Core Execution Environment does not have a garbage collector.

Core objects shall be accessible to Baseline threads. For this reason, it is not possible to support explicit deallocation. Otherwise, a Core task might deallocate an object while the Baseline world is still trying to make use of the object. This is why we have designed a protocol that allows cooperation between garbage collection and explicit dynamic memory management. In particular, the Core task “releases” a collection of objects after it is done using the objects. We trust the developers of Core components to correctly manage their dynamic memory. The effect of the allocation context’s release operation is to make the objects eligible for garbage collection (and possible relocation). The objects shall not be reclaimed, however, until the garbage collector verifies that the objects are unreachable from the Baseline domain.

C.4 Comments Regarding the Core Verifier

Use of a Core Verifier is required in the deployment of any conforming Core application. Core developers might ask: “What are the risks in the absence of a Core Verifier?” If a system did not enforce these restrictions, this would introduces a number of possible risks. Here we list some of the risks that might arise in the absence of enforcement.

1. Interrupts might remain disabled for too long
2. Memory leaks might result from temporary object allocation in Core tasks
3. Objects might be reclaimed by the garbage collector while a Core or Baseline task is still looking at the objects
4. The garbage collector might become confused because of premature deallocation of objects, resulting in fatal termination of the Core Execution Environment, or of the Baseline Virtual Machine
5. A request to abort a task doesn’t really abort the task, because the task does not cooperate with the abort request

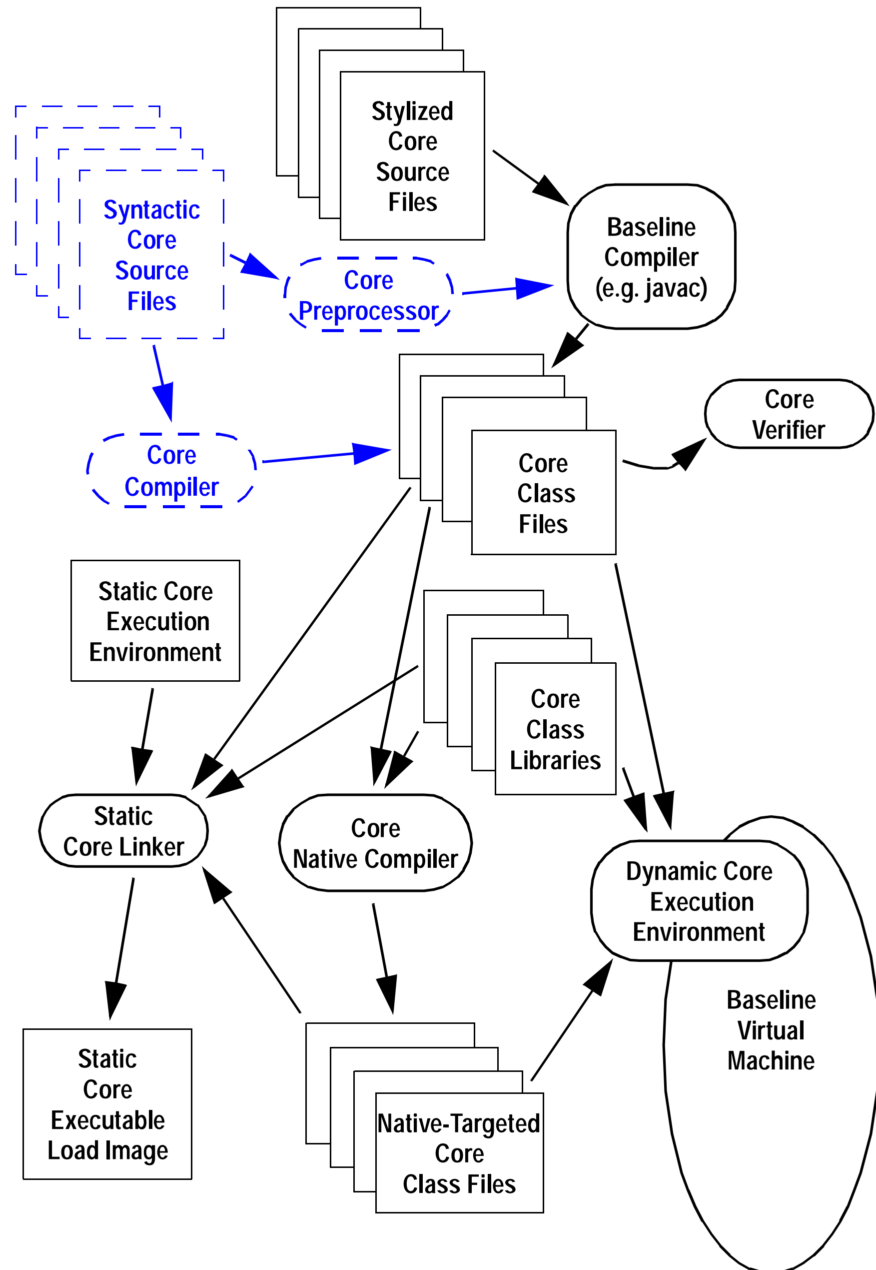
C.5 Comments on Syntactic Core Extensions

During early development of the Core Specification, two optional syntax extensions were proposed for inclusion in the specification. Subsequently, it was decided by the Real-Time Java Working Group to remove these syntax extension from the Core Specification and to describe the proposed technologies for possible implementation and use within vendor-specific Core development tools. The Core development architecture is shown in Figure 4 on page 139.. In this figure, the components drawn with solid black outlines are described in Section 3.4 (starting on page 12). The components drawn with dashed blue outlines are special components that are not defined by this specification. Rather, these represent technologies and tools that independent tool developers might implement to simplify the development and maintenance of Core software components. The special components are:

1. **Syntactic Core Source Files:** Syntactic Core Source Files are Java 1.1 source files written to take advantage of special syntaxes that have been designed to simplify the development of Core Components. In particular, Syntactic Core Source Files make use of two special keywords, `stackable` and `baseline`, which are not a part of the traditional Baseline syntax.
2. **Core Preprocessor:** A Core Preprocessor transforms Syntactic Core Source Files to Java 1.1 source files which do not contain any uses of the `baseline` and `stackable` keywords.

Copyright 1999, 2000 J Consortium, All Rights Reserved

Figure 4. Overview of Real-Time Core Development Architecture



3. Core Compiler: The Core Compiler translates Syntactic Core Source Files to real-time Core Class Files. At the same time, it performs all of the verification checking that is performed by the Core Verifier.

The baseline keyword would be used to identify Core-Baseline methods. Rather than inserting an invocation of CoreRegistry.registerBaseline() as part of the class' static initializer (as described in "CoreRegistry.registerBaseline()" on page 77), a Core programmer who chooses to use the Core Compiler or Core Preprocessor might instead insert the baseline keyword into the declaration of the method's prototype, as suggested by the following example:

```
public baseline void foo(int i, float x) {
   ...
}
```

Special Notations for Syntactic Core Source Code. Syntactic Core Source Code is code written for the Core Execution Environment which is intended to be compiled by a special Core Compiler. This compiler provides all of the functionality of a traditional javac compiler, with the following additional functionality:

1. For each class file produced by this special compiler, the Core Compiler shall insert an invocation of CoreRegistry.registerCoreClass() as the first executable code in the static initializer for the class.
2. The Core Compiler shall allow the special baseline keyword as an attribute for method definitions. For each method that is identified as a Core-Baseline method (by the presence of the baseline keyword), the Core Compiler catenates the name and signature of this method into the CoreString argument for this class's invocation of CoreRegistry.registerBaseline(). For each class compiled by the Core Compiler that has at least one Core-Baseline method, the Core Compiler shall insert an invocation of CoreRegistry.registerBaseline() into the static initializer for the class immediately following the invocation of CoreRegistry.registerCoreClass().
3. The Core Compiler shall allow the special stackable keyword as an attribute for local variable and argument definitions. For each variable or parameter that is identified as stackable (by the presence of the stackable keyword in its declaration), the Core Compiler inserts the variable or parameter name into the CoreString argument for the invocation of CoreRegistry.registerStackable() method as the first executable line of code in the method.
4. For any class that fails to identify which class it extends, the Core Compiler generates code to indicate that the class extends java.lang.Object, with the understanding that the Core Class Loader shall replace the reference to java.lang.Object with a reference to org.rtjwg.CoreObject.
5. All throw statements, catch statements, and method declarations from which exceptions are thrown are understood to refer to objects extending from org.rtjwg.CoreThrowable, and type checking is performed to enforce conformance with this understanding. However, the class file produced by the Core Compiler replaces references to CoreThrowable with references to java.lang.Throwable, references to CoreException with references to java.lang.Exception, and references to CoreRuntimeException with references to java.lang.RuntimeException, with the understanding that the Core Class Loader will replace each of these types with its original representation.
6. All string constants are treated as CoreString objects for purposes of type consistency checking. The Core Compiler shall represent string constants as Baseline String_CONSTANT objects in the class-file constant pool, recognizing that the Core

Copyright 1999, 2000 J Consortium, All Rights Reserved

Class Loader shall replace all String_CONSTANT objects with appropriate CoreString replacements representing the same sequence of characters.

7. Each variable that is declared to be of type array is treated as a variable of type CoreArray (or an appropriate subclass of CoreArray). See Section 3.17.7 (starting on page 66) for additional description of the CoreArray class. Each allocation of a new array object is treated as if it produced an instance of the CoreArray class (or the appropriate subclass of CoreArray). The Core Compiler shall enforce type consistency checking by using the appropriate CoreArray type as the type of each allocated array and of each variable that is declared to hold a reference to an array object. In the class-file representation that is emitted from the Core Compiler, each CoreArray type is represented as a Baseline array type. The Core Class Loader shall replace each reference to a Baseline array type with a reference to an appropriate derivative of CoreArray when it loads the class.
8. Except for the specific exceptions described above, the Core Compiler shall enforce all of the requirements of the Java 1.1 language specification as described in Reference 5. Further, except for the specific exceptions described above, the translated output from the Core Compiler shall be compatible with the output produced by existing Baseline Compilers and shall comply with the existing conventions for translation of the Java language as described in References 5 and 8.
9. The Core Compiler shall ensure that the translated Core Class Files that it produces conform with all of the rules and constraints described in Section 3.5. If the Core source code is such that complying with these constraints is not possible, the Core Compiler shall issue appropriate implementation-defined diagnostic messages and shall not produce a translation of the offending source program.

C.6 Clarification and Rationale re: Stack Allocation

Note that being able to allocate objects on the run-time stack might benefit the Baseline language as well. The motivations for supporting stack allocation in Core are several fold: (1) to enable better throughput performance, (2) to enable dynamic allocation (and deallocation) of temporary objects in the absence of a garbage collector, and (3) to facilitate creation and verification of certified software for safety critical applications.

Certification agencies, such as the Nuclear Regulatory Commission, the Federal Aviation Administration, and the Food and Drug Association, are generally very conservative. The general sense among companies who have been involved in certification of safety critical software is that automatic garbage collection is much more complicated that stack allocation, both to implement correctly and to prove implemented correctly.

In order to safely allocate objects on the run-time stack, we must assure that no references to a stack-allocated object survives beyond disappearance of the stack activation frame within which the object is allocated. The various restrictions described in Section 3.12, all of which can be enforced at compile and link time, are sufficient to guarantee that all references to stack-allocated objects disappear by the time the stack-allocated object is reclaimed from its run-time stack.

In general, objects should only be stack allocated if it has been verified that all of the special restrictions and conditions for stack allocation described in Section 3.12 have been satisfied. The process of verifying compliance with these conditions is intention-

ally conservative, meaning that there may exist situations in which a more sophisticated analysis would conclude that particular objects are stack allocatable even though the rules of the Core specification do not so permit. We prefer a conservative approach in that it makes very clear to Core programmers exactly which objects shall be allocated from the run-time stack.

A careful reader of the Core specification suggested that the following sample code represents a loophole in the specification:

```
import org.rtjwg.*;
class C extends CoreObject {                    // core class
  static void foo() {
    stackable final int [] intarray = new int[100];
    class MyTask extends CoreTask {             // declaration of an inner class
      public void work() {
        sleep(1000);
         // refer to elements of intarray
      }
    }

    MyTask mt = new MyTask();
    mt.start();
    return;
  }


  static {                                      // static initializer for this class
    foo();
  }
}
```

When C is loaded, foo() will be invoked by the static initializer. This constructs an instance of MyTask, assigning a reference to its local variable mt, and starts this task up. Then foo returns, releasing the activation frame within which the stackable array of integers intarray. However, the mt task is still running and is continuing to access the stack-allocated intarray object.

The above example appears to demonstrate that the safeguards designed to prevent the existence of dangling pointers are insufficient to serve this purpose. The fact is, however, that the above example is not a valid Core program. The reason for this is as follows:

1. MyTask is a “member class” of class C.
2. When C.foo() creates a new MyTask, the Baseline Compiler silently inserts code at the constructor call to pass a copy of the intarray reference into MyTask’s constructor.
3. MyTask’s constructor silently saves its copy of the intarray reference in a hidden member field. According to the Java Language specification, this is permitted if and only if the object referenced by intarray is declared to be final.

Copyright 1999, 2000 J Consortium, All Rights Reserved

It appears from examination of Java source code that this sample program conforms to all of the requirements for a Core application. However, examination of the corresponding class file reveals that this program does not conform to the Core class-file specification. In particular, this program passes as an argument to a method (MyTask's constructor) a reference to a stackable object, and the formal argument is not declared to be stackable. Even if this implicit argument were declared to be stackable, MyTask's constructor would not be allowed by the Core restrictions on uses of stackable variables to copy the intarray reference into a "hidden member field" of a heap object.

C.7 Motivation for Special Class Loading Semantics

The Baseline specification requires that classes be initialized upon first access. Implementation of these semantics is burdensome, requiring run-time checks on frequently used operations and/or self-modifying code. Self-modifying code does not work well for code executing out of ROM. Furthermore, code that resolves and initializes itself on the fly is difficult to analyze with respect to execution time.

C.8 Clarifications re: Execution Time Analyzability

Section 3.14 describes a number of constraints on the byte-code generator for the Core Compiler. Clearly, it would be desirable to impose these same requirements on the Baseline Compiler. However, the specification for compliant behavior of Baseline Compilers is in the hands of Sun Microsystems and the J Consortium does not control how that might evolve. To the extent that Baseline Compilers continue to conform to the requirements stated in Section 3.14, it will continue to be possible to use Baseline Compilers for development of Core Class Files.

Note that the restrictions on analyzable loops are more strict than is really necessary. Certainly, it would be possible to analytically determine the worst-case execution times for more loops than satisfy our fairly restrictive criteria. Our main objective, however, is to provide reliable support for execution-time analysis of a restricted subset of the Java language, and we want to make sure that programmers can easily understand the rules (though not necessarily the implementation) that characterize this restricted subset.

C.9 Rationale for Core Class Loading Requirements

The rationale for requiring that the dynamic Core Class Loader be implemented as a Baseline component is that class loading is a complicated activity, and it is desirable for the Core Class Loader implementation to take full advantage of the Baseline language's high-level benefits, such as garbage collection and the full breadth of Baseline APIs. Further, the expectation is that Core class loading is relatively rare (thus, it is not performance critical) and does not have stringent timing constraints. For these reasons, we felt there was no need for the Core Class Loader to run within the Core Execution Environment.

C.10 Comments on Run-Time Differentiation between Core and Baseline Tasks

In the NIST requirements document (see reference 1), Section 5, core requirement 8 states: "The RTJ specification must provide a mechanism to allow code to query whether it is running under a real-time Java thread or a non-real-time Java thread."

The Core APIs do not provide any run-time mechanism to address this requirement. Instead, Core programmers distinguish code intended for execution in a Baseline thread from code intended for execution as a Core task with static (syntactic) notations. In particular, all of the methods of any class for which the static initializer code starts with an invocation of the CoreRegistry.registerCoreClass() method that are not identified as Core-Baseline methods are executed as Core tasks. Any other methods are executed as Baseline tasks.

C.11 Comments re: the PCP Interface

One of the key benefits of using the Priority Ceiling Protocol for task synchronization is that it enables non-blocking implementations of synchronization. Whenever any task has the lock, its priority is automatically increased to the highest priority of any task that might attempt to lock the object. Thus, any other task that might attempt to access the same object shall not be allowed to run (because of priority) while a particular task has the monitor locked. Another way to think of this: For any given task, if the system scheduler has dispatched the task for execution, the task can be assured that no other task owns access to any of the monitor locks that this task might want to use.

Given the specification as drafted, the implementation of Priority Ceiling Protocol does not require a queue of objects waiting for access to the monitor's lock. This allows for a small-memory, easy-to-analyze implementation of synchronization locks.

Though the current specification does not address the special needs of multiprocessor systems, it is important to recognize that the specification is designed to generalize to such targets in the future. It is the intent that a future variant or profile of this specification will provide support for an N-processor SMP computer, in which PCP synchronization shall block the currently running task no longer than the time required for each of the other N-1 processors in the system to execute at most one segment of code associated with the same PCP object. Further, it is desirable to avoid deadlock conditions which might arise when multiprocessors attempt to enter multiple shared PCP-protected contexts in different orders. For this reason, the specification requires that the priority ceilings associated with nested PCP contexts be strictly increasing.

Note that we allow the system to disable time slicing while any task is executing with a PCP lock. Otherwise, some other task of equal priority might attempt to access the same monitor lock and would necessarily block. This would require that each lock maintain a queue of waiting tasks.

Note that we prohibit Core tasks from executing blocking operations while they hold a PCP lock. Otherwise, a task might block while it holds the PCP lock, making it possible for some other task of equal or lower priority to run and attempt to lock the same resource. In this case, the new task would have to block on a queue, waiting for the first task to complete its I/O operation and release the lock. But this contradicts our assertion that no queues are required in the implementation of priority ceiling locks.

Note also that we prohibit nesting of PCP locks. Otherwise, a multiprocessor implementation of the Core specification would likely experience deadlock for programs that run correctly on a single-processor implementation of the Core specification.

Copyright 1999, 2000 J Consortium, All Rights Reserved

C.12 Rationale for the CoreString and DynamicCoreString Specifications

At the March 30, 1999 meeting of the Real-Time Java Working Group, there were several requests to make CoreString very simple. However, there were also people who desired to retain a broader set of capabilities for CoreString. To satisfy both audiences, the API supports two classes: CoreString and DynamicCoreString. CoreString is intended to support string constants, as required for error messages and interactive user prompts. DynamicCoreString, which extends CoreString, supports additional capabilities. The expectation is that the DynamicCoreString class would be pruned from the load image in static applications for which it is not needed.

C.13 Rationale for Semaphores to Complement Built-In Java Primitives

Note that wait() and notify() are not appropriate signaling mechanisms for use from within interrupt handlers. The difficulty with using notify() from within an interrupt handler is that the interrupt handler must acquire the monitor lock before it can invoke the notify() operation. Since interrupt handlers are triggered by hardware (and not necessarily by the system dispatcher), it is not possible for interrupt handlers to block waiting for access to the monitor.

C.14 Rationale for the Mutex Class

The reason for providing Mutex lock() and unlock() operations in addition to providing the built-in locking mechanisms for synchronized statements is that the use of synchronized statements requires all locks to be released in LIFO order. There are particular algorithms that require locks to be released in a different order than LIFO.

Furthermore, though it would be possible for Core programmers to implement their own Mutex class by building upon the built-in synchronization wait() and notify() mechanisms, it would not be possible for application developers to implement priority inheritance for their Mutex implementation.

C.15 Comments on Loading and Starting Core Tasks from Baseline Domain

With a Dynamic Core Execution Environment, the Baseline domain is responsible for starting up the Core Execution Environment. It does so by instantiating a BaselineCoreClassLoader object using either one of the two constructors for this class (See Section 4.1). For example:

```
org.rtjwg.BaselineCoreClassLoader bccl = new BaselineCoreClassLoader();
```

Having created the primordial instance of BaselineCoreClassLoader, the Baseline component obtains a reference to the primordial instance of the org.rtjwg.CoreDomain class by executing code of the following form:

```
java.lang.Class cdc = bccl.findSystemClass("org.rtjwg.CoreDomain");
org.rtjwg.CoreDomain cd = null;
cd = cd.core;
```

The Baseline component uses the CoreDomain object to load and instantiate Core objects. The following code sequence, for example, loads a Core class named Sam-

pleCoreClass and instantiates it, assigning the instantiated object's reference to the ct variable. This code template assumes that SampleCoreClass extends org.rtjwg.CoreTask.

```
org.rtjwg.CoreClass cc = cd.loadClass("SampleCoreClass");
org.rtjwg.CoreTask ct = (org.rtjwg.CoreTask) cd.instantiate(cc);
```

To cause the newly instantiated ct task to begin running, the Baseline component invokes its Core-Baseline _start() method, as in the following code sample:

```
ct._start();
```

Note that the Baseline domain can only start CoreTask tasks. It cannot directly start periodic or interrupt-driven tasks. To start up other kinds of tasks, the Baseline domain creates a proxy CoreTask object to start up the periodic or interrupt-driven task, and then starts up the proxy CoreTask object.

C.16 Comments on Explicit Memory Management

By default, all memory allocated within a particular Core task is automatically released when that Core task terminates. This requires great care by Core programmers to make sure that no other task is allowed to see references to the objects it allocated. Otherwise, that other task will end up with a dangling pointer to the reclaimed object's memory. There are a few programming practices that are recommended to Core programmers:

1. Keep all references to objects allocated by your task local to your task, or
2. Make sure that your task runs forever, so its memory will never be released, or
3. Whenever it is necessary to allocate objects that must be visible to other tasks, allocate those objects from special AllocationContext regions which persist as long as the objects continue to be referenced.

C.17 Rationale and Discussion Regarding Asynchronous Transfer of Control

Asynchronous transfer of control describes the ability for one Core task to cause the control flow of some other Core task to change, asynchronously. We say this change is asynchronous because the affected task does not know or exert any control over when the control transfer takes place.

Asynchronous transfer of control is a common programming tool for dealing with real-world processes and events. As motivation for providing this programming language feature, consider the many ways that humans handle asynchronous events:

1. A telephone rings and we suspend whatever we are doing to answer it. Following completion of the phone call, we resume the previously suspended task.
2. A fire alarm sounds at work. In response, we abort the task on which we are currently working, lock our confidential papers into a fireproof safe, and leave the building.
3. While we are driving a car, we hear a siren. In response, we check rear view mirror and scan the road ahead for flashing lights. None is seen so we continue driving our established course.

Copyright 1999, 2000 J Consortium, All Rights Reserved

4. A student is taking a timed college entrance examination. She is notified that only five minutes remains for completion of the test. She abandons work in progress and begins darkening circles for the computer answer sheet.
5. While we are driving a car, we hear a siren. In response, we check the rear view mirror and scan the road ahead for flashing lights. A fire truck is seen in the rear view mirror so we pull to the side of the road and wait for it to pass. Once passed, we pull into the road and continue driving towards our intended destination.
6. A researcher is working on a 3-year federally funded project. Two years into the project, his administrative assistant informs him that he is 40% over the proposed spending budget. In response, the researcher modifies the remainder of the research plan in order to bring the project back into budget before proceeding with the last year's research efforts.
7. In a crowded meeting room, a cell phone rings. Five different people check to see if it is theirs. Only one interrupts his work to answer the phone. The others resume whatever activity they were already participating in.
8. A team of five developers is working on a six-month engineering project. Two months into the project, they are notified that funding cuts force the project to be abandoned. Each of the five developers is reassigned to other efforts.

These examples highlight the importance of supporting two forms of asynchronous transfer of control: (1) abortion and (2) resumption. The abortion form abandons whatever work was in progress when the asynchronous event is triggered. The resumption form allows a certain amount of work to be performed in response to the asynchronous event, following which the original work which was preempted is resumed.

An earlier revision (1.0.2) of the Core specification provided no general purpose asynchronous transfer of control mechanism. Instead, it provided explicit timeout forms of particular Core Library methods. During the public review period of that draft specification, the absence of general asynchronous transfer of control support was identified as a shortcoming in the Core specification. In discussing whether to add asynchronous transfer of control at our Dec. 7, 1999 meeting, the Real-Time Java Working Group considered the following:

1. Against adding asynchronous transfer of control:
    a. This would complicate the implementation of the Core Execution Environment, especially the implementations of operating system services that might block a Core task (e.g. a semaphore operation that must be timed out).
    b. This would represent a significant change to the Core specification, delaying publication of the final specification and probably requiring another public review period.
2. In favor of adding asynchronous transfer of control:
    a. The Core specification as originally drafted already required that blocking operating system services be timed out. Thus, the burden of implementing full asynchronous transfer of control is not perceived to be significantly greater than the burden of implementing the originally described specification.
    b. The group felt it would be better to have a stronger specification later than a weaker specification earlier.

c. Having fully general asynchronous transfer of control increases the relevance of the Core specification to a broader set of potential users. It also improves the expressive power available to Core programmers, making it easier to solve particular classes of programming problems.

d. Using asynchronous transfer of control in place of explicit timeout arguments for particular methods replaces many special-case situations with a single general-purpose solution. This makes it easier for programmers to use and maintain software components that interact with timeouts.

In the end, the Real-Time Java Working Group decided in favor of adding asynchronous transfer of control to the Core specification, provided that the various identified requirements could be satisfied to the mutual satisfaction of the member organizations.

There has been discussion and conflicting viewpoints on certain topics related to the design of the asynchronous transfer of control mechanism. In particular:

1. Why defer asynchronous event handling during execution of `finally` statements? The main observation is that `finally` statements generally represent cleanup code that is necessary to maintain the integrity of shared data structures and system-wide logical invariants. If an asynchronous event results in abortion of a particular code segment, all of the `finally` statements associated with that code segment will be executed as a side effect of the abort operation. If an asynchronous event is delivered during execution of a `finally` statement, we have two options:

   a. We could immediately interrupt the `finally` statement to execute the event handler, and resume execution of the `finally` statement after the interrupt handler completes, or

   b. We could defer execution of the event handler until after the `finally` statement completes its execution.

   In either case, the `finally` statement runs to completion. However, the first option introduces the risk that certain shared data structures may be in an inconsistent state during execution of the asynchronous event handler. For this reason, we chose to pursue the second option for the Core specification.

2. Why not defer asynchronous event handling during execution of all `synchronized` contexts? Programmers who are accustomed to programming real-time systems in the Ada programming language have come to expect that all synchronization is "abort deferred". A primary objection to adopting the Ada semantics is that deadlock situations cannot be remedied by aborting the offending tasks. For this reason, the Core specification does not defer asynchronous event handling during execution of `synchronized` code. We note that the type(s) of programming for which the Core specification is intended are more general than typical Ada applications (having more dynamic behavior, and using priority inheritance in addition to priority ceiling protocols for synchronization), which is part of the reason that we feel a different approach toward abortion of synchronized contexts is appropriate. Given that there do not currently exist any legacy Core applications, there is relatively low cost in adopting a different semantics than has been used for the Ada programming language.

Among the requirements for asynchronous transfer of control (See paragraph 5 of Section B.2 (starting on page 127)) is the ability to support common asynchronous pro-

Copyright 1999, 2000 J Consortium, All Rights Reserved

gramming idioms, such as abortion of a task, timeouts and nested timeouts for particular code sequences, software interrupts, and application mode changes. Here, we discuss how each of these idioms would be addressed with the proposed asynchronous transfer of control mechanism.

Abortion of a task. To abort a running Core task t, invoke its abort() method, as shown here:

```
t.abort();
```

Timing out a sequence of code. To establish a timeout on the sequence of code represented by the method named arbitraryCode(), structure the code as shown below:

```
class ScopedTimeoutException extends ScopedException { }

class TimeoutEvent extends ATCEvent {
  CoreThrowable exception;

  public TimeoutEvent(CoreThrowable scoped_exception) {
    exception = scoped_exception;
  }

  public void defaultAction() throws CoreThrowable {
    throw exception;
  }
}
```

Given the ScopedTimeoutException and TimeoutEvent classes defined above, the following code fragment demonstrates how to run the arbitraryCode() method with a watchdog timeout to abort its execution if it runs too long:

```
ScopedTimeoutException timeout_x = new ScopedTimeoutException();
TimeoutEvent timeout_e = new TimeoutEvent(timeout_x);
Alarm alarm;
alarm = timer.createAlarm();
try {
  alarm.setAlarmRelative(Time.ms(3), timeout_e);
  this.arbitraryCode();
} catch (ScopedTimeoutException z) {
  System.out.println("Code timed out after 3 ms.");
} finally {
  alarm.cancelAlarm();
}
```

In this sample code, we assume that this thread has an asynchronous event signal handler which simply invokes the defaultAction() of whatever event is signaled to this task. We also assume the existence of an application-defined timer object and application-defined Alarm class, the definitions of which are not provided here. The timer object supports a createAlarm() factory method, which creates an instance of Alarm that is bound to this particular timer object. The returned Alarm object is used by this thread to register requests for the timer object to deliver asynchronous timeout events at appropriate future moments in time. The Alarm class supports a setAlarmRelative() method, which takes as

arguments a long integer specifying the number of nanoseconds from the current time in which the timer service should send the asynchronous timeout event to this task and a reference to the scope-specific timeout event object that the timer service is to send at the appropriate time. Setting of the alarm by the setAlarmRelative() method is atomic in the sense that if execution of this method is aborted because this thread receives an asynchronous transfer of control signal, we are guaranteed that either the alarm has been completely set or that it has not been set. The Alarm class also supports a cancel() method, which has the effect of turning off the alarm if it was previously set. A side effect performed by the Alarm.cancelAlarm() method is to re-signal all "active" alarms. An active alarm is an alarm that was previously signaled and has not yet been canceled. The reason for specifying this behavior for cancelAlarm() is to simplify the handling of nested timeouts.

Nested timeouts. The approach described immediately above for implementing timeouts works properly for nested timeouts. Suppose, for example, that the implementation of this.arbitraryCode() includes code to set a nested timeout, using the same protocol detailed above. The possible interplay between nested timeouts is described by the following four scenarios:

1. If the inner-nested timeout occurs first, the inner TimeoutEvent object will be signaled to the task, and this will trigger event handling associated with the inner scope. The outer timeout remains pending.
2. If the outer-nested timeout occurs first, the outer TimeoutEvent object will be signaled to the task, and this will trigger event handling associated with the outer scope. Because TimeoutEvent objects use ScopedTimeoutException objects for their implementation, we are assured that a timeout event corresponding to an outer nested scope will not be mistakenly processed by an inner scope's timeout event handler. When exception handling for the outer timeout's exception unwinds the context within which the inner timeout context was established, the inner timeout is canceled.
3. Suppose the inner timeout occurs first, and then the outer timeout occurs while we are still "handling" the inner timeout's event. There are two cases to consider:
    a. If the defaultAction() method has already thrown its exception object, handling of the outer nested timeout is deferred until after all of the finally statements associated with handling of the thrown exception have completed their execution. Note that the timeout context's catch clause will not be allowed to execute.
    b. If the defaultAction() method has not yet thrown its exception object, the outer timeout's event handler immediately preempts the inner timeout's defaultAction() method and throws its exception object. In this case, the inner timeout event handler never gets a chance to throw its exception, because the outer timeout aborts the inner timeout's event handler.
4. Suppose the outer timeout occurs first, and then the inner timeout occurs while we are still "handling" the outer timeout's event. There are two cases to consider:
    a. If the outer timeout's defaultAction() method has already thrown its exception object, handling of the inner nested timeout is deferred until after all of the finally clauses associated with handling of the thrown exception have completed their execution. In the process of unwinding the stack for the outer timeout, we cancel the alarm and disable the ScopedTimeoutException object for the inner nested timeout. When the inner timeout's event handler is eventually executed,

Copyright 1999, 2000 J Consortium, All Rights Reserved

it will throw the disabled ScopedTimeoutException. This has the effect of simply resuming the code that immediately follows the context of the outer-nested timeout exception.

b. If the outer timeout's defaultAction() method has not yet thrown its exception object, the inner timeout's event handler preempts the outer timeout's event handler. When the inner timeout's defaultAction() method throws its ScopedTimeoutException object, this causes the stack to unwind to the point of the inner timeout's context. That context's finally clause causes the alarm to be canceled. Execution of alarm.cancel() causes the outer context's timeout event to be re-signaled.

Software interrupts. The idea of software interrupts is to allow one Core task to cause some other Core task to execute a special code sequence (an "interrupt handler") and then resume whatever code was previously executing.

It is straightforward to implement software interrupts using the asynchronous transfer of control system described in this specification. To cause another task to execute its "software interrupt handler" (also known as its event handler), invoke the task's signalAsync() method, passing as an argument a reference to an ATCEvent object that provides whatever application-specific information is required as parameters to the event handler.

The task to which the event is signaled must provide an appropriate asynchronous event handler which executes the desired interrupt handling code and then returns. Upon return from the asynchronous event handler, the code within the task that was executing when the asynchronous event was signaled is resumed.

System mode changes. The notion of a system mode change is that a complex system comprised of many cooperating tasks may operate in multiple modes. For example, the control software for a fighter aircraft may have modes dedicated to such independent activities as takeoff, cruise, evade incoming missiles, engage enemy aircraft, and land. Each time the system transitions from one mode to another, multiple tasks need to be informed of the transition.

The two most common ways of supporting mode changes in complex software systems comprised of multiple cooperating tasks are:

1. Each task is required to periodically poll a system state variable which reports when the system is transitioning to another mode. Each task is independently responsible for performing whatever work is necessary to effect the transition.
2. When a mode change is required, a supervisor activity signals this requirement by delivering an asynchronous event to each of the cooperating tasks. Each task's asynchronous event handler is responsible for performing whatever work is necessary to effect the transition.

C.18 Comments re: low-level I/O Services

A previous draft of this specification included a much more sophisticated collection of I/O services. The design of that earlier set of services was patterned after the Real-Time Data Access profile, which is currently under development within a working group of the J Consortium. A July 2000 teleconference call involving the memberships of both

the Real-Time Java Working Group and the Real-Time Access Working Group concluded that it would be best to remove the Real-Time Data Access compatibility from the Core specification. The main reason for this change was that the Real-Time Data Access profile is maturing and evolving independently of the Core specification, and it is very difficult to keep the two documents synchronized. Instead, it was felt that the Real-Time Data Access profile could be written so as to complement the Core specification. Ultimately, we expect the Real-Time Access Working Group to produce two variants of the Real-Time Data Access profile - one describing extensions to the Baseline environment, and the other describing extensions to the Core.

Having removed the generality of the Real-Time Data Access services, it was necessary to replace these with simpler primitive API libraries. Thus, the IOPort, ISR_Task, and SporadicTask classes were introduced.

Copyright 1999, 2000 J Consortium, All Rights Reserved

Annex D Implementation Suggestions

D.1 Comments on the Implementation of Partitioned Heaps (Section 3.3)

There are many possible implementations for the memory management system described in this section. Here, we offer comments describing one possible implementation.

1. When a Core object is allocated, it is allocated from a region of memory that is normally garbage collected using mark-and-sweep (non-relocating) techniques. Note that techniques are available to allow coexistence of mark-and-sweep garbage collection with copying garbage collection.
2. At the moment a Core object is allocated, a reference to the object is stored into a Baseline hash table. As long as this reference to the object continues to exist in the hash table, the object shall not be garbage collected. Since the object was allocated from a mark-and-sweep region, the object shall not be relocated.
3. The garbage collector marks and scans the anchored Core object, treating it like every other object in the mark-and-sweep region. The object shall not be treated as garbage because we know the hash table holds a live pointer to the object.
4. When a Core task releases an allocation context, the references to all of the objects belonging to that allocation context which were stored into the Baseline hash table in step 2 above are removed from the hash table. If garbage collection is active at the moment the allocation context is released, all of the newly released objects are marked as live for purposes of this pass of the garbage collector. At this point, the released objects are now eligible to be garbage collected.

   The reason the object must be marked as live for this pass of the garbage collector is because the recent actions of the Core tasks are not necessarily visible to the garbage collector. Recent Core actions may have affected the pointer paths by which this object is known to be reachable (i.e. live). After an object's allocation context has been released, any further changes to the object's reachability graph must be performed by Baseline components, all of which implement appropriate read and write barriers. Thus, subsequent passes of the garbage collector shall be able to identify the object as unreachable and reclaim its memory.

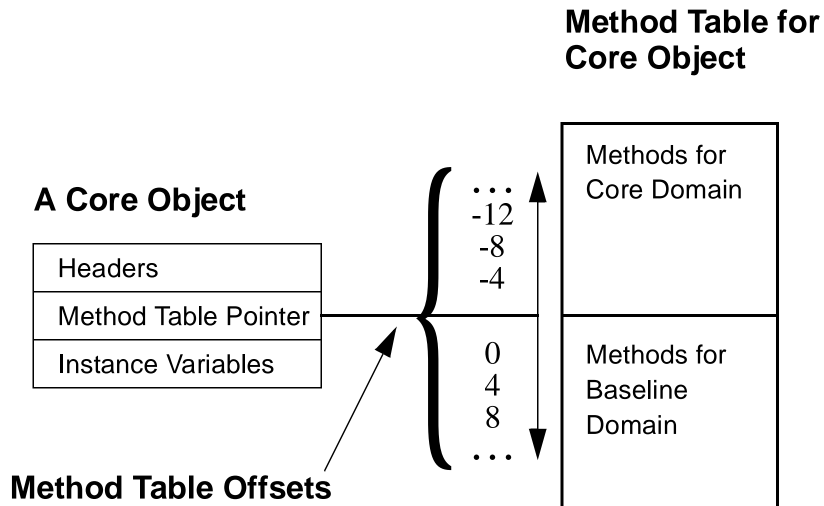
D.2 Comments on Implementation of Multiple Method Tables (Section 3.3)

Each Core object must implement two method tables, one to support the Core-Baseline methods and the other to support the Core methods. There exist many different possible implementations of Core object method tables. Here, we describe one possible implementation.

Note that each Core object must support two different interfaces. Within Core tasks, the Core object must support the Core API (all the Core methods). If the Core object is published to the Baseline world, the Core object must also support the Baseline API (everything inherited from the Baseline `java.lang.Object` class, plus any Core-Baseline methods declared for that object or its Core superclasses). One way to efficiently implement the two different method interfaces is to augment the traditional virtual method table so that it represents two tables in a single data structure, using positive offsets to represent the

Baseline method table, and using negative offsets to represent the Core method table, as illustrated in Figure 5 on page 154.

Figure 5. Method Tables for Core Objects



D.3 Comments on Implementation of Stack Allocation

Once a class loader has determined which objects are stack allocatable, there are at least two possible approaches for the implementation of the new memory allocation requests that correspond to the stack allocatable objects. Assume that the Core Class Loader replaces `new` invocations with a special `stack-new` operation for each new memory allocation request that assigns its result to a stackable variable.

Dynamic stack allocation. One possible approach toward stack allocation is to implement the `stack-new` operation using the same implementation that is typical for implementations of the C `alloca()` service. In particular, each time the `stack-new` operation is invoked, the stack is expanded to make space for the new stack-allocatable object and the object is allocated and initialized from the newly available stack space.

Static stack allocation. An alternative approach toward stack allocation of objects is to have the Core Class Loader arrange for space in the method's stack activation frame to represent one copy of each stack-allocatable object for each `stack-new` operation found within the method. The memory for these objects shall be initialized at the moment the corresponding `stack-new` operation is executed. Note that stack-allocation of arrays whose size is not known until run-time must uses a form of dynamic stack allocation.

 Copyright 1999, 2000 J Consortium, All Rights Reserved

A

Aborting a Task 25
Acknowledgments 109
Active Priority 4
Allocation Context 3
AllocationContext 69
  allocated() 39
  available() 39, 71
  constructors 39, 70
  release() 39, 71
Analyzability 33
Architectural Overview of the Core Implementation 12
  illustration 13, 140
Asynchronous Transfer of Control 3, 25, 26, 26
  ATCEvent 77
  ATCEventHandler 76
  rationale 147
ATC 3
ATCEvent 77
  constructor 43, 77
  defaultAction() 43, 77
ATCEventHandler 76, 102
  constructor 43, 77
  handleATCEvent() 43, 77
Atomic 73

B

Base Priority 4
Baseline 2
baseline 5
Baseline API 104
  Core Execution Profiles 105
  semaphore operations 105
  Starting up a CoreTask 105
Baseline APIs
  Starting up a Core Execution Environment 105
Baseline Compiler 13, 13
baseline keyword 5
Baseline Virtual Machine 13
BaselineCoreClassLoader 105
  constructors 105
  semantics 106

C

C/Native API 57
can (as a normative term) 2
Class Initialization 29
Class Loading 29
Class Resolution
  rationale 144
Configuration 82
  default_stack_size 83
  little_endian 84
  min_core_priority 83
  stack_overflow_checking 83
  system_priority_map 83
  tick_duration 83
  ticks_per_slice 83
  uptime_precision 83
Conformity Assessment 6
Cooperation between Core and Baseline components 127
Cooperation between Core and High-Level Real-Time Profiles 127
Core 3
Core Class File 6, 12, 13, 15
Core Class Files 6, 12, 15
Core Class Libraries 13, 14
Core Class Loader 15, 18, 66, 144
  Initialization and Class Loading 29
  unloading classes 66
Core Class Loading 57
Core Compiler 140
Core Components 3
Core Execution Environment 14
Core Memory Model 24
Core Methods 3
Core Native Compiler 13, 14
Core Native Interface Compiler 8
Core Objects 8
Core Preprocessor 139
Core Priorities 21
Core Source File 12
Core Static Linker 15
Core Task Execution Model 24
Core Throwable Types 101
Core Verifier 6, 13, 14, 18
  rationale 139
CoreArithmeticOverflowException 56, 104
CoreArray 67
  atGet() 39, 69
  atPut() 39, 69
  constructors 39
  length() 39, 69
CoreArrayIndexOutOfBoundsException 56, 103
CoreATCEventsIgnoredException 56, 104

CoreBadArgumentException 56, 103
CoreBadPriorityException 56, 103
Core-Baseline Methods 3, 24
CoreBaselineException 108
  constructors 108
  getCoreException() 109
CoreBaselineRuntimeException 108
  constructors 108
  getCoreException() 108
CoreBaselineThrowable 107
  constructors 107
  getCoreThrowable() 108
CoreBoolArray 56, 68
CoreByteArray 56, 68
CoreCharArray 56, 68
CoreClass 65
  forName() 37, 65
  getComponentType() 37, 65
  isArray() 37, 65
  isAssignableFrom() 37, 65
  isInstance() 37, 65
  isInterface() 37, 66
  isPrimitive() 37, 66
  loadClass() 38, 66
  newInstance() 38, 66
  toString() 38, 66
  unloadClass() 38, 66
  verification() 38, 66
CoreClassFormatError 56, 103
CoreClassInUseException 56, 104
CoreClassLoader 57
CoreClassNotFoundException 56, 104
CoreDomain 106
  core 106
  defineClass() 106
  instantiate() 107
  loadClass() 107
  lookup() 106
  profiles() 107
CoreDoubleArray 56, 69
CoreEmbeddedConflictException 56, 103
CoreException 56, 64, 101
  constructors 36, 64
CoreFloatArray 56, 68
CoreIllegalMonitorStateException 56, 103
CoreIntArray 56, 68
coreInterruptLevels() 56, 58
CoreLongArray 56, 68
CoreObject 61
  arrayAddress() 35, 62
  clone() 35, 61
  constructor 61
  constructors 35
  equals() 35, 61
  getClass() 35, 61
  hashCode() 35, 62
  notify() 35, 62
  notifyAll() 35, 62
  sizeof() 35, 63
  toString() 35, 62
  wait() 35, 62
CoreObjectNotAddressableException 56, 104
CoreOperationNotPermittedException 56, 103
CoreOutOfMemoryException 56, 103
corePriorityMap() 56, 58
CoreRefArray 56, 69
CoreRegistry 77
  coerce() 44, 78
  profiles() 44, 79
  publish() 45, 79
  registerBaseline() 44, 78
  registerCoreClass() 44, 78
  registerStackable() 44, 78
  stackAllocation() 44, 77
  unpublish() 45, 79
coreRegistryLookup() 56, 57
CoreRuntimeException 36, 56, 63, 102
  constructors 36, 63
CoreSecurityException 56, 103
CoreShortArray 56, 68
CoreString 74
  _charAt() 40, 74
  _hashCode() 40, 74
  _length() 40, 75
  charAt() 40, 74
  constructors 40, 74
  equals() 40, 74
  hashCode() 40, 74
  length() 40, 75
CoreTask 89
  _start() 94
  abort() 51, 92

Copyright 1999, 2000 J Consortium, All Rights Reserved

abortWorkException() 51, 92
asyncHandler() 51, 92
constructor 50, 89
currentTask() 50, 90
defaultStackSize() 50, 90
join() 51, 92
maxBaselinePriority() 50, 90
maxCorePriority() 50, 90
maxSystemPriority() 50, 91
minBaselinePriority() 50, 91
minCorePriority() 50, 91
minSystemPriority() 50, 91
numInterruptPriorities() 50, 91
resume() 51, 93
setPriority() 51, 93
signalAsync() 51, 93
sleep() 51, 94
sleepUntil() 51, 94
stackDepth() 51, 94
stackOverflowChecking() 50, 91
stackSize() 51, 94
start() 51, 94
stop() 51, 94
suspend() 51, 95
systemPriority() 52, 95
systemPriorityMap() 50, 91
ticksPerSlice() 51, 92
work() 52, 95
yield() 52, 95
CoreThrowable 63, 101
constructors 36, 37, 63
getMessage() 36, 37, 63
CoreUnsignedCoercionException 56, 104
CountingSemaphore 81
_count() 81
_numWaiters() 81
_P() 81
_V() 81
constructor 47
count() 47, 81
numWaiters() 47, 81
P() 47, 81
V() 47, 81

D

Differences between CoreString and DynamicCoreString
rationale 146
Differentiating Core and Baseline Tasks 144
Dynamic Class Loading
eager resolution and initialization 29
rationale 138, 144
requirement 128, 129
Dynamic Core Application 6
Dynamic Core Development Environment 7
Dynamic Core Execution Environment 13, 15
dynamic properties 2
DynamicCoreString 75
_length() 41
concat() 41, 75
constructors 41, 75
getChars() 41, 75
length() 41
substring() 42, 76
toCharArray() 42, 76
toLowerCase() 42, 76
toUpperCase() 43, 76

E

enterSynchronized() 57, 59
Execution-Time Analyzable Code 29
clarification 144
specification of conforming behavior 29
exitSynchronized() 57, 60
Explicit Memory Management
rationale 147
Extended Baseline Virtual Machine 3

G

Garbage Collection Overhead
requirement 127
Green Threads 3

H

Historical Background 131

I

I/O Channel 4
I/O-Space Access 4
implementation defined (as a normative term) 2
Initialization of Core Classes 29
IOPort 99, 153
createIOPort() 54, 100

readByte() 54, 100
readInt() 54, 101
readLong() 54, 101
readShort() 54, 100
writeByte() 54, 100
writeInt() 54, 101
writeLong() 54, 101
writeShort() 54, 101
IOPort16I 100
IOPort16IO 100
IOPort16O 100
IOPort32I 100
IOPort32IO 100
IOPort32O 100
IOPort64I 100
IOPort64IO 100
IOPort64O 100
IOPort8I 100
IOPort8IO 100
IOPort8O 100
ISR_Task 95, 153
arm() 52
ceilingPriority() 52, 97
constructor 52, 96
disarm() 52
serviced() 52, 97
trigger() 52
work() 52, 97
ISR_Task.arm() 98
ISR_Task.disarm() 98
ISR_Task.trigger() 97

J

Java 2

L

Limited Cooperation between Core and Baseline
requirement 127
Limited Cooperation between Core and Baseline Java
requirement 127
Limited Cooperation between Core and High-Level Real-Time Profiles
rationale 133
Limited Cooperations between Core and Baseline
rationale 133
Limited Cooperations between Core and High-Level Real-Time Profiles
requirement 127
Loading and Starting Core Tasks from Baseline domain 146

M

maxBaselinePriority() 56, 58
maxCorePriority() 56, 58
Maximal Throughput
rationale 134
requirement 127
maximal throughput 127
may 1
may not 1
Memory footprint
requirement 128
Memory-Mapped Access 4
minBaselinePriority() 56, 58
minCorePriority() 56, 58
Minimal Latency
rationale 133
requirement 127
minimal latency 127
Mutex 82
_lock() 48, 82
_unlock() 48, 82
constructor 48
constructors 82
lock() 48, 82
unlock() 48, 82
Mutex Class
rationale 146

N

Native-Targeted Core Class File 13
Native-Targeted Core Class Files 14
NCITS Principles of the Real-Time Core 131
Nested timeouts 151
Never-Scheduled Priority 4
NIST Requirements for the Real-Time Core 131

O

ObjectNotFoundException 107
overhead of coordinating with a garbage collector 127

P

Partitioning of Memory 9
comments on implementation 154
comments on implementation of multiple method tables 154
Perspective of the Baseline programmer 12
Perspective of the Core programmer 11

Copyright 1999, 2000 J Consortium, All Rights Reserved

rationale 134, 138
PCP 72
ceilingPriority() 73
Portability
rationale 137
requirement 128
Predictability of Core Execution Environment 34
Priority Ceiling Protocol
rationale 145
Profiles
requirement 128
Programming Language Security
rationale 137
requirement 127

R

Real-Time Java Working Group 127
RTOS 3
Run-time Queues
specification of behavior 21

S

Scope 1
ScopedException 64, 102
constructors 64
disable() 102
enable() 102
ScopedException.disable() 65
ScopedException.enable() 64
Semaphore Classes
rationale 146
semaphoreP() 56, 59
semaphoreV() 56, 59
semaphoreVall() 56, 59
Sending feedback to the J Consortium 1
shall 1
shall not 1
should 1
should not 1
SignalingSemaphore 80
_numWaiters() 80, 81
_P() 80
_V() 80
_Vall() 80
constructor 46
numWaiters() 46, 80
P() 46, 80
V() 46, 80
Vall() 46, 80
SpecialAllocation 71
context() 39, 71
execute() 39, 72
run() 39, 71
SporadicTask 98, 153
clearPending() 53, 99
constructor 53, 98
pendingCount() 53, 99
trigger() 53, 99
work() 53, 99
Stack Allocation 27
comments on implementation of 155
rationale 142
requirement 128
specification of behavior 27
stackable 5
stackable keyword 5
Static Core Application 6
Static Core Development Environment 6
Static Core Executable Load Image 7, 13, 14
Static Core Execution Environment 13, 14
Static Core Linker 7, 13
static properties 2
Stylized Core Source Code
special notations 20
Stylized Core Source File 12, 13, 20
Synchronization Issues 21
Synchronizing and Coordinating Between Core and Baseline 59
Syntactic Core Extensions 139
Syntactic Core Source Code
special notations 141
Syntactic Core Source Files 139

T

Terminology 1
The Core Verifier 18
Time 84
day() 49, 84
h() 49, 84
hertz() 49, 85
m() 49, 85
ms() 49, 85
ns() 49, 85
s() 49, 85
tickDuration() 49, 84

toString() 49, 85
uptime() 49, 86
uptimePrecision() 49, 84
us() 49, 86

U

undefined behavior 2
Unsigned 86
compare() 55, 86
eq() 55, 87
ge() 55, 86
gt() 55, 86
le() 55, 87
lt() 55, 87
neq() 55, 87
toByte() 55, 88
toHexString() 55, 89
toInt() 55, 88
toLong() 55, 88
toShort() 55, 88
toString() 55, 88
unsigned integers 86, 99
unspecified behavior 2

W

WCET 29
Working Principles of the Real-Time Java Working Group 127

Copyright 1999, 2000 J Consortium, All Rights Reserved