

INTERNATIONAL J CONSORTIUM[®] SPECIFICATION

High Integrity Profile

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



P. O. Box 1565
Cupertino, CA 95015-1565
USA
www.j-consortium.org

Copyright J Consortium 2000, 2001

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.

J Consortium Specification No. TBD

Table of Contents

1. GENERAL.....	4
1.1 SCOPE	4
1.2 EXTENT	4
1.3 STRUCTURE.....	5
2. TERMINOLOGY CONVENTIONS.....	6
2.1 NORMATIVE TERMS.....	6
2.2 PROGRAM LANGUAGE AND TECHNICAL TERMINOLOGY.....	6
3. HIGH INTEGRITY PROFILE SPECIFICATION.....	8
3.1 EXECUTION ENVIRONMENT.....	8
3.1.1 <i>Program Link and Class Load</i>	8
3.1.2 <i>System Configuration</i>	9
3.1.3 <i>Program Execution</i>	9
3.1.4 <i>Memory Management</i>	11
3.1.5 <i>Modes of Operation</i>	13
3.2 RESTRICTIONS APPLYING TO THE CORE SPECIFICATION	14
3.2.1 <i>Static Execution Environment</i>	14
3.2.2 <i>Stand-Alone Execution Environment</i>	14
3.2.3 <i>HIP Tasks</i>	15
3.2.4 <i>Task Creation</i>	16
3.2.5 <i>Task Activation</i>	17
3.2.6 <i>Task Dispatching</i>	18
3.2.7 <i>Task Termination</i>	18
3.2.8 <i>Synchronized Code</i>	19
3.2.9 <i>Synchronization Primitives</i>	20
3.2.10 <i>Asynchronous Operations</i>	20
3.2.11 <i>Timing Operations</i>	20
3.2.12 <i>Throwable Classes</i>	21
3.3 ACTION ROUTINES	21
APPENDIX A EXCLUSIONS FROM CORE APIS AND CLASSES	22
APPENDIX B EXTENSIONS TO CORE APIS AND CLASSES	23

High Integrity Profile

This document represents a draft revision to the specification of the High Integrity Profile of the Real-Time Core Extensions for the Java Platform, based on the collective work of the members of the J Consortium's Real-Time and High Integrity Working Groups.

1. General

The High Integrity Profile is a specification designed to support the construction of highly reliable software systems. The semantics of the operational model defined by this specification are described in English, together with a notation based on the Java programming language and programming platforms as these were originally described by Sun Microsystems, Inc in references TBD, including the many variants that have come into existence since the original specifications were published.

The profile is intended for the development of high integrity programs. The term “high integrity” applies to the following application categories:

1. Safety Critical / High Integrity, for which all the data and executable code must be functionally and spatially deterministic in order to undergo thorough analysis and verification, and must be assured not to be corruptible by less trusted code;
2. High Reliability / Fault Tolerance, for which the code must be resilient enough to detect faults and to recover with minimal disturbance to the overall system;
3. Hard Real-Time, for which the timing of code execution must be deterministic to ensure that hard deadlines are met;
4. Embedded, for which a very small memory footprint and fast execution times are required.

1.1 Scope

This International Specification defines the meaning of programs written to conform to the High Integrity Profile operational model. The purpose of this International Specification is to promote safety, portability, reliability, maintainability, and efficient execution of such programs to a variety of computer systems.

1.2 Extent

This International Specification specifies:

- The effect of executing a program that conforms to the High Integrity Profile operational model;
- The classes that a conforming implementation is required to supply;
- The permissible variations within the specification that do not affect conformance;
- Those violations of the specification that a conforming implementation is required to detect, and the effect of executing a program containing such violations;
- Those violations of the specification that a conforming implementation is not required to detect.

This International Specification does not specify:

- The means whereby a program that conforms to the High Integrity Profile operational model is transformed into executable form, including the source code in which it originates (if any);
- The means whereby execution of a program is invoked;
- The size or speed of execution, or the relative execution speed of different operations;

- The form or contents of any outputs produced automatically by implementations; in particular, the form or contents of error or warning messages;
- The effect of unspecified or undefined behavior;
- The size of a program or usage of implementation-defined resources that exceeds the capacity of a conforming implementation.

1.3 Structure

This International Specification contains three chapters and two annexes.

Chapter 1 is a general introduction to this specification.

Chapter 2 defines the terminology conventions used within this specification.

Chapter 3 contains the technical details of the specification.

Appendix A lists the APIs, classes and methods defined in the Core specification that are not supported by this specification. The material in this Appendix is informative.

Appendix B lists the APIs, classes and methods that are extensions to the Core specification. The material in this Appendix is informative.

2. Terminology Conventions

2.1 Normative terms

Throughout this International Specification, 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 has the semantics of “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.

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

Implementation-defined behavior. 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.

Unspecified behavior. This identifies behavior for a correct program construct and correct data, for which the specification explicitly imposes no requirements.

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.

Notes. The text within sections headed “Notes” provides further information on the consequences of the rules described elsewhere. This material is informative.

2.2 Program Language and Technical Terminology

Application Programming Interface (API). A set of predefined classes and interfaces made available to the application developer in order to support the programming concepts defined in this International Specification.

Baseline Java. Throughout this International Specification, the word “Java” and the term “Baseline Java” are used to describe the Java programming language and programming platforms as these were originally described by Sun Microsystems, Inc, in references TBD, including the many variants that have come into existence since the original specifications were published. Java is a trademark of Sun Microsystems, Inc in the U.S. and other countries.

Core specification. The International Specification “Real-Time Core Extensions” reference TBD

HIP specification. This “High Integrity Profile” International Specification.

Interrupt Service Routine (ISR). A function or method whose execution is triggered automatically by an interrupt that is triggered either by software or hardware.

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.

Task. An independently schedulable thread of control.

3. High Integrity Profile Specification

This specification defines the J Consortium High Integrity Profile (hereafter known as the “HIP”) of the Real Time Core Extensions International Specification (hereafter known as the “Core specification”). The profile is named `org.j-consortium.hip`.

The HIP specification consists of the definition of the HIP Execution Environment, the HIP-specific API and a set of restrictions that apply to the Core specification for a HIP-conformant implementation.

3.1 Execution Environment

3.1.1 Program Link and Class Load

The HIP Execution Environment shall be based on the Static Core Execution Environment as defined in the Core specification. A HIP executable load image is built by a static linking system that links together:

1. the application Core class files
2. Core class libraries that are determined by the static linking system as being potentially used by the application
3. the HIP runtime environment.

Verification of conformance to the HIP specification shall be performed prior or during the static linking process. A HIP executable load image shall be generated only if all constituent Core class files conform to the restrictions imposed by the HIP specification. A HIP executable load image is defined to be a HIP program.

The following static method that is defined as relating to optional class verification in the Core specification shall not be supported by a HIP implementation:

`CoreClass.verify()`

The HIP Execution Environment shall not support the equivalent of the Dynamic Core Execution Environment. Consequently all aspects of the Core API that relate to the Dynamic Core Execution Environment, and hence to the definition of interaction with a Baseline Java Virtual Machine (JVM), are restricted in the HIP subset (see section 3.2.1).

Implementation-Defined Behavior:

It is implementation-defined whether the resulting HIP program is executed by an underlying operating system or not.

It is implementation-defined whether native method programming is supported and if so, how the native methods are linked into the HIP executable load image. If native method programming is supported then the `CoreObject.arrayAddress()` method should be supported.

The means by which verification of conformance to the HIP specification is implemented is implementation-defined (but is required).

Since all interaction with a Baseline JVM is restricted, the dynamic loading of HIP classes is not supported. An implementation may perform dynamic loading of an entire HIP program, or any part of such a program, e.g. a dynamic linked library. If an

implementation provides this capability, the means by which such a load is achieved is implementation-defined.

Notes:

The Core classes that form part of a HIP program are identified explicitly using a call to the `CoreRegistry.registerCoreClass()` method as defined in the Core specification.

All HIP classes are derived from `CoreObject`.

There is no requirement for the HIP execution environment to include a Java byte code interpreter. HIP programs are intended to be executed using native code. This may be machine object code, or Java byte codes executed directly by the hardware.

3.1.2 System Configuration

The HIP Execution Environment is configured via the settings of the constants in the class `Configuration`. The HIP implementation shall implement the configuration that is defined by these constants. The following constants whose semantics are defined in the Core specification are in class `Configuration`:

`Configuration.tick_duration`
`Configuration.uptime_precision`
`Configuration.little_endian`

The following constant is added to the class `Configuration` and shall be supported by a HIP implementation:

`Configuration.num_interrupt_priorities`

The constant `Configuration.num_interrupt_priorities` defines the number of interrupt priority values that are supported by this implementation. The actual priority values shall be such as to occupy the high end of the priority range, as defined in the Core specification.

Notes:

The HIP Configuration constants are not merely guidelines, or requests, as in the Core specification. They can be used reliably to provide the actual configuration settings.

The constant `Configuration.num_interrupt_priorities` is defined instead of supporting static method `CoreTask.numInterruptPriorities()` as in the Core specification since the number of interrupt levels is a property of the configuration of the underlying architecture, rather than a property of any specific task.

If `Configuration.num_interrupt_priorities` were 16 for example, the interrupt priority range would be from priority 113 to priority 128 inclusive.

3.1.3 Program Execution

A HIP program shall include one class, known as the main class, that defines the main method. The main method is executed by an implicit task, known as the Environment task. The main class shall implement the interface `HIPProgramActionRoutines`.

Prior to execution of the Environment task, the implementation shall:

- ◆ inhibit delivery of interrupts to the HIP program
- ◆ defer task dispatching

- ◆ construct the predefined instance of each of the following sub-classes of `CoreThrowable`, using the no-argument constructor as defined in the Core specification:

`CoreArithmeticOverflowException`
`CoreArrayIndexOutOfBoundsException`
`CoreBadArgumentException`
`CoreBadPriorityException`
`CoreEmbeddedConflictException`
`CoreIllegalMonitorStateException`
`CoreOutOfMemoryException`
`CoreUnsignedCoercionException`

The following additional sub-class of `CoreException`, as defined in the Core specification, is defined by this specification:

`HIPSubsetViolationException`

The signature of the constructor for class `HIPSubsetViolationException` is as follows:

```
public HIPSubsetViolationException();
```

The usage of sub-class `HIPSubsetViolationException` is to report violations of the restrictions imposed by this specification for HIP conformance that are detectable during HIP program execution.

Prior to execution of the Environment task, the implementation shall construct the predefined instance of `HIPSubsetViolationException`.

The implementation shall start the Environment task.

The Environment task shall initialize every class that is part of the HIP program. The order in which class initialization is performed shall be determined prior or during the static linking process, and shall be consistent with the following partial orderings:

- ◆ If an interface `A` extends another interface `B`, then the initialization of `B` shall occur prior to the initialization of `A`.
- ◆ If a class `C` extends another class `D`, then the initialization of `D` shall occur prior to the initialization of `C`.

After all class initialization is complete, the Environment task shall call the action routine `onStartup()` (see section 3.3).

After exit from the `onStartup()` action routine, the implementation shall enable the delivery of interrupts for the HIP program and shall invoke deferred task dispatching. The Environment task shall then call the main method.

If an unchecked exception of the class `CoreRuntimeException` is thrown and not caught by any class initialization code, or by the `onStartup()` action routine, the Environment task terminates. The Environment task also terminates when the main method exits. Execution of the HIP program becomes completed when all HIP tasks, including the Environment task, are either terminated or not started. At completion of the HIP program, the implementation shall call the action routine `onShutdown()`.

Implementation-Defined Behavior:

If the `CoreObject.arrayAddress()` method is supported, the implementation shall construct the predefined instance of the class `CoreObjectNotAddressableException`, using the no-

argument constructor as defined in the Core specification, prior to execution of the Environment task.

The order in which class initialization is performed is implementation defined, subject to the partial ordering requirements above. An implementation may provide the means to specify a full or partial ordering of the initialization of the classes in the HIP program.

If the `onShutdown()` action routine returns control to its caller, or throws an unchecked exception, the behavior is implementation defined.

Notes:

The construction of the predefined instances of the exception classes allows the implementation to throw any of these (state-less) exception objects as necessary throughout the execution of the HIP program, including during the class initialisation code.

The calls to the action routines provide hooks for application-specific action at program startup and shutdown. The routines that are called are methods belonging to the implicit object of the main class (that implements the interface which defines these methods).

3.1.4 Memory Management

The implementation shall support stack allocation of objects; that is, the programmatic identification of stackable objects as defined in the Core specification.

The following static method that is defined as relating to optional support for stackable objects in the Core specification shall not be supported by a HIP implementation:

`CoreRegistry.stackAllocation()`

The dynamic method call stack for a HIP task can be created by construction of an object of class `HIPStack` which extends `CoreObject`.

There are two constructors for `HIPStack`; the first allocates the `HIPStack` object within the current default allocation context; the second allocates the `HIPStack` object within a named special memory block. If the named memory block does not exist, or the HIP Execution Environment does not permit allocation of a `HIPStack` object within the named memory block, the second constructor shall throw the predefined instance of `CoreEmbeddedConflictException`.

Each constructor shall have an argument `maximum_bytes` that defines the maximum total number of bytes authorized to be allocated contiguously for the HIP stack object. If `maximum_bytes` is negative or zero, the implementation determines the maximum total number of bytes by calling the action routine `defaultStackSize()` and uses the returned value instead of `maximum_bytes`.

The following configuration constant that is defined as relating to default stack size in the Core specification shall not be supported by a HIP implementation:

`Configuration.default_stack_size()`

Each constructor shall create a `HIPStack` object that is configured to allocate no more than the specified number of bytes. Each constructor shall throw the predefined instance of `CoreOutOfMemoryException` if insufficient memory exists to allocate the `HIPStack` object.

The signature for the first constructor is:

```
public HIPStack (long maximum_bytes)
throws CoreOutOfMemoryException, HIPSubsetViolationException;
```

The signature for the second constructor is:

```
public HIPStack (long maximum_bytes, CoreString block_name)
throws CoreOutOfMemoryException, CoreEmbeddedConflictException,
HIPSubsetViolationException;
```

The following instance method for class **HIPStack** is defined:

HIPStack.stackSize(): The method returns the actual number of bytes that the **HIPStack** object is configured to allocate, when used as a method call stack.

The implementation shall support stack overflow checking for all HIP tasks. The following constant and static method that are defined as relating to optional support for stack overflow checking in the Core specification shall not be supported by a HIP implementation:

```
Configuration.stack_overflow_checking
CoreTask.stackOverflowChecking()
```

Each HIP object that is not stackable shall be allocated in the current default allocation context as in the Core specification. There shall be a global memory area that represents the default allocation context at program startup. The default allocation context is changed during program execution as defined in the Core specification.

If insufficient memory exists in an allocation context to satisfy an allocation request, the allocation shall throw the predefined instance of **CoreOutOfMemoryException**.

An allocation context shall have a fixed size and shall map to a contiguous memory address range. An allocation context may be located in a specially-named memory area, as defined in the Core specification. The following constructor that is defined for class **AllocationContext** in the Core specification shall not be supported by a HIP implementation:

```
public AllocationContext ();
```

All instantiations of the classes **HIPStack** and **AllocationContext** shall occur prior to the call to the main method; that is, as part of either the class initialization code or the action routine **onStartup()**. If an attempt is made to construct a **HIPStack** or **AllocationContext** object after the call to the main method, the constructor shall throw the predefined instance of **HIPSubsetViolationException**.

Implementation-Defined Behavior:

It is implementation-defined how the static linking system is used to create application-specific global memory areas for allocation of objects during program startup.

The implementation may provide the means to specify one of the application-specific global memory areas that are created during the static linking process to be the default allocation context at program startup.

The implementation shall configure a **HIPStack** object according to the architecture-specific characteristics of a dynamic method call stack. In particular, all allocations shall obey any architecture-specific alignment constraints.

The implementation may define implicit storage reclamation mechanisms for the objects that are allocated in an allocation context. Any such mechanism shall ensure that it is not possible to access the object at the point of reclamation of that object's storage.

The implementation may exclude support for the `AllocationContext.release()` method if the application domain requires static assurance that it is not possible to access an object at the point of reclamation of that object's storage.

Notes:

The `CoreRegistry.registerStackable()` method is used to identify stackable objects explicitly, as defined in the Core specification.

The default task stack size is application-dependent, rather than implementation-dependent as in the Core specification. Hence it is supplied via a call to an action routine, rather than as a configuration constant.

Allocation contexts exist for the lifetime of the program. An allocation context may be associated either with a task or with a special allocation context at the point of their construction, as defined in the Core specification. The `AllocationContext.release()` method can be used for explicit programmatic storage reclamation of all objects within an allocation context. No implicit storage reclamation mechanisms are defined or required by this specification for the objects that are allocated in allocation contexts.

3.1.5 Modes of Operation

A HIP program shall be executable in one of two operation modes:

1. Execution in Deployment mode implies that the Core classes and methods that are characterized as having the Development mode characteristic are not supported.
2. Execution in Development mode implies that the Core classes and methods that are characterized as having the Development mode characteristic are supported.

The following Core methods have the Development mode characteristic:

`CoreObject.getClass()`
`CoreObject.toString()`
`CoreThrowable.getMessage()`
`CoreClass.toString()`
`CoreRegistry.profiles()`

`AllocationContext.available()`
`AllocationContext.allocated()`

`Time.toString()`

`Unsigned.toString()`
`Unsigned.toHexString()`

The following method in the `HIPTask` class has the Development mode characteristic:

`HIPTask.stackDepth()`

Implementation-Defined Behavior:

As part of execution in Development mode, the implementation should provide a means by which state information relating to thrown exceptions, including the predefined instances of exception classes, can be obtained by the HIP application.

3.2 Restrictions Applying to the Core Specification

3.2.1 Static Execution Environment

The Baseline API that is defined in the Core specification shall not be supported by a HIP implementation.

The following static and instance methods that are defined as relating to the Dynamic Core Execution Environment in the Core specification shall not be supported by a HIP implementation:

CoreArray.length()
CoreArray.atGet()
CoreArray.atPut()

CoreClass.loadClass()
CoreClass.unloadClass()

CoreRegistry.lookup()
CoreRegistry.registerBaseline()
CoreRegistry.publish()
CoreRegistry.unpublish()

CoreString._charAt()
CoreString._hashCode()
CoreString._length()
DynamicCoreString._length()

CoreTask._start()
CoreTask.maxBaselinePriority()
CoreTask.minBaselinePriority()

SignalingSemaphore._P()
SignalingSemaphore._V()
SignalingSemaphore._Vall()
SignalingSemaphore._numWaiters()

The following static and instance methods that are defined as relating to dynamic classes in the Core specification shall not be supported by a HIP implementation:

CoreClass.forName()
CoreClass.newInstance()

Notes:

Communication with a Baseline Java Virtual Machine is not defined by the HIP specification, and so all related APIs, classes and methods are restricted.

3.2.2 Stand-Alone Execution Environment

The following constants and static methods that are defined as relating to an underlying operating system in the Core specification shall not be supported by a HIP implementation:

Configuration.min_core_priority
Configuration.system_priority_map

CoreTask.maxCorePriority()
CoreTask.maxSystemPriority()
CoreTask.minCorePriority()
CoreTask.minSystemPriority()

CoreTask.systemPriorityMap()
CoreTask.SystemPriority()

Notes:

Existence of an underlying operating system is not required by the HIP specification. In the interests of determinism and portability, a HIP-conformant program should not make use of any knowledge that defines how the HIP task priorities (other than interrupt priorities) are implemented by the underlying implementation.

3.2.3 HIP Tasks

The class `CoreTask` that is defined in the Core specification shall not be supported by a HIP implementation. All tasks in a HIP implementation shall be HIP tasks. A HIP task is defined to be either of the class `HIPTask`, or of any of three class extensions of `HIPTask`: `HIPPeriodicTask`, `HIPSporadicTask` and `HIP_ISR_Task`.

The class `HIPTask` extends `CoreObject` as defined in the Core specification, and implements the interface `HIPTaskActionRoutines`.

The class `HIPTask` includes the following methods that have equivalent signature and semantics as the corresponding methods in class `CoreTask` in the Core specification:

`HIPTask.currentTask()`
`HIPTask.defaultStackSize()`
`HIPTask.sleepUntil()`
`HIPTask.stackSize()`
`HIPTask.work()`
`HIPTask.yield()`

The class `HIPSporadicTask` includes the following methods that have equivalent signature and semantics as the corresponding methods in class `SporadicTask` in the Core specification:

`HIPSporadicTask.clearPending()`
`HIPSporadicTask.pendingCount()`
`HIPSporadicTask.trigger()`
`HIPSporadicTask.work()`

The class `HIP_ISR_Task` implements the `Atomic` interface, as defined in the Core specification. The class includes the following methods that have equivalent signature and semantics as the corresponding methods in class `ISR_Task` in the Core specification:

`HIP_ISR_Task.arm()`
`HIP_ISR_Task.ceilingPriority()`
`HIP_ISR_Task.disarm()`
`HIP_ISR_Task.serviced()`
`HIP_ISR_Task.trigger()`
`HIP_ISR_Task.work()`

The class `HIPPeriodicTask` defines the following instance methods:

`HIPPeriodicTask.pendingCount()`: The `pendingCount()` method returns the difference between the number of times this periodic task has been made schedulable due to expiry of its periodic cycle time since the task was started, and the number of times this task has completed its `work()` method. The signature is:

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

`HIPPeriodicTask.clearPending()`: The `clearPending()` method clears all pending invocations of the periodic task's `work()` method. Any incomplete execution of the task's `work()` method is unaffected. The signature is:

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

`HIPPeriodicTask.work()`: The HIP execution environment shall invoke the periodic task's `work()` method when the task is selected to execute by the task dispatching policy, after having been made schedulable due to expiry of its periodic cycle time. The signature is:

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

Notes:

The class `HIPTask` is defined instead of `CoreTask` in the Core specification in order to amend the signatures of the constructors and to add the sub-class `HIPPeriodicTask`.

The class `HIPTask` also includes `start()` and `stop()` methods, but these have different signature and semantics to their equivalents in `CoreTask` in the Core specification, and so are defined separately.

Although there is no support for the `ATCEvent` class that is defined in the Core specification, overrun of Periodic and Sporadic tasks can be programmatically detected by interrogation of the task's `pendingCount()` method. If `pendingCount()` returns greater than one during execution of the task's `work()` method, then overrun has occurred.

3.2.4 Task Creation

All HIP tasks shall be created prior to the call to the main method by the Environment task. A HIP task is created via the execution of a constructor for the class `HIPTask` or one of its sub-classes. If an attempt is made to construct a HIP task object after the Environment task has called the main method, the constructor shall throw the predefined instance of `HIPSubsetViolationException`.

The signatures of the constructors are as follows:

```
public HIPTask(HIPStack stack, AllocationContext context, int priority)
    throws CoreBadPriorityException, CoreOutOfMemoryException,
           HIPSubsetViolationException;
```

```
public HIPPeriodicTask(HIPStack stack, AllocationContext context, int priority, long period)
    throws CoreBadPriorityException, CoreOutOfMemoryException,
           CoreBadArgumentException, HIPSubsetViolationException;
```

```
public HIPSporadicTask(HIPStack stack, AllocationContext context, int priority)
    throws CoreBadPriorityException, CoreOutOfMemoryException,
           HIPSubsetViolationException;
```

```
public HIP_ISR_Task(HIPStack stack, int priority, int interrupt_no)
    throws CoreBadPriorityException, CoreEmbeddedConflictException,
           CoreOutOfMemoryException, HIPSubsetViolationException;
```

The argument `stack` refers to the stack object that is to be used as the dynamic method call stack for this task's execution.

The argument `context` refers to the allocation context object that is the initial default allocation context for execution of the task's `work()` method. If `context` is null, the task has an initial default allocation context of zero-size.

The argument **priority** specifies the base priority for the HIP task. The constructor shall throw the predefined instance of **CoreBadPriorityException** if the value of **priority** is not in the range defined for HIP task priorities.

The argument **period** specifies the periodicity for the HIP periodic task in nanoseconds. The constructor shall throw the predefined instance of **CoreBadArgumentException** if the value of **period** is less than or equal to zero.

The argument **interrupt_no** specifies the identity of the interrupt that is to trigger execution of the HIP ISR task's **work()** method. The constructor shall throw the predefined instance of **CoreEmbeddedConflictException** if the HIP ISR task cannot be bound to the specified interrupt.

Implementation-Defined Behavior:

The implementation shall define the minimum memory requirements for each class of HIP task. If the **stack** argument does not designate a **HIPStack** object that satisfies the minimum memory requirements for the HIP task being constructed, the constructor shall throw the predefined instance of **CoreOutOfMemoryException**.

If the **stack** argument is null, the behavior of the constructor is implementation-defined.

The implementation may define one or more action routines to be called as part of task construction, for example to acquire implementation-specific per-task storage or other resources deterministically.

Notes:

If the **context** argument is null, any attempt to construct a non-stackable array or object other than within the execution scope of a special allocation context (that establishes a new default allocation context) throws the predefined instance of **CoreOutOfMemoryException**.

A HIP ISR task does not have an allocation context since its (atomic synchronized) code is not permitted to allocate objects in such a context. (Only stackable objects can be allocated by this code).

3.2.5 Task Activation

A HIP task is started by an explicit call to the **start()** instance method in the **HIPTask** class. The call to **start()** may occur either before or after the call to the main method. A call to **start()** for a task that has already been started shall throw the predefined instance of **HIPSubsetViolationException** (even if the task is terminated).

The signature of **start()** is as follows:

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

Notes:

Since the implementation defers task dispatching until after the class initialization code and the **onStartup()** action routine are complete, any call to **start()** that is executed prior to the call to the main method does not cause that task to execute immediately. Instead, the task will execute when it is schedulable and when selected to run after the implementation enables task dispatching.

3.2.6 Task Dispatching

The preemptive, priority-ordered task dispatching policy `FIFO_Within_Priorities`, as defined in the Core specification, shall be supported by the implementation and shall be the default task dispatching policy. The minimum priority range that is supported by the implementation is as in the Core specification. The number of interrupt priority values is implementation-defined and is the value of the configuration constant `Configuration.num_interrupt_priorities`.

A task of class `HIPTask` is made schedulable when it is started, and ceases to be schedulable when it is terminated, as defined for `CoreTask` in the Core specification.

A task of class `HIPeriodicTask` is made schedulable when it is started, and ceases to be schedulable when it has completed its `work()` method. It is made schedulable again at its next periodic release time, as defined by its `period` argument, for re-execution of its `work()` method.

A task of class `HISporadicTask` is made schedulable when it is started and its `trigger()` method has been called, and ceases to be schedulable when it has completed its `work()` method. It is made schedulable again at the next call to its `trigger()` method for re-execution of its `work()` method, as defined for `SporadicTask` in the Core specification.

A task of class `HIP_ISR_Task` is made schedulable when it is started, and its `arm()` method has been called, and its triggering event occurs. The triggering events are: the delivery of the interrupt that it is bound to; a call to the task's `trigger()` method. The task ceases to be schedulable when it has completed its `work()` method. It is made schedulable again when its next triggering event occurs for re-execution of its `work()` method, as defined for `ISR_Task` in the Core specification.

The implementation shall not support time-slicing of equal priority HIP tasks as part of the `FIFO_Within_Priorities` dispatching policy. The following constant and static method that are defined as relating to time-slicing in the Core specification shall not be supported by a HIP implementation:

`Configuration.ticks_per_slice`
`CoreTask.ticksPerSlice()`

Implementation-Defined Behavior:

An implementation may define alternative task dispatching policies. The means by which such a policy is selected is implementation-defined.

3.2.7 Task Termination

A HIP task terminates itself by calling its own `stop()` instance method in the `HIPTask` class. The effect of task termination for a HIP task is to execute all `finally` clauses that are associated with active `try` clauses, and then to be no longer schedulable. A call to `stop()` for any other task other than `this` task shall throw the predefined instance of `HIPSubsetViolationException`.

A call to `stop()` shall not occur within any method other than the `work()` method. A call to `stop()` shall not occur within a `finally` clause or within any synchronized code.

A HIP task that is not in the `HIPeriodicTask`, `HISporadicTask` or the `HIP_ISR_Task` class (and thus is a `HIPTask`) terminates as a result of completion of its `work()` method, as defined for `CoreTask` in the Core specification. In this case, the terminating task calls its action routine `terminatedTask()` after the `work()` method is complete, and the task becomes terminated (and no longer schedulable) when this action routine completes.

The signature of `stop()` is as follows:

```
public final void stop() throws HIPSubsetViolationException;
```

Implementation-Defined Behavior:

An implementation may define additional action routines to be called as part of task termination, for example to release implementation-specific per-task storage or other resources deterministically.

Notes:

Since all task creation occurs during program startup, task termination that occurs implicitly (other than via the `stop()` method) might be an abnormal event for the application. Hence an action routine is called so that the application can take recovery action if necessary.

The detection of a call to `stop()` within a `finally` clause or within any synchronized code is a static property.

A terminated task cannot be made schedulable again.

3.2.8 Synchronized Code

The implementation shall support the **PCP** interface as defined in the Core specification.

The class `CoreObject` shall implement the **PCP** interface. The implementation shall implement `CoreObject.ceilingPriority()` to return the largest non interrupt priority value (i.e. `127 - Configuration.num_interrupt_priorities`). A HIP class may explicitly implement the **PCP** interface and provide an overriding `ceilingPriority()` method that returns the required ceiling priority for the class.

The implementation shall support the **Atomic** interface as defined in the Core specification.

The following instance methods that are defined as relating to synchronized code in the Core specification shall not be supported by a HIP implementation:

```
CoreObject.notify()  
CoreObject.notifyAll()  
CoreObject.wait()
```

Implementation-Defined Behavior:

An implementation may define alternative semantics to those in the Core specification relating to when the predefined instance of `CoreIllegalMonitorStateException` is thrown as a result of priority ceiling violation, when a task with priority higher than the ceiling value attempts to synchronize on the **PCP** object.

Notes:

All HIP objects are synchronized using the modified priority ceiling protocol, as defined in the Core specification, as the locking mechanism for mutual exclusion. If the object does not contain synchronized code, then this synchronization has no effect.

Since all synchronized code is locked using priority ceiling protocol, execution of blocking operations such as `wait()` in synchronized code is restricted so as to allow an efficient and resource-free implementation of mutual exclusion on a mono-processor.

A HIP ISR task implements the **Atomic** interface as for Core ISR tasks. This interface provides the interrupt priority value for execution of the ISR task's `work()` method.

3.2.9 Synchronization Primitives

The following classes that are defined as relating to synchronization primitives in the Core specification shall not be supported by a HIP implementation:

CountingSemaphore class
Mutex class

The following instance method that is defined as relating to task synchronization in the Core specification shall not be supported by a HIP implementation:

CoreTask.join()

Notes:

Signaling semaphores are supported as in the Core specification.

Counting semaphores are restricted due to the lack of temporal determinism that may occur whilst waiting to obtain a semaphore (in the form of unbounded priority inversion).

Mutexes are restricted due to the lack of functional determinism that is inherent in explicit lock and unlock operations in the form of ensuring that usage is always balanced and the lock is always released.

Mutexes are also restricted since the Core specification defines priority inheritance as the means to solve unbounded priority inversion when contending for the lock. The combination of support for priority inheritance and priority ceiling within the HIP implementation significantly increases complexity, and hence also increases footprint, decreases performance, and makes formal verification much more difficult to achieve.

The join() method is not required since all tasks are created during program startup and are assumed not to terminate.

3.2.10 Asynchronous Operations

The following classes and methods that are defined as relating to asynchronous events and asynchronous task operations in the Core specification shall not be supported by a HIP implementation:

ATCEvent class
ATCEventHandler class

CoreTask.abort()
CoreTask.abortWorkException()
CoreTask.asyncHandler()
CoreTask.resume()
CoreTask.setPriority()
CoreTask.signalAsync()
CoreTask.suspend()

ScopedException class

Notes:

These operations undermine deterministic behavior.

3.2.11 Timing Operations

The following static and instance methods that are defined as relating to time and task timing operations in the Core specification shall not be supported by a HIP implementation:

CoreTask.sleep()
Time.tickDuration()
Time.uptimePrecision()

Notes:

Relative sleep operations are non-deterministic, and so only the absolute sleep method `HIPTask.sleepUntil()` is supported as defined in the Core specification.

The methods `tickDuration()` and `uptimePrecision()` in class `Time` as defined in the Core specification are not supported since the corresponding constants in class `Configuration` provide the same information in a reliable manner.

3.2.12 Throwable Classes

The following sub-classes of `CoreThrowable` that are defined in the Core specification shall not be supported by a HIP implementation:

CoreClassFormatError
CoreSecurityException
CoreOperationNotPermittedException
CoreATCEventsIgnoredException
CoreClassInUseException
CoreClassNotFoundException

Notes:

The unsupported exception classes relate to features in the Core specification that are not supported by the HIP specification, or that are required to be detected prior to HIP program execution.

3.3 Action Routines

Action routines are public abstract instance methods that belong either to the interface `HIPProgramActionRoutines` or to the interface `HIPTaskActionRoutines`.

The interface `HIPProgramActionRoutines` is implemented by the class that defines the main method (see section 3.1.3). The methods defined for the interface `HIPProgramActionRoutines` have the following signatures:

`abstract void onStartUp();`
Called as part of HIP program startup (see section 3.1.3).

`abstract void onShutdown();`
Called as part of HIP program shutdown (see section 3.1.3).

`abstract long defaultStackSize();`
Called as part of HIP task creation (see section 3.2.4).

The interface `HIPTaskActionRoutines` is implemented by the class `HIPTask` (see section 3.2.3). The methods defined for the interface `HIPTaskActionRoutines` have the following signatures:

`abstract void terminatedTask();`
Called as part of HIP task termination (see section 3.2.7).

The methods are called implicitly by the implementation using the current task's execution resources (i.e. in the context of the current thread of control) in response to certain runtime states.

APPENDIX A Exclusions from Core APIs and Classes

The following APIs, classes, methods and constants that are defined in the Core specification are not supported by the HIP specification:

Dynamic Core Execution Environment and Baseline API

ATCEvent class
ATCEventHandler class
CoreATCEventsIgnoredException class
CoreClassFormatError class
CoreClassInUseException class
CoreClassNotFoundException class
CoreOperationNotPermittedException class
CoreSecurityException class
CoreTask class
CountingSemaphore class
Mutex class
ScopedException class

AllocationContext(); constructor

Configuration.ticks_per_slice
Configuration.min_core_priority
Configuration.system_priority_map
Configuration.stack_overflow_checking
Configuration.default_stack_size()

CoreArray.length()
CoreArray.atGet()
CoreArray.atPut()

CoreClass.forName()
CoreClass.newInstance()
CoreClass.loadClass()
CoreClass.unloadClass()
CoreClass.verification()

CoreObject.notify()
CoreObject.notifyAll()
CoreObject.wait()

CoreRegistry.lookup()
CoreRegistry.registerBaseline()
CoreRegistry.publish()
CoreRegistry.unpublish()
CoreRegistry.stackAllocation()

CoreString._charAt()
CoreString._hashCode()
CoreString._length()
DynamicCoreString._length()

SignalingSemaphore._P()
SignalingSemaphore._V()
SignalingSemaphore._Vall()
SignalingSemaphore._numWaiters()

Time.tickDuration()
Time.uptimePrecision()

APPENDIX B Extensions to Core APIs and Classes

The following interfaces, classes and constants are defined in the HIP specification that are extensions to the Core specification:

HIPProgramActionRoutines interface

HIPTaskActionRoutines interface

HIPSubsetViolationException class

HIPStack class

HIPTask class

HIPPeriodicTask, HIPSporadicTask and HIP_ISR_Task classes

Configuration.num_interrupt_priorities