IBM WebSphere Real Time for RT Linux
Version 3

User Guide

IBM WebSphere Real Time for RT Linux
Version 3

User Guide

Note

Before using this information and the product it supports, read the information in “Notices” on page 151.

Fifth edition (February 2014)

This edition of the user guide applies to IBM WebSphere Real Time for RT Linux, Version 3, and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

Figures

Tables

Preface

This user guide provides general information about IBM® WebSphere® Real Time for RT Linux.

Chapter 1. Introduction

This information tells you about IBM WebSphere Real Time for RT Linux.

Any new modifications made to this user guide are indicated by vertical bars to the left of the changes.

Late breaking information about the IBM WebSphere Real Time for RT Linux that is not available in the user guide can be found here: http://www.ibm.com/support/docview.wss?uid=swg21501145

- "Overview of WebSphere Real Time for RT Linux"
- "What's new" on page 2
- "Benefits" on page 3

Overview of WebSphere Real Time for RT Linux

WebSphere Real Time for RT Linux bundles real-time capabilities with the IBM J9 virtual machine (JVM).

WebSphere Real Time for RT Linux is a Java™ Runtime Environment with a Software Development Kit that extends the IBM SDK for Java with real-time capabilities. Applications that are dependent on precise response times can take advantage of the real-time features provided with WebSphere Real Time for RT Linux on standard Java technology.

Features

Real-time applications need consistent run time rather than absolute speed.

When the JVM is run in real-time mode, additional memory areas are available in addition to the garbage collected heap. Programs might request or specify any number of reusable scoped and nonreusable immortal memory areas, which are not garbage collected. This capability provides the application with more control over memory usage. It also uses the Metronome Garbage Collector to achieve time-based collections. When the JVM is run in a traditional throughput mode, various work-based garbage collectors can be used that optimize throughput but can have larger individual delays than the Metronome Garbage Collector.

The main concerns when deploying real-time applications with traditional JVMs are as follows:
- Unpredictable (potentially long) delays from Garbage Collection (GC) activity.
- Delays to method run time as Just-In-Time (JIT) compilation and recompilation occurs, with variability in execution time.
- Arbitrary operating system scheduling.

WebSphere Real Time for RT Linux removes these obstacles by providing:
- The Metronome Garbage Collector, an incremental, deterministic garbage collector with very short pause times.
- Ahead-Of-Time (AOT) compilation.
- Priority-based FIFO scheduling.

In addition, WebSphere Real Time provides the real-time programmer with the RTSJ facilities; see "Support for RTSJ" on page 9.

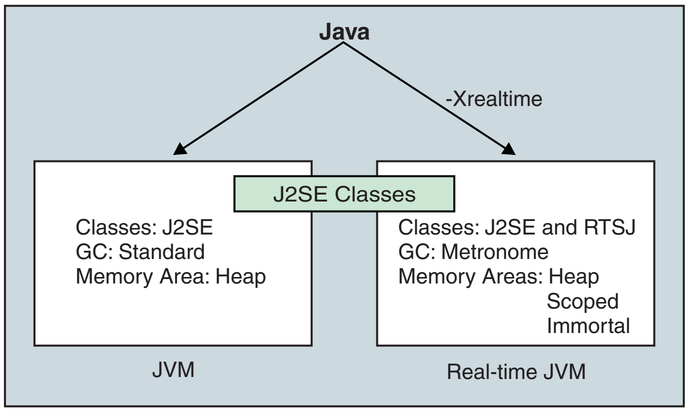


Figure 1. Overview of WebSphere Real Time for RT Linux

You enable real-time capabilities by using the -Xrealtime option when running the JVM or any of the tools provided. By default, the JVM and the tools provided run without real-time capabilities enabled. Figure 1 shows the relationships of the two JVMs that are supplied with WebSphere Real Time for RT Linux.

The following Java commands recognize the -Xrealtime option:

Table 1. Java commands used in real-time mode

Command	Function
java	Runs in standard mode by default but also runs in real-time mode when the -Xrealtime option is specified. In real-time mode, the programmer accesses classes from the javax.realtime package. You can use precompiled jar files and the Metronome deterministic garbage collection technology.
javac, javah, javap	Runs in standard mode by default; but, when the -Xrealtime option is specified, it includes the javax.realtime.* classes in the class path.
admincache	Can be run both with and without -Xrealtime, but populating a shared cache with the **admincache** tool is only possible in the real-time mode. In regular mode, only the cache utilities are available (such as listAllCaches, or printStats). Like **jdmpview**, **admincache** must be run with -Xrealtime to access caches for the real-time JVM, and must be run without -Xrealtime to access caches for the regular JVM.
jextract	jextract runs in standard mode by default, but must be run with the -Xrealtime option when processing system dumps generated by the JVM in real-time mode

What's new

This topic introduces changes for IBM WebSphere Real Time for RT Linux .

WebSphere Real Time for RT Linux V3

WebSphere Real Time for RT Linux V3 is an extension to the IBM SDK for Java V7, building on the features and functions available with this release to include real-time capabilities. Earlier versions of WebSphere Real Time for RT Linux were based on earlier releases of the IBM SDK for Java.

To learn more about what's new in IBM SDK for Java V7, see: What's new in the IBM SDK for Java 7 information center.

Any new modifications made to this user guide are indicated by vertical bars to the left of the changes.

Real Time Linux operating system support

Support is now included for Red Hat Enterprise MRG 2.2 for Red Hat Enterprise Linux 6 Update 3. For more information about supported hardware and software, including any required updates or patches, see “Hardware and software prerequisites” on page 23.

Use of large pages

In service refresh 4, the IBM SDK for Java V7 introduces the use of large pages by default for Linux x86 and AMD64/EM64T platforms. Although WebSphere Real Time for RT Linux is an extension to the IBM SDK for Java V7, large pages are not enabled by default because of a known problem.

Symbol resolution

By default, the JVM immediately resolves symbol resolution for each function in a user native library. Use the **`-XX:+LazySymbolResolution`** option to force the JVM to delay symbol resolution for all functions in a user native library, until the function is called. For more information, see “Non-standard options” on page 145.

Real Time Linux operating system support

Support is now included for SUSE Linux Enterprise Real Time version 11. For more information about supported hardware and software, including any required updates or patches, see “Hardware and software prerequisites” on page 23.

jxeinajar

WebSphere Real Time for RT Linux V3 no longer supports the use of jxeinajar. For reference purposes, earlier information regarding jxeinajar and in particular how to migrate to admincache can be found in the WebSphere Real Time for RT Linux V2 documentation.

Benefits

The benefits of the real-time environment are that Java applications run with a greater degree of predictability than with the standard JVM and provide consistent timing behavior for your Java application. Background activities, such as compilation and garbage collection, occur at given times and thus remove any unexpected peaks of background activity when running your application.

You obtain these advantages by extending the JVM with the following functions:
- Metronome real-time garbage collection technology
- Ahead-of-time (AOT) compilation
- Support for the Real-Time Specification for Java (RTSJ)

All Java applications can be run in a real-time environment without modification, benefiting from the Metronome Garbage Collector and its deterministic garbage collection that occurs at regular intervals. To achieve the maximum benefit from WebSphere Real Time for RT Linux, you can write applications specifically for the real-time environment using both real-time threads and no-heap real-time threads. The approach that you take depends on the timing specification of your application.

Many real-time Java applications can exploit the low pause times of the Metronome Garbage Collector and AOT to achieve their goals, retaining the benefits of Java portability. Applications with tighter requirements must use the RTSJ facilities of real-time threads and no-heap real-time threads, with scoped and immortal memory. This approach limits your application to run in a real-time environment only, losing the advantage of portability to JSE Java. You also have to develop a more complex programming model.

Accessibility

Accessibility features help users who have a disability, such as restricted mobility or limited vision, to use information technology products successfully.

IBM strives to provide products with usable access for everyone, regardless of age or ability.

For example, you can operate WebSphere Real Time for RT Linux without a mouse, by using only the keyboard.

To read about issues that affect accessibility of the underlying IBM SDK for Java V7, see IBM Information Center. There are no accessibility issues affecting unique features and capabilities in WebSphere Real Time for RT Linux.

Keyboard navigation

This product uses standard Microsoft Windows navigation keys.

For users who require keyboard navigation, a description of useful keystrokes for Swing applications can be found here: Swing Key Bindings.

IBM and accessibility

See the IBM Human Ability and Accessibility Center for more information about the commitment that IBM has to accessibility.

Chapter 2. Understanding IBM WebSphere Real Time for RT Linux

This section introduces key components of IBM WebSphere Real Time for RT Linux.

Introduction to the Metronome Garbage Collector

The Metronome Garbage Collector replaces the standard Garbage Collector in WebSphere Real Time for RT Linux.

The key difference between Metronome garbage collection and standard garbage collection is that Metronome garbage collection occurs in small interruptible steps but standard garbage collection stops the application while it marks and collects garbage.

For example:

```
java -Xrealtime -Xgc:targetUtilization=80 yourApplication
```

The example specifies that your application runs for 80% in every 60ms. The remaining 20% of the time might be used for garbage collection, if there is garbage to be collected. The Metronome Garbage Collector guarantees utilization levels provided that it has been given sufficient resources. Garbage collection begins when the amount of free space in the heap falls below a dynamically determined threshold.

Garbage collection and priorities

The garbage collection thread has to run at a priority higher than the highest priority thread that generates garbage in the heap; otherwise, it might not run as specified by the configured utilization. Both regular Java threads and real-time threads can generate garbage and, therefore, garbage collection must run at a priority higher than all regular and real-time threads. This prioritization is handled automatically by the JVM and garbage collection runs at 0.5 priority above the highest priority of all regular and real-time threads. However, it is important to ensure that no-heap real-time threads (NHRTs) are not affected by garbage collection. Run all NHRTs at a higher priority than the highest priority real-time threads. This means that NHRTs run at a priority higher than garbage collection and are not delayed.

Table 2 on page 6 shows a typical example of priorities that you can define and the related garbage collection priorities that follow from your choice

For the comparison of Java priorities and OS priorities, see "Priority mapping and inheritance" on page 11.

Table 2. Example of garbage collection and priorities

Threads	Priorities (examples)
If the highest priority real-time thread is:	20 (OS priority 43)
Then the Garbage Collector is:	20.5 (OS priority 44)
To ensure that a NHRT runs independently of the garbage collector, set a higher priority than the GC:	21 (OS priority 45) or higher.
The Metronome alarm thread is:	Priority 46 (OS priority 89)

Note: Even with this configuration, no-heap real-time threads are not completely unaffected by garbage collection because the metronome alarm thread runs at the highest priority in the system to ensure that it can wake up regularly and work out if garbage collection needs to do anything. The work to do that is, of course, tiny and thus not a major consideration.

Metronome garbage collection and class unloading

The Metronome Garbage Collector does not unload classes in IBM WebSphere Real Time because it can require a non-deterministic amount of work causing pause time outliers.

Metronome Garbage Collector threads

The Metronome Garbage Collector consists of two types of threads: a single alarm thread, and a number of collection (GC) threads. By default, there is one GC thread. You can set the number of GC threads for the JVM using the **`-Xgcthreads`** option.

You cannot change the number of alarm threads for the JVM.

The Metronome Garbage Collector periodically checks the JVM to see if the heap memory has sufficient free space. When the amount of free space falls below the limit, the Metronome Garbage Collector triggers the JVM to start garbage collection.

Alarm thread

The single alarm thread guarantees to use minimal resources. It "wakes" at regular intervals and makes these checks:

- The amount of free space in the heap memory
- Whether garbage collection is currently taking place

If insufficient free space is available and no garbage collection is taking place, the alarm thread triggers the collection threads to start garbage collection. The alarm thread does nothing until the next scheduled time for it to check the JVM.

Collection threads

Each collection thread checks Java and real-time threads for heap objects. They check the memory areas in the following sequence:

1. Scoped memory to identify and mark any live objects in the heap that are being used by objects from scoped memory.
2. Immortal memory to identify and mark any live objects in the heap that are being used by objects from immortal memory.
3. Heap memory to identify and mark live objects.

When the live objects have been marked, the unmarked objects are available for collection.

After the garbage collection cycle has completed, the Metronome Garbage Collector checks the amount of free heap space. If there is still insufficient free heap space, another garbage collection cycle is started using the same trigger ID. If there is sufficient free heap space, the trigger ends and the garbage collection threads are stopped. The alarm thread continues to monitor the free heap space and will trigger another garbage collection cycle when it is required.

For more information about using the Metronome Garbage Collector, see "Using the Metronome Garbage Collector" on page 68.

Compilers

IBM WebSphere Real Time for RT Linux supports several models of code compilation that provide varying levels of code performance and determinism.

The options available for compiling Java code with IBM WebSphere Real Time for RT Linux include:

Low priority Just-In-Time (JIT) compilation
: The default compilation model in WebSphere Real Time for RT Linux uses a Just-In-Time compiler to compile the important methods of a Java application while the application runs. In this mode, the JIT compiler works in a similar way to the operation of the JIT compiler in a non-real-time JVM. The difference is that the WebSphere Real Time for RT Linux JIT compiler runs at a lower priority level than any real-time threads. The lower priority means that the JIT compiler uses system resources when the application does not need to perform real-time tasks. The effect is that the JIT compiler does not significantly affect the performance of real-time tasks.

Ahead-Of-Time (AOT) precompiled code
: WebSphere Real Time for RT Linux compiles Java methods to native code in a precompilation step before running the application. Using AOT precompiled code provides the highest level of determinism, with good performance.

Mixed mode, combining AOT precompiled code and low priority JIT compilation
: AOT and JIT compiled code can be used together while the application runs. This mode of operation can provide very good determinism with good performance, and very high performance for methods that run frequently.

Interpreted operation
: The interpreter runs a Java application, but does not use code compilation at all.

For more information about using compiled code, see "Using compiled code with WebSphere Real Time for RT Linux" on page 40.

Comparing JIT and AOT compilation

Ahead-of-Time (AOT) compilation allows you to compile Java classes and methods before you run your code. AOT compilation avoids the unpredictable timing effect the JIT compiler can have on sensitive performance paths. To ensure your code is compiled before it executes and for the highest level of deterministic performance, you can precompile your code into a shared class cache using the AOT compiler.

Note: AOT-compiled code does not typically run as quickly as JIT compiled code.

The Just-In-Time (JIT) compiler runs as a high-priority SCHED_OTHER thread, running above the priority of standard Java threads, but running below the priority of real-time threads. Just-in-time compilation, therefore, does not cause nondeterministic delays in real-time code. As a result, important real-time work is performed on-time because it won't be preempted by the JIT compiler. Real-time code might run as interpreted code, however, because the JIT has not had enough time to compile the hot methods that have queued up. A comparison is shown, Figure 2.

In general, if your application has a warm-up phase, it is more efficient to run with the JIT and, if necessary, disable the JIT when the warm-up phase is complete. This approach allows the JIT compiler to generate code for the environment in which your application runs.

If the application has no warm-up phase and it is not clear if key paths of execution are compiled through standard application operation, AOT compilation works well in this environment.

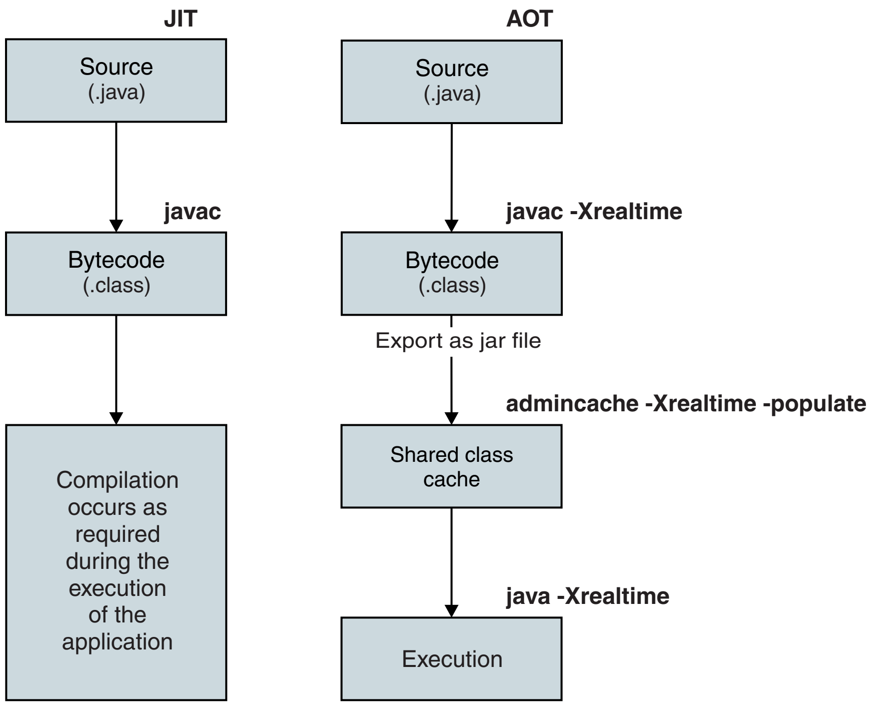


Figure 2. Comparing JIT compiler and AOT compiler.

Thread scheduling

Linux scheduling policies can be used with regular Java threads as well as real-time Java threads to tune real-time applications.

With WebSphere Real Time for RT Linux, you can run regular Java threads with the SCHED_RR or SCHED_FIFO scheduling policy. Using the SCHED_RR or

SCHED_FIFO policy gives you finer control over your application, which can improve the real-time performance of Java threads. The JVM detects the priority and policy of the main thread when Java is started with the SCHED_RR or SCHED_FIFO policy. The JVM alters the priority and policy mappings accordingly. For more information about altering regular Java thread priorities and policies, see "Thread scheduling and dispatching" on page 37

Thread scheduling and dispatching of real-time Java threads is part of the Real Time Specification for Java (RTSJ). This topic, including the scheduling policies and priority handling of real-time Java threads can be found in "Support for RTSJ."

The Linux scheduling policies include:

SCHED_OTHER
: The default universal time-sharing scheduling policy that is used by most threads. These threads must be assigned with a priority of zero.

  SCHED_OTHER uses time slicing, which means that each thread runs for a limited time period, after which the next thread is allowed to run.

SCHED_FIFO
: Can be used only with priorities greater than zero. When a SCHED_FIFO thread becomes available, the thread has priority over any normal SCHED_OTHER thread.

  If a SCHED_FIFO thread that has a higher priority becomes available, this thread has priority over existing SCHED_FIFO threads with a lower priority. This thread is then kept at the top of the queue for its priority.

  There is no time slicing.

SCHED_RR
: Is an enhancement of SCHED_FIFO. The difference is that each thread is allowed to run only for a limited time period. If the thread exceeds that time, it is returned to the list for its priority.

For more details on these Linux scheduling policies, see the **`man`** page for **`sched_setscheduler`**.

For more information about using Linux scheduling policies with WebSphere Real Time for RT Linux, see "Thread scheduling and dispatching" on page 37.

Support for RTSJ

WebSphere Real Time for RT Linux implements the Real-Time Specification for Java (RTSJ).

WebSphere Real Time for RT Linux version 3.0 has been certified as RTSJ 1.0.2 compliant against the RTSJ Technology Compatibility Kit version 3.1.0 FCS, and is compliant with the Java Compatibility Kit (JCK) for version 7.0.

Real-time thread scheduling and dispatching

Thread scheduling and dispatching of real-time Java threads is part of the Real Time Specification for Java. The scheduling policy SCHED_FIFO is used to prioritize real-time Java threads using Linux operating system priorities 11 - 89.

Information about Linux scheduling policies can be found in "Thread scheduling and dispatching" on page 37.

Schedulables and their Parameters

There are two main types of real-time schedulable objects: real-time threads and asynchronous event handlers.

These schedulable objects have the following parameters associated with them:

SchedulingParameters
: **PriorityParameters** schedules real-time schedulable objects by priority.

ReleaseParameters

- **PeriodicParameters** describes periodic release of real-time schedulable objects. A periodic real-time thread is one that is released at regular intervals.
- **AperiodicParameters** describes the release of real-time schedulable objects. Aperiodic real-time threads are released at irregular intervals.

MemoryParameters
: Describes memory allocation constraints for real-time schedulable objects.

ProcessingGroupParameters
: Unsupported in WebSphere Real Time for RT Linux.

The priority scheduler

In WebSphere Real Time for RT Linux, the scheduler is a priority scheduler. As its name implies, it manages the running of schedulable objects according to their active priorities.

The scheduler maintains the list of schedulable objects and determines when each object can be released to run in the CPU. The scheduler must abide by the various parameters that are associated with each schedulable object. The methods addToFeasibility, isFeasible, and removeFromFeasibility are provided for this purpose.

Priorities and policies

Regular Java threads, that is, threads allocated as `java.lang.Thread` objects, can use scheduling policies SCHED_OTHER, SCHED_RR or SCHED_FIFO. Real-time threads, that is, threads allocated as `java.lang.RealtimeThread`, and asynchronous event handlers use the SCHED_FIFO scheduling policy.

Regular Java threads use the default scheduling policy of SCHED_OTHER, unless the JVM is started by a thread with policy SCHED_RR or SCHED_FIFO. Regular Java threads that use the policy SCHED_OTHER have the operating system thread priority set to 0. Regular Java threads that use the policy SCHED_RR or SCHED_FIFO inherit the priority of the thread that starts the JVM. For more information about priorities and policies for regular Java threads, see "Regular Java thread priorities and policies" on page 38.

For real-time threads, the SCHED_FIFO policy has no time slicing and supports 99 priorities from 1 (the lowest) to 99 (the highest). This WebSphere Real Time for RT Linux implementation supports 28 user priorities in the range 11-38 inclusive, so:

```
javax.realtime.PriorityScheduler().getMinPriority()
```

returns 11, and:

```
javax.realtime.PriorityScheduler().getMaxPriority()
```

returns 38.

OS priorities 81 - 89 are used by the IBM JVM for dispatching worker threads. These threads are all designed to do a small amount of work before going back to sleep. The threads are as follows:

- The Metronome Garbage Collector alarm thread runs at a priority of 89. This thread runs regularly and dispatches a GC work unit.
- Two Asynchronous Signal Threads, which process asynchronous signals, one being a no-heap real-time (NHRT) thread at priority 88 and the other at priority 87.
- Two Timer Threads, which dispatch timer events, one being a no-heap real-time thread for no-heap timers at priority 85 and the other at priority 83.
- The Async Event Handler Threads, which are dispatched to run asynchronous event handlers, and, while running an async event handler, are assigned the priority of that handler. The system starts up with two no-heap real-time handler threads at priority 85 and 8 others at priority 83.
- The Asynchronous Signal no-heap real-time thread at priority 88 handles requests for heap dumps, core dumps, and javacore dumps. It temporarily boosts its priority to 89 while creating dump files.

The Metronome GC Trace thread runs at OS priority 12, and the JIT Sampler thread, which samples Java methods for compilation, runs at OS priority 13.

The JIT Compilation thread (which is different from the JIT Sampler thread) runs with the SCHED_OTHER policy at OS priority 0.

The JIT compilation and JIT sampler threads are both disabled if **`-Xnojit`** or **`-Xint`** is specified.

The Metronome Garbage Collector and finalizer priority constantly changes (before each round of collection) to be above the highest priority heap-allocating thread. You must ensure that the priority of heap-allocating threads is below that of NoHeapRealtimeThreads.

A heap-allocating thread is any non-NHRT user thread that is not asleep or blocked on a monitor. A user thread running native code outside the JNI interface is not considered to be heap-allocating. If a garbage collection is in progress when a heap-allocating thread wakes up, is no longer blocked on a monitor, or leaves JNI, it is forced to wait until the garbage collection has finished before it can continue.

OS priority 81 is reserved for internal JVM threads that are allocating from the heap. If an internal JVM thread is at OS priority 81, the garbage collector runs at OS priority 82. When the only heap-allocating user threads are not real-time threads, the GC priority runs at OS priority 11. Otherwise, the GC runs at a priority that is one OS priority higher than the highest priority heap-allocating user thread.

The GC priority is adjusted just before a round of collection.

Priority mapping and inheritance

Each Java priority is mapped to an associated operating system base priority, and each operating system priority is associated with a scheduling policy. The WebSphere Real Time for RT Linux Linux operating system scheduling policies are SCHED_OTHER, SCHED_RR and SCHED_FIFO.

Real-time Java threads use policy SCHED_FIFO, while regular Java threads use the policy of the thread that starts the JVM. The default scheduling policy for regular Java threads is SCHED_OTHER, but you can use a utility like **chrt** to set policies SCHED_RR or SCHED_FIFO. For more information about thread priorities and policies, see "Thread scheduling and dispatching" on page 37.

The following table shows how the Java priorities are mapped to native operating system priorities. Some Java priorities are reserved for use by the JVM, and some native priorities that have no corresponding Java priorities are used by the JVM as well.

Note:

- Priorities 1-10 are used by regular Java threads.
  - For policy SCHED_OTHER, Java priorities 1-10 map to operating system priority 0.
  - For policies SCHED_FIFO or SCHED_RR, Java priorities 1-10 inherit the priority of the thread that starts the JVM.
- Priorities 11 upwards are used by real-time threads and no-heap real-time threads
- A schedulable object always runs with its active priority. The active priority is initially the base priority of the schedulable object, but the active priority can be temporarily raised by priority inheritance. The base priority of a schedulable object can be changed while it is running.

User base priorities:

```
Java priorities 1-10: SCHED_OTHER, OS priority 0

Java priority 11: SCHED_FIFO, OS priority 25
Java priority 12: SCHED_FIFO, OS priority 27
Java priority 13: SCHED_FIFO, OS priority 29
Java priority 14: SCHED_FIFO, OS priority 31
Java priority 15: SCHED_FIFO, OS priority 33
Java priority 16: SCHED_FIFO, OS priority 35
Java priority 17: SCHED_FIFO, OS priority 37
Java priority 18: SCHED_FIFO, OS priority 39
Java priority 19: SCHED_FIFO, OS priority 41
Java priority 20: SCHED_FIFO, OS priority 43
Java priority 21: SCHED_FIFO, OS priority 45
Java priority 22: SCHED_FIFO, OS priority 47
Java priority 23: SCHED_FIFO, OS priority 49
Java priority 24: SCHED_FIFO, OS priority 51
Java priority 25: SCHED_FIFO, OS priority 53
Java priority 26: SCHED_FIFO, OS priority 55
Java priority 27: SCHED_FIFO, OS priority 57
Java priority 28: SCHED_FIFO, OS priority 59
Java priority 29: SCHED_FIFO, OS priority 61
Java priority 30: SCHED_FIFO, OS priority 63
Java priority 31: SCHED_FIFO, OS priority 65
Java priority 32: SCHED_FIFO, OS priority 67
Java priority 33: SCHED_FIFO, OS priority 69
Java priority 34: SCHED_FIFO, OS priority 71
Java priority 35: SCHED_FIFO, OS priority 73
Java priority 36: SCHED_FIFO, OS priority 75
Java priority 37: SCHED_FIFO, OS priority 77
Java priority 38: SCHED_FIFO, OS priority 79
```

Internal base priorities:

```
Internal Java priority 39: SCHED_FIFO, OS priority 81
Internal Java priority 40: SCHED_FIFO, OS priority 83
Internal Java priority 41: SCHED_FIFO, OS priority 84
Internal Java priority 42: SCHED_FIFO, OS priority 85
Internal Java priority 43: SCHED_FIFO, OS priority 86
Internal Java priority 44: SCHED_FIFO, OS priority 87
Internal Java priority 45: SCHED_FIFO, OS priority 88
OS priorities 11, 12, 13
OS priorities even numbers 26, 28, 30, ..., 82
OS priority 89
```

See also: the "Synchronization" section in http://www.rtsj.org/specjavadoc/book_index.html.

Priority inheritance:

The active priority of a thread can be temporarily boosted because it holds a lock required by a higher priority thread. These locks might be internal JVM locks or user-level monitors associated with synchronized methods or synchronized blocks. The priority of a regular Java thread, therefore, might temporarily have a real-time priority until the point at which the thread has released the lock.

One consequence of priority inheritance means that the thread policy of a SCHED_OTHER thread is temporarily changed to SCHED_FIFO.

For more information about base and active priorities, see the "Synchronization" section in the RTSJ specification.

Memory management

Garbage collecting memory heaps has always been considered an obstacle to real-time programming because of the unpredictable behavior introduced by garbage collection. The Metronome garbage collector in IBM WebSphere Real Time for RT Linux can provide high deterministic GC performance. At the same time, the Real-Time Specification for Java (RTSJ) provides several extensions to the memory model for objects outside the garbage-collected heap, so that a Java programmer can explicitly manage both short-lived and long-lived objects.

Memory areas

The RTSJ introduces the concept of a memory area that can be used for the allocation of objects. Some memory areas exist outside the heap and place restrictions on what the system and garbage collector can do with objects. For example, objects in some memory areas are never garbage collected but the Garbage Collector can scan these memory areas for references to any object in the heap to preserve the integrity of the heap.

Memory management has three basic types:

- Heap memory is the traditional Java heap but is managed by the Metronome Garbage Collector.
- Scoped memory must be specifically requested by applications and can be used only by real-time threads, including no-heap real-time threads and no-heap asynchronous event handlers.
- Immortal memory represents an area of memory containing objects that can be referenced by any schedulable object, specifically including no-heap real-time threads and no-heap asynchronous event handlers. It is used by class loading and static initialization even if the application does not use it.

Immortal or scoped memory can be designated to use physical memory, which consists of memory regions having specific characteristics such as substantially faster access. In general, physical memory is not used often and is unlikely to affect the standard JVM user.

Heap memory

The maximum size is controlled by **-Xmx** but remember *not* to set the initial heap size (**-Xms**) or set it equal to maximum heap size **-Xmx**, because, in real time, the heap never expands from the initial heap size to the maximum heap size. When you reach maximum heap size with no free space, OutOf MemoryError results. In general, the real time JVM consumes more heap memory than the traditional JVM because supporting deterministic collection requires objects to be organized differently, resulting in higher heap fragmentation. In addition, arrays are broken into fragments, each of which has a header. It depends on the ratio of large to small objects and the amount of array usage, but it is likely to find an application needing 20% more heap space.

The Metronome Garbage Collector is similar to the "mostly concurrent" collector that exists in the mainstream JVM in that it collects garbage while the application is running. In a perfect world, the collection cycle completes before the application runs out of memory, but some applications with very high allocation rates can allocate faster than the Metronome Garbage Collector can collect. Various detailed controls affect the collection rate, but there is one control that forces Metronome to revert to traditional stop-the-world GC before finally throwing OutOf MemoryError. The runtime parameter is **-Xgc:synchronousGCOnOOM** and the counterpart is **-Xgc:nosynchronousGCOnOOM**. The default is **-Xgc:synchronousGCOnOOM**.

Scoped memory

The RTSJ introduces the concept of scoped memory. It can be used by objects that have a well-defined lifetime. A scope can be entered explicitly, or it can be attached to a schedulable object (a real-time thread or an asynchronous event handler) that effectively enters the scope before it runs the run() method of the object. Each scope has a reference count and when this reaches zero the objects that are resident in that scope can be closed (finalized) and the memory associated with that scope is released. Reuse of the scope is blocked until finalization is complete.

Scoped memory can be divided into two types: VTMemory and LTMemory. These types of scoped memory vary by the time required to allocate objects from the area. LTMemory guarantees linear time allocation when memory consumption from the memory area is less then the initial size of the memory area. VTMemory offers no such guarantee.

Scopes can be nested. When a nested scope is entered, all subsequent allocations are taken from the memory associated with the new scope. When the nested scope is completed, the previous scope is restored and subsequent allocations are again taken from that scope.

Because of the lifetimes of scoped objects, it is necessary to limit the references to scoped objects, by means of a restricted set of assignment rules. A reference to a scoped object cannot be assigned to a variable from an outer scope, or to a field of an object in either the heap or the immortal area. A reference to a scoped object can be assigned only into the same scope or into an inner scope. The virtual machine detects incorrect assignment attempts and throws an IllegalAssignmentError exception when they occur. The flexibility provided in choice of scoped memory types allows the application to use a memory area that has characteristics that are appropriate to a particular syntactically defined region of the code.

The size of the area must be specified during construction of the area and a command-line parameter, **`-Xgc:scopedMemoryMaximumSize`**, controls the maximum value. The default is 8 MB and is adequate for most purposes.

Immortal memory

Immortal memory is a memory resource shared among all schedulable objects and threads in an application. Objects allocated in immortal memory are always available to non-heap threads and asynchronous event handlers and are not subject to delays caused by garbage collection. Objects are freed by the system when the program terminates.

The size is controlled by **`-Xgc:immortalMemorySize`**; for example, **`-Xgc:immortalMemorySize=20m`** sets 20 MB, The default is 16 MB, which is usually adequate, unless you are doing a lot of class loading. Class loading is the likely cause of most OutOfMemoryError exceptions.

Estimating memory requirements

How to obtain information required to allocate sufficient memory

A reasonable approach is to identify the memory required to hold the expected objects, with a sensible safety margin. Analysis of the application helps identify the number and nature of objects required, although the actual size required for an object can vary between different systems. Using the SizeEstimator class takes actual object size into account, providing more portable information.

The SizeEstimator class

The SizeEstimator class provides guidance information about the amount of memory needed to store an object. The estimate is an indication of the minimum memory space that should be allocated for the object itself, and does not take into account memory requirements for any other resources that might be required by the object, for example during its construction.

For details about this class, see http://www.rtsj.org/specjavadoc/book_index.html

Using memory

A comparison of Java threads, real-time threads, and no-heap real-time threads.

The Real-Time Specification for Java (RTSJ) adds two classes to support real-time threads: RealtimeThread class and NoHeapRealtimeThread class.

- Both real-time threads and no-heap real-time threads are schedulable objects. As schedulable objects, they have the following parameters: release, scheduling, memory, and processing group.
- Real-time threads can access objects in heap memory as well as in scoped and immortal memory.
- No-heap real-time threads access only scoped and immortal memory areas.
- No-heap real-time threads need a higher priority than other real-time threads. If their priority is less than other real-time threads, they lose their advantage of running without interference from the Garbage Collector.

Note: A no-heap real-time thread with priority higher than other real-time threads will not be interrupted by garbage collection.

Table 3. Memory access by real-time and no-heap real-time threads

Threads	Immortal memory	Scoped memory	Heap memory
Normal threads	✓	✗	✓
Real-time threads	✓	✓	✓
no-heap real-time threads	✓	✓	✗

Types of memory area

Immortal memory

Immortal memory is not subject to garbage collection. After space has been allocated in immortal memory, the space cannot be reclaimed until the application exits.

- Because of the nature of immortal memory, you might want to find ways to reuse the memory. One possibility is to create a pool of reusable objects. Using scoped memory is an alternative.
- Objects in immortal memory cannot reference anything in scoped memory. If a field of an object in immortal memory is assigned an object from scoped memory, an IllegalAssignmentError exception is thrown.

Scoped memory

Scoped memory can be used as the initial memory area of a schedulable object or can be entered by one. When no longer referenced, the area is cleared of all objects. Schedulable objects running in a scoped memory area perform all their object allocations from that area. When a scoped memory area is unused, the objects inside it are finalized and the memory is reclaimed, preparing the scope for reuse. When the scoped memory area is no longer available to any schedulable objects, the memory is reclaimed for other uses.

The memory area described by a ScopedMemory instance does not exist in the Java heap and is not subject to garbage collection. It is safe to use a ScopedMemory object as the initial memory area associated with a NoHeapRealtimeThread or to enter the memory area using the ScopedMemory.enter method inside a NoHeapRealtimeThread.

Physical memory

Use physical memory when the characteristics of the memory itself are important; for example, non-pageable or non-volatile.

Linear time allocation scheme (LTMemory)

LTMemory represents a memory area guaranteed by the system to have linear time allocation when memory consumption from the memory area is less than the initial size of the memory area. Run time for allocation is allowed to vary when memory consumption is between the initial size and the maximum size for the area. Furthermore, the underlying system is not required to guarantee that memory between initial and maximum is always available.

Variable time allocation scheme (VTMemory)
: VTMemory is similar to LTMemory except that the running time of an allocation from a `VTMemory` area need not complete in linear time.

Heap Memory
: Objects in heap memory cannot reference anything in scoped memory. If a field of an object in heap memory is assigned an object from scoped memory, an IllegalAssignmentError exception is thrown.

Synchronization and resource sharing

In a real-time system, when three or more threads that are running at different priorities and are synchronizing with each other, a condition called priority inversion can occasionally result, in which a higher priority thread is blocked from running by a lower priority thread for an extended period of time. WebSphere Real Time for RT Linux uses a scheme called priority inheritance to avoid this condition.

When a higher priority task is blocked from running by a lower priority task, the priority of the lower priority task is temporarily boosted to match the higher priority until the higher priority task is no longer blocked.

Periodic and aperiodic parameters

Real-time threads have a number of release parameters, determining how often a schedulable object is released. Periodic and aperiodic parameters are examples of release parameters.

Periodic parameters

This class is for those schedulable objects that are released at regular intervals.

AbsoluteTime
: Is expressed in milliseconds and nanoseconds.

RelativeTime
: Is the length of time of a given event expressed in milliseconds and nanoseconds. For example, you can measure the absolute time when an event starts and finishes. You can then calculate the relative time as the difference between the two measurements.

Aperiodic parameters

This class is used by those schedulable objects that are released at irregular intervals. Because a second aperiodic event might occur before the first one has completed, you can define the length of the queue of outstanding requests.

Asynchronous event handling

Asynchronous event handlers react to events that occur outside a thread; for example, input from an interface of an application. In real-time systems, these events must respond within the deadlines that you set for your application.

Asynchronous events can be associated with system interrupts and POSIX signals, and asynchronous events can be linked to a timer.

Like real-time threads, asynchronous event handlers have a number of parameters associated with them. For a list of these parameters, see "Schedulables and their Parameters" on page 10.

Signal Handlers

The POSIXSignalHandler supports the signals SIGQUIT, SIGTERM, and SIGABRT. The default behavior for SIGQUIT causes a Javadump to be generated. The generation of the Javadump does not interfere with the operation of a running program, apart from CPU time and file reading and writing. The generation of a Javadump interrupts the program until the Javadump has been completed; application performance will not be predictable while Javadumps are being generated.

To suppress all core and Javadump generation on a failure, use **`-Xdump:none`**.

To suppress only system dump and Javadump generation on a SIGQUIT signal, specify **`-Xdump:java:none -Xdump:java:events=gpf+abort`**.

The following signals can be attached to asynchronous event handlers (AEHs) by the POSIXSignalHandler mechanism (signal descriptions as defined in `/usr/include/bits/signum.h`):

```
#define SIGQUIT         3       /* Quit (POSIX).  */
#define SIGABRT         6       /* Abort (ANSI).  */
#define SIGKILL         9       /* Kill, unblockable (POSIX).  */
```

No other signals are currently supported. All of the signals listed previously are asynchronous signals, and it is impossible to support attaching to synchronous signals (such as SIGILL and SIGSEGV) because they indicate a failure of your application or the JVM code, not an externally generated event.

Note: SIGQUIT by default causes the Java application to generate dumps (for example, a Javadump) when received by the JVM. Although additionally it is delivered to any attached AEH, this delivery might cause confusing or undesirable behavior and you can disable it by using the **`-Xdump:none:events=user`** option on the Java command line.

Required documentation

WebSphere Real Time for RT Linux implements the Real-Time Specification for Java (RTSJ).

WebSphere Real Time for RT Linux version 2.0 has been certified as RTSJ 1.0.2 compliant against the RTSJ Technology Compatibility Kit version 3.0.13 FCS and is compliant with the Java Compatibility Kit (JCK) for version 6.0.

Supported facilities

The following facilities are supported:

- Allocation-rate enforcement on heap allocation to limit the rate at which a schedulable object creates objects in the heap.

Unsupported facilities

The following facilities are not supported:

- Priority Ceiling Emulation Protocol. For example, it does not permit PriorityCeilingEmulation to be used as a monitor control policy.
- Atomic access support, except where required for conformance to the specification.
- No schedulers other than the base priority scheduler are available to applications.
- Cost enforcement.

Required documentation for Real-Time Specification for Java

The *Required Documentation* section of the Real-Time Specification for Java (RTSJ) is quoted in this section. Any deviations from the standard implementation of RTSJ are noted.

1. **The feasibility testing algorithm is the default.**

   "If the feasibility testing algorithm is not the default, document the feasibility testing algorithm."
2. **Only the base priority scheduler is available to applications.**

   "If schedulers other than the base priority scheduler are available to applications, document the behavior of the scheduler and its interaction with each other scheduler as detailed in the Scheduling chapter. Also document the list of classes that constitute schedulable objects for the scheduler unless that list is the same as the list of schedulable objects for the base scheduler."
3. **A schedulable object that is preempted by a higher priority schedulable object will be placed at the front of the queue for its priority.**

   "A schedulable object that is preempted by a higher-priority schedulable object is placed in the queue for its active priority, at a position determined by the implementation. If the preempted schedulable object is not placed at the front of the appropriate queue the implementation must document the algorithm used for such placement. Placement at the front of the queue can be required in a future version of this specification."
4. **Cost enforcement is not supported.**

   "If the implementation supports cost enforcement, the implementation is required to document the granularity at which the current CPU consumption is updated."
5. **Simple sequential mapping is supported.**

   "The memory mapping implemented by any physical memory type filter must be documented unless it is a simple sequential mapping of contiguous bytes."
6. **There are no subclasses for the Metronome Garbage Collector supplied with WebSphere Real Time for RT Linux.**

   "The implementation must fully document the behavior of any subclasses of GarbageCollector."
7. **There are no MonitorControl subclasses supplied with WebSphere Real Time for RT Linux.**

   "An implementation that provides any MonitorControl subclasses not detailed in this specification must document their effects, particularly with respect to priority inversion control and which (if any) schedulers fail to support the new policy."
8. **A schedulable object holding a monitor required by a higher priority schedulable object has its priority boosted to the higher priority until such time as it releases the monitor. If, at that point, the schedulable object is to be made no longer runnable (that is, there is higher priority work to be**

done) it will be placed at the back of the queue for its original (unboosted) priority when running on kernels prior to SUSE Linux Enterprise Real Time 10 SP2 update kernel version 2.6.22.19-0.16, and Red Hat Enterprise Linux 5.1 MRG 2.6.24.7-73 Errata 1. Kernels at these levels or later place the schedulable object at the front of the queue.

"If on losing "boosted" priority because of a priority inversion avoidance algorithm, the schedulable object is not placed at the front of its new queue, the implementation must document the queuing behavior."

9. **The base scheduler is the only scheduler provided with WebSphere Real Time for RT Linux.**

   "For any available scheduler other than the base scheduler an implementation must document how, if at all, the semantics of synchronization differ from the rules defined for the default PriorityInheritance instance. It must supply documentation for the behavior of the new scheduler with priority inheritance (and, if it is supported, priority ceiling emulation protocol) equivalent to the semantics for the base priority scheduler found in the Synchronization chapter."

10. **The worst case time from the firing of an event to the scheduling of an associated bound event handler will average 40µs and not exceed 100µs providing that there are no competing schedulable objects or system activity of equal or higher priority and providing that garbage collection does not interfere. If the schedulable object driving the fire method, the AsyncEvent object or the handler references the heap then the potential influence of garbage collection is as documented in (A). This assumes that the code is being interpreted and that a single handler (which is bound) is configured on the event.**

    "The worst-case response interval between firing an AsyncEvent because of a bound happening to releasing an associated AsyncEventHandler (assuming no higher-priority schedulable objects are runnable) must be documented for some reference architecture."

11. **The worst case interval between firing an AsynchronouslyInterruptedException at an ATC-enabled thread and the first delivery of that exception will average 35µs and not exceed 160µs providing that there are no competing schedulable objects or system activity of equal or higher priority and providing that garbage collection does not interfere. ATC-enabled in this case means that the thread is executing in an AI enabled method in a region that is not ATC-deferred and those conditions remain true until delivery of the exception. The potential influence of garbage collection is as documented in (A). If the target thread is in native code then the delay is potentially unbounded. This assumes that the code is being interpreted.**

    "The interval between firing an AsynchronouslyInterruptedException at an ATC-enabled thread and first delivery of that exception (assuming no higher-priority schedulable objects are runnable) must be documented for some reference architecture."

12. **Not applicable see response 4.**

    "If cost enforcement is supported, and the implementation assigns the cost of running finalizers for objects in scoped memory to any schedulable object other than the one that caused the scope's reference count to drop to zero by leaving the scope, the rules for assigning the cost shall be documented."

13. **There are no changes to the standard implementation of RealtimeSecurity.**

"If the implementation of RealtimeSecurity is more restrictive than the required implementation, or has runtime configuration options, those features shall be documented."

14. **The finalizers for objects in a scoped memory area will be run by the last thread to reference that area, that is, they will be run when the thread decrements the reference count from one to zero. Any cost associated with running the finalizers will be assigned to that thread.**

    "An implementation can run finalizers for objects in scoped memory before the scope is reentered and before it returns from any call to getReferenceCount() for that scope. It must, however, document when it runs those finalizers."
15. **The resolution is not settable.**

    "For each supported clock, the documentation must specify whether the resolution is settable, and if it is settable the documentation must indicate the supported values."
16. **There are no other clocks other than the real-time clock provided with WebSphere Real Time for RT Linux.**

    "If an implementation includes any clocks other than the required real-time clock, their documentation must indicate in what contexts those clocks can be used."

Note:

A The reference architecture for the tests will be an LS20, 4–way, 2 GHz with 1 MB cache and 4 GB of memory.

B Garbage collection can cause a delay at any point in a thread that is associated with the heap. The collector can operate in one of two basic modes governing the behavior when heap memory is exhausted. If the collector is set to immediately throw OutOfMemoryError in these circumstances, the worst case garbage collection delay will typically be below 1 ms. Currently, in some circumstances, the delay can be higher; for example, if there are many threads with deeply nested stacks or large numbers of large-sized scopes. If the collector is set to perform a synchronous GC before throwing an OutOfMemoryError, the potential collection delay is related to the number of live objects in the heap and the numbers of objects in other memory areas. In these circumstances, the delay is considered to be unbounded because it can be many seconds for typical heap sizes.

Chapter 3. Planning

Read this section before installing WebSphere Real Time for RT Linux.
- "Migration"
-
- "Hardware and software prerequisites"
- "Considerations" on page 24

Migration

WebSphere Real Time for RT Linux runs in a Linux environment that is modified for real-time applications. You can use standard Java applications in a real-time environment. Alternatively, you can modify your applications to exploit the features of WebSphere Real Time.

System migration

Follow the instructions provided by the Linux support team.

Hardware and software prerequisites

Use this list to check the hardware, operating system, and Java environment that is supported for WebSphere Real Time for RT Linux.

Hardware

WebSphere Real Time for RT Linux certified hardware configurations are multiprocessor variants of the following systems:
- IBM BladeCenter® LS20 (Types 8850-76U, 8850-55U, 7971, 7972)
- IBM eServer™ xSeries 326m (Types 7969-65U, 7969-85U, 7984-52U, 7984-6AU)
- IBM BladeCenter LS21 (Type 7971-6AU)
- IBM BladeCenter HS21 XM Dual Quad Core (Type 7995)

To remain certified for WebSphere Real Time for RT Linux, IBM systems with hyperthreading must not have hyperthreading enabled.

In addition, WebSphere Real Time for RT Linux is supported on hardware that runs a supported operating system, and that has these characteristics:
- A minimum of 512 MB of physical memory.
- Minimum Intel Pentium 4, AMD Opteron, or Intel Atom Processor.

For systems that are not certified hardware configurations, IBM does not make any performance statements. Performance considerations for certified hardware configurations are detailed here: Chapter 7, "Performance," on page 93

On systems with hyperthreading support, ensure that it is not enabled to avoid adverse performance effects when using WebSphere Real Time for RT Linux.

Operating system

- Red Hat Enterprise Messaging, Realtime, Grid (MRG) 1.3 for Red Hat Enterprise Linux 5 Update 5.
- Red Hat Enterprise MRG 2.2 for Red Hat Enterprise Linux (RHEL) 6 Update 5. See note.
- SUSE Linux Enterprise Real Time (SLERT) 10.
- SLERT 11 service pack 1, with patch level 2.6.33.18.

Note: In high workload conditions, an intermittent problem is observed when running applications under IBM WebSphere Real Time V3 for Real Time Linux on MRG 2.2 with RHEL 6 Update 3. This problem is resolved in RHEL 6 Update 5, which is the recommended base level. For more information, see http://www.ibm.com/support/docview.wss?uid=swg21624408.

See "Installing a Real Time Linux environment" on page 27.

Considerations

You must be aware of a number of factors when using WebSphere Real Time for RT Linux.

- Where possible, do not run more than one real-time JVM on the same system. The reason is that you would then have multiple garbage collectors. Each JVM does not know about the memory areas of the other. One effect is that GC cycles and pause times cannot be coordinated across JVMs, meaning that it is possible for one JVM to affect adversely the GC performance of another JVM. If you must use multiple JVMs, ensure that each JVM is bound to a specific subset of processors by using the **`taskset`** command.
- You cannot use the **`-Xdebug`** option and the **`-Xnojit`** option with code that has been precompiled by using the Ahead-of-Time compiler. The reason is that **`-Xdebug`** compiles code in a different way from the Ahead-of-Time (AOT) compiler and is not supported.

  To debug your code, use interpreted or JIT-compiled code.
- If you are using the com.sun.tools.javac.Main interface to compile Java source code that uses the javax.realtime package, you must ensure that *`sdk/jre/lib/i386/realtime/jclSC170/realtime.jar`* is included in the class path. One common example of this type of compilation is ant compilation.
- The optional JavaComm package can be installed into WebSphere Real Time for RT Linux and accessed from both the real-time and non-real-time JVM. For more information about installation and configuration, see http://publib.boulder.ibm.com/infocenter/java7sdk/v7r0/topic/com.ibm.java.lnx.70.doc/user/jcommchapter.html. The real-time JVM in WRT supports the JavaComm API for use with regular Java threads. However, no guarantee exists in respect to determinism or real-time performance when accessing external devices with JavaComm. As such, do not use JavaComm with no-heap real-time and real-time threads, or when real-time behavior is required.
- The shared caches used by earlier WebSphere Real Time for RT Linux releases to store precompiled code and classes are not compatible with the caches used by this release of WebSphere Real Time for RT Linux. You must regenerate the contents of the earlier caches.
- When using shared class caches, the cache name must not exceed 53 characters.
- The **`ps`** command truncates Java thread names.

  The **`ps`** command is limited to 15 characters. If you set a thread name to more than 15 characters, the name is truncated by the **`ps`** command.

- WebSphere Real Time for RT Linux does not support NTLoginModule (NTLM) authentication.

  NTLoginModule (NTLM) is used to help authenticate access to a Windows service. Authentication by using NTLM is supported on the Windows platform only. This means that WebSphere Real Time for RT Linux does not support NTLM authentication.

Chapter 4. Installing WebSphere Real Time for RT Linux

Follow these steps to install the product.

- "Installation files"
- "Installing a Real Time Linux environment"
- "Installing from an InstallAnywhere package" on page 28
  - "Completing an attended installation" on page 29
  - "Completing an unattended installation" on page 30
  - "Known issues and limitations" on page 31
- "Setting the path" on page 32
- "Setting the classpath" on page 33
- "Testing your installation" on page 33
- "Uninstalling WebSphere Real Time for RT Linux" on page 34

Installation files

You require these installation files.

IBM WebSphere Real Time for RT Linux is provided in two types of InstallAnywhere package.

Installable packages
Installable packages configure your system. For example, the programs might set environment variables.

- `wrt-3.0-0.0-rtlinux-x86_32-sdk.bin`
- `wrt-3.0-0.0-rtlinux-x86_32-jre.bin`

Archive packages
These packages extract the files to your system, but do not perform any configuration.

- `wrt-3.0-0.0-rtlinux-x86_32-sdk.archive.bin`
- `wrt-3.0-0.0-rtlinux-x86_32-jre.archive.bin`

Installing a Real Time Linux environment

Before installing WebSphere Real Time for RT Linux, you must install a 64-bit version of Real Time Linux.

Red Hat Enterprise Messaging, Realtime, Grid (MRG 1.3) for RHEL 5.5
For more information about installing the real time component of Red Hat Enterprise Linux 5.5 MRG 1.3, see the installation instructions for RT-Linux RHEL 5.5 MRG 1.3: https://access.redhat.com/site/documentation/en-US/Red_Hat_Enterprise_MRG/1.3/html/Realtime_Installation_Guide/index.html

Red Hat Enterprise MRG 2.2 for RHEL 6.3
For more information about installing the real time component of Red Hat Enterprise MRG 2.2, see the installation instructions: https://access.redhat.com/site/documentation/en-US/Red_Hat_Enterprise_MRG/2/html/Realtime_Installation_Guide/index.html

SUSE Linux Enterprise Real Time (SLERT) 10
For more information about installing SLERT 10, see: http://www.novell.com/products/realtime/eval.html

SUSE Linux Enterprise Real Time (SLERT) 11 service pack 1
You can obtain SLERT 11 from http://download.novell.com/. Service pack 1 is required to work correctly with WebSphere Real Time for RT Linux. You must also apply patch level 2.6.33.18, which you can download from Real Time Linux Kernel 5075. For more information about SLERT 11, see: https://www.suse.com/products/realtime/.

When using a large number of file descriptors to load different instances of classes, you might see an error message "`java.util.zip.ZipException: error in opening zip file`", or some other form of IOException advising that a file could not be opened. The solution is to increase the provision for file descriptors, using the **`ulimit`** command. To find the current limit for open files, use the command:

```
ulimit -a
```

To allow more open files, use the command:

```
ulimit -n 8196
```

Installing from an InstallAnywhere package

These packages provide an interactive program that guides you through the installation options. You can run the program as a graphical user interface, or from a system console.

Before you begin

Your system must have both the following shared libraries:
- GNU C library V2.3 (glibc)
- `libstdc++.so.5`

If you do not have the `libstdc++.so.5` shared library, you might see a Java core dump when you install, containing the following errors:

```
JVMJ9VM011W Unable to load j9dmp24: libstdc++.so.5: cannot open shared object file:
No such file or directory
JVMJ9VM011W Unable to load j9gc24: libstdc++.so.5: cannot open shared object file:
No such file or directory
JVMJ9VM011W Unable to load j9vrb24: libstdc++.so.5: cannot open shared object file:
No such file or directory
```

If you are installing an installable package, you must have the rpm-build tool installed on your system, otherwise the installation program cannot register the new package in the RPM database. To find out if the rpm-build tool is installed, enter the following command:

```
rpm -q rpm-build
```

About this task

The InstallAnywhere packages have a `.bin` file extension.

There are two types of package:

Installable
Installing these packages also configures your system, for example by setting environment variables.

Archive
Installing these packages extracts the files to your system, but does not perform any configuration.

Procedure

- To install the package in an interactive way, complete an attended installation.
- To install the package without any additional user interaction, complete an unattended installation. You might choose this option if you want to install many systems.
- When the installation process is completed, follow the configuration steps in this section, such as settting path and classpath environment variables.

Results

The product is installed.

Note: Do not interrupt the installation process, for example by pressing Ctrl+C. If you interrupt the process, you might have to reinstall the product. For more information, see "Interrupted installation" on page 31.

If you are using an installable package, you might see messages advising that a problem has been found. Installation of the archive packages does not produce any messages. Some of the messages that you might see when using an installable package are shown in the following list:

`The installer cannot run on your configuration. It will now quit.`
This error message occurs when your user ID is not authorized to run the installation process. Because it cannot continue, the installation program ends. To fix the problem, start the installation again but with a user ID that has root authority.

`An RPM package is already installed. Uninstall the package before proceeding.`
This message indicates that an RPM package is already installed. Because it cannot continue, the installation program ends. To fix the problem, uninstall the RPM package before proceeding.

Completing an attended installation

Install the product from an InstallAnywhere package, in an interactive way.

Before you begin

Check the following conditions before you begin the installation process:

- If you have previously installed WebSphere Real Time for RT Linux from an RPM package, you must uninstall this package before proceeding.
- You must have a user ID with root authority.

Procedure

1. Download the installation package file to a temporary directory.
2. Change to the temporary directory.
3. Start the installation process by typing `./package` at a shell prompt, where *package* is the name of the package that you are installing.
4. Select a language from the list shown in the installer window, then click **Next**. The list of available languages is based on the locale setting for your system.

5. Read the license agreement, using the scroll bar to reach the end of the license text. To proceed with the installation you must accept the terms of the license agreement. To accept the terms, select the radio button, then click **OK**.

   Note: You cannot select the radio button to accept the license agreement until you have read to the end of the license text.
6. You are asked to choose the target directory for the installation. If you do not want to install into the default directory, click **Choose** to select an alternative directory, by using the browser window. When you have chosen the installation directory, click **Next** to continue.
7. You are asked to review the choices that you made. To change your selection, click **Previous**. If your choices are correct, click **Install** to proceed with installation.
8. When the installation process is complete, click **Done** to finish.

Completing an unattended installation

If you have more than one system to install, and you already know the installation options that you want to use, you might want to use the unattended installation process. You install once by using the attended installation process, then use the resulting response file to complete further installations without any additional user interaction.

Procedure

1. Create a response file by completing an attended installation. Use one of the following options:
   - Use the GUI and specify that the installation program creates a response file. The response file is called `installer.properties`, and is created in the installation directory.
   - Use the command line and append the -r option to the attended installation command, specifying the full path to the response file. For example:

     ```
     ./package -r /path/installer.properties
     ```

   Example response file contents:

   ```
   INSTALLER_UI=silent
   USER_INSTALL_DIR=/my_directory
   ```

   In this example, *`/my_directory`* is the target installation directory that you chose for the installation.
2. Optional: If required, edit the response file to change options.

   Note: Archive packages have the following known issue: installations that use a response file use the default directory even if you change the directory in the response file. If a previous installation exists in the default directory, it is overwritten.

   If you are creating more than one response file, each with different installation options, specify a unique name for each response file, in the format *`myfile`*`.properties`.
3. Optional: Generate a log file. Because you are installing silently, no status messages are displayed at the end of the installation process. To generate a log file that contains the status of the installation, complete the following steps:
   a. Set the required system properties by using the following command.

      ```
      export _JAVA_OPTIONS="-Dlax.debug.level=3 -Dlax.debug.all=true"
      ```

b. Set the following environment variable to send the log output to the console.

```
export LAX_DEBUG=1
```

4. Start an unattended installation by running the package installer with the **-i** silent option, and the **-f** option to specify the response file. For example:

```
./package -i silent -f /path/installer.properties 1>console.txt 2>&1
./package -i silent -f /path/myfile.properties 1>console.txt 2>&1
```

You can use a fully qualified path or relative path to the properties file. In these examples, the string `1>console.txt 2>&1` redirects installation process information from the stderr and stdout streams to the `console.txt` log file in the current directory. Review this log file if you think there was a problem with the installation.

Note: If your installation directory contains multiple response files, the default response file, `installer.properties` is used.

Interrupted installation

If the package installer is unexpectedly stopped during installation, for example if you press Ctrl+C, the installation is corrupted and you cannot uninstall or reinstall the product. If you try to uninstall or reinstall you might see the message `Fatal Application Error`.

About this task

To solve this problem, delete files and reinstall, as described in the following steps.

Procedure

1. Delete the `/var/.com.zerog.registry.xml` registry file.
2. Delete the directory containing the installation, if it was created. For example opt/IBM/javawrt3/.
3. Run the installation program again.

Known issues and limitations

The InstallAnywhere packages have some known issues and limitations.

- If you do not have the `libstdc++.so.5` shared library on your system, the installation fails, producing a Java core dump. For more information, see "Installing from an InstallAnywhere package" on page 28.
- The installation package GUI does not support the Orca screen-reading program. You can use the unattended installation mode as an alternative to the GUI.
- If, after installation, you enter `./package` to start the program again, the program displays the following message:

```
ENTER THE NUMBER OF THE DESIRED CHOICE, OR PRESS <ENTER> TO ACCEPT THE DEFAULT:
```

  If you press Enter to accept the default, the program does not respond. Type a number, then press Enter.
- If you install the package, then attempt to install again in a different mode, for example console or silent, you might see the following error message:

```
Invocation of this Java Application has caused an InvocationTargetException.
This application will now exit
```

You should not see this message if you installed by using the GUI mode and are running the installation program again in console mode. If you see this error, and are running the program to select the uninstallation option (installable packages only), use the **./_uninstall/uninstall** command instead, as described in "Uninstalling WebSphere Real Time for RT Linux" on page 34.

Installable packages only

- You cannot upgrade an existing installation by using the InstallAnywhere packages. To upgrade WebSphere Real Time for RT Linux, you must first uninstall any previous versions.
- You cannot install two different instances of the same version of WebSphere Real Time for RT Linux on the same system at the same time, even if you use different installation directories. For example, you cannot simultaneously have WebSphere Real Time for RT Linux V3 in directory `/previous`, and an WebSphere Real Time for RT Linux service refresh installation in directory `/current`. The installation program checks the version number. If the program finds an existing package with the same version number, you are asked to uninstall the existing package.
- If the package is installed, and you run the package installer again by using the GUI, you can select to uninstall the package. This uninstallation option is not available in unattended mode. If you run the package installer again in unattended mode, the program runs but does not perform any actions.

Archive packages only

- If you change the installation directory in a response file, and then run an unattended installation by using that response file, the installation program ignores the new installation directory and uses the default directory instead. If a previous installation exists in the default directory, it is overwritten.

Setting the path

When you have set the **PATH** environment variable, you can run an application or program by typing its name at a shell prompt.

About this task

Note: If you alter the **PATH** environment variable as described in this section, you override any existing Java executables in your path.

You can specify the path to a tool by typing the path before the name of the tool each time. For example, if the SDK is installed in `opt/IBM/javawrt3/`, you can compile a file named *myfile.java* by typing the following command at a shell prompt:

```
opt/IBM/javawrt3/bin/javac myfile.java
```

To avoid typing the full path each time:

1. Edit the shell startup file in your home directory (usually **.bashrc**, depending on your shell) and add the absolute paths to the **PATH** environment variable; for example:
   ```
   export PATH=opt/IBM/javawrt3/bin:opt/IBM/javawrt3/jre/bin:$PATH
   ```
2. Log on again or run the updated shell script to activate the new **PATH** setting.
3. Compile the file with the **javac** tool. For example, to compile the file *myfile.java*, at a shell prompt, enter:
   ```
     javac -Xrealtime myfile.java
   ```

The **PATH** environment variable enables Linux to find executable files, such as **javac**, **java**, and the **javadoc** tool, from any current directory. To display the current value of your path, type the following command at a command prompt:

```
echo $PATH
```

What to do next

See "Setting the classpath" to determine whether you need to set your CLASSPATH environment variable.

Setting the classpath

The **CLASSPATH** environment variable tells the SDK tools, such as **java**, **javac**, and **javadoc** tool, where to find the Java class libraries.

About this task

Set the **CLASSPATH** environment variable explicitly only if one of the following conditions applies:

- You require a different library or class file, such as one that you develop, and it is not in the current directory.
- You change the location of the `bin` and `lib` directories and they no longer have the same parent directory.
- You plan to develop or run applications using different runtime environments on the same system.

To display the current value of your **CLASSPATH**, enter the following command at a shell prompt:

```
echo $CLASSPATH
```

If you develop and run applications that use different runtime environments, including other versions that you have installed separately, you must set **CLASSPATH** and **PATH** explicitly for each application. If you run multiple applications simultaneously and use different runtime environments, each application must run in its own shell.

If you run only one version of Java at a time, you can use a shell script to switch between the different runtime environments.

What to do next

See "Testing your installation" to verify that your installation has been successful.

Testing your installation

Use the **-version** option to check if your installation is successful.

About this task

The Java installation consists of a standard JVM and a real-time JVM.

Procedure

Test your installation by following these steps:

1. To see version information for the standard JVM, type the following command at a shell prompt:

   ```
   java -version
   ```

   This command returns the following messages if it is successful:

   ```
   java version "1.7.0"
   WebSphere Real Time V3 (build pxi3270rt-20110518_02)
   IBM J9 VM (build 2.6, JRE 1.7.0 Linux x86-32 20110516_82445 (JIT enabled,
   AOT enabled)
   J9VM - R26_head_20110515_0456_B82363
   JIT  - r11_20110510_19526
   GC   - R26_head_20110513_1009_B82250
   J9CL - 20110516_82445)
   JCL - 20110516_01 based on Oracle 7b145
   ```

   If you intend to use the standard JVM and not the real-time JVM, refer to the IBM User Guides for Java v7 on Linux.

   Note: The version information is correct but the dates might be later than the ones in this example. The format of the date string is: `yyyymmdd` followed possibly by additional information specific to the component.
2. To see version information for the real-time JVM, type the following command at a shell prompt:

   ```
   java -Xrealtime -version
   ```

   This command returns the following messages if it is successful:

   ```
   java version "1.7.0"
   WebSphere Real Time V3 (build pxi3270rt-20110518_02)
   IBM J9 VM (build 2.6, JRE 1.7.0 real-time Linux x86-32 20110516_82445 (JIT
   enabled, AOT enabled)
   J9VM - R26_head_20110515_0456_B82363
   JIT  - r11_20110510_19526
   GC   - R26_head_20110513_1009_B82250
   J9CL - 20110516_82445)
   JCL - 20110516_01 based on Oracle 7b145
   ```

   Note: The version information is correct but the platform architecture and dates might differ from the example. The format of the date string is: `yyyymmdd` followed possibly by additional information specific to the component.

Uninstalling WebSphere Real Time for RT Linux

The process that you use to remove WebSphere Real Time for RT Linux depends on what type of installation you used.

Before you begin

For InstallAnywhere installable packages, you must have a user ID with root authority.

About this task

There is no uninstallation process for InstallAnywhere archive packages. To remove an archive package from your system, delete the target directory that you chose when you installed the package. For InstallAnywhere installable packages, you uninstall the product by using a command, or by running the installation program again, as described in the following steps.

Procedure

- Optional: Uninstall manually by using the **`uninstall`** command.
  1. Change to the directory that contains the installation. For example:
     ```
     cd /opt/IBM/javawrt3
     ```
  2. Start the uninstallation process by entering the following command:
     ```
     ./_uninstall/uninstall
     ```
- Optional: If you cannot locate the uninstall program easily, as an alternative you can run another attended installation. The installation program detects that the product is already installed, then gives you the opportunity to uninstall the previous installation.

Chapter 5. Running IBM WebSphere Real Time for RT Linux applications

Important information to assist you when running real time applications.

- "Thread scheduling and dispatching"
- "Using compiled code with WebSphere Real Time for RT Linux" on page 40
- "Using no-heap real-time threads" on page 58
- "Class data sharing between JVMs" on page 66
- "Using the Metronome Garbage Collector" on page 68

Thread scheduling and dispatching

The Linux operating system supports various scheduling policies. The default universal time sharing scheduling policy is SCHED_OTHER, which is used by most threads. SCHED_RR and SCHED_FIFO can be used by threads in real-time applications. .

The kernel decides which is the next runnable thread to be run by the processor. The kernel maintains a list of runnable threads. It looks for the thread with the highest priority and selects that thread as the next thread to be run.

Thread priorities and policies can be listed using the following command:

```
ps -emo pid,ppid,policy,tid,comm,rtprio,cputime
```

where policy:

- `TS` is SCHED_OTHER
- `RR` is SCHED_RR
- `FF` is SCHED_FIFO
- `-` has no policy reported

The output looks like this example:

```
  PID  PPID POL   TID COMMAND         RTPRIO     TIME
18314 30285 -       - java                 - 00:01:40
    -     - RR  18314 -                    6 00:00:00
    -     - RR  18315 -                    6 00:01:40
    -     - FF  18318 -                   88 00:00:00
    -     - RR  18323 -                    6 00:00:00
    -     - FF  18324 -                   13 00:00:00
    -     - RR  18325 -                    6 00:00:00
    -     - RR  18326 -                    6 00:00:00
    -     - FF  18327 -                   11 00:00:00
    -     - FF  18328 -                   89 00:00:00
```

This output shows the Java process, the scheduling policy in force, the main thread with priority "-" (other), and some real-time threads with priorities from 11 to 89.

To query the current scheduling policy, use **`sched_getscheduler`**, or the **`ps`** command shown in the example.

For more information about processes, see "General debugging techniques" on page 98.

Regular Java thread priorities and policies

Regular Java threads, that is, threads allocated as `java.lang.Thread` objects, use the default scheduling policy of SCHED_OTHER. From WebSphere Real Time for RT Linux V3 service refresh 1, you can run regular Java threads with the SCHED_RR or SCHED_FIFO scheduling policy.

By default, Java threads are run using the default SCHED_OTHER policy. This policy maps Java threads to the operating system priority 0.

Using the SCHED_RR or SCHED_FIFO policy gives you finer control over your application, which can improve the real-time performance of Java threads. The JVM detects the priority and policy of the main thread when Java is started with the SCHED_RR or SCHED_FIFO policy. The JVM alters the priority and policy mappings accordingly. All Java threads are run at the same operating system priority as the main thread. Although SCHED_RR or SCHED_FIFO can be assigned priorities 1 - 99, the usable SCHED_RR or SCHED_FIFO priorities for WebSphere Real Time for RT Linux V3 are priorities 1 - 10. If the priority is set higher than 10, the priority of the main thread is lowered to 10 and the mapping applied based on the value of 10.

One way to alter the real-time scheduling property of a process on the command line is to use the command **`chrt`**. In the following example, the priority of the main Java thread is set to use the SCHED_FIFO scheduling policy, with an operating system priority of 6.

```
chrt -f 6 java
```

You might need to configure your system to allow priorities to be changed. See "Configuring the system to allow priority changes" on page 39 for more information.

Table 4. Java and operating system priorities

Java priority	Java priority value for thread	Operating system priority
1	MIN_PRIORITY	6
2		6
3		6
4		6
5	NORM_PRIORITY (default)	6
6		6
7		6
8		6
9		6
10	MAX_PRIORITY	6

All threads associated with the main Java thread are run at the same operating system priority. The metronome alarm thread, and other threads internal to the JVM with precise real-time timing requirements, might run at a priority higher than the priorities used by regular Java threads.

If you run the command `chrt -f 11 java`, the result is the same as running `chrt -f 10 java`. This is because you cannot apply a priority above 10 to the priority mapping used by JVM threads, although the thread that launches the JVM and waits for JVM termination remains at priority 11.

For more information about the **chrt** command, see http://publib.boulder.ibm.com/infocenter/lnxinfo/v3r0m0/index.jsp?topic=/liaai/realtime/liaairtchrt.htm.

Configuring the system to allow priority changes

By default, non-root users on Linux cannot raise the priority of a thread or process. You can change the system configuration to allow priority changes using the `pam_limits` module of the Pluggable Authentication Modules (PAM) for Linux.

If you cannot change the priority of a thread or process using the **chrt** utility, you typically see the following message:

```
sched_setscheduler: Operation not permitted
```

On recent Linux kernels, you can change the configuration of the system to allow priority changes using the `pam_limits` module. This module allows you to configure the limits on system resources in the limits configuration file. The default file is `/etc/security/limits.conf`.

An entry in the `/etc/security/limits.conf` file has the following form:

```
<domain> <type> <item> <value>
```

where:

`<domain>` is either:
- a user name on the system that can alter limits on a resource.

- a group name, with the syntax `@group`, whose members can alter limits on a resource.

- a wildcard "*", which indicates that any user or group can alter limits on a resource.

`<type>` is either:
- `hard`, where hard limits are enforced by the kernel.

- `soft`, where soft limits apply, which can be altered within the range specified by the hard limits.

- a dash "-", which indicates hard and soft limits.

`<item>` is:
- a resource. Use `rtprio` for real-time priorities.

`<value>` is:
- a limit. Use a value in the range 1 - 100 to indicate the maximum limit for real-time priority setting.

For example,

```
* - rtprio 100
```

allows all users to change the priority of real-time processes, using **chrt** or other mechanisms.

By default, the root user can increase real-time priorities without limits. To apply a limit to root, the root user must be explicitly specified. Group and wildcard limits in the configuration file do not apply to the root user.

If you specify individual user limits in the file, these limits have priority over group limits.

Changes to `limits.conf` do not take effect immediately. You must restart the affected services or reboot the system for a configuration change to take effect.

To enable priority changes on a real-time Linux system you can add a user to the `realtime` group, shown in the `limits.conf` file.

Launching secondary processes

The java.lang.Runtime.exec methods in the Java virtual machine (JVM) API give your Java application the ability to execute a command in a separate process.

From that method call, a new java.lang.Process object is created. The object can be used to control the new process, or to obtain information about it.

Several threads are created by the exec methods for this purpose. In IBM WebSphere Real Time for RT Linux, modifications of the procedure enable more deterministic behavior in a real-time environment.

The Runtime.exec call creates a "reaper" thread for each forked subprocess. The reaper thread is the only thread that waits for the subprocess to terminate. When the subprocess terminates, the reaper thread records the subprocess exit status. The reaper thread spawns the new process, and gives it the same scheduling parameters as the thread that originally called Runtime.exec.

If the spawned process is another WebSphere Real Time for RT Linux JVM, and the Runtime.exec method was called by another method running with a Linux real-time policy and priority, then the main thread of the new virtual machine maps its policy and priority to the same Linux real-time policy and priority. This Java thread priority is between 1 and 10.

The reaper thread also creates two new threads that listen to the `stdout` and `stderr` streams of the new process. These new threads are regular Java threads, not real-time Java threads. The `stdout` and `stderr` data is saved into buffers used by these threads. The buffers persist beyond the lifetime of the spawned process. This persistence allows the resources held by the spawned process to be cleared immediately when the process terminates.

If you want the `stdout` and `stderr` reader threads to run at a Linux real-time priority, launch the original JVM owning these threads with a Linux SCHED_FIFO or SCHED_RR policy and priority. The effect is to map all regular threads to a real-time policy and priority as high as 10 in the Linux operating system.

Real-time Java thread priorities and policies

Real-time threads, that is, threads allocated as `java.realtime.RealtimeThread`, and asynchronous event handlers use the SCHED_FIFO scheduling policy.

Thread scheduling and dispatching of real-time Java threads is part of the Real Time Specification for Java (RTSJ). This topic, including the scheduling policies and priority handling of real-time Java threads is discussed in the section "Support for RTSJ" on page 9.

Using compiled code with WebSphere Real Time for RT Linux

IBM WebSphere Real Time for RT Linux supports several models of code compilation, providing varying levels of code performance and determinism.

Interpreted operation

This is the simplest code compilation model. The interpreter runs a Java application, but does not use code compilation at all. The interpreter exhibits good determinism, but provides very low performance, therefore avoid using this mode of operation for production systems.

To use interpreted operation, specify the **`-Xint`** option on the Java command line.

Low priority Just-In-Time (JIT) compilation

The default compilation model in WebSphere Real Time for RT Linux uses a Just-In-Time compiler to compile the important methods of a Java application while the application runs. In this mode, the JIT compiler works in a similar way to the operation of the JIT compiler in a non-real-time JVM. The difference is that the WebSphere Real Time for RT Linux JIT compiler runs at a lower priority level than any real-time threads. The lower priority means that the JIT compiler uses system resources when the application does not need to perform real-time tasks. The effect is that the JIT compiler does not significantly affect the performance of real-time tasks.

The JIT compiler uses two threads for compilation-related activities: the compilation thread, and the sampler thread. These threads run at lower priority than real-time tasks. The compilation thread runs in an asynchronous fashion to the application. This means that an application thread does not wait for the compilation thread to finish compiling a method at any time. The sampler thread periodically sends an asynchronous message to the application threads to identify the currently running method on each thread. Processing the message takes little time on the application thread. No messages are sent if the sampling thread cannot run because of higher priority real-time tasks. Using the JIT compiler has a small effect on determinism, but this compilation mode provides the best performance for many users.

To run an application with the JIT at low priority, see "Enabling the JIT" on page 57.

Ahead-Of-Time (AOT) precompiled code

WebSphere Real Time for RT Linux compiles Java methods to native code in a precompilation step before running the application. Prior to WebSphere Real Time for RT Linux V2, the precompilation step used the jxeinajar tool to compile methods using an Ahead-Of-Time compiler, and stored the results in special Java executable files. These files might be collected into bound jar files. When running an application, bound jar files are added to the application class path so that the JVM can load AOT code when the classes for methods were loaded from the JXE. Using this approach, the JIT compiler is made completely unavailable by specifying the **`-Xnojit`** option on the command-line. The application can use any precompiled AOT code that has been created, and the interpreter for other methods. This mode of operation provides high determinism because the JIT compiler is not present, so there are no sampling thread or context switch performance reductions. The difficulty of compiling Java code ahead of time, while conforming with the Java specification, means that AOT compiled code typically performs slower than JIT compiled code, although it is typically much faster than interpreting.

WebSphere Real Time for RT Linux V2 and subsequent versions stores AOT code in a shared class cache rather than in JXE files, using the shared classes technology provided in the IBM Java 6 JVMs. The admincache tool lets you query the contents of a cache, list all existing caches, and populate a cache with classes and AOT code. The advantages of storing AOT compiled code are that the application jar files are not modified, and no class path changes are needed when running the application.

A shared class cache has a practical size limit, based on the available virtual address space. This means that AOT compilation for all jar files is not practical. Selective AOT compilation must be performed.

When an application runs with AOT code in a shared class cache, the AOT code for methods of a class loads automatically when the class loads into the JVM. The additional cost in loading a class to install AOT code for its methods makes it important to preload as many classes as possible before the performance critical parts of the application run.

Using AOT precompiled code provides the highest level of determinism, with good performance. AOT code can be used when your application runs by specifying the **-Xshareclasses** and **-Xaot** options. The **-Xaot** option is on by default.

To store and use AOT code with a shared class cache using the admincache tool, see “Using the admincache tool” on page 43. Information regarding migration from jxeinajar to admincache can be found in the WebSphere Real Time for RT Linux V2 documentation.

For an example of running an application with AOT compiled code, see “Running the sample application while using AOT” on page 87.

Mixed mode, combining AOT precompiled code and low priority JIT compilation

AOT and JIT compiled code can be used together while the application runs. This mode of operation can provide very good determinism with good performance, and very high performance for methods that run frequently. The main benefit of this mode is that AOT precompilation is used to ensure that the most important parts of your application never run in the interpreter, which is typically far slower than AOT or JIT compiled code. You do not need to precompile every method because the JIT compiler can identify dynamically any interpreted methods that run frequently, without disturbing the application performance significantly. Mixed mode is the default mode when the **-Xshareclasses** option is added to the command line.

To run your application with mixed AOT and JIT compilation, see “Running the sample application while using AOT” on page 87

Managing compilation explicitly

In compilation modes with the JIT compiler enabled, the java.lang.Compiler API can be used to control JIT compiler operation explicitly. The JIT compiler compiles methods of the class passed using the compileClass() method. compileClass() is synchronous, therefore it does not return until the supplied methods have been compiled. An application might use compileClass() in an initialization phase, by iterating over the classes used by the main phase of the application run time. When the initialization phase finishes, call the Compiler.disable() method to disable the compilation and sampling threads entirely. The main difficulty with this technique is the problem of managing the list of classes to load and compile in the application initialization phase, especially during application development.

For more information about managing compilation in an application, see IBM Real-Time Class Analysis Tool for Java.

Overview of Compilation Command-Line Options

You can run an application with JIT enabled using the **`-Xjit`** option or without the JIT using the **`-Xnojit`** option. **`-Xjit`** is the default mode.

You can run an application with AOT code enabled using the **`-Xshareclasses -Xaot`** options. Disable AOT code by using the **`-Xnoaot`** option. **`-Xaot`** is the default option, but has no effect unless the **`-Xshareclasses`** option is also specified, because AOT code must be stored in a shared class cache.

Using the AOT compiler

Use these steps to precompile your Java code. This procedure describes the use of the **`-Xrealtime`** option in a **`javac`** command, the **`admincache`** tool, and the **`-Xrealtime`** and **`-Xnojit`** options with the **`java`** command.

About this task

Using the ahead-of-time compiler means that compilation is separate from the run time of the application. Also, you can compile more methods at the same time rather than just the frequently used methods. You can compile everything in an application or just individual classes, as shown in the following steps.

Note: When using shared class caches, the name of the cache must not exceed 53 characters.

Procedure

1. From a shell prompt, enter:
   ```
   javac -Xrealtime source
   ```
   This command creates the Java bytecode from your source for use in the real-time environment. See Figure 2 on page 8.
2. Package the class files generated into a JAR file. For example, to create test.jar:
   ```
   jar cvf test.jar source
   ```
3. From a shell prompt, enter:
   ```
   admincache -Xrealtime -populate -aot test.jar -cacheName myCache -cp test.jar
   ```
   This command pre-compiles the test.jar file and writes the output to the output directory `./aot`.
4. To run the file using the AOT code in the shared class cache, in a shell prompt enter:
   ```
   java -Xrealtime -Xshareclasses:name=myCache -cp test.jar -Xnojit MyTestClass
   ```
   To run the file using the AOT code in the shared class cache, recompile frequently called methods. Then, run the following command without creating a new JAR file:
   ```
   java -Xrealtime -Xshareclasses:name=myCache -cp test.jar MyTestClass
   ```
   These commands use the same JAR files that you precompiled in step 3.

Using the admincache tool

The admincache tool is used to manage the shared class caches on a workstation.

In the IBM WebSphere Real Time for RT Linux product, the admincache tool can be used to create a shared class cache containing classes or classes and AOT compiled code. After caches have been created, this tool can also be used to inspect existing caches.

The shared class cache is used to reduce the memory footprint in multi-JVM scenarios, and to accelerate application start-up.

Shared class caches can be used by WebSphere Real Time for RT Linux in both non-real-time and real-time modes, but the cache format, creation and population techniques differ. Real-time mode caches are not compatible with non-real-time mode caches. In non-real-time mode, caches are created and populated in the same way as the standard JVM. This means the cache is created and populated by the JVM as it runs an application, transparently to the user. In real-time mode, using the **`-Xrealtime`** option, shared class caches must be created and prepopulated by admincache, using the **`-populate`** option. Applications running in real-time mode might read content from the prepopulated cache, but cannot modify its contents.

Shared class caches created in the real-time mode can only be used when running an application in real-time mode. Shared class caches created in the non-real-time mode can only be used when running an application in non-real-time mode. This also applies to the admincache tool. To manage caches created by the JVM in real-time mode, use admincache with the **`-Xrealtime`** option. To manage caches created by the JVM in non-real-time mode, do not use the **`-Xrealtime`** option. To connect to a shared class cache at run time, add the **`-Xshareclasses`** option to the command-line.

Multiple shared class caches can be created on a workstation, each with a specific name and in a specific directory. When a new cache is created, the name of the cache can be specified by the **`-cacheName`** <*name*> option. The cache name must not exceed 53 characters.

By default, shared class caches are created in the `/tmp/javasharedresources` directory, but this location can be overridden by specifying the **`-cacheDir`** <*directory*> option. The internal format for a shared class caches depends on the characteristics of the workstation where it is created. This means shared class caches cannot be created on networked drives as a safety measure. Another reason for this restriction is the slower and unpredictable performance effects when accessing a shared class cache from a networked file system.

If no cache name is specified on the command line, the default is **`sharedcc_<user_login>`**

For more information about shared cache operation in the non-real-time mode, see "Class data sharing between JVMs for non-Real-Time mode" on page 93.

Note: From IBM WebSphere Real Time for RT Linux V2 SR1 and later, you must use the **`-classpath`** option with the **`-populate`** option.

Creating a Real-Time Shared Class Cache:

The admincache tool is used to create shared class caches that are accessible in real-time mode.

Note: You must be aware of the security considerations when creating shared class cache files with default settings. See "Security considerations for the shared class cache" on page 95 for more information about security considerations for the shared class cache and information about changing the default permissions.

The admincache tool **-populate** option is used to create shared class caches. The option is used in combination with a list of .jar files, or a directory, or a directory tree to search for .jar files. For each .jar file specified or found, admincache stores each class in the .jar file in the shared class cache. The class methods are also AOT compiled and stored in the shared class cache, unless you specify the **-noaot** option.

You must use the **-classpath** option with **-populate** or you will see the following error message:

```
-populate action requires -classpath <class path> option to be specified
```

The admincache **-help** option lists the suboptions that can be used to control how admincache populates the cache.

```
$ admincache -Xrealtime -help
Usage: admincache [option]*
  where [option] can be:
     -help | -?              Action: show this help
     -Xrealtime              use in real time environment
     -cacheName <name>       specify name of shared cache (Use %%u to substitute username)
     -cacheDir <dir>         set the location of the JVM cache files
     -listAllCaches          Action: list all existing shared class caches
     -printStats             Action: print cache statistics
     -printAllStats          Action: print more verbose cache statistics
     -destroy                Action: destroy the named (or default) cache
     -destroyAll             Action: destroy all caches
     -populate               Action: Create a new cache and populate it
       -searchPath <path>    specify the directory in which to find files if no files specified
                             (default is .)
                             only one -searchPath option can be specified
       -classpath <class path> specify the classpath that will be used at runtime to access this cache
                             the -classpath option is required
       -[no]recurse          [do not] recurse into subdirectories to find files to convert
                             (default do not recurse)
       -[no]grow             if specified cache exists already, [do not] add to it (default no grow)
                             if -grow is not selected, specified cache will be removed if present
       -verbose              print out progress messages for each jar
       -noisy                print out progress messages for each class in each jar
       -quiet                suppress all output
       -[no]aot              also perform AOT compilation on methods after storing classes into cache
       -aotFilter <signature>  only matching methods will be AOT compiled and stored into cache
          e.g. -aotFilter {mypackage/myclass.mymethod(I)I} compiles only mymethod(I)I
          e.g. -aotFilter {mypackage/myclass.mymethod*} compiles any mymethod
          e.g. -aotFilter {mypackage/myclass.*} compiles all methods from myclass
       -aotFilterFile <file>   only methods matching those in file will be AOT compiled and stored into
                             cache (input file must have been created by -Xjit:verbose={precompile},
                             vlog=<file>)
       -printvmargs          print VM arguments needed to access populated cache at runtime
       [jar file]*.[jar][zip]  explicit list of jar files to populate into cache
                             if no files are specified, all files.[jar][zip] in the searchPath
                             will be converted.
Exactly one action option must be specified
```

Note: When using shared class caches, the name specified by the **-cacheName** option must not exceed 53 characters.

A list of .jar files can be specified, in which case only the classes from those .jar files will be added to the shared class cache. If you do not specify a list of .jar files, use the **-searchPath <path>** option to specify a directory tree to search for .jar or

.zip files. The **-recurse** option is the default, and means that the directory tree is searched recursively for .jar or .zip files. The **-norecurse** options means that only the specified directory is searched. Specify the **-classpath <class path>** option so that admincache can find all the classes needed to process the specified .jar files. The classes are loaded into the JVM as part of populating the shared class cache, so it is important that all referenced classes and superclasses can be found by admincache when it tries to load a class from a .jar file.

The **-grow** option specifies that a new .jar file is added to the existing cache contents, if there is an existing shared class cache of the same name in the cache directory. The **-nogrow** option specifies that a new .jar file replaces the old cache contents, if there is an existing shared class cache of the same name in the old cache directory. The **-grow** option is used to add new .jar files that do not currently exist in the shared class cache, not to replace classes that have changed. Do not use the **-grow** option to update classes that are already in the cache but have changed because of application modifications. To update existing classes, create a completely new cache with the current class contents. If you do not update your shared class cache when you change a class, your application will run properly with the new class contents, but will not be taking advantage of the shared class cache. This is because the changed class will be loaded from disk rather than from the shared class cache. Loading the class from disk means that AOT compiled code cannot be used for that class. Regenerate your shared class cache when you change a class.

Use the **-quiet**, **-verbose**, and **-noisy** options to control the level of detail provided by admincache.

To specify Ahead-Of-Time (AOT) precompilation for the methods in the classes populating the shared class cache, use the **-aot** option. To prevent AOT precompilation and only store classes into the shared class cache, use the **-noaot** option. The **-aot** option is the default setting.

To precompile some methods selectively, use the **-aotFilter** *<signature>* or **-aotFilterFile** *<file>* options. The *<signature>* is a simplified regular expression for a method signature, enclosed in curly braces, where '*' can replace any sequence of characters. You might need to enclose *<signature>* in single quotation marks so that the shell does not interpret any of the characters in the method signature.

Table 5 shows some examples of the *<signature>* option.

Table 5. Examples of the <signature> option

Signature	Meaning
`-aotFilter '{java/lang/*}'`	AOT compiles methods in the java/lang package.
`-aotFilter '{*.sample*}'`	AOT compiles methods beginning with "sample".
`-aotFilter '{mypackage/ myclass.mymethod(I)I}'`	AOT compiles the method with this exact signature.

The **-aotFilterFile** *<file>* option uses the contents of *<file>* to select the methods for AOT compilation. No other methods are AOT compiled. The contents of *<file>* are generated during an earlier run of the application using the

-Xjit:verbose={precompile},vlog=<*file*> option. The verbose output stored in <*file*> uses an internal format. This format is required by the **-aotFilterFile** option.

Note: The **-vlog=**<*file*> option does not directly generate a file called "file". A date and process ID string is appended to "file" when the verbose output is generated. By specifying the option **-Xjit:verbose={precompile},vlog=my_file**, the generated file name is similar to `my_file.<date>.<#>.<process id>` The extra fields make it easier to generate individual verbose log files in multiple-JVM scenarios where it can be difficult to supply command-line options to one particular JVM or to use different **-Xjit** command-line options with different JVMs. In a single JVM scenario, these numbers are appended to the file name supplied on the command-line.

A generated file can be used with the **-aotFilterFile** option, without requiring any editing. Multiple verbose log files generated by several application runs using the **-Xjit:verbose={precompile},vlog=<file>** option can be concatenated and supplied to admincache using the **-aotFilterFile** option.

The **-printvmargs** option helps ensure that the correct arguments are supplied on the command line when the application runs.

```
$ admincache -Xrealtime -classpath myapp.jar -cacheDir myCacheDir -cacheName
myCache -populate myapp.jar -printvmargs

admincache 1.02
Converting files
Processing classes in /team/triage/180724/bin/myapp.jar into shared class cache
No errors while processing jar file /team/triage/180724/bin/myapp.jar

Processing complete

VM args needed at runtime: -Xshareclasses:name=myCache,cacheDir=/tmp/peter
-classpath myapp.jar -Xaot
```

In this example, the final line of output shows the options that should be added to the command line when running the application, so that the classes and AOT methods stored in the shared class cache are used. To use the options from this example, enter the command:

```
java -Xshareclasses:name=myCache,cacheDir=myCacheDir -classpath myapp.jar -Xaot
myMainClass <application arguments>
```

Managing Shared Class Caches with admincache:

The admincache tool includes several utilities to manage the shared class caches on your system.

The admincache tool provides utilities to help with several activities.
- Listing the shared class caches present in a cache.
- Providing details about the contents of a shared class cache.
- Removing some or all of the caches in a specific cache directory.

Listing Available Shared Class Caches:

The admincache tool provides a list of shared class caches present in a cache.

To obtain a list of all shared class caches present in a cache, use the **-listAllCaches** option and specify the cache directory using the **-cacheDir** option.

```
$ admincache -Xrealtime -listAllCaches

admincache 1.02

Listing all caches in cacheDir /tmp/javasharedresources/

Cache name             level         persistent  last detach time

Compatible shared caches
sharedcc_username      Java6 32-bit  yes         Thu Oct 16 17:02:39 2008
rtCache                Java6 32-bit  yes         Thu Oct 16 17:03:12 2008

Incompatible shared caches
nonrtCache             Java6 32-bit  yes         Thu Oct 16 17:17:32 2008
```

In this example, there are two compatible shared class caches in the default cache directory:

- The default named cache for a user with the login *username*
- Another cache called *rtCache*

The example also shows an incompatible cache called *nonrtCache*. The *nonrtCache* was created by the JVM while executing in the non-real-time mode. This means it cannot be accessed using the **-Xrealtime** option.

The real-time mode JVM can see caches created in non-real-time mode. The non-real-time mode JVM cannot see caches created in real-time mode.

```
$ admincache -listAllCaches
J9 Java(TM) admincache 1.0
Licensed Materials - Property of IBM

(c) Copyright IBM Corp. 1991, 2008  All Rights Reserved
IBM is a registered trademark of IBM Corp.
Java and all Java-based marks and logos are trademarks or registered
trademarks of Oracle Corporation


Listing all caches in cacheDir /tmp/javasharedresources/

Cache name             level         persistent  last detach time

Compatible shared caches
nonrtCache             Java6 32-bit  yes         Thu Oct 16 17:17:32 2008
```

In this example, *nonrtCache* is listed, and it is shown as compatible because **-Xrealtime** is not specified.

Inspecting Contents of Shared Class Caches:

The admincache tool describes the contents of a shared class cache.

You can use the admincache tool **-printStats** option to obtain an overview describing the main contents of a shared class cache. For information about a specific cache, in a specific cache directory, use the **-cacheName** and **-cacheDir** options. The following example gives information about the *nonrtCache* cache in the default cache directory.

```
$ admincache -cacheName nonrtCache -printStats

admincache 1.02

Current statistics for cache "nonrtCache":
```

```
base address       = 0xD5445000
end address        = 0xD6437000
allocation pointer = 0xD5529FA8

cache size         = 16776852
free bytes         = 14070360
ROMClass bytes     = 1166004
AOT bytes          = 1437412
Data bytes         = 57440
Metadata bytes     = 45636
Metadata % used    = 1%

# ROMClasses       = 372
# AOT Methods      = 981
# Classpaths       = 1
# URLs             = 0
# Tokens           = 0
# Stale classes    = 0
% Stale classes    = 0%

Cache is 16% full
```

Note: When using shared class caches, the name of the cache must not exceed 53 characters.

There are several pieces of useful information about this cache:
- The size of the cache, shown as `cache size = 16776852`.
- The space available in the cache, shown as `free bytes = 14070360`. You can calculate that the cache is approximately 16% full.
- The number of classes stored in the cache, shown as `# ROMClasses = 372`.
- The number of AOT methods stored in the cache, shown as `# AOT Methods = 981`.

For more details about the information provided by the **`-printStats`** option in the admincache tool, see printStats utility.

The **`-printAllStats`** option provides a more detailed description of the contents of a shared class cache. The information includes the list of classes and AOT methods stored in the cache. Output from the **`-printAllStats`** option is verbose.

Classes contained in the cache are indicated by lines similar to:

```
1: 0xD643B788 ROMCLASS: java/lang/ClassLoader at 0xD5469B88.
```

This line indicates that the class java/lang/ClassLoader is contained in the cache. The addresses are internal to the shared class cache, and are rarely useful except for diagnostic purposes.

AOT methods contained in the cache are indicated by lines similar to:

```
1: 0xD643B290 AOT: callerClassLoader
        for ROMClass java/lang/ClassLoader at 0xD5469B88.
```

These lines indicate that the callerClassLoader method from the java/lang/ClassLoader class is contained in the cache. The addresses listed are internal shared cache addresses. Output from the **`-printAllStats`** option does not show the signature for each AOT method in the cache, where the signature consists of the parameter types and return type.

For more details about the information provided by the **-printAllStats** option in the admincache tool, see printAllStats utility.

Destroying Shared Class Caches:

The admincache tool has options to erase a particular cache or all caches in a specified cache directory.

The admincache tool **-destroy** option is used to erase a particular cache in a specific cache directory, if the user has permission to do so. The **-destroyAll** option is used to erase all the caches, if the user has permission to do so. For example:

```
$ admincache -Xrealtime -destroy

admincache 1.02

JVMSHRC256I Persistent shared cache "sharedcc_username" has been destroyed
```

After erasing the cache, a listing of the available shared class caches in the default cache directory shows that the erased cache no longer appears:

```
$ admincache -Xrealtime -listAllCaches

admincache 1.02

Listing all caches in cacheDir /tmp/javasharedresources/

Cache name              level         persistent  last detach time

Compatible shared caches
rtCache                 Java6 32-bit  yes         Thu Oct 16 17:03:12 2008

Incompatible shared caches
nonrtCache              Java6 32-bit  yes         Thu Oct 16 17:17:32 2008
```

The **-destroyAll** option removes all caches in the specified cache directory, regardless of whether they are compatible or not with the current JVM. The **-destroyAll** option must be used with great care:

```
$ admincache -Xrealtime -destroyAll

admincache 1.02

Attempting to destroy all caches in cacheDir /tmp/javasharedresources/

JVMSHRC256I Persistent shared cache "rtCache" has been destroyed
JVMSHRC256I Persistent shared cache "nonrtCache" has been destroyed
```

The result is that there are no longer any shared class caches available on the machine:

```
$ admincache -Xrealtime -listAllCaches

admincache 1.02

JVMSHRC005I No shared class caches available
```

If the current user does not have permission to access a cache, then the cache is not be destroyed by either the **-destroy** or **-destroyAll** options.

Practical Sizes for Shared Class Caches:

The admincache tool provides information for sizing shared class caches.

For smaller applications, a shared class cache can be populated with all classes and methods without producing a prohibitively large cache size. For larger applications, the size of the resulting shared class cache might become too large for practical purposes. This is because a JVM process must have sufficient virtual address space to address the entire contents of the shared class cache. There are some considerations you can apply when using the shared class cache technology.

The shared class cache must be virtually addressable in its entirety, in any JVM connecting to it. This means avoiding using shared class caches larger than 700 MB. The admincache tool can predict the size of a cache. If the tool indicates that the cache will be larger than the 700 MB limit, a message is displayed advising that you store a smaller number of classes, or are more selective about the AOT methods stored in the cache.

```
$ admincache -Xrealtime -populate veryBigJar.jar -cp <my class path>

admincache 1.02

WARNING: predicted cache size (15960MB) exceeds recommended maximum shared class cache size of 700MB
If your jar files contain primarily class files then you may not be able to create a cache of this size
        or you may not be able to connect to the created cache when you run your application.
Alternatively, you may want to more selectively compile AOT methods by using -aotFilterFile
To override this warning message, please directly specify -Xscmx15960M on your command-line
        but beware that the resulting failure may not occur until the very end of the population
        procedure.
```

The admincache tool predicts a conservative cache size, based upon the total size of the .jar files specified or found for population. This means that the prediction might not be accurate if the .jar file contains many files that are not class files. To get a more accurate prediction of the cache size, create temporary versions of the .jar files that contain only the class files. If the admincache tool still produces a warning message, you might consider AOT precompiling the methods in the .jar file more selectively, by using the **-aotFilter** <*pattern*> or **-aotFilterFile** <*file*> options. The admincache tool message reminds you that the prediction does not take into account AOT methods filtered out by these options.

To override the warning message and proceed to the cache population step, add the indicated **-Xscmx** option to the admincache command line. If the predicted size is very large, the admincache tool might not be able to create a shared class cache of the required size. To resolve this, reduce the cache size until admincache tool is able to proceed.

When the final cache is written to disk, it is only as big as is needed to hold the classes and AOT methods specified. This means that specifying a large initial cache size is not a problem.

Storing SDK classes in a Shared Class Cache:

Creating a cache containing all the jar files from the SDK might not be necessary for all applications.

The number and size of the jar files in the SDK means that trying to create a cache containing all of these jar files results in a warning message advising that the resulting cache will be too large. For many applications, most of the SDK jar files are never be referenced.

The main SDK jar files are located in the `SDK/jre/lib` directory. For most applications, the most important of these jar files is `rt.jar`, which is new in Java 6 releases. `rt.jar` is a collection of classes previously stored in separate jar files before the Java 6 release. Populating a shared class cache with `rt.jar` alone, and compiling all of its methods with the AOT compiler, creates a cache approximately 300 MB in size. Most of the methods from the `rt.jar` classes will not be referenced by a typical application. To populate a shared class cache with `rt.jar`:

1. Populate the classes only from `rt.jar` into the shared class cache. This consumes approximately 50 MB of space in the cache.
2. Use the **`-aotFilterFile`** *<file>* option to compile only the methods your program might use. You can generate the *<file>* by running the application.

There are other commonly-used and important jar files in the SDK, including:

- `sdk/jre/lib/i386/realtime/jclSC160/realtime.jar`
- `sdk/jre/lib/i386/realtime/jclSC160/vm.jar`
- `sdk/jre/lib/java.util.jar`

`realtime.jar` contains the IBM implementation of the Real Time Specification for Java (RTSJ). If your application uses any of the features of the RTSJ, store the `realtime.jar` file in the shared class cache for better deterministic performance. `vm.jar` contains several internal JVM classes, commonly used in all applications. `java.util.jar` contains several container classes, and must be stored into every application's shared class cache for better deterministic performance.

Other jar files in `sdk/jre/lib` and `sdk/jre/lib/ext` directories can be stored into a shared class cache if an application uses those classes. The easiest way to identify if your application uses these classes is to use the **`-verbose:dynload`** option when running your program. The **`-verbose:dynload`** option describes only the classes loaded by the current run of the application. For example:

```
<Loaded java/io/InputStreamReader from /myjdk/sdk/jre/lib/rt.jar>
<  Class size 2126; ROM size 2280; debug size 0>
<  Read time 54 usec; Load time 47 usec; Translate time 86 usec>
<Loaded java/util/LinkedHashSet from /myjdk/sdk/jre/lib/java.util.jar>
<  Class size 1218; ROM size 1136; debug size 0>
<  Read time 48 usec; Load time 31 usec; Translate time 55 usec>
<Loaded java/util/HashSet from /myjdk/sdk/jre/lib/java.util.jar>
<  Class size 3171; ROM size 2664; debug size 0>
<  Read time 71 usec; Load time 70 usec; Translate time 118 usec>
```

This example output shows three classes loaded from two different SDK jar files. The java/io/InputStreamReader class was loaded from `rt.jar`. The java/util/LinkedHashSet and java/util/HashSet classes were loaded from `java.util.jar`.

Other admincache considerations:

Useful information for working with admincache.

Cache Population and Immortal Memory Sizing

When the **`admincache`** tool populates a shared class cache in real-time mode, it must load each class as it goes through the population process. Each class consumes some immortal memory, thus it is possible that the default immortal memory size will not be large enough for all the classes requested. If the **`admincache`** tool throws an OutOfMemory error while populating a cache with many classes, try increasing the immortal memory size beyond the default 16 MB, using the **`-Xgc:immortalMemorySize=32M`** option.

When you Change Classes

If a class file is changed on disk, the shared class cache technology automatically detects that the cached version of that class in a shared class cache must not be used. Your program will operate correctly, but cannot take full advantage of the shared class cache, and any AOT methods for that class will not be used. If you change a class in your application, re-create your shared class cache. Do not try to use the **`-grow`** option to repopulate only the .jar file containing the modified class, because this option is not designed for the scenario where the .jar file already exists in the cache.

Managing shared caches

Shared caches require address space even if there are no files loaded. See "How the IBM JVM manages memory" on page 105 for more information on how shared class caches consume memory in the JVM process.

Storing precompiled jar files into a shared class cache

You can store all, some, or include Java classes provided by IBM into a shared class cache. This process uses the **`-Xrealtime`** option with **`javac`** and the **`admincache`** tool to store the classes into a shared class cache.

Before you begin

Jar files stored ahead of time into a shared class cache is supported only with the **`-Xrealtime`** option and when running java with the **`-Xrealtime`** option. You can use the same jar files when running with or without **`-Xrealtime`**, but the jar files stored into the cache can only be used when **`-Xrealtime`** is specified.

Note: When using shared class caches, the cache name must not exceed 53 characters.

About this task

You can store jar files into a shared class cache using the **`admincache`** tool. **`admincache`** enables you to build your application in one of three ways.

Note:
- If you have set a timeout on your Linux system, you might need to override it when precompiling large jar files; otherwise, compilation will time out and the jar file is not created.

Precompiling all classes and methods in an application:

This procedure precompiles all the classes in an application. It stores a set of jar files into a shared class cache. All methods in all classes in those jar files are stored into the cache. The optimized jar files have all methods compiled.

About this task

For the purposes of this example, the application resides under the directory specified by the environment variable *$APP_HOME* and the jar files are in the subdirectory *$APP_HOME*`/lib`. The application also uses some classes from those provided by IBM in `core.jarrt.jar`. In this case, you can precompile only the application code, namely `main.jar` and `util.jar`.

By default, the shared class cache is in `/tmp/javasharedresources`. Use the **`-cacheDir`** option to put the cache into a different directory. You cannot create a cache on a networked file system.

Procedure

1. From a shell prompt, enter: `cd` *`$APP_HOME`*

   where *$APP_HOME* is the directory of your application.
2. From a shell prompt, enter: `cd` *`$APP_HOME`*`/lib`. *`$APP_HOME`*`/lib` is the directory where `main.jar` and `util.jar` are stored.
3. From a shell prompt, enter: `admincache -Xrealtime -populate -aot -classpath` *`$APP_HOME`*`/lib -searchPath` *`$APP_HOME`*`/lib -norecurse .` This procedure optimizes each of the jar files found in *`$APP_HOME`*`/lib`, writing out progress information to the screen, and then creating the new jar file in the *`$APP_HOME`*`/aot` directory. You can specify a cache name with **`-cacheName <name>`**, but a default name based on the user's login is used if none is specified.

   Note: The name specified by the **`-cacheName`** option must not exceed 53 characters.
4. From a shell prompt, entering: `admincache -Xrealtime -listAllCaches` shows the existence of the cache.

What to do next

For more options, specify: `admincache -Xrealtime -help`.

Precompiling frequently used methods:

You can use profile-directed AOT compilation to precompile only the methods that are frequently used by the application. AOT compilation stores a set of jar files into a shared class cache using an option file generated by running the application with a special option **`-Xjit:verbose={precompile},vlog=optFile`**. Only the methods listed in the option file are precompiled.

Before you begin

Before you start, create a list of those methods that are typically compiled by a JIT compiler.

About this task

You can edit the file generated by the **`-Xjit:verbose={precompile}`** option. The file is an explicit specification of the methods that are to be precompiled. These methods are specific; that is, they contain the full signature for each method to be compiled, which lets you compile `com/acme/sample.myMethod(J)V` but not `com/acme/sample.myMethod(I)V`.

Note: When using shared class caches, the name of the cache must not exceed 53 characters.

Procedure

1. From a shell prompt, enter:
   ```
   cd $APP_HOME
   ```
   where *$APP_HOME* is the directory of your application.

2. From a shell prompt, enter:

```
java -Xjit:verbose={precompile},vlog=$APP_HOME/app.precompileOpts \
     -cp $APP_HOME/lib/demo.jar applicationName
```

   where:

   - *app.precompileOpts* is the name of the log file that lists the methods compiled with JIT.
   - *applicationName* is the name of your application.

   This command creates a list of the methods that are compiled using JIT.
3. From a shell prompt, enter:

   ```
   cd $APP_HOME/lib
   ```

   $APP_HOME/lib is the directory where the jar files for your application are stored.
4. To compile all the sample application methods into the cache, enter:

   ```
   admincache -Xrealtime -populate -cacheName myCache \
              -aotFilterFile $APP_HOME/app.precompileOpts \
              -cp $APP_HOME/lib/demo.jar
   ```
5. To compile `realtime.jar` and `vm.jar` into the cache, enter:

```
admincache -Xrealtime -populate -grow -cacheName myCache \
          -aotFilterFile $APP_HOME/app.precompileOpts \
          -searchPath $JAVA_HOME/jre/bin/realtime/jclSC160
          -cp $APP_HOME/lib/demo.jar
```

6. To compile `rt.jar` into the cache, enter:

```
admincache -Xrealtime -populate -grow -cacheName myCache \
          -aotFilterFile $APP_HOME/app.precompileOpts \
          $JAVA_HOME/jre/lib/rt.jar
          -cp $APP_HOME/lib/demo.jar
```

7. To test this command run your application with the **-nojit** option, which uses the code in the cache. From the shell prompt, enter:

```
java -Xrealtime -Xshareclasses:name=myCache -Xnojit \
     -cp $APPHOME/aot/demo.jar applicationName
```

   where *applicationName* is the name of your application.

Precompiling files provided by IBM:

You can precompile files provided by IBM, for example `rt.jar`, to achieve a compromise between performance and predictability.

About this task

The precompilation is similar to the task of precompiling your application jars but an additional requirement applies at run time; you must ensure that your boot class path is specified correctly to use these files instead of the files in the JRE. You can do this with the **-Xshareclasses** option, which instructs the JVM to look first in the specified class cache ahead of the default class path locations.

Note: When using shared class caches, the name of the cache must not exceed 53 characters.

Precompile `rt.jar` for use with the application:

Procedure

1. From a shell prompt, enter: `cd $JAVA_HOME/lib` where *$JAVA_HOME* is your Java home directory.

2. Run the **admincache** tool. At a shell prompt, enter:

   ```
   admincache -Xrealtime -populate -cacheName myCache
       -classpath <class path> rt.jar
   ```

   This command populates the cache called `myCache` with the results of precompiling the IBM-provided file called `rt.jar`.
3. Run your application specifying the **-Xshareclasses** option to specify the cache name. To run your application, enter:

   ```
   java -Xrealtime -Xnojit -Xshareclasses:name=myCache
           -classpath:$APP_HOME/main.jar:$APP_HOME/util.jar ...
   ```

The Just-In-Time (JIT) compiler

You can control when and how the JIT compiler operates using the java.lang.Compiler class that is provided as part of the standard SDK class library. IBM fully supports the Compile.compileClass(), Compiler.enable() and Compiler.disable() methods.

For example, if you want to warm up your application and know that the key methods in your application have been compiled, you can call the Compiler.disable() method after you have warmed up your application and be confident that JIT compilation will not occur during the remainder of the execution of your application.

You can control method compilation in two ways:
- Specify a set of methods that you can compile:

  ```
  Compiler.command("{<method specification>}(compile)");
  ```

  where *<method specification>* is a list of all the methods that have been loaded at this point and are to be compiled. *<method specification>* describes a fully qualified method name. An asterisk denotes a wildcard match.

  For example, to compile all methods starting with java.lang.String that were already loaded, you specify:

  ```
  Compiler.command("{java.lang.String*}(compile)");
  ```

  Note: This command compiles not only methods in the java.lang.String class, but also in the java.lang.StringBuffer class, which might not be what you wanted. To compile only methods in the java.lang.String class, you specify:

  ```
  Compiler.command("{java.lang.String.*}(compile)");
  ```
- Specify that all methods in the compilation queue will be compiled before this thread runs and continues:

  ```
  Compiler.command("waitOnCompilationQueue");
  ```

  You might want to ensure that the compilation queue was empty before disabling the compiler. A typical technique for compiling a set of methods and classes might be:

```
Compiler.enable();                           // ensure compiler is active
Compiler.command("{com.mycompany.*}(compile)"); // queue up all the methods you want to compile
Compiler.command("waitOnCompilationQueue");     // wait until all those methods are compiled
Compiler.disable();                          // turn the compiler off
```

Determinism during JNI transitions

By default, the JIT generates optimized code for high performance Java-to-native (J2N) JNI transitions. Reduced determinism might possibly occur when reloading a native library using the following code sequence:

```
RegisterNatives / UnregisterNatives / RegisterNatives
```

To revert to the slower, more deterministic code, use the command line option **`-Xjit:disableDirectToJNI`**.

Enabling the JIT

You can explicitly enable the JIT in several ways. Both command-line options override the **`JAVA_COMPILER`** environment variable.

Procedure

- Set the **`JAVA_COMPILER`** environment variable to "jitc" before running the Java application. At a shell prompt, enter:
  - **For the Korn shell:** `export JAVA_COMPILER=jitc`

    Note: Korn shell commands are used in this information unless otherwise stated.
  - **For the Bourne shell:**
    ```
    JAVA_COMPILER=jitc
    export JAVA_COMPILER
    ```
  - **For the C shell:** `setenv JAVA_COMPILER jitc`

  If the **`JAVA_COMPILER`** environment variable is an empty string, the JIT remains disabled. To disable the environment variable, at a shell prompt, enter `unset` **`JAVA_COMPILER`**.
- Use the **`-D`** option on the JVM command line to set the java.compiler property to "jitc". At a shell prompt, enter: `java -Djava.compiler=jitc` *<MyApp>*
- Use the **`-Xjit`** option on the JVM command line. You must *not* specify the **`-Xint`** option at the same time. At a shell prompt, enter: `java -Xjit` *<MyApp>*

Disabling the JIT

You can disable the JIT in several ways. Both command-line options override the **`JAVA_COMPILER`** environment variable.

About this task

Procedure

- Set the **`JAVA_COMPILER`** environment variable to "NONE" or the empty string before running the Java application. At a shell prompt enter:
  - **For the Korn shell:** `export JAVA_COMPILER=NONE`

    Note: Korn shell commands are used for the remainder of this information.
  - **For the Bourne shell:**
    ```
    JAVA_COMPILER=NONE
    export JAVA_COMPILER
    ```
  - **For the C shell:** `setenv JAVA_COMPILER NONE`
- Use the **`-D`** option on the JVM command line to set the java.compiler property to "NONE" or the empty string. At a shell prompt, enter: `java -Djava.compiler=NONE` *<MyApp>*
- Use the **`-Xint`** option on the JVM command line. At a shell prompt, enter: `java -Xint` *<MyApp>*

Determining whether the JIT is enabled

You can determine the status of the JIT using the **`-version`** option.

Procedure

Enter the following command at a shell prompt:

```
java -version
```

If the JIT is not in use, a message is displayed that includes the following text: `(JIT disabled)`
If the JIT is in use, a message is displayed that includes the following text: `(JIT enabled)`

Using no-heap real-time threads

Metronome garbage collection provides more consistent response times, but sometimes it is appropriate to completely avoid interruptions from garbage collection.

NoHeapRealtimeThreads (NHRT) are an extension to RealtimeThreads. They differ from RealtimeThreads in that they do not have access to the heap memory. Without access to the heap, NHRTs can continue to run even during a garbage collection cycle, with some restrictions. It follows that without access to the heap the programming model is different from the one for real-time threads.

Considerations when using NHRTs

Consider these points about NHRTs:

- The main reason for using NHRTs is with a task that cannot tolerate garbage collection. For example, if your application is time-critical and cannot tolerate any interruptions.
- If time is so critical that you are using NHRTs, also consider using the ahead-of time (AOT) compiler; that is, use the **`-Xnojit`** option.
- When you use the **`-Xrealtime`** option, you automatically use the Metronome Garbage Collector. The benefits of Metronome Garbage Collector might be sufficient for your enterprise, thus reducing the need to code NHRTs.
- NHRT threads run independently of the Garbage Collector because they have a priority higher than the priority of the Garbage Collector. Java threads can have a priority in the range 1 - 10. If NHRTs are present, the priority of Java threads is reset to 0 regardless of the priority set in your program. The Garbage Collector is automatically set to half a step higher than the highest real-time thread. You set the priority of your NHRTs to be at least one higher than the highest real-time thread. In this way, the NHRTs are independent of the garbage collector.

  Note: NHRTs are not entirely free of garbage collection because the Metronome alarm thread garbage collector runs at the highest priority in the system. This priority ensures that the JVM can be activated to check if the Garbage Collector must do anything. The work to run the Metronome alarm thread is small and does not affect performance significantly. On a multi-processor system, the alarm thread can run simultaneously with NHRT threads and thus no garbage collection interruptions occur.
- Because NHRTs are restricted to the scoped and immortal memory areas, Java methods perform checks to ensure that they are not allocated from the heap. The start method checks and returns an exception (`MemoryAccessError`) if NHRTs are allocated from the heap. NHRTs can access only `ImmortalMemory` and `ScopedMemory`.

- The semantics of locking are unchanged, so that NHRT threads can be blocked by normal threads if a lock is shared.
- A thread that is using the heap can have its priority boosted on a synchronized method when an NHRT tries to use the same method.
- Use nonblocking queues for communications between NHRTs and heap threads. Otherwise, separate the two types of threads.

Exceptions

These exceptions can occur when using NHRTs:
- `IllegalAssignmentError`. As an example, this error can occur when an attempt is made to create a reference to scoped memory in immortal memory.
- `MemoryAccessError`. As an example, this error can occur when an NHRT tries to reference heap memory.

Asynchronous event handling constraints

There are multiple cases where NHRTs can be blocked during garbage collection, including:
1. When an NHRT calls either fire(), setHandler(), or addHandler() on an AsyncEvent that is already associated with handlers that are allocated from the Heap memory
2. When an NHRT calls either destroy(), start(), or stop() on a Timer which is associated with handlers that are allocated from the Heap memory
3. An NHRT is the last thread exiting a scope and is shutting down Timers or AsyncEvents from the scope. However, the Timers or AsyncEvents have associated handlers that are allocated from the Heap memory

To avoid these situations with NHRTs:
1. Avoid adding handlers allocated from the Heap to AsyncEvents or Timers that might be fired by an NHRT.
2. Avoid conditions where an NHRT exits last from a scope that has AsyncEvents or Timers that have handlers allocated from the Heap memory.

Memory and scheduling constraints

The JVM prevents no-heap real-time threads from loading references to objects that are on the heap onto its operand stack. To do so throws a javax.realtime.MemoryAccessError.

The JVM also guards against references to objects in scoped memory being stored in heap or immortal memory. Although scoped memory is not used exclusively by NHRTs, it is likely to be used if immortal memory is inappropriate and memory deallocation is required in an NHRT context.

While an NHRT is executing, if it populates a field with a reference to an object, it can successfully overwrite any pre-existing reference to an object on the heap in that field. The pre-existing reference will be successfully overwritten by the NHRT without generating a MemoryAccessError.

Class loading constraints

Classes are loaded into the same memory areas as the class loader. The default area for class loaders is immortal memory.

In order for applications to provide the expected response times, they must be "warm". Applications should load their classes early so that class loading does not interrupt real-time threads and asynchronous event handlers later.

Constraints on Java threads when running with NHRTs

Because system properties are shared in a JVM and any thread can access the system properties, some care is required when using the getProperties and setProperties methods in JVMs in which NHRTs are run. For system properties to be accessible to NHRTs, they must be in immortal memory.

The `java.lang.System` class provides several methods that allow threads to interact with the system properties; including these methods:

```
String getProperty(String)
String getProperty(String,String)
Properties getProperties()

String setProperty(String,String)
void setProperties(Properties)
```

The real-time JVM uses an instance of the com.ibm.realtime.ImmortalProperties class that was created specifically for the real-time JVM object to store all system properties. Use of this instance ensures that any calls to the System.setProperty() or System.getProperties.setProperty() methods result in the property being stored in immortal memory. No special user code is required in this case but it is important to understand that each time a property is set some immortal memory is consumed.

Calls to the setProperties() method are a little more difficult because the shared Properties object is used to store system properties. If an application runs in a real-time JVM that has NHRTs running, calls to the setProperties method must pass in an instance of a com.ibm.realtime.ImmortalProperties class, or subclass, that was created in immortal memory. Use of this instance ensures that all properties that are set by using the setProperties method are stored in immortal memory.

Note: Calling setProperties(null) results in a new ImmortalProperties object being created internally with a default set of properties, which consumes additional immortal memory.

Calls to the getProperties() method return either the object that was set or the default properties object, which is an com.ibm.realtime.ImmortalProperties object. To maximize compatibility with existing code that calls the getProperties() method, the ImmortalProperties object serializes the object and then deserializes in a standard JVM. The default behavior for the serialization of ImmortalProperties is to serialize a regular Properties object, because standard JVMs do not have the ImmortalProperties object and deserialization fails. To override this default behavior, the ImmortalProperties class provides the enabledReplacement(boolean) method, which, if called with false, disables the default behavior. In this case, serialization serializes the ImmortalPropeties object and it is then possible to deserialize this and use the resulting object in a call to the System.setProperties method in a real-time JVM.

Note: Deserialization takes place in immortal memory, which might consume too much of this limited resource.

Security manager

The security manager set for the system is used by all types of threads in the JVM. For this reason, in a real-time JVM in which NHRTs run, the security manager must be allocated in immortal memory. The real-time JVM ensures that any security manager specified in the command-line options is allocated in immortal memory. The security manager can also be set through calls to the System.setSecurityManager(SecurityManager) method. If the application sets the security manager in this way, it must ensure that the security manager was allocated from Immortal so that NHRTs are able to run correctly.

The exceptions thrown and any objects returned by the security manager must either be in immortal memory, if cached, or be allocated in the current allocation context.

Synchronization

The MonitorControl class and its subclass PriorityInheritance manage synchronization, in particular priority inversion control. These classes allow the setting of a priority inversion control policy either as the default or for specific objects.

The WaitFreeReadQueue, WaitFreeWriteQueue, and WaitFreeDequeue classes allow wait-free communication between schedulable objects (especially instances of NoHeapRealtimeThread) and regular Java threads.

The WaitFree classes provide safe, concurrent access to data shared between instances of NoHeapRealtimeThread and schedulable objects subject to garbage collection delays.

No-heap real-time class safety

In some circumstances, portions of the JSE API cannot necessarily be used in a no-heap context. Restrictions are placed on classes that are shared between heap and no-heap threads. Be aware of the classes supplied with the JVM that can be safely used.

Sharing objects

Methods that run in no-heap real-time threads throw a javax.realtime.MemoryAccessError whenever they try to load a reference to an object on a heap.

Figure 3 on page 62 is an example of the sort of code to be avoided:

```
/**
 * NHRTError1
 *
 * This example is a simple demonstration of an NHRT accessing
 * a heap object reference.
 *
 * The error generated is:
 *
 * Exception in thread "NoHeapRealtimeThread-0" javax.realtime.MemoryAccessError
 *      at NHRT.run(NHRTError1.java:56)
 *      at javax.realtime.RealtimeThread.runImpl(RealtimeThread.java:1754)

 */
import javax.realtime.*;

public class NHRTError1 {
    public static void main(String[] args) {
        NHRTError1 example = new NHRTError1();

        example.run();
    }

    public NHRTError1() {
        message = new String("This on the heap.");
    }

    static public String message; /* The NHRT can access static fields directly - they are always Immortal. */
    static public NHRT myNHRT = null;

    public void run() {
                ImmortalMemory.instance().executeInArea(new Runnable() {
            public void run() {
                NHRTError1.this.myNHRT = new NHRT();
            }
        });

        myNHRT.start();

        try {
            myNHRT.join();
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

    }
```

Figure 3. Example of an NHRT accessing a heap object reference

```
    /* A NHRT class */
    class NHRT extends NoHeapRealtimeThread {
        public NHRT() {
            super(null, ImmortalMemory.instance());
        }

        /* Prints the String via the static reference in NHRTError1.message */
        public void run() {
            System.out.println("Message: " + NHRTError1.message);
        }
    }

}
```

Figure 4. Example of an NHRT accessing a heap object reference (continued from Figure 1)

Figure 3 produces a **javax.realtime.MemoryAccessError**:

```
Exception in thread "NoHeapRealtimeThread-0" javax.realtime.MemoryAccessError
        at NHRTError1$NHRT.run(NHRTError1.java:56)
        at javax.realtime.RealtimeThread.runImpl(RealtimeThread.java:1754)
```

If an object is to be accessible to both a no-heap real-time thread and a standard Java thread, the object must be allocated in immortal memory. Similarly, if an object is to be accessible to a no-heap real-time thread and a real-time thread, the object can also be held in a scoped memory area.

In Figure 3 on page 62, the reference to the String "This on the heap." was held in a class variable. This variable is accessible to NHRTs because all classes are allocated in immortal memory. Alternatively, the String could have been passed to the NHRTs constructor.

Most objects contain references to other objects, and so care must be taken when sharing such objects between ordinary threads and NHRTs. A typical example is of a LinkedList allocated in immortal memory, shared between an ordinary thread and a NHRT. If sufficient care is not taken, the standard thread might introduce objects into the LinkedList that are on the heap. Of greater concern is that the data structures that are allocated by the LinkedList to track objects are allocated on the heap by the ordinary thread, easily causing a MemoryAccessError in the NHRT.

Some classes cannot be safely shared between NHRTs and other threads, regardless of where individual instances of them might be allocated. These are classes that rely on objects stored in class variables, usually for caching purposes. InetAddress is a typical example that caches addresses; if the first thread to call certain methods in InetAddress is running on the heap, the same methods are unsafe to be called by NHRTs in the future.

Locking on objects with NHRTs

NHRTs must avoid synchronizing with other threads. Consider the following scenario:

- A real-time thread of low priority enters a synchronized block or method, synchronizing on an object.
- An NHRT of high priority is blocked when attempting to synchronize on the same object.
- Priority inheritance causes the real-time thread to temporarily assume the same priority as the NHRT.
- Garbage collection then runs at a higher priority than the NHRT, and can therefore interrupt the NHRT. The reason for using the NHRT is to avoid interruption by garbage collection, so this scenario negates the use of the NHRT.

It is sometimes unavoidable that NHRTs and other threads synchronize on the same object, but you must minimize the possibility. Be careful to avoid unnecessary synchronization when sharing objects.

Restrictions on safe classes

Some considerations apply when an application contains both real-time thread and no-heap real-time thread objects.

- The no-heap real-time thread can suffer MemoryAccessErrors caused by interaction with the real-time thread.
- The no-heap real-time thread might be accidentally delayed by garbage collection caused by the real-time thread.

MemoryAccessErrors caused on a no-heap real-time thread

When the two types of thread both call methods on the same class, the real-time thread might “pollute” the static variables of the class with objects allocated from the heap. The no-heap real-time thread will receive a MemoryAccessError when trying to access those heap objects. The pollution can also happen on instances of the class. Unfortunately, both problems are quite likely to be seen in typical coding patterns and so it is worth exploring a couple of cases.

If a class is performing a time-consuming operation, it often chooses to cache the result to improve performance of the subsequent operations. The cache is typically a collection such as a HashMap anchored in a static variable in the class. A real-time thread operating in heap context can store a heap object in this collection, which not only adds the object itself but also adds infrastructure objects to the collection; for example, parts of the index. When a no-heap real-time thread later tries to access the collection, even if it is not trying to access the object added by the other thread, it attempts to load the infrastructure objects and hence receive a MemoryAccessError. As class libraries develop and are tuned for performance, these caches become more common.

A class instance can also become polluted by heap objects in a variety of ways. Consider an instance built in immortal memory and thus accessible to both types of thread. If the first use of the object is by a realtime thread in heap context, you might find that a secondary object is stored in a field of the original object. If the secondary object is in heap context, subsequent use by the no-heap real-time thread again shows a MemoryAccessError. These secondary objects might not always be added on first use but after a number of uses, and might be designed to improve the performance of heavily used methods.

NoHeap thread delayed by garbage collection

No-heap threads must be assigned priorities that are higher than other threads to avoid being delayed by garbage collection.

Additionally, if a class contains any synchronized methods, it is possible that a no-heap real-time thread calling such methods might unintentionally be delayed by garbage collection. This scenario is described in “Locking on objects with NHRTs” on page 63.

If a class contains any synchronized methods (either static or instance methods), it is possible that a no-heap real-time thread calling such methods might unintentionally be delayed by garbage collection. The problem is caused if a real-time thread is accessing a synchronized method (static or instance) at the point where a no-heap real-time thread attempts to call another synchronized method that will block waiting for the other thread to complete. If the no-heap real-time thread has a higher priority than the real-time thread, the priority of the real-time thread is raised. If that thread is then forced to wait for a garbage collection interrupt, a priority inversion is possible, because the garbage collector thread has a priority higher than the highest priority real-time thread, which might not be as high as the no-heap real-time thread that is currently blocked waiting to enter the synchronized method.

The only way to fix such problems is to ensure that no-heap real-time threads never call synchronized methods on classes or instances that are shared with other thread types. Unfortunately, it is not always clear from a method signature whether a method is synchronized; it might, for example, contain a synchronized block or call a synchronized method.

Summary

The NoHeapRealtimeThread class adds a significant amount of complexity to the real-time environment and a significant number of problems can be caused when a mixture of thread types operate in an environment. During development of an application, you must carefully design areas in which you have shared use of classes by the different thread types. Of particular importance is the use that these threads make of classes in the SDK. Because of the complexity of analysis, it is impossible to give any guarantee that all the classes provided in the SDK are safe for such shared use. Instead, a small subset of the classes have been verified. Initially, verification has concentrated on the MemoryAccessError aspect and the result is a list of classes that have been analyzed, tested, and modified where necessary to ensure that they can be used by both no-heap and other types of threads.

Safe classes

This section lists the set of classes that are intended to be safely used by NoHeapRealtimeThread and other thread types.

The main concern is focused on the MemoryAccessError aspect of safety. The following list details classes that can be used by all three thread types in the same JVM.

Note: Individual instances of the class might not always be safely shared.

Follow these rules to ensure that a class can be used safely by all thread types:
- The instance must be built in a memory area that is accessible to the thread intending to access the instance.
- If the class has public static fields, avoid storing heap objects in these fields.
- If the class has public instance fields, avoid storing heap objects in these fields.

Not all IBM-provided classes are NHRT-safe. The following packages contain classes that are NHRT-safe:
- java.lang package
- java.lang.reflect package
- java.lang.ref package (all classes)
- java.net package
- java.io package
- java.math package

These tables show classes within these packages that are not NHRT-safe:

Table 6. Classes in the java.lang package that are not NHRT-safe

Class	**Method**
java.lang.ProcessBuilder	*
java.lang.Thread	getAllStackTraces()Ljava.util.Map;
java.lang.ThreadGroup	*
java.lang.ThreadLocal	*
java.lang.InheritableThreadLocal	*

Table 7. Classes in the java.lang.reflect package that are not NHRT-safe

Class	**Method**
java.lang.reflect.Proxy.*	*

Table 8. Classes in the java.net package that are not NHRT-safe

Class	Method
java.net.SocketPermission.*	newPermissionCollection()Ljava.net.SocketPermissionCollection;

Table 9. Classes in the java.io package that are not NHRT-safe

Class	Method
java.io.ExpiringCache	*
java.io.SequenceInputStream	*
java.io.FilePermission	newPermissionCollection()Ljava.io.FilePermissionCollection;
java.io.ObjectInputStream	*
java.io.ObjectOutputStream	*
java.io.ObjectStreamClass	*

Table 10. Classes in the java.math package that are not NHRT-safe

Class	Method
java.math.BigInteger	*

The packages might include sub packages that contain non-safe classes. For example, the following classes are not NHRT-safe:
- java.lang.management.*
- java.lang.annotation.*
- java.lang.instrument.*

Even if a class is considered as NHRT-safe, the class might not be suitable for use in an NHRT. Application developers must determine the real-time requirements of the classes on a case-by-base basis, regardless of whether a class is NHRT-safe.

Class data sharing between JVMs

The Java Virtual Machine (JVM) allows you to share class data between JVMs by storing it in a memory-mapped cache file on disk.

Sharing reduces the overall virtual storage consumption when more than one JVM shares a cache. Sharing also reduces the startup time for a JVM after the cache has been created. The shared class cache is independent of any active JVM and persists until it is destroyed. A shared cache can contain:
- Bootstrap classes
- Application classes
- Metadata that describes the classes
- Ahead-of-time (AOT) compiled code

Shared class caches can be used by IBM WebSphere Real Time for RT Linux in both non-real-time and real-time modes, but the cache format, creation, and population techniques differ. Real-time mode caches are not compatible with non-real-time mode caches. In non-real-time mode, caches are created and populated in the same way as the standard JVM. This means the cache is created and populated by the JVM as it runs an application, transparently to the user. In real-time mode, using the **`-Xrealtime`** option, shared class caches must be created and prepopulated by **admincache**, using the **-populate** option. Applications running in real-time mode might read content from the prepopulated cache, but cannot modify its contents.

Use the **admincache** tool to create, populate, and destroy caches.

To enable an application to use a shared class cache, add the **-Xshareclasses** option to its command-line. Because real-time mode caches are read-only, some non-real-time mode suboptions of **-Xshareclasses** are not available in real-time mode.

For further information, see "Using the admincache tool" on page 43, "Class data sharing between JVMs for non-Real-Time mode" on page 93, and "Shared classes diagnostic data" on page 141.

Running Applications with a Shared Class Cache

To run an application with a shared class cache, use the **-Xshareclasses** option on the command-line.

Table 11 shows the suboptions available when running an application in real-time mode, using the **-Xshareclasses** option.

Table 11. Suboptions available when running an application in real-time mode

Option	**Meaning**
`cacheDir=<`*directory*`>`	Sets the directory in which shared class cache data is read and written. By default, <*directory*> is `/tmp/javasharedresources`. The directory name must match that specified in the **-cacheDir** option used in the admincache command to create the cache.
`name=<`*name*`>`	The name of the shared class cache to use. The name must match that specified in the **-cacheName** option used in the admincache command to create the cache. The name must not exceed 53 characters.
`none`	Explicitly disable class sharing. Can be added to the end of a command line to disable class data sharing. This suboption overrides class sharing arguments found earlier on the command line.
`nonfatal`	Always start JVM regardless of errors or warnings. Allows the JVM to start even if class data sharing fails. Typical behavior for the JVM is to refuse to start if class data sharing fails. If you select **nonfatal** and the shared classes cache fails to initialize, the JVM attempts to connect to the cache in read-only mode. If this attempt fails, the JVM starts without class data sharing.
`silent`	Suppress all output messages. Turns off all shared classes messages, including error messages. Unrecoverable error messages, which prevent the JVM from initializing, are displayed.

Table 11. Suboptions available when running an application in real-time mode (continued)

Option	Meaning
`verbose`	Enable verbose output, providing overall status on the shared class cache and more detailed error messages.
`verboseAOT`	Enables verbose output when compiled AOT code is being found in the cache, for example during AOT method load requests.
`verboseHelper`	Enables verbose output for the Java Helper API. This output shows you how the Helper API is used by your class loader.
`verboseIO`	Enable verbose output of class load requests. This option provides detailed output about the cache I/O activity, listing information about classes being found.

To ensure these options are correct, use the **-printvmargs** option with admincache (see **-printvmargs** for more information). The **nonfatal** option is not suitable for general use, because it forces the JVM to ignore warnings and errors about the shared class cache. The **none** option explicitly disables class sharing, and is equivalent to omitting the **-Xshareclasses** option on the command line.

For more detailed information about the **-Xshareclasses** suboptions, see Class data sharing command-line options.

Using the Metronome Garbage Collector

Metronome Garbage Collector replaces the standard Garbage Collector in WebSphere Real Time for RT Linux.

Controlling processor utilization

You can limit the amount of processing power available to the metronome garbage collector.

You can control garbage collection with the Metronome Garbage Collector using the **-Xgc:targetUtilization=N** option to limit the amount of CPU used by the Garbage Collector.

For example:

```
java -Xrealtime -Xgc:targetUtilization=80 yourApplication
```

The example specifies that your application runs for 80% in every 60 milliseconds. The remaining 20% of the time is used for garbage collection. The Metronome Garbage Collector guarantees utilization levels provided that it has been given sufficient resources. Garbage collection begins when the amount of free space in the heap falls below a dynamically determined threshold.

Tuning Metronome Garbage Collector

You can tune the real-time environment by controlling the amount of memory that your application uses. For example, use the **-Xmx**, **-Xgc:immortalMemorySize=size**, **-Xgc:scopedMemoryMaximumSize=size**, and the **-Xgc:targetUtilization=N** options.

- Use the **-Xmx** option to limit the size of the heap.

  The value chosen is used as the upper limit of heap size and thus reflects the likely usage over time. Choosing a value that is too low increases the garbage collection frequency and leads to a lower overall throughput although it reduces the memory footprint. For good real-time performance, avoid paging. It is normal to ensure that the footprint of all the running processes on a machine does not exceed the physical memory size.

- Use the **-Xgc:immortalMemorySize=size** option to control the size of the immortal memory area.

  You must analyze carefully the use of immortal memory. The "ideal" application uses immortal memory during startup but thereafter stops using it. If allocation of immortal objects continues, the application is able to continue to run until immortal memory has been exhausted. The current usage can be obtained by adding:

  ```
  long used = ImmortalMemory.instance().memoryConsumed();
  ```

  to your code.
- Use the **-Xgc:scopedMemoryMaximumSize=size** option to ensure that applications do not request excessive amounts of scoped memory. Use this option for diagnosis rather than tuning.
- Set the **-Xgc:targetUtilization=N** option to ensure that under the worst-case conditions (maximum allocation rate of heap objects), the garbage collector can collect garbage at a higher rate than the application generates it.

  Typically, the default value is sufficient but application performance might be improved by increasing the utilization to the point at which the collector is able to collect garbage slightly faster than the application can create it.
- Use the **-Xgcthreads <n>** option to create additional threads to run garbage collection in parallel.

  The default is to use one thread. If your workload has a high rate of garbage generation, and runs on a symmetric multiprocessor with CPU cycles available, performance could benefit by setting this parameter to >1.

  Note: Setting this parameter too high can have a negative affect on throughput.

Metronome Garbage Collector limitations

This topic captures any known issues or limitations that affect the metronome GC policy.

AESNI support on x86 platforms

Software exploitation of AESNI instructions on x86 architectures is currently not supported with the metronome GC policy.

Long pause times during garbage collection

Under rare circumstances, you might experience longer than expected pauses during garbage collection. During garbage collection, a root scanning process is used. The garbage collector walks the heap, starting at known live references. These references include:

- Live reference variables in the active thread call stacks.
- Static references.
- All object references in immortal and scoped memories.

To find all the live object references on an application thread's stack, the garbage collector scans all the stack frames in that thread's call stack. Each active thread stack is scanned in an uninterruptible step. Therefore the scan must take place within an individual GC pause.

The effect is that the system performance might be worse than expected if you have some threads with very deep stacks, because of extended garbage collection pauses at the beginning of a collection cycle.

Immortal memory is processed incrementally. All other scoped memory areas are processed in one atomic, uninterruptible step. Therefore significant use of scoped memory areas might lead to worse system performance than expected, because of extended garbage collection pauses when the root scan is processing scoped memory.

Chapter 6. Developing applications

Important information about writing real time applications, including code samples.

Writing Java applications to exploit real time

These examples describe how to exploit the real-time environment. They range from the simplest example, running a Java application in real time without any modifications to the code, through to a more complex process of planning and writing no-heap real-time threads. Reasons are provided to help you decide which approach might be most suitable for your applications.

Introduction to writing real-time applications

You do not have to write elaborate no-heap real-time applications to exploit the features of real-time technology. Some of the benefits can be used with very little change to your existing code.

For application programers, here are the steps that you can take to exploit WebSphere Real Time for RT Linux:

1. You can run a standard Java application in a real-time JVM to give you the benefit of Metronome garbage collection and achieve significant improvement in the predictability of the run time of your application.
2. Add the **`-Xnojit`** option after you have precompiled your code to use the ahead-of-time (AOT) compiler. See "Storing precompiled jar files into a shared class cache" on page 53.
3. Replace java.lang.Thread with javax.realtime.RealtimeThread in your application. You might see a slight improvement when compared with the AOT option.

   The main advantage of using real-time threads is the ability to control the priority that you give to each of the threads. Real-time threads can also be made periodic. To exploit these advantages, you must be prepared to make changes to the application itself.
4. Plan and write a specific application to use real-time threads and asynchronous event handlers to deal with timers or external events. Consider these three factors:
   - Planning the priority that you assign to your real-time threads
   - Deciding which memory areas you will use to hold objects
   - Communicating with the event handlers
5. Plan and write a specific application to use no-heap real-time threads. No-heap real-time threads are extensions of real-time threads and you have to consider the priority that you assign and the memory area. In general, take this step only if the application must handle events in times comparable to the GC pause time (sub-millisecond). Do not underestimate the complexity of developing with no-heap real-time threads.

Figure 5 shows the steps described previoulsy.

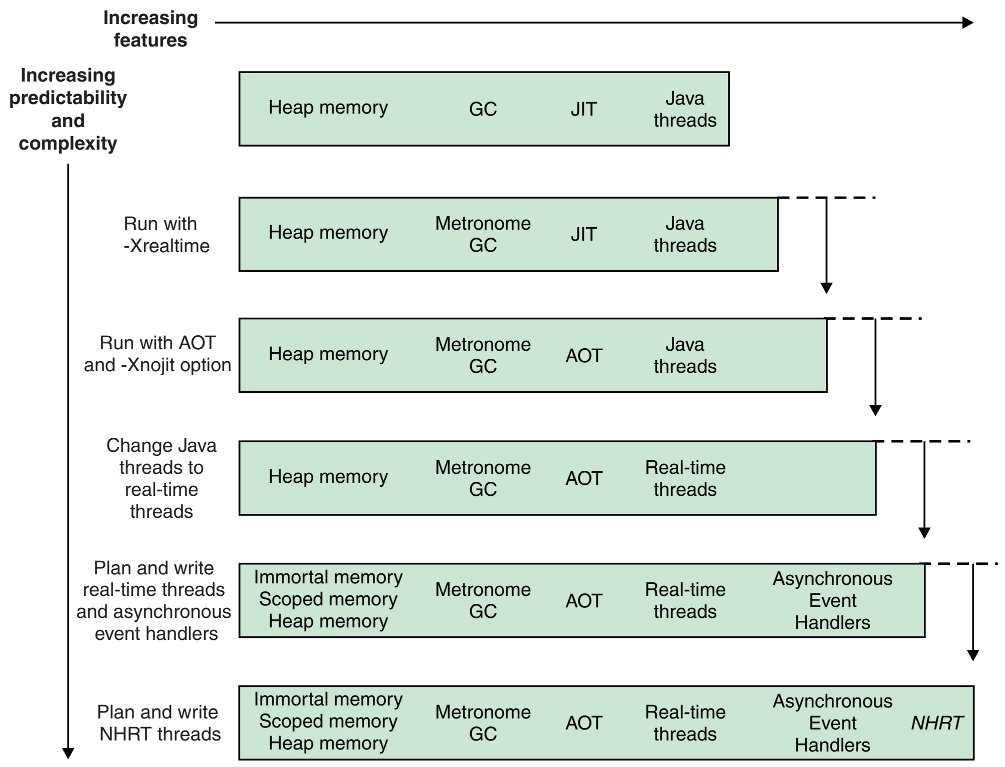

Increasing features
Increasing predictability and complexity
Heap memory
GC
JIT
Java threads
Run with -Xrealtime
Metronome GC
Run with AOT and -Xnojit option
AOT
Change Java threads to real-time threads
Real-time threads
Plan and write real-time threads and asynchronous event handlers
Immortal memory
Scoped memory
Asynchronous Event Handlers
Plan and write NHRT threads
NHRT

Figure 5. A comparison of the features of RTSJ with the increased predictability.

Planning your WebSphere Real Time for RT Linux application

When you are preparing to write real-time Java applications, you must consider whether to use Java threads, real-time threads, or no-heap real-time threads. In addition, you can decide which memory area that your threads will use.

About this task

When planning your application, these steps describe the decisions you need to make:

Procedure

1. Identify your tasks.
2. Decide the timing periods:
   - Responses greater than 10 ms, choose Java threads, just exploiting the Metronome Garbage Collector.

     These threads use only the heap memory for storage. Their disadvantage is that garbage collection interrupts your application but, because it is controlled by the Metronome Garbage Collector, the length and timing of the interruptions are predictable.
   - Responses less than 10 ms, choose real-time threads.

     Real-time threads can be placed in heap, scoped, or immortal memory. The benefits of using real-time threads are as follows:
     – They can run at a higher priority than standard Java threads.

    - Garbage collection is under the control of the Metronome Garbage Collector. However, the garbage collector runs at a higher priority than the highest priority of a real-time thread and interrupts the running of your program.
  - Responses less than a millisecond, choose no-heap real-time threads.

    The priority of no-heap real-time threads can be set higher than garbage collection and therefore is not interrupted significantly by the Metronome. Only the Metronome alarm thread runs at the top level of priority and that uses very small amounts of CPU.
3. Determine if your application requires asynchronous event handlers. This requirement depends on the structure of your program.
   - A time response less than 10 ms, choose real-time threads.
   - A time response less than a millisecond, choose no-heap real-time threads
4. Determine thread priorities. In general, the shorter the time period, the higher the priority.
5. Decide memory characteristics.
   - If a task has a variable or high allocation rate, which might overwhelm the GC, consider imposing a rate limit (using MemoryParameters) or consider allocating into a scoped memory area.
   - If a task generates a large amount of temporary data during a calculation, consider using a scoped memory area.
   - If a task generates some data during startup that is required for the lifetime of the JVM, consider using immortal memory. Try to avoid using immortal memory in cases where objects will continue to be created over the life of the JVM.
   - If tasks need to communicate, particularly if one is running under a no-heap real-time thread, consider using a scoped memory area for the communication.
   - If a task is running under a no-heap real-time thread, consider building a scoped memory area, for example LTMemory, to contain the no-heap thread, the runtime parameters, and possibly the wait-free queues that are used to communicate with the task. The LTMemory object must be built either in immortal or another scope to avoid errors when the no-heap thread attempts to reference it.
6. Modify the runtime options to improve the performance of your application, when you have decided the structure and content of your application. The next steps describe how to do this:
   a. During initial testing of your application, set generous amounts of space in heap, scoped, and immortal memory using the **-Xmx**, **-Xgc:immortalMemorySize=size** and **-Xgc:scopedMemoryMaximumSize=size** options.

      Note: With the Metronome GC the initial and maximum heap sizes must be the same because Metronome GC does not increase the size of the heap. Growing the heap is a nondeterministic operation.
   b. Use the **-verbose:gc** option to determine the amount of memory used.
   c. Modify the **-Xgc:targetUtilization** option to allow sufficient time for garbage collection to occur. The default is 70% and this percentage is usually adequate for most applications. Ensure that the garbage collection rate is slightly higher than the allocation rate.
   d. Set a realistic size for heap memory using the **-Xmx** option.

Modifying Java applications

To write code that makes use of the real-time Java features, use `javax.realtime.RealtimeThread` to replace `java.lang.Thread` for threads.

Before you begin

This example is based on JavaRadar.java class found in the `demo/realtime/sample_application.zip` file.

About this task

The programming model for real-time threads is similar to that for standard Java applications. However, this rather crude way of adding real-time threads to your programs does not take full advantage of the features of WebSphere Real Time for RT Linux. To do so, you must modify the threads so that they had a priority associated with them and also consider what memory areas they will use.

Just by changing the classes of your threads, you gain only a slight benefit for your application because the default priority of real-time threads is greater than that of standard Java threads.

To change JavaRadar to a RealtimeThread, you change the class it extends from `Thread` to `RealtimeThread`.

Replacement of `java.lang.Thread` by `javax.realtime.RealtimeThread`

The JavaRadar class in the sample application extends `java.lang.Thread`. For example:

```
public class JavaRadar extends Thread implements Radar
```

To make this Java thread a real-time thread, you redefine this class definition as follows:

```
public class RTJavaRadar extends RealtimeThread implements Radar
```

Writing real-time threads

So far, you have just modified an application; now it is time to write some code. You can write applications that use real-time threads to take advantage of the real-time priority levels and memory areas.

Before you begin

This example is based on JavaRadar.java, RTJavaRadar.java, and RTJavaControlLauncher.java classes found in the `demo/realtime/sample_application.zip` file.

This sample shows you how to use immortal memory with the same sample that is described in “Modifying Java applications.”

About this task

The programming model for real-time threads is similar to that for standard Java applications.

The benefits of using real-time threads are as follows:

- Full support for OS-level thread priorities on real-time threads.
- The use of scoped or immortal memory areas.
  - With scoped memory you can explicitly control the deallocation of memory without affecting Garbage Collection.
  - With no-heap real-time threads, you can use immortal memory to avoid garbage collection pauses.
  - Those real-time threads that reference objects in the heap are subject to garbage collection as are those real-time threads that are stored in the heap memory.
  - No-heap real-time threads cannot reference objects in heap memory and, as a consequence, they are not affected by garbage collection.

In Table 12, the priorities are assigned on the basis that the SimulationThread has the highest priority because it represents external events and must not be allowed to be preempted by anything in the program. The RadarThread needs to respond quickly to the pings from the controller. The quicker the response, the more accurate the measurement of the height of the lunar lander. The ListenThread also has to respond quickly to commands from the controller but takes second place to the RadarThread.

These three threads are in scoped memory because the simulation runs as a server. After the server has run a simulation, it can exit the scoped memory area and then reenter it to wait for another run of the simulation. The server uses scoped memory so that it can reset itself.

RTJavaRadarthread has the highest priority of the controller threads because it is more sensitive to timing because it is using this time to derive the height. It is immortal because it is running as a NHRT and the controller is run only once and the memory is released when the JVM exits.

For RTJavaControlThread and RTJavaEventThread, the time constraints are not as critical and therefore using heap memory is acceptable.

RTLoadThread performs no useful function for the lunar lander. However, RTLoadThread demonstrates that significant memory allocation and deallocation can be performed at a lower priority than other threads, without affecting the performance of the higher priority threads.

Table 12. Relationship of threads to memory areas in the sample application

Memory	Thread	Priority
Scoped	demo.sim.SimulationThread	38
	demo.sim.RadarThread	37
	demo.sim.SimulationThread.ListenThread	36
Immortal	demo.controller.RTJavaRadarThread	15
Heap	demo.controller.RTJavaControlThread	14
	demo.controller.RTJavaEventThread	13
Scoped and Heap	demo.controller.RTLoadThread	12

Examples

This code from demo.sim.SimulationThread shows where the priority of 38 has been set. **1** This line of code retrieves the maximum priority that is available in the JVM.

```
        super(null, area);

        // Set priority separately, as we are using "this".
        // Note that PriorityScheduler.MAX_PRIORITY has been deprecated.
        this.setSchedulingParameters(new PriorityParameters(PriorityScheduler
                .getMaxPriority(this)));     1
```

This code from demo.sim.SimLauncher shows where scoped memory has been defined. **2** shows the allocation of LTMemory, which is a scoped memory area that allocates memory in linear time.

```
        final IndirectRef<MemoryArea> myMemRef = new IndirectRef<MemoryArea>();

        /*
         * The LTMemory object has to be created in a memory area that the
         * NHRTs can access.
         */
        ImmortalMemory.instance().enter(new Runnable() {
            public void run() {
                myMemRef.ref = new LTMemory(10000000);  2
            }
        });

       final MemoryArea simMemArea = myMemRef.ref;
```

The ScopedMemoryArea object referenced by `simMemArea` is being allocated in immortal memory, because the NHRT must be able to reference the object that represents the ScopedMemoryArea. Allocating it on the heap results in the NHRT constructor throwing an IllegalArgumentException, because its memory area argument was on the heap.

```
            simMemArea.enter(new Runnable() {
                public void run() {
                    try {
                        CommsControl commsControl = new CommsControl();
```

This code from demo.controller.RTJavaControlLauncher shows where immortal memory has been defined and used by RTJavaRadar. Because RTJavaRadar runs once during the whole lifetime of the controller JVM, it is designed to allocate memory only on startup; it can be safely run in immortal memory. The design of the application benefits because the Controller can access the RTJavaRadar methods without having to first enter the scoped memory area. Entering the scoped memory area is difficult because Controller was written to run in ordinary Java as well as in real-time Java.

```
            final RadarPort radarPort = commsControl.getRadarPort();
            EventPort eventPort = commsControl.getEventPort();

            final IndirectRef<RTJavaRadar> radarRef = new IndirectRef<RTJavaRadar>();

            // Create RTJavaRadar in Immortal, it is an NHRT.
            // If it was in scoped, it's interaction with the other threads would
            // be more complex.
            ImmortalMemory.instance().enter(new Runnable() {
                public void run() {
                    // Realtime version of Radar.
                    radarRef.ref = new RTJavaRadar(radarPort, ImmortalMemory
                            .instance());
                }
            });

            RTJavaRadar radarJava = radarRef.ref;
```

Writing asynchronous event handlers

Asynchronous event handlers react to timer events or to events that occur outside a thread; for example, input from an interface of an application. In real-time systems, these events must respond inside the deadlines that you set for your application.

Before you begin

This example is based on RTJavaEventThread.java and RTJavaControlLauncher.java classes found in the `demo/realtime/sample_application.zip` file.

About this task

In the sample application, the event thread waits on events from the simulation that signals a crash or a landing. In the real-time version of this thread, the AsyncEvent mechanism is used. These events are used to print out the appropriate status message and to cause the controller to exit.

The RTJavaEventThread has two asynchronous events defined. They both have no parameters.

```
public class RTJavaEventThread extends RealtimeThread {

    private AsyncEvent landEvent = new AsyncEvent(),   Land
            crashEvent = new AsyncEvent();  Crash
```

These events create and register two asynchronous event handlers:

```
    /**
     * Pass a runnable object that will be fired when the land event occurs.
     *
     * @param runnable code to be executed when land event is triggered.
     */
    public void addLandHandler(Runnable runnable) {
        AsyncEventHandler handler = new AsyncEventHandler(runnable);
        this.landEvent.addHandler(handler);
    }

    /**
     * Pass a runnable object that will be run when the crash event occurs.
     *
     * @param runnable code to be executed when crash event is triggered.
     */
    public void addCrashHandler(Runnable runnable) {
        AsyncEventHandler handler = new AsyncEventHandler(runnable);
        this.crashEvent.addHandler(handler);
    }
```

When the crash or land messages are received, their corresponding asynchronous event handler is fired, causing Runnable objects to be released.

```
                tag = this.eventPort.receiveTag();

                switch (tag) {
                case EventPort.E_CRSH:
                    // Crash
                    this.crashEvent.fire();
                    this.running = false;
                    break;
                case EventPort.E_LAND:
                    // Land
                    this.landEvent.fire();
                    this.running = false;
                    break;
                }
```

Results

RTJavaControlLauncher.java contains invocations to the addLandHandler and addCrashHandler methods. The Runnable objects passed cause a message to be printed onto the console and the control thread is stopped when their associated asynchronous event handlers are fired. See RTJavaEventThread.java for the point where they are triggered.

```
            // AEH runnable for land handler.
            javaEventThread.addLandHandler(new Runnable() {
                public void run() {
                    System.out.println("LAND!");
                }
            });

            // AEH runnable for crash handler.
            javaEventThread.addCrashHandler(new Runnable() {
                public void run() {
                    System.out.println("CRASH!");
                }
            });
```

Writing NHRT threads

To add no-heap real-time threads (NHRT) to a Java application, use this tutorial to develop or modify your own programs.

Before you begin

This example is based on SimulationThread.java and SimLauncher.java classes found in the `demo/realtime/sample_application.zip` file.

About this task

The demo.sim.SimulationThread class is part of the simulation in the demo application. It is intended to act as a substitute for the real world, and, therefore, will probably run without interruption from the rest of the system. The thread is created as a NoHeapRealtimeThread with the highest available priority, to ensure that the thread is not interrupted by garbage collection or by other threads on the system.

In SimulationThread, the following constructor calls the super constructor "NoHeapRealtimeThread(SchedulingParameters scheduling, MemoryArea area)", before then setting its SchedulingParameters and ReleaseParameters separately:

```
public SimulationThread(MemoryArea area, ControlPort controlPort,
            EventPort eventPort, RadarThread radarThread) {

        super(null, area);

        // Set priority separately, as we are using "this".
        // Note that PriorityScheduler.MAX_PRIORITY has been deprecated.
        this.setSchedulingParameters(new PriorityParameters(PriorityScheduler
                .getMaxPriority(this)));

        ReleaseParameters releaseParms = new PeriodicParameters(null,
                new RelativeTime(period, 0)); // 20ms cycle (50Hz)
        this.setReleaseParameters(releaseParms);
        // It is good practice to identify each of the threads.
        this.setName("SimulationThread");

        this.controlPort = controlPort;
        this.eventPort = eventPort;
        this.radarThread = radarThread;
    }
```

The other active threads in the simulation are also created as no-heap real-time threads (NHRTs), but of slightly lower priority. See "Writing real-time threads" on page 74 for the arrangement of the priorities.

The simulation has the option of running indefinitely, so that after a simulation has completed it restarts. Because the simulation is composed of NHRTs, you can choose ScopedMemory or ImmortalMemory. The sample application uses ScopedMemory for the simulation because it is appropriate to exit the ScopeMemoryArea that was allocated when the simulation finished and then reenter it to wait for the next run. In this case, no state is carried over from one run to the next.

Most classes are NHRT safe; however, most classes can be run in a manner that is not NHRT safe. For example, if the DatagramSockets were kept in immortal memory, or in an outer scoped memory area, problems might occur because they are not designed to span memory areas. The sample application uses just the one ScopedMemory area to prevent such problems.

Memory allocation in RTSJ

In RTSJ, you can allocate an object in a specific memory area in a number of ways and it is not always obvious which of these to choose at any given point.

Each approach has some characteristics, which vary between implementations of RTSJ, and make a difference to either the performance or the eventual memory footprint. This section outlines the available options and suggests occasions where they might be the most appropriate choice for allocating an object.

Static initializer

The simplest way to allocate an object in the immortal memory area is to allocate it in a static initializer. The advantage is that you do not have to deal with the issues of changing memory context, but the circumstances where this pattern is appropriate are quite limited. This approach is efficient in that the amount of immortal memory consumed is limited to that required for the object itself.

MemoryArea.newInstance(Class c)

This approach is straightforward if a thread is in a memory context and wants to allocate an object in another area, which must already be in the scope stack of the thread. The advantage is that you need access only to the class to be instantiated, but the newInstance method must build an appropriate constructor. This pattern is most appropriate if objects of a given class must be allocated infrequently, but otherwise tends to show high memory usage.

MemoryArea.newInstance(Constructor c, Object[] args)

Again, a simple approach if a thread is in a memory context and wants to allocate an object in another context, which must already be in the scope stack of the thread. In this case, you must pass a Constructor and some arguments and assumes the responsibility of ensuring that Constructor is valid in the current memory context. Because the newInstance method does not have to build a Constructor, the memory usage is lower than newInstance(Class c) and thus this pattern is more appropriate if objects are to be allocated more frequently and you are willing to pay the price of allocating the constructor in advance and storing it somewhere like ImmortalMemory.

MemoryArea.enter(Runnable r) followed by new <class>()

This approach makes the given MemoryArea the new default for allocations and removes the need for reflection and the attendant Constructor objects. Hence it is most appropriate if many objects are to be created because no additional memory usage occurs above the object itself. This approach works only if the given area is not already active in the scope stack of any thread. The requirement to build a Runnable memory area makes this approach more complex than using newInstance because you generally should pass parameters either on the Runnable or through static or instance fields.

MemoryArea.executeInArea(Runnable r) followed by new <class>()

Again, this approach makes the given MemoryArea the new default for allocations and removes the need for reflection and the attendant Constructor objects. Hence it is most appropriate if many objects are to be created because no additional memory usage occurs above the object itself. You can use this approach if the given area is already in the scope stack of the current thread and hence is more flexible than MemoryArea.enter. The requirement to build a Runnable makes this approach more complex than using newInstance because you generally should pass parameters either on the Runnable or through static or instance fields.

Class.newInstance()

This approach builds the new instance in the current memory area and therefore must be used with either MemoryArea.enter or executeInArea. No additional memory usage occurs above the object itself.

Using the high-resolution timer

The real-time clock provides more precision that the clocks associated with the standard JVM.

Before you begin

This example is based on RTJavaRadar.java class found in the `demo/realtime/sample_application.zip` file.

About this task

Ordinary Java has limited ability for dealing with clocks and timers. The Real-Time Specification for Java allows absolute times to be specific with nanosecond precision and sufficient magnitude for wall-clock time. javax.realtime.HighResolutionTime and its subclasses are used to represent time with two components, milliseconds and nanoseconds.

WebSphere Real Time for RT Linux uses the support of the underlying operating system to supply the high resolution time. Current® Linux kernels supply a clock with, at best, 4 millisecond guaranteed precision. The Linux patches supplied with WebSphere Real Time for RT Linux provide a clock with a precision of closer to 1 microsecond.

The RTJavaRadar class demonstrates the use of the high-resolution timer:

- 1 gets the real-time clock.
- 2 gets the current absolute time.
- 3 gets the nanosecond component of time. The accuracy of the real-time clock means that using nanoseconds is reasonable.
- 4 gets the time before and after the ping.
- 5 returns the speed of descent of the lander.
- 6 makes the thread wait for 5 milliseconds before performing another iteration.

```
public void run() {
       // The following objects are created in advance and reused each
       // iteration.
       Clock rtClock = Clock.getRealtimeClock();         1
       AbsoluteTime time = rtClock.getTime();            2

       try {
           double height = 0.0, lastheight;
           long millis = time.getMilliseconds(), lastmillis;
           long nanos = time.getNanoseconds(), lastnanos;     3

           while (this.running) {

               lastmillis = millis;
               lastnanos = nanos;
               lastheight = height;

               // Rather than use the time = rtClock.getTime() form, this
               // method
               // replaces the values in a preexisting AbsoluteTime object.
               rtClock.getTime(time);                 4
               millis = time.getMilliseconds();
               nanos = time.getNanoseconds();

               // We time the time it takes to send the ping and receive the
               // pong.
               this.radarPort.ping();

               rtClock.getTime(time);                 4

               height = (time.getMilliseconds() - millis)
                       / demo.sim.RadarThread.timeScale;
               height += ((time.getNanoseconds() - nanos) / 1.0e6)      5
                       / demo.sim.RadarThread.timeScale;

               double difference = ((double) (millis - lastmillis)) / 1.0e3
                       + ((double) (nanos - lastnanos)) / 1.0e9;
               double speed = (height - lastheight) / difference;

               this.myHeight = height;
               this.mySpeed = speed;

               try {
                   sleep(5);                    6
```

```
                } catch (InterruptedException e) {
                    // This is not important.
                }
            }
```

The preceding code can be compared with the following standard JVM code in the JavaRadar class:

```
public void run() {
        try {
            double height = 0.0, lastheight;

            long nanos = System.nanoTime(), lastnanos;
            while (this.running) {
                /* Set the height every x milliseconds */
                Thread.sleep(5);
                lastnanos = nanos;
                lastheight = height;

                nanos = System.nanoTime();

                this.radarPort.ping();

                // Time scale is height units per millisecond
                height = ((System.nanoTime() - nanos) / 1.0e6)
                        / demo.sim.RadarThread.timeScale;

                double speed = (height - lastheight)
                        / (((double) (nanos - lastnanos)) / 1.0e9);

                this.myHeight = height;
                this.mySpeed = speed;
            }
```

The sample application

The sample application uses a series of examples to demonstrate the features of WebSphere Real Time for RT Linux that can be used to improve the real-time characteristics of Java programs.

The source files for the sample application are in the `demo/realtime/sample_application.zip` file.

The sample consists of two main components:

- **A simulation,** a simple example of a lunar lander. The lander's position is defined by its height above the ground, derived from timed pulses, and its horizontal distance from the landing area. See Figure 6 on page 83.

  The simulation class is written using no-heap real-time threads (NHRTs) and is not modified any further in this documentation.
- **A controller** that sends commands to the simulation. The controller sends radar pings to judge the lander's height and control the rate of descent of the lander based on this information. The controller also receives a stream of information from the lander; for example, the lander's distance from the landing area.

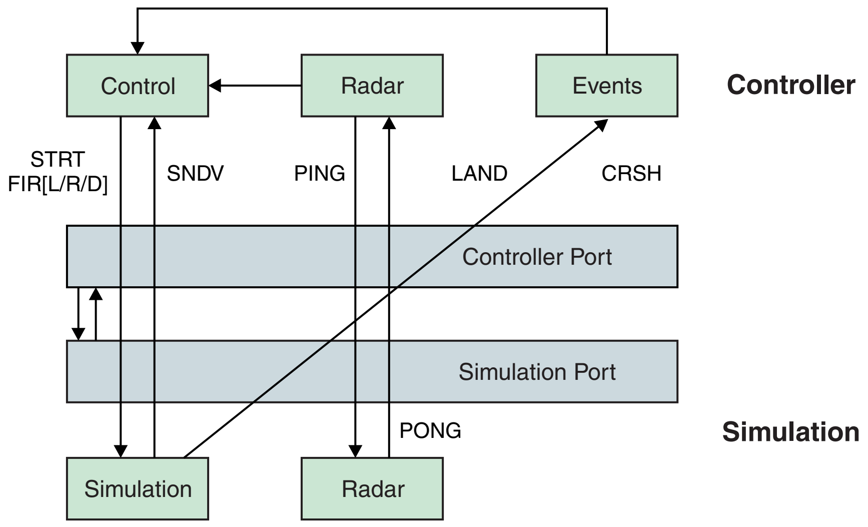
  The controller is written initially in standard Java. In "Modifying Java applications" on page 74, it is developed into a real-time Java program

Depending on the outcome of the landing, the controller is sent one of two messages: either crash or land.

Using the sample application you can perform these operations:

- Run both the simulation and the controller together to demonstrate a combination of real-time and standard Java classes running together. For information, see "Building the sample application" on page 84 and "Running the sample application" on page 84, where you will also see output that you can expect from the sample application.

  Note: You can start both the simulation and the controller at the same time using the LaunchBoth class.
- Compare the difference when using the Metronome Garbage Collector and the standard Garbage Collector. For information, see "Running the sample application without Real Time" on page 84 and "Running the sample application with Metronome Garbage Collector" on page 86.
- Run the application using the ahead-of-time (AOT) complier. For information, see "Running the sample application while using AOT" on page 87.



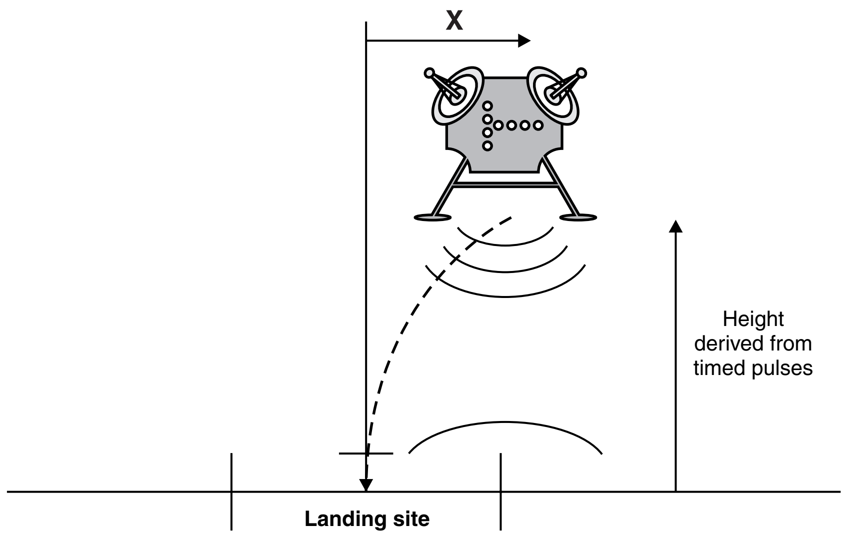


Figure 6. Diagram of the lunar lander

This diagram shows the relationship of the modules provided in the sample. The controller and simulation components communicate with each other through the controller and simulation ports. The controller component has three threads: Control, Radar, and Events. The simulation component has two threads: Simulation and Radar. The Control thread starts the simulation, then sends firing messages to the simulation component to control the direction of the lander. The Simulation thread sends back values which represent its state. The Radar threads of each component send PING and PONG messages to each other. The time between the exchange of these messages is used by the Control thread to calculate the height of the lander. The Simulation thread also sends appropriate ending events, either crash or land, to the Events thread. The Events thread passes the ending events back to the Control thread, which then ends the simulation.

Building the sample application

The sample application source code is provided for guidance. Preparation requires unpacking and compilation of the Java source code before it can be run.

Procedure

1. Create a working directory.
2. Extract the sample application into your working directory:
   ```
   unzip sample_application.zip
   ```
3. Create a new directory for your output:
   ```
   mkdir classes
   ```
4. Compile the source.
   a. Generate a list of the files:
      ```
      find -name "*.java" > source
      ```
   b. Compile the source:
      ```
      javac -Xrealtime -Xlint:deprecated -g -d classes @source
      ```
   c. Create a jar file of the class files:
      ```
      jar cf demo.jar -C classes/ .
      ```

What to do next

You can now run the sample application.

Running the sample application

WebSphere Real Time provides a standard JVM as well as a real-time JVM, started with the **`-Xrealtime`** command-line argument.

The sample application has two components, designed to be run in separate JVMs:

- The Simulation, which only runs in Real-Time Java.
- The Controller, which can run either non-Real-Time or Real-Time Java.

Running the Controller code in a variety of modes demonstrates the benefits of the IBM Real-Time Java technology.

Running the sample application without Real Time

In this procedure, you run the sample application without taking advantage of IBM WebSphere Real Time.

Before you begin

To run the sample application, you must first build the sample source code. See "Building the sample application" on page 84 for more information.

Procedure

1. Start the simulation:

   ```
   java -Xrealtime -classpath ./demo.jar -Xgc:scopedMemoryMaximumSize=11m
   demo.sim.SimLauncher <port>
   ```

   In this command, *<port>* is a deallocated port for the workstation.
2. Start the controller:

   ```
   java -classpath ./demo.jar -mx300m demo.controller.JavaControlLauncher <host>
   <port>
   ```

   In this command, *<host>* is the hostname of the workstation running the simulation, and *<port>* is the port specified in the previous step.

Results

The application produces a message showing that the simulation and the controller have started:

```
SimLauncher: Waiting for connections...
Starting control thread...
```

Some point samples of the values in the controller are printed out to the console:

```
x=99.50, radar=199.11, y=198.34, vx=-0.71, vy=-0.43, timeSinceLast=0.19, targetVx=-6.01, targetVy=-9.00
x=95.50, radar=194.59, y=192.70, vx=-2.70, vy=-2.43, timeSinceLast=0.20, targetVx=-5.94, targetVy=-9.00
x=87.50, radar=186.57, y=183.06, vx=-4.70, vy=-4.40, timeSinceLast=0.20, targetVx=-5.77, targetVy=-9.00
x=76.46, radar=172.84, y=169.42, vx=-5.42, vy=-6.75, timeSinceLast=0.20, targetVx=-5.60, targetVy=-9.00
x=65.36, radar=155.58, y=151.84, vx=-5.50, vy=-9.19, timeSinceLast=0.20, targetVx=-5.57, targetVy=-9.00
x=54.36, radar=138.06, y=135.24, vx=-5.44, vy=-7.63, timeSinceLast=0.20, targetVx=-5.56, targetVy=-9.00
x=43.26, radar=120.57, y=117.22, vx=-5.67, vy=-9.62, timeSinceLast=0.20, targetVx=-5.52, targetVy=-9.00
x=32.36, radar=103.60, y=100.72, vx=-5.47, vy=-9.06, timeSinceLast=0.20, targetVx=-5.43, targetVy=-9.00
x=21.52, radar=84.60, y=82.86, vx=-5.32, vy=-9.09, timeSinceLast=0.20, targetVx=-5.60, targetVy=-9.00
x=10.72, radar=67.07, y=65.56, vx=-5.30, vy=-10.54, timeSinceLast=0.20, targetVx=-5.65, targetVy=-9.00
x=0.76, radar=51.08, y=49.78, vx=-4.30, vy=-7.52, timeSinceLast=0.20, targetVx=-0.50, targetVy=-9.00
x=-5.24, radar=37.07, y=35.94, vx=-2.30, vy=-8.26, timeSinceLast=0.20, targetVx=0.50, targetVy=-9.00
x=-7.24, radar=20.05, y=19.90, vx=-0.30, vy=-6.15, timeSinceLast=0.20, targetVx=0.50, targetVy=-9.00
x=-6.36, radar=2.68, y=2.80, vx=0.27, vy=-10.08, timeSinceLast=0.20, targetVx=0.50, targetVy=-9.00
```

Just before the simulation stops, an event summary message is issued:

```
Fire down transitions 141, fire horizontally transitions 141
LAND!
```

In addition to point samples and the event summary message, the controller produces a graph called `graph.svg` in the same directory. The graph contains a plot of the point samples. The graph shows the effect of garbage collection pauses on the JavaRadar thread when running the application with a standard non-Real-Time JVM. The data representing the Radar Height has spikes. The spikes are caused by standard garbage collection pauses affecting the Controller application. On some runs, the garbage collection pauses are long enough to cause failures, leading to the message:

```
CRASH!
```

To see the pause times caused by garbage collection, add the **`-verbose:gc`** option to the controller launch command:

```
java -classpath ./demo.jar -verbose:gc -mx300m demo.controller.JavaControlLauncher
<host> <port>
```

Running the sample application with Metronome Garbage Collector

You can run a standard Java application in a real-time environment without any need to rewrite the code, by adding the -Xrealtime option. The option enables both Real-Time Java language features, and the Metronome Garbage Collector.

Before you begin

To run the sample application, you must first build the sample source code. See "Building the sample application" on page 84 for more information.

Procedure

1. Start the simulation:

   ```
   java -Xrealtime -classpath ./demo.jar -Xgc:scopedMemoryMaximumSize=11m
   demo.sim.SimLauncher <port>
   ```

   In this command, *<port>* is a deallocated port on the workstation.
2. Start the controller:

   ```
   java -Xrealtime -classpath ./demo.jar -mx300m
   demo.controller.JavaControlLauncher <host> <port>
   ```

   In this command, *<host>* is the hostname of the workstation running the simulation, and *<port>* is the port specified in the previous step. Running both JVMs on the same workstation can lead to less deterministic behavior. See "Considerations" on page 24 for more information.

Results

The application runs and generates several outputs, including:
1. Messages showing that the simulation and the controller have started.
2. Point samples of the controller values.
3. A graph called `graph.svg` in the same directory, containing a plot of the point samples.
4. An event summary message.

When running the application with Metronome garbage collection, the point samples and corresponding graph tends to show:
- No spikes in the Radar Height data.
- Accurate tracking of the Real Height data.

The reason is that the Controller code is now running with shorter garbage collection pauses.

Metronome garbage collection pauses are frequent, but typically less than 1 millisecond in duration. Non-real-time garbage collection pauses are fewer, but typically last for tens or hundreds of milliseconds. The difference between the pauses can be seen by adding the **`-verbose:gc`** option to the Controller run command.

See "Using verbose:gc information" on page 135 for more information about the verbose garbage collection output.

Running the sample application while using AOT

This procedure runs a standard Java application in a real-time environment while using the ahead-of-time (AOT) compiler, without the need to rewrite code. Use this sample to compare running the same application while using the JIT compiler.

See "Using compiled code with WebSphere Real Time for RT Linux" on page 40 for more details about ahead-of-time compilation.

Before you begin

To run the sample application, you must first build the sample source code. See "Building the sample application" on page 84 for more information.

About this task

The ahead-of-time compiler compiles your Java application to native code before running it. You can predict more precisely how the application runs, because there are no interruptions caused by just-in-time (JIT) compilation.

Procedure

1. Convert the application bytecodes into native code.
   a. The conversion takes place by first running the sample with the normal JIT compiler.
   ```
   java -Xrealtime -Xjit:verbose={precompile},vlog=./sim.aotOpts \
        -classpath ./demo.jar -Xgc:scopedMemoryMaximumSize=11m
        demo.sim.SimLauncher <port>
   ```
   In this command, *<port>* is a deallocated port for the workstation.
   b. In a different window, run the application.
   ```
   java -Xrealtime -Xjit:verbose={precompile},vlog=./control.aotOpts \
        -classpath ./demo.jar -Xmx300m demo.controller.JavaControlLauncher
        localhost <port>
   ```
   In this command, *<port>* is the port specified in the previous step. The application outputs results similar to the following messages:
   ```
   Fire down transitions 141, fire horizontally transitions 141
   ```
   and:
   ```
   Land!
   ```
   c. Combine the AOT options files created in the previous steps.
   ```
   cat sim.aotOpts.20081014.234958.13205 control.aotOpts.20081014.234958.13205
   > sample.aotOpts
   ```
   The names used for the log files created in the previous steps have date and process ID information appended to the file name. The format for the file name is specified by the **`vlog=`** option. For example, **`vlog=sim.aotOpts`** generates a file name similar to **`sim.aotOpts.20081014.234958.13205`**:
   d. Compile the files in the `sample.aotOpts` file in `realtime.jar`,`vm.jar` `rt.jar`, and the application `demo.jar`. When using shared class caches, the name of the cache must not exceed 53 characters.

```
admincache -Xrealtime -populate -cacheName "sample" -aotFilterFile
sample.aotOpts -classpath ./demo.jar \
$JAVA_HOME/jre/lib/i386/realtime/jclSC160/vm.jar \
$JAVA_HOME/jre/lib/i386/realtime/jclSC160/realtime.jar \
$JAVA_HOME/jre/lib/rt.jar \
./demo.jar
```

The compilation results are reported:

```
J9 Java(TM) admincache 1.0
Licensed Materials - Property of IBM

(c) Copyright IBM Corp. 1991, 2008 All Rights Reserved
IBM is a registered trademark of IBM Corp.
Java and all Java-based marks and logos are trademarks or registered
trademarks of Oracle Corporation

JVMSHRC256I Persistent shared cache "sample" has been destroyed
Converting files
Converting /team/mstoodle/demo/sdk/jre/lib/i386/realtime/jclSC160/vm.jar into shared class cache
Succeeded to convert jar file /team/mstoodle/demo/sdk/jre/lib/i386/realtime/jclSC160/vm.jar
Converting /team/mstoodle/demo/sdk/jre/lib/i386/realtime/jclSC160/realtime.jar into shared class cache
Succeeded to convert jar file /team/mstoodle/demo/sdk/jre/lib/i386/realtime/jclSC160/realtime.jar
Converting /team/mstoodle/demo/sdk/jre/lib/rt.jar into shared class cache
Succeeded to convert jar file /team/mstoodle/demo/sdk/jre/lib/rt.jar
Converting /team/mstoodle/demo/demo.jar into shared class cache
Succeeded to convert jar file /team/mstoodle/demo/demo.jar

Processing complete
```

Note: The line:

```
JVMSHRC256I Persistent shared cache "sample" has been destroyed
```

means that any existing cache called "sample" is destroyed by this command, to create the specified cache.

e. Display the contents of the populated cache.

```
admincache -Xrealtime -cacheName "sample" -printStats
```

2. Start the simulation:

```
java -Xrealtime -Xnojit -Xmx300m -Xshareclasses:name="sample" \
     -classpath ./demo.jar -Xgc:scopedMemoryMaximumSize=11m \
     demo.sim.SimLauncher <port>
```

In this command, *<port>* is a deallocated port for this workstation.

3. Start the controller:

```
java -Xrealtime -Xnojit -Xmx300m -Xshareclasses:name="sample" \
     -classpath ./demo.jar \
     demo.controller.JavaControlLauncher <host> <port>
```

In this command, *<host>* is the hostname of the workstation running the simulation, and *<port>* is the port specified in the previous step. Running both JVMs on the same workstation can lead to less deterministic behavior. See "Considerations" on page 24 for more information.

Results

The application runs and generates several outputs, including:

1. Messages showing that the simulation and the controller have started.
2. Point samples of the controller values.

3. A graph called `graph.svg` in the same directory, containing a plot of the point samples.
4. An event summary message.

When running the application with ahead-of-time compilation, the point samples and corresponding graph tends to show:

- No spikes in the Radar Height data.
- Accurate tracking of the Real Height data.

The reason is that the Controller code is now running with shorter garbage collection pauses and no just-in-time compilation interruptions.

A benefit of using the shared class cache to run this application is that the controller and the simulation JVMs share some of the memory used by classes loaded by both JVMs.

The sample real-time hash map

WebSphere Real Time for RT Linux includes HashMap and HashSet implementations that provide more consistent performance for the put method than the standard HashMap in the IBM SDK for Java 7.

The standard java.util.HashMap that IBM provides works well for high throughput applications. It also helps with applications that know the maximum size their hash map needs to grow to. For applications that need a hash map that could grow to variable sizes, depending on usage, there is a potential performance problem with the standard hash map. The standard hash map provides good response times for adding new entries into the hash map using the put method. However, when the hash map fills up, a larger backing store must be allocated. This means that the entries in the current backing store must be migrated. If the hash map is large, the time to perform a put could also be large. For example, the operation could take several milliseconds.

WebSphere Real Time for RT Linux includes a sample real-time hash map. It provides the same functional interface as the standard java.util.HashMap, but enables much more consistent performance for the put method. Instead of creating a backing store and migrating all the entries when the hash map fills up, the sample hash map creates an additional backing store. The new backing store is chained to the other backing stores in the hash map. The chaining initially causes a slight performance reduction while the empty backing store is allocated and chained to the other backing stores. Once the backing hash map is updated, it is faster than having to migrate all the entries. A disadvantage of the real-time hash map is that the get, put and remove operations are slightly slower. The operations are slower because each look-up must to proceed through a set of backing hash maps instead of just one.

To try out the real-time hash map, add the `RTHashMap.jar` file to the start of your boot class path. If you installed WebSphere Real Time for RT Linux into the directory `$WRT_ROOT`, then add the following option to use the real-time hash map with your application, instead of the standard hash map:

```
-Xbootclasspath/p:$WRT_ROOT/demo/realtime/RTHashMap.jar
```

The source and class files for the real-time hash map implementation are included in the `demo/realtime/RTHashMap.jar` file. In addition, a real time java.util.LinkedHashMap and java.util.HashSet implementation are also provided.

Developing WebSphere Real Time for RT Linux applications using Eclipse

Using Eclipse provides you with a fully-featured IDE when developing your real-time applications.

Before you begin

If this is the first time that you have used the Eclipse application development environment to develop real-time applications, use this procedure to configure your environment.

WebSphere Real Time for RT Linux supplies the standard Oracle **`javac`** compiler. There are no restrictions on which compiler you use, but it must produce valid Java 5.0 class files. However, the javax.realtime.* Java classes have to be on the build path.

About this task

To develop your applications on Eclipse, follow these instructions:

Procedure

1. Download Eclipse from http://www.eclipse.org/downloads/. It is recommended that you use Eclipse 3.1.2 for correct Java 5.0 compilation.
2. Download IBM SDK and Runtime Environment for Linux platforms, Java 2 Technology Edition, Version 5.0 compliant JVM for running Eclipse.
3. Extract this file, `opt/IBM/javawrt3/jre/lib/i386/realtime/jclSC160/realtime.jar`, from the WebSphere Real Time for RT Linux package.
4. Open Eclipse and create a project. Click **File** > **New**. Select **Java project** from the **New Project** panel.
5. Click **Next** to display the **New Java Project** panel.
   a. Enter a project name, for example, RTSJ-Tests.
   b. Check that the JDK compiler is set to 5.0.
6. Click **Finish**.
7. Create a working directory and import the `opt/IBM/javawrt3/jre/lib/i386/realtime/jclSC160/realtime.jar` file.
8. Click **File** > **New** > **Folder** to open the **New Folder** panel. Enter a new folder name, for example, *deplib*.
9. Click **Finish**.
10. To import your `realtime.jar` file, click **File** > **Import** to open the **Import** panel.
11. Click **File System** and click **Next**.
12. Open the `opt/IBM/javawrt3/jre/lib/i386/realtime/jclSC160/` directory on the filesystem where the JVM was unpacked.
13. Select the check box next to the realtime.jar file, specify a folder to import into, for example `RTSJ-Tests/deplib`, and ensure that the **Create selected folders only** option is selected.
14. Click **Finish**.
15. Add the jar file to the library path. Right-click your project and click **Properties** to open the **Properties** panel.
16. Click **Java Build Path** and the **Libraries** tab. Click **Add Jars**.

17. Click **realtime.jar** under your project directory. Click **OK**.

Results

If this procedure is successful, the realtime.jar file appears in the list of .jar files on the **Libraries** tab.

Example

Eclipse can use the `realtime.src.jar` to present additional information on the RTSJ classes. To do this, open the properties window for the imported `realtime.jar` file, click **Java Source Attachment** and enter in **Location path:** the location of the `realtime.src.jar` file.

What to do next

If you want to build applications by using Apache Ant with Eclipse, add the `realtime.jar` file to the class path in your Ant build script. For example:

```
<property name="rtsj.src" location="." />
<property name="rtsj.deplib" location="deplib" />
<property name="rtsj.jar.dir" location="build/rtsj-jar.dir" />

<!-- Generate .class files for this package -->
<target name="compile" depends="init">
<javac destdir="${rtsj.jar.dir}"
srcdir="${rtsj.src}"
target="1.5"
classpath="${rtsj.deplib}/realtime.jar:${rtsj.src}"
debug="true"/>
</target>
```

This is only a part of an ant build script.

Debugging your applications

Using the Eclipse Application developer you can debug your applications either locally or remotely.

About this task

To debug your real-time application remotely, the JVM being debugged requires the following option.
`-agentlib:jdwp=transport=dt_socket,server=y,suspend=y,address=10100`

Procedure

1. In the Linux environment where your application is running, enter:

   ```
   java -Xrealtime -agentlib:jdwp=transport=dt_socket,server=y,suspend=y,address=10100
   ```

   where:
   - `server=y` indicates that the JVM is accepting connections from debuggers.
   - `suspend=y` makes the JVM wait for a debugger to attach before running.
   - `address=10100` is the port number to which the debugger should attach to the JVM. This number should normally be above 1024.

   The JVM displays the following message:

   ```
   Listening for transport dt_socket at address: 10100
   ```
2. Open your application in Eclipse and select **Debug**.

3. A new configuration for debugging remote applications should be created. You need only to create one if an application in the same project is run, and is listening on the same port for each run.
4. When you have created the configuration, fill in the name of the Configurations, the name of the project that contains the application you are debugging, the *hostname* of the workstation where the application is running, and the port number you passed in the **-agentlib** options.
5. Click **Debug** to start the debugging session. The **Debug** perspective should be open for you to view the state of the remotely debugged JVM.

Running Eclipse with the JVM

This section explains how to run Eclipse with the WebSphere Real Time for RT Linux JVM.

To run Eclipse with the JVM, specify the following items with the **eclipse** command:

- The fully qualified directory to the Java executable file of the WebSphere Real Time for RT Linux JVM that you intend to use
- The **-Xrealtime** JVM option
- The size of the Immortal Memory that you want Eclipse to use. The size should be at least 128M.

For example:

```
eclipse -vm $JAVA_HOME/jre/bin/java -vmargs -Xrealtime -Xgc:immortalMemorySize=128M
```

Note: The Eclipse SDK does not take advantage of the various real-time memory options which are available to WebSphere Real Time for RT Linux applications. One consequence of this behavior is that Immortal Memory can become exhausted, especially when Eclipse is used for many hours or days without being restarted. If an **OutOfMemory** error occurs, you can increase the value of the **-Xgc:immortalMemorySize** option to increase the amount of Immortal Memory that you want Eclipse to use.

Chapter 7. Performance

WebSphere Real Time for RT Linux is optimized for consistently short GC pauses rather than the highest throughput performance or smallest memory footprint.

On systems where hyperthreading is supported, you must ensure that it is not enabled. The reason is to avoid adverse performance effects when using WebSphere Real Time for RT Linux.

Reducing timing variability and support for the Real-Time Specification for Java (RTSJ) required some standard IBM Java runtime optimizations to be disabled. Consequently, a reduction in overall performance is likely to be observed when a standard Java application is run with the **`-Xrealtime`** parameter.

Performance on certified hardware configurations

Certified systems have sufficient clock granularity and processor speed to support WebSphere Real Time for RT Linux performance goals. For example, a well-written application running on a system that is not overloaded, and with an adequate heap size, would normally experience GC pause times that are well below 1 millisecond, typically about 500 microseconds. During GC cycles, an application with default environment settings is not paused for more than 30% of elapsed time during any sliding 10 millisecond window. The collective time spent in GC pauses over any 10 millisecond period normally totals less than 3 milliseconds.

Reducing timing variability

The two main sources of variability in a standard JVM are handled in WebSphere Real Time for RT Linux as follows:

- Java code preparation: loading and Just-In-Time (JIT) compilation is dealt with by Ahead-Of-Time (AOT) compilation. See “Using the AOT compiler” on page 43.
- Garbage Collection pauses: the potentially long pauses from standard Garbage Collector modes are avoided by using the Metronome Garbage Collector. See “Using the Metronome Garbage Collector” on page 68.

Class data sharing between JVMs for non-Real-Time mode

Class sharing is supported in non-real-time mode, but operates differently than in real-time mode.

You can share class data between Java Virtual Machines (JVMs) by storing it in a memory-mapped cache file on disk. Sharing reduces the overall virtual storage consumption when more than one JVM shares a cache. Sharing also reduces the startup time for a JVM after the cache has been created. The shared class cache is independent of any running JVM and persists until it is deleted.

A shared cache can contain:

- Bootstrap classes
- Application classes
- Metadata that describes the classes

- Ahead-of-time (AOT) compiled code

Note: A real-time shared classes cache cannot be removed by a non real-time JVM.

Chapter 8. Security

This section contains important information about security.

Security considerations for the shared class cache

The shared class cache is designed for ease of cache management and usability, but the default security policy might not be appropriate.

When using the shared class cache, you must be aware of the default permissions for new files so that you can improve security by restricting access.

File	**Default permissions**
new shared caches	read permissions for group and other
`javasharedresources` directory	world read, write, and execute permission

You require write permission on both the cache file and the cache directory to destroy or grow a cache.

Changing the file permissions on the cache file

To limit access to a shared class cache, you can use the **`chmod`** command.

Change required	**Command**
Limit access to the user and group	`chmod 770 /tmp/javasharedresources`
Limit access to the user	`chmod 700 /tmp/javasharedresources`
Limit the user to read and write access only for a particular cache	`chmod 600 /tmp/javasharedresources/`*`<file for shared cache>`*
Limit the user and group to read and write access only for a particular cache	`chmod 660 /tmp/javasharedresources/`*`<file for shared cache>`*

See "Creating a Real-Time Shared Class Cache" on page 44 for more information about creating a shared class cache.

Connecting to a cache that you do not have permission to access

If you try to connect to a cache that you do not have the appropriate access permissions for, you see an error message:

```
JVMSHRC226E Error opening shared class cache file
JVMSHRC220E Port layer error code = -302
JVMSHRC221E Platform error message: Permission denied
JVMJ9VM015W Initialization error for library j9shr25(11): JVMJ9VM009E J9VMDllMain
failed
Could not create the Java virtual machine.
```

Chapter 9. Troubleshooting and support

Troubleshooting and support for WebSphere Real Time for RT Linux

- "General problem determination methods"
- "Troubleshooting OutOfMemory Errors" on page 103
- "Using diagnostic tools" on page 112

General problem determination methods

Problem determination helps you understand the kind of fault you have, and the appropriate course of action.

When you know what kind of problem you have, you might do one or more of the following tasks:

- Fix the problem.
- Find a good workaround.
- Collect the necessary data with which to generate a bug report to IBM.

Linux problem determination

This section describes problem determination on Linux.

The IBM SDK for Java V7 User guide contains useful guidance on diagnosing problems on Linux, covering:

- Setting up and checking your Linux environment
- General debugging techniques
- Diagnosing crashes
- Debugging hangs
- Debugging memory leaks
- Debugging performance problems

You can find this information here: IBM SDK for Java 7 - Linux problem determination.

The following information is supplementary for IBM WebSphere Real Time for RT Linux

Setting up and checking your Linux environment

On IBM WebSphere Real Time for RT Linux, check that the JVM is configured correctly to generate a system dump.

Linux system dumps (core files)

When a crash occurs, the most important diagnostic data to obtain is the Linux system dump (core file). To ensure that this file is generated, you must check your operating system settings, and your available disk space, as described in the IBM SDK for Java V7 user guide.

Java virtual machine settings

The JVM must be configured to generate core files when a crash occurs. Run `java -Xrealtime -Xdump:what` on the command line. The output from this option is:

```
-Xdump:system:
    events=gpf+abort+traceassert+corruptcache,
    label=/mysdk/sdk/jre/bin/core.%Y%m%d.%H%M%S.%pid.dmp,
    range=1..0,
    priority=999,
    request=serial
```

The values shown are the default settings. At least `events=gpf` must be set to generate a core file when a crash occurs. You can change and set options with the command-line option **`-Xdump:system[:name1=value1,name2=value2 ...]`**

General debugging techniques

Because Java thread names are visible in the operating system, you can use the ps command to help with debugging. When using tracing tools, you must use the correct commands for IBM WebSphere Real Time for RT Linux.

Examining process information

The output you can expect to see when running the ps command on IBM WebSphere Real Time for RT Linux is:

```
ps -eLo pid,tid,rtprio,comm,cmd
29286 29286      - java            jre/bin/java -Xrealtime -jar example.jar
29286 29287      - main            jre/bin/java -Xrealtime -jar example.jar
29286 29290     88 Signal Reporter jre/bin/java -Xrealtime -jar example.jar
29286 29295      - JIT Compilation jre/bin/java -Xrealtime -jar example.jar
29286 29296     13 JIT Sampler     jre/bin/java -Xrealtime -jar example.jar
29286 29297      - Signal Dispatch jre/bin/java -Xrealtime -jar example.jar
29286 29298      - Finalizer maste jre/bin/java -Xrealtime -jar example.jar
29286 29299     11 Gc Slave Thread jre/bin/java -Xrealtime -jar example.jar
29286 29300     89 Metronome GC Al jre/bin/java -Xrealtime -jar example.jar
29286 29301      - Thread-2        jre/bin/java -Xrealtime -jar example.jar
29286 29302     43 Realtime AEH Se jre/bin/java -Xrealtime -jar example.jar
29286 29303     83 Realtime AEH Se jre/bin/java -Xrealtime -jar example.jar
29286 29304     83 Realtime AEH Se jre/bin/java -Xrealtime -jar example.jar
29286 29305     83 Realtime AEH Se jre/bin/java -Xrealtime -jar example.jar
29286 29306     83 Realtime AEH Se jre/bin/java -Xrealtime -jar example.jar
29286 29307     83 Realtime AEH Se jre/bin/java -Xrealtime -jar example.jar
29286 29311     83 Realtime AEH Se jre/bin/java -Xrealtime -jar example.jar
29286 29312     83 Realtime AEH Se jre/bin/java -Xrealtime -jar example.jar
29286 29313     85 Realtime AEH No jre/bin/java -Xrealtime -jar example.jar
29286 29314     85 Realtime AEH No jre/bin/java -Xrealtime -jar example.jar
29286 29315     87 Realtime Schedu jre/bin/java -Xrealtime -jar example.jar
29286 29316     79 Realtime AEH Se jre/bin/java -Xrealtime -jar example.jar
29286 29317     85 Realtime Non-he jre/bin/java -Xrealtime -jar example.jar
29286 29318     83 Realtime Heap T jre/bin/java -Xrealtime -jar example.jar
29286 29319     83 Realtime Heap T jre/bin/java -Xrealtime -jar example.jar
29286 29321     45 RealtimeThread- jre/bin/java -Xrealtime -jar example.jar
29286 29343     43 RealtimeThread- jre/bin/java -Xrealtime -jar example.jar
29286 29345      - stdout reader j jre/bin/java -Xrealtime -jar example.jar
29286 29346      - stderr reader j jre/bin/java -Xrealtime -jar example.jar
```

e Selects all processes.

L Shows threads.

o Provides a pre-defined format of columns to display. The columns specified are the process ID, thread ID, scheduling policy, real-time thread priority, and the command associated with the process. This information is useful for understanding what threads in your application as well as the virtual machine are running at a given time.

Tracing tools

Three tracing tools on Linux are **`strace`**, **`ltrace`**, and **`mtrace`**. The command `man strace` displays a full set of available options.

`strace`
The strace tool traces system calls. You can either use it on a process that is already available, or start it with a new process. strace records the system calls made by a program and the signals received by a process. For each system call, the name, arguments, and return value are used. strace allows you to trace a program without requiring the source (no recompilation is required). If you use strace with the **-f** option, it will trace child processes that have been created as a result of a forked system call. You can use strace to investigate plug-in problems or to try to understand why programs do not start properly.

To use strace with a Java application, type `strace java -Xrealtime <class-name>`.

You can direct the trace output from the strace tool to a file by using the `-o` option.

`ltrace`
The ltrace tool is distribution-dependent. It is very similar to strace. This tool intercepts and records the dynamic library calls as called by the executing process. strace does the same for the signals received by the executing process.

To use ltrace with a Java application, type `ltrace java -Xrealtime <class-name>`

`mtrace`
mtrace is included in the GNU toolset. It installs special handlers for malloc, realloc, and free, and enables all uses of these functions to be traced and recorded to a file. This tracing decreases program efficiency and should not be enabled during normal use. To use mtrace, set **IBM_MALLOCTRACE** to 1, and set **MALLOC_TRACE** to point to a valid file where the tracing information will be stored. You must have write access to this file.

To use mtrace with a Java application, type:

```
export IBM_MALLOCTRACE=1
export MALLOC_TRACE=/tmp/file
java -Xrealtime <class-name>
mtrace /tmp/file
```

Diagnosing crashes

When gathering information about running processes and the Java environment prior to a crash, follow these guidelines.

Gathering process information

When researching what was happening before the crash occurred, use **`gdb`** and the **`bt`** command to display the stack trace of the failing thread, instead of analyzing the core file.

Finding out about the Java environment

Use Javadump to determine what each thread was doing and which Java methods were being run. Match function addresses against library addresses to determine the source of code running at various points.

Use the **-verbose:gc** option to look at the state of the Java heap and the Immortal and Scoped Memory areas, . Ask these questions:
- Was there a shortage of memory in one of the memory areas that could have caused the crash?
- Did the crash occur during garbage collection, indicating a possible garbage collection fault?
- Did the crash occurred after garbage collection, indicating a possible memory corruption?

Debugging performance problems

When debugging performance problems, consider these specific items for IBM WebSphere Real Time for RT Linux in addition to the topics in the IBM SDK for Java V7 user guide.

Sizing memory areas

The JVM can be tuned by varying the sizes of the heap, immortal and scoped memory. Choose the correct size to optimize performance. Using the correct size can make it easier for the Garbage Collector to provide the required utilization.

For more information about varying the size of the memory areas, see "Troubleshooting the Metronome Garbage Collector" on page 134.

JIT compilation and performance

When using the JIT, you should consider the implications to real-time behavior.

If you require predictable behavior but also need better performance then you should consider using ahead-of-time (AOT) compilation. For further information see "Using compiled code with WebSphere Real Time for RT Linux" on page 40.

Known limitations on Linux

Linux has been under rapid development and there have been various issues with the interaction of the JVM and the operating system, particularly in the area of threads.

Note the following limitations that might be affecting your Linux system.

Threads as processes

If the number of Java threads exceeds the maximum number of processes allowed, your program might:
- Get an error message
- Get a **SIGSEGV** error
- Stop

For more information, see *The Volano Report* at http://www.volano.com/report/index.html.

Floating stacks limitations

If you are running without floating stacks, regardless of what is set for **`-Xss`**, a minimum native stack size of 256 KB for each thread is provided.

On a floating stack Linux system, the **`-Xss`** values are used. If you are migrating from a non-floating stack Linux system, ensure that any **`-Xss`** values are large enough and are not relying on a minimum of 256 KB.

glibc limitations

If you receive a message indicating that the `libjava.so` library could not be loaded because of a symbol not found (such as __bzero), you might have an earlier version of the GNU C Runtime Library, glibc, installed. The SDK for Linux thread implementation requires glibc version 2.3.2 or greater.

Font limitations

When you are installing on a Red Hat system, to allow the font server to find the Java TrueType fonts, run (on Linux IA32, for example):

```
/usr/sbin/chkfontpath --add opt/IBM/javawrt3/jre/lib/fonts
```

You must do this at installation time and you must be logged on as “root” to run the command. For more detailed font issues, see the *Linux SDK and Runtime Environment User Guide*.

Performance issues on Linux Red Hat MRG kernels

A configuration issue with Red Hat MRG kernels can cause unexpected pauses to application threads when WebSphere Real Time starts with verbose garbage collection enabled. These pauses are not reported in the verbose GC output, but can last several milliseconds, depending on the network configuration. JVMs started from remotely defined LDAP users are affected the most, because the name service cache daemon (`nscd`) is not started, causing network delays. Solve the problem by starting `nscd`. Follow these steps to check on the status of the `nscd` service and correct the problem:

1. Check that the `nscd` daemon is running by typing the command:
   ```
   /sbin/service nscd status
   ```
   If the daemon is not running you see the following message:
   ```
   nscd is stopped
   ```
2. As root user, start the `nscd` service with the following command:
   ```
   /sbin/service nscd start
   ```
3. As root user, change the startup information for the `nscd` service with the following command:
   ```
   /sbin/chkconfig nscd on
   ```

The `nscd` process is now running, and starts automatically after reboot.

NLS problem determination

The JVM contains built-in support for different locales.

The IBM SDK for Java V7 User guide contains useful guidance on diagnosing problems with NLS, covering:

- Overview of fonts
- Font utilities
- Common NLS problems and possible causes

You can find this information here: IBM SDK for Java 7 - NLS problem determination.

ORB problem determination

One of your first tasks when debugging an ORB problem is to determine whether the problem is in the client-side or in the server-side of the distributed application. Think of a typical RMI-IIOP session as a simple, synchronous communication between a client that is requesting access to an object, and a server that is providing it.

The IBM SDK for Java V7 User guide contains useful guidance on diagnosing problems with ORB, covering:
- Identifying an ORB problem
- Interpreting the stack trace
- Interpreting ORB traces
- Common problems
- IBM ORB service: Collecting data

You can find this information here: IBM SDK for Java 7 - ORB problem determination.

The following information is supplementary for IBM WebSphere Real Time for RT Linux.

IBM ORB service: collecting data

When collecting the Java version output for service, run the following command:

```
java -Xrealtime -version
```

Preliminary tests

When a problem occurs, the ORB might generate a `org.omg.CORBA.*` exception that includes:
- text to indicate the cause
- a minor code
- a completion status

Before you assume that the ORB is the cause of the problem, check these items:
- The scenario can be reproduced in a similar configuration.
- The JIT is disabled.
- No AOT compiled code is being used

Other actions include:
- Turn off additional processors.
- Turn off Simultaneous Multithreading (SMT) where possible.
- Eliminate memory dependencies with the client or server. The lack of physical memory can be the cause of slow performance, apparent hangs, or crashes. To remove these problems, ensure that you have a reasonable headroom of memory.

- Check physical network problems such as firewalls, communication links, routers, and DNS name servers. These are the major causes of CORBA COMM_FAILURE exceptions. As a test, ping your own workstation name.
- If the application is using a database such as DB2®, switch to the most reliable driver. For example, to isolate DB2 AppDriver, switch to Net Driver, which is slower and uses sockets, but is more reliable.

Troubleshooting OutOfMemory Errors

Dealing with OutOfMemoryError exceptions, memory leaks, and hidden memory allocations.

For general troubleshooting information on the Metronome Garbage Collector, see "Troubleshooting the Metronome Garbage Collector" on page 134.

Diagnosing OutOfMemoryErrors

Diagnosing OutOfMemoryError exceptions in Metronome Garbage Collector can be more complex than in a standard JVM because of the periodic nature of the garbage collector.

The characteristics of the different types of heap are described in "Memory management" on page 13. In general, an RTSJ application requires approximately 20% more heap space than a standard Java application.

By default, the JVM produces the following diagnostic output when an uncaught OutOfMemoryError occurs:

- A snap dump; see "Using dump agents" on page 114.
- A Heapdump; see "Using Heapdump" on page 123.
- A Javadump; see "Using Javadump" on page 118
- A system dump; see "Using system dumps and the dump viewer" on page 126.

The dump file names are given in the console output:

```
JVMDUMP006I Processing dump event "systhrow", detail "java/lang/OutOfMemoryError" - please wait.
JVMDUMP007I JVM Requesting Snap dump using 'Snap.20081017.104217.13161.0001.trc'
JVMDUMP010I Snap dump written to Snap.20081017.104217.13161.0001.trc
JVMDUMP007I JVM Requesting Heap dump using 'heapdump.20081017.104217.13161.0002.phd'
JVMDUMP010I Heap dump written to heapdump.20081017.104217.13161.0002.phd
JVMDUMP007I JVM Requesting Java dump using 'javacore.20081017.104217.13161.0003.txt'
JVMDUMP010I Java dump written to javacore.20081017.104217.13161.0003.txt
JVMDUMP013I Processed dump event "systhrow", detail "java/lang/OutOfMemoryError".
```

The Java backtrace shown on the console output, and also available in the Javadump, indicates where in the Java application the OutOfMemoryError occurred. The next step is to find out which RTSJ memory area is full. The JVM memory management component issues a tracepoint that gives the size, class block address, and memory space name of the failing allocation. You find this tracepoint in the snap dump:

```
<< lines omitted... >>
09:42:17.563258000 *0xf2888e00       j9mm.101  Event       J9AllocateIndexableObject() returning NULL! 80
bytes requested for object of class 0xf1632d80 from memory space 'Metronome' id=0xf288b584
```

The tracepoint ID and data fields might vary from that shown, depending on the type of object being allocated. In this example, the tracepoint shows that the allocation failure occurred when the application attempted to allocate a 33.6 MB object of type `class 0x81312d8` in the `Metronome heap`, memory segment `id=0x809c5f0`.

You can determine which RTSJ memory area is affected by looking at the memory management information in the Javadump:

```
NULL           ---------------------------------------------------------------
0SECTION       MEMINFO subcomponent dump routine
NULL           =================================
NULL
1STMEMTYPE     Object Memory
NULL           region     start      end        size       name
1STHEAP        0xF288B584 0xF2A1C000 0xF6A1C000 0x04000000 Default
NULL
1STMEMUSAGE    Total memory available: 67108864 (0x04000000)
1STMEMUSAGE    Total memory in use:    66676824 (0x03F96858)
1STMEMUSAGE    Total memory free:      00432040 (0x000697A8)
NULL
NULL           region     start      end        size       name
1STHEAP        0xF288B5A4 0xF17FF008 0xF27FF008 0x01000000 Immortal
NULL
1STMEMUSAGE    Total memory available: 16777216 (0x01000000)
1STMEMUSAGE    Total memory in use:    00450816 (0x0006E100)
1STMEMUSAGE    Total memory free:      16326400 (0x00F91F00)
NULL
1STSEGTYPE     Internal Memory
NULL           segment    start      alloc      end        type       size
1STSEGMENT     0x0808DA48 0x0814A0A8 0x0814A0A8 0x0815A0A8 0x01000040 0x00010000
1STSEGMENT     0x0808DB50 0x08131EB8 0x08131EB8 0x08141EB8 0x01000040 0x00010000
<< lines removed for clarity >>
```

You can determine the type of object being allocated by looking at the classes section of the Javadump:

```
NULL           ---------------------------------------------------------------
0SECTION       CLASSES subcomponent dump routine
NULL           =================================
<< lines omitted... >>
1CLTEXTCLLOD    ClassLoader loaded classes
2CLTEXTCLLOAD    Loader *System*(0xF182BB80)
<< lines omitted... >>
3CLTEXTCLASS      [C(0xF1632D80)
```

Information in the Javadump confirms that the attempted allocation was for a character array, in the normal heap (`ID=0xF288B584`) and that the total allocated size of the heap, indicated by the appropriate `1STHEAP` line, is 67108864 decimal bytes or 0x04000000 hex bytes, or 64 MB.

In this example, the failing allocation is large in relation to the total heap size. If your application is expected to create 33 MB objects, the next step is to increase the size of the heap, using the **`-Xmx`** option.

It is more common for the failing allocation to be small in relation to total heap size. This is because of previous allocations filling up the heap. In these cases, the next step is to use the Heapdump to investigate the amount of memory allocated to existing objects.

The Heapdump is a compressed binary file containing a list of all objects with their object class, size, and references. Analyze the Heapdump using the IBM Monitoring and Diagnostic Tools for Java - Memory Analyzer tool, which is available for download from the IBM Support Assistant (ISA).

Using MDD4J, you can load a Heapdump and locate tree structures of objects that are suspected of consuming large amounts of heap space. The tool provides various views for objects on the heap. For example, MDD4J can show a view that details likely leak suspects, and gives the top five objects and packages contributing to the heap size. Selecting the tree view gives further information about the nature of the leaking container object.

By default, a single Heapdump file containing all objects in all RTSJ memory spaces is produced. Use the command-line option **-Xdump:heap:request=multiple** to request a separate Heapdump for each memory space. With multiple dumps, you can examine just the set of objects that are allocated in a specific memory area. You identify the Heapdumps by the file name given on the console output:

```
JVMDUMP006I Processing Dump Event "uncaught", detail "java/lang/OutOfMemoryError" - Please Wait.
<< lines omitted... >>
JVMDUMP007I JVM Requesting Heap Dump using '/home/test/heapdump-Default0809DCD8-0002.phd'
JVMDUMP010I Heap Dump written to /home/test/heapdump-Default0809DCD8-0002.phd
JVMDUMP007I JVM Requesting Heap Dump using '/home/test/heapdump-Immortal0809DCF4-0002.phd'
JVMDUMP010I Heap Dump written to /home/test/heapdump-Immortal0809DCF4-0002.phd
JVMDUMP007I JVM Requesting Heap Dump using '/home/test/heapdump-Scope0809DD10-0002.phd'
JVMDUMP010I Heap Dump written to /home/test/heapdump-Scope0809DD10-0002.phd
<< lines omitted... >>
JVMDUMP013I Processed Dump Event "uncaught", detail "java/lang/OutOfMemoryError".
Exception in thread "RTJ Memory Consumer (thread_type=Realtime)" java.lang.OutOfMemoryError
 at tests.com.ibm.jtc.ras.runnable.DepleteMemory.depleteMemory(DepleteMemory.java:57)
<< lines omitted... >>
```

How the IBM JVM manages memory

The IBM JVM requires memory for several different components, including memory regions for classes, compiled code, Java objects, Java stacks, and JNI stacks. Some of these memory regions must be in contiguous memory. Other memory regions can be segmented into smaller memory regions and linked together.

Dynamically loaded classes and compiled code are stored in segmented memory regions for dynamically loaded classes. Classes are further subdivided into writable memory regions (RAM classes) and read-only memory regions (ROM classes). At run time, the class cache is memory mapped, but not necessarily loaded, into a contiguous memory region on application startup. As classes are referenced by the application, classes and compiled code in the class cache are mapped into storage. The ROM component of the class is shared between multiple processes referencing this class. The RAM component of the class is created in the segmented memory regions for dynamically loaded classes when the class is first referenced by the JVM. AOT-compiled code for the methods of a class in the class cache are copied into an executable dynamic code memory region, because this code is not shared by processes. Classes that are not loaded from the class cache are similar to cached classes, except that the ROM class information is created in segmented memory regions for dynamically loaded classes. Dynamically generated code is stored in the same dynamic code memory regions that hold AOT code for cached classes.

All Java objects are stored in the standard heap memory when running the JVM without the **-Xrealtime** option. If the **-Xrealtime** option is used, objects can also be allocated out of two additional memory regions called immortal memory and scoped memory.

The stack for each Java thread can span a segmented memory region. The JNI stack for each thread occupies a contiguous memory region.

To determine how your JVM is configured, run with the **-verbose:sizes** option. This option prints out information about memory regions where you can manage the size. For memory regions that are not contiguous, an increment is printed describing how much memory is acquired every time the region needs to grow.

Here is example output using the **-Xrealtime -verbose:sizes** options:

```
-Xmca32K                        RAM class segment increment
-Xmco128K                       ROM class segment increment
-Xms64M                         initial memory size
-Xgc:immortalMemorySize=16M     immortal memory space size
-Xgc:scopedMemoryMaximumSize=8M scoped memory space maximum size
-Xmx64M                         memory maximum
-Xmso256K                       operating system thread stack size
-Xiss2K                         java thread stack initial size
-Xssi16K                        java thread stack increment
-Xss256K                        java thread stack maximum size
```

This example indicates that the RAM class segment is initially 0, but grows by 32 KB blocks as required. The ROM class segment is initially 0, and grows by 128 KB blocks as required. You can use the **-Xmca** and **-Xmco** options to control these sizes. RAM class and ROM class segments grow as required, so you will not typically need to change these options.

The immortal memory is a contiguous region and might need to be preallocated a larger space. In this example, the immortal memory region is preallocated at 16 MB. If you try to write more than 16 MB of objects into this immortal memory region, you receive an OutOfMemory exception because, by definition, this memory area is not garbage collected.

The scoped memory region is contiguous and in this example is pre-allocated at 8 MB. If you have many scoped memory areas active when the program is running, you might need to specify a larger scoped memory region.

Use the admincache utility to determine how large your memory mapped region will be if you use the class cache. Here is a sample of the output from the command `admincache -Xrealtime -printStats -nologo`:

```
J9 Java(TM) admincache 1.0

Current statistics for cache "sharedcc_localuser":


base address       = 0xA52B4000
end address        = 0xA59B7000
allocation pointer = 0xA59B4000

cache size         = 7356040
free bytes         = 330604
ROMClass bytes     = 3798460
AOT bytes          = 3101560
Data bytes         = 3812
Metadata bytes     = 121604
Metadata % used    = 1%

# ROMClasses       = 1044
# AOT Methods      = 1652
# Classpaths       = 2
# URLs             = 1
# Tokens           = 0
# Stale classes    = 0
% Stale classes    = 0%

Cache is 95% full
```

The cache size indicates that the memory mapped region will be slightly over 7 MB in space. The ROM class and AOT bytes take up the majority of this space, using slightly over 3 MB each.

Example OutOfMemoryError in immortal memory space

This example shows how to identify an OutOfMemoryError in the immortal memory space and describes steps to take to prevent the problem.

The snap dump shows that two small allocation requests have failed in the immortal memory area id=0x809dd1c:

```
16:08:04.876087000 083d4000       j9mm.100  Event       J9AllocateObject() returning NULL!
  16 bytes requested for object of class 0x8110e60 from memory space 'Immortal' id=0x809dd1c
16:08:04.876171000 083d4000       j9mm.100  Event       J9AllocateObject() returning NULL!
  32 bytes requested for object of class 0x81180f0 from memory space 'Immortal' id=0x809dd1c
```

The Javadump shows that the immortal memory space is full:

```
NULL           ---------------------------------------------------------------
0SECTION       MEMINFO subcomponent dump routine
NULL           =================================
1STHEAPFREE    Bytes of Heap Space Free: 3f0c000
1STHEAPALLOC   Bytes of Heap Space Allocated: 4000000
1STHEAPFREE    Bytes of Immortal Space Free: 0
1STHEAPALLOC   Bytes of Immortal Space Allocated: 1000000
<< lines omitted... >>
1STSEGTYPE     Object Memory
NULL           segment  start    alloc    end       type     bytes
1STSEGSUBTYPE  Immortal Segment  ID=0809DD1C
1STSEGMENT     0809D510 B279D008 B379D008 B379D008  00001008 1000000
```

An MDD4J analysis shows that a very large LinkedList has been allocated, consuming a large proportion of the available memory.

You are recommended to minimize the number of objects allocated in the immortal memory area because objects in the immortal area are not garbage collected. The most common immortal memory usage is class loading, which is a finite activity occurring mostly during JVM and application initialization. Applications with high numbers of loaded classes (or other immortal memory usage) can increase the size of the immortal memory area using the **-Xgc:immortalMemorySize=<size>** option. The default size for the immortal memory area is 16 MB.

If increasing the size of the immortal memory area only delays the OutOfMemoryError for immortal memory, investigate the pattern of continued allocation of immortal data, either related to class loading or other application objects.

Example OutOfMemoryError in scoped memory space

This example shows you how to identify an OutOfMemoryError in scoped memory space and describes steps to take to prevent the problem.

Use the command-line option **-Xdump:heap:request=multiple** to produce separate dumps for each memory space:

```
VMDUMP006I Processing Dump Event "uncaught", detail "java/lang/OutOfMemoryError" - Please Wait.
JVMDUMP007I JVM Requesting Snap Dump using '/home/test/snap-0001.trc'
JVMDUMP010I Snap Dump written to /home/test/snap-0001.trc
JVMDUMP007I JVM Requesting Heap Dump using '/home/test/heapdump-Default0809DCD8-0002.phd'
JVMDUMP010I Heap Dump written to /home/test/heapdump-Default0809DCD8-0002.phd
JVMDUMP007I JVM Requesting Heap Dump using '/home/test/heapdump-Immortal0809DCF4-0002.phd'
JVMDUMP010I Heap Dump written to /home/test/heapdump-Immortal0809DCF4-0002.phd
JVMDUMP007I JVM Requesting Heap Dump using '/home/test/heapdump-Scope0809DD10-0002.phd'
JVMDUMP010I Heap Dump written to /home/test/heapdump-Scope0809DD10-0002.phd
JVMDUMP007I JVM Requesting Java Dump using '/home/test/javacore-0003.txt'
JVMDUMP010I Java Dump written to /home/test/javacore-0003.txt
JVMDUMP013I Processed Dump Event "uncaught", detail "java/lang/OutOfMemoryError".
Exception in thread "RTJ Memory Consumer (thread_type=Realtime)" java.lang.OutOfMemoryError
   at tests.com.ibm.jtc.ras.runnable.DepleteMemory.depleteMemory(DepleteMemory.java:57)
   at tests.com.ibm.jtc.ras.runnable.DepleteMemory.run(DepleteMemory.java:26)
<< lines omitted... >>
```

The snap dump shows that two allocation requests have failed in scoped memory area id=0x809dd10:

```
16:14:45.887176823 08480900       j9mm.100  Event       J9AllocateObject() returning NULL!
  16 bytes requested for object of class 0x8110e38 from memory space 'Scoped' id=0x809dd10
16:14:45.887252747 08480900       j9mm.100  Event       J9AllocateObject() returning NULL!
  32 bytes requested for object of class 0x81180c8 from memory space 'Scoped' id=0x809dd10
```

The Javadump shows that for the scoped memory area with id=0x809dd10, the allocated size of the memory area is quite small, only 60 KB; in this case, increase the size of the scoped memory area in the application code.

```
0SECTION       MEMINFO subcomponent dump routine
NULL           =================================
1STHEAPFREE    Bytes of Heap Space Free: 3eb0000
1STHEAPALLOC   Bytes of Heap Space Allocated: 4000000
1STHEAPFREE    Bytes of Immortal Space Free: f47474
1STHEAPALLOC   Bytes of Immortal Space Allocated: 1000000
1STHEAPFREE    Bytes of Scoped Space ID=0809DD10 Free: eb00
1STHEAPALLOC   Bytes of Scoped Space Allocated: eb00
.......
1STSEGTYPE     Object Memory
NULL           segment  start    alloc    end       type     bytes
1STSEGSUBTYPE  Scoped Segment  ID=0809DD10
1STSEGMENT     0809D560 08416350 08424E50 08424E50  00002008 eb00
1STSEGSUBTYPE  Immortal Segment  ID=0809DCF4
1STSEGMENT     0809D4E8 B2857008 B3857008 B3857008  00001008 1000000
```

In the example Javadump, the scoped memory area appears to be empty. It appears empty because the Javadump is produced when the OutOfMemoryError reaches the JVM, at which time the scope has been exited and cleaned up. You can produce a Javadump at the point of failure by using the **-Xdump:java:events=throw,filter=java/lang/OutOfMemoryError** command-line option. By using this option, the free space in the scoped memory area is correctly reported.

It is also possible to exhaust the total space available for scoped memory; in this case, increase the size of the scoped memory area using the command-line option **-Xgc:scopedMemoryMaximumSize=<size>**. The default size for the scoped memory area is 8 MB. If the total space available for scoped memory becomes exhausted, you see different messages on the console; for example:

```
Exception in thread "main" java.lang.OutOfMemoryError: Creating (LTMemory) Scoped memory # 0 size=16777216
        at javax.realtime.MemoryArea.create(MemoryArea.java:808)
        at javax.realtime.MemoryArea.create(MemoryArea.java:798)
        at javax.realtime.ScopedMemory.create(ScopedMemory.java:1359)
        at javax.realtime.ScopedMemory.create(ScopedMemory.java:1351)
        at javax.realtime.ScopedMemory.initialize(ScopedMemory.java:1705)
        at javax.realtime.ScopedMemory.<init>(ScopedMemory.java:216)
        at javax.realtime.ScopedMemory.<init>(ScopedMemory.java:164)
```

Diagnosing problems in multiple heaps

You can use the address ranges provided in the Javadump with the occupancy information in the Heapdump to help analyze OutOfMemoryErrors in multiple RTSJ memory areas.

In this Javadump, the immortal segment ranges from 0xB281C008 to 0xB381C008, and the normal heap segment ranges from 0xB381D008 to 0xB781D008:

```
0SECTION       MEMINFO subcomponent dump routine
NULL           =================================
1STHEAPFREE    Bytes of Heap Space Free: 58000
1STHEAPALLOC   Bytes of Heap Space Allocated: 4000000
1STHEAPFREE    Bytes of Immortal Space Free: b319d8
1STHEAPALLOC   Bytes of Immortal Space Allocated: 1000000
NULL
1STSEGTYPE     Internal Memory
<< lines omitted... >>
1STSEGTYPE     Object Memory
NULL           segment  start    alloc    end       type     bytes
1STSEGSUBTYPE  Immortal Segment  ID=0809C68C
1STSEGMENT     0809BE80 B281C008 B381C008 B381C008  00001008 1000000
1STSEGSUBTYPE  Heap Segment  ID=0809C670
1STSEGMENT     0809BE08 B381D008 B781D008 B781D008  00000009 4000000
NULL
1STSEGTYPE     Class Memory
NULL           segment  start    alloc    end       type     bytes
1STSEGMENT     08158154 083FFD68 083FFEF0 08407D68  00010040 8004
```

The Heapdump is a compressed binary file containing a list of all objects with their object class, size, and references. Analyze the Heapdump using the IBM Monitoring and Diagnostic Tools for Java - Memory Analyzer tool, which is available for download from the IBM Support Assistant (ISA).

You can use the object memory locations listed by MDD4J to determine the memory space in which an object is. Addresses in the 0xB28*nnnnn* range are in the immortal memory area. Addresses in the 0xB61*nnnnn* range are in the normal heap.

Avoiding memory leaks

The garbage collector does not process immortal or scoped memory areas. In the case of immortal memory, the memory is freed only when the JVM exits. Scoped memory areas are freed only after their reference counts go to zero. Long-running tasks running in these contexts must be written in such a way that, after the task has warmed up, no additional memory from the immortal memory area is allocated.

Loading classes uses a small amount of immortal memory. These classes are not garbage collected in the real-time environment. As such, loading classes that are not required by the application can cause the application to use more immortal memory than necessary.

If your application contains classes that implement the Serializable interface, adjust the initial immortal memory size to account for the footprint of generated classes. Each constructor has one generated object per class, in the form of "GeneratedSerializationConstructorAccessorXXX" (whereXXX is a number) that is loaded into immortal memory the first time the object is serialized.

Avoid using immortal memory, because objects allocated from immortal memory cannot be garbage collected. Consider object pooling in immortal memory if the immortal memory area is being used more than occasionally.

Hidden memory allocation through language features

In a scoped or immortal memory context, avoid the variable arguments language feature because these methods allocate hidden memory.

Variable arguments (vararg)

The Java language implements variable arguments by passing them to the method as an array. The compiler makes calling variable argument methods easy by creating and initializing the array for you.

Memory can be lost by calling a variable argument method in an immortal or scoped memory context. Do not use variable arguments in scoped or immortal memory contexts. Instead, explicitly create an array and use it in place of the variable arguments.

Here are two examples showing equivalent ways of calling a variable argument method:

```
public class VarargEx {

    public static void main(String[] args) {
        System.out.println("Sum: "+ sum(1.0, 2.0 , 3.0, 4.0));

     }
    static double sum(double... params) {
        double total=0.0;

        for(double num : params) {
            total += num;
        }

        return total;
    }
}
```

```
public class VarargEx {

    public static void main(String[] args) {
        double array[] = new double[4];

        array[0] = 1.0; array[1] = 2.0; array[2] = 3.0; array[3] = 4.0;
        System.out.println("Sum: " + sum(array));
    }

    static double sum(double... params) {
        double total=0.0;
        for(double num : params) {
            total += num;
        }

        return total;
    }
}
```

The second example is preferred. Because the double array allocation becomes visible in the code, the allocation can be directed into a particular memory area.

String concatenation

Adding to an existing string to produce a longer string is implemented using java.lang.StringBuilder objects, which requires memory allocations.

Autoboxing

Autoboxing involves creating an object to contain a basic type, which requires memory allocations.

Using reflection across memory contexts

If a constructor object has been built in a scoped memory area, it can be used only in the same scope or an inner scope. Any attempt to use that constructor object in immortal, heap, or an outer scope memory context will fail.

The exception thrown when reflection has occurred across memory contexts will be similar to the following output:

```
Exception in thread "NoHeapRealtimeThread-14" javax.realtime.IllegalAssignmentError
   at java.lang.reflect.Constructor$1.<init>(Constructor.java:570)
   at java.lang.reflect.Constructor.acquireConstructorAccessor(Constructor.java:568)
   at java.lang.reflect.Constructor.newInstance(Constructor.java:521)
   at testMain$TestRunnable$1.run(testMain.java:40)
   at javax.realtime.MemoryArea.activateNewArea(MemoryArea.java:597)
   at javax.realtime.MemoryArea.doExecuteInArea(MemoryArea.java:612)
   at javax.realtime.ImmortalMemory.executeInArea(ImmortalMemory.java:77)
   at testMain$TestRunnable.allocate(testMain.java:36)
   at testMain$TestRunnable.run(testMain.java:12)
   at java.lang.Thread.run(Thread.java:875)
   at javax.realtime.ScopedMemory.runEnterLogic(ScopedMemory.java:280)
   at javax.realtime.MemoryArea.enter(MemoryArea.java:159)
   at javax.realtime.ScopedMemory.enterAreaWithCleanup(ScopedMemory.java:194)
   at javax.realtime.ScopedMemory.enter(ScopedMemory.java:186)
   at javax.realtime.RealtimeThread.runImpl(RealtimeThread.java:1824)
```

It is possible to work around this restriction by using the constructor in the same scope as it was allocated.

Using inner classes with scoped memory areas

When using inner classes in the context of scoped memory areas, you must take care when instantating inner class objects if the outer and inner objects are located in different memory areas. An IllegalAssignmentError will result from compiler-generated code that is not visible in the original source code, if the inner object is not capable of storing a reference to the outer object.

An inner class object must be able to store an implicit reference to its outer class object. If the reference violates RTSJ memory reference rules then an IllegalAssignmentError will be generated.

Most inner classes (including local and anonymous inner classes) will contain a compiler-generated (synthetic) non-static field for the instance of the lexically enclosing outer class. The only exception occurs when an inner class instance does not have an enclosing outer object, such as an anonymous class object instantiated in a static intializer block. The synthetic field of the inner object will contain a reference to the outer object. This is implemented by the compiler as a convenience to the java programmer. The field will not be visible in the original source code, although it is possible to write similar code using static nested classes with a reference that is visible. If the implicit reference violates RTSJ memory area rules, then an IllegalAssignmentError will be thrown, when the inner object is contructed, as it attempts to store the reference to the outer object.

In general, you cannot violate RTSJ memory reference rules when using inner classes. You cannot create an inner object if a reference to the associated outer object violates RTSJ memory reference rules. This rule means that an inner object allocated in immortal or heap cannot have a reference to an outer object from scoped memory. An inner object from scoped memory can have a reference to an outer object from scoped memory, but the outer object must be allocated from the same scoped memory area or an outer scoped memory area.

There are work-arounds, including:

- use static nested classes to eliminate the implicit reference
- choose memory areas to ensure inner and outer object relationships do not violate memory area reference restrictions

Using diagnostic tools

There are a number of diagnostic tools that are available to help diagnose problems with the IBM WebSphere Real Time for RT Linux JVM.

The IBM SDK for Java 7 provides a number of diagnostic tools that can be used to diagnose problems with the IBM WebSphere Real Time for RT Linux JVM. This section introduces the tools that are available, and provides links to further information about using the tools.

There is an important point to remember when using the SDK diagnostic tools. When you invoke the real time JVM, you use the following option:

```
java -Xrealtime
```

This option must be used when running diagnostic tools for the real time JVM. For example, to show the registered dump agents for the IBM WebSphere Real Time for RT Linux JVM, type:

```
java -Xrealtime -Xdump:what
```

Any further differences in using these tools with IBM WebSphere Real Time for RT Linux is provided here as supplementary information, together with sample output to assist you with diagnosis.

For a summary of the diagnostic information that is generated by the IBM SDK for Java 7, see Summary of diagnostic information.

Using the IBM Monitoring and Diagnostic Tools for Java

IBM provides tooling and documentation to help you understand, monitor, and diagnose problems with applications using the IBM JRE.

The following tools are available:
- Health Center
- Garbage Collection and Memory Visualizer
- Interactive Diagnostic Data Explorer
- Memory Analyzer

Garbage Collection and Memory Visualizer

Garbage Collection and Memory Visualizer (GCMV) helps you understand memory use, garbage collection behavior, and performance of Java applications.

GCMV parses and plots data from various types of log, including the following types:
- Verbose garbage collection logs.
- Trace garbage collection logs, generated by using the -Xtgc parameter.
- Native memory logs, generated by using the ps, **svmon**, or **perfmon** system commands.

The tool helps to diagnose problems such as memory leaks, analyze data in various visual formats, and provides tuning recommendations.

GCMV is provided as an IBM Support Assistant (ISA) add-on. For information about installing and getting started with the add-on, see: http://www.ibm.com/developerworks/java/jdk/tools/gcmv/.

Further information about GCMV is available in an IBM Information Center.

Health Center

Health Center is a diagnostic tool for monitoring the status of a running Java Virtual Machine (JVM).

The tool is provided in two parts:
- The Health Center agent that collects data from a running application.
- An Eclipse-based client that connects to the agent. The client interprets the data and provides recommendations to improve the performance of the monitored application.

Health Center is provided as an IBM Support Assistant (ISA) add-on. For information about installing and getting started with the add-on, see: http://www.ibm.com/developerworks/java/jdk/tools/healthcenter/.

Further information about Health Center is available in an IBM Information Center.

Interactive Diagnostic Data Explorer

Interactive Diagnostic Data Explorer (IDDE) is a GUI-based alternative to the dump viewer (**jdmpview** command). IDDE provides the same functionality as the dump viewer, but with extra support such as the ability to save command output.

Use IDDE to more easily explore and examine dump files that are produced by the JVM. Within IDDE, you enter commands in an investigation log, to explore the dump file. The support that is provided by the investigation log includes the following items:
- Command assistance
- Auto-completion of text, and some parameters such as class names

- The ability to save commands and output, which you can then send to other people
- Highlighted text and flagging of issues
- The ability to add your own comments
- Support for using the Memory Analyzer from within IDDE

IDDE is provided as an IBM Support Assistant (ISA) add-on. For information about installing and getting started with the add-on, see IDDE overview on developerWorks®.

Further information about IDDE is available in an IBM Information Center.

Memory Analyzer

Memory Analyzer helps you analyze Java heaps using operating system level dumps and Portable Heap Dumps (PHD).

This tool can analyze dumps that contain millions of objects, providing the following information:

- The retained sizes of objects.
- Processes that are preventing the Garbage Collector from collecting objects.
- A report to automatically extract leak suspects.

This tool is based on the Eclipse Memory Analyzer (MAT) project, and uses the IBM Diagnostic Tool Framework for Java (DTFJ) feature to enable the processing of dumps from IBM JVMs.

Memory Analyzer is provided as an IBM Support Assistant (ISA) add-on. For information about installing and getting started with the add-on, see: http://www.ibm.com/developerworks/java/jdk/tools/memoryanalyzer/.

Further information about Memory Analyzer is available in an IBM Information Center.

Using dump agents

Dump agents are set up during JVM initialization. They enable you to use events occurring in the JVM, such as Garbage Collection, thread start, or JVM termination, to initiate dumps or to start an external tool.

The IBM SDK for Java V7 User guide contains useful guidance on dump agents, covering:

- Using the **`-Xdump`** option
- Dump agents
- Dump events
- Advanced control of dump agents
- Dump agent tokens
- Default dump agents
- Removing dump agents
- Dump agent environment variables
- Signal mappings
- Dump agent default locations

You can find this information here: IBM SDK for Java 7 - Using dump agents.

Supplementary information for IBM WebSphere Real Time for RT Linux is provided here:

Dump events

Dump agents are triggered by events occurring during JVM operation. For IBM WebSphere Real Time for RT Linux, the default value for the slow event is 5 milliseconds.

Some events can be filtered to improve the relevance of the output. See "filter option" on page 116 for more information.

Note: The unload and expand events currently do not occur in WebSphere Real Time. Classes are in immortal memory and cannot be unloaded.

Note: The gpf and abort events cannot trigger a heap dump, prepare the heap (request=prepwalk), or compact the heap (request=compact).

The following table shows events available as dump agent triggers:

Event	Triggered when...	Filter operation
gpf	A General Protection Fault (GPF) occurs.	
user	The JVM receives the SIGQUIT signal from the operating system.	
abort	The JVM receives the SIGABRT signal from the operating system.	
vmstart	The virtual machine is started.	
vmstop	The virtual machine stops.	Filters on exit code; for example, `filter=#129..#192#-42#255`
load	A class is loaded.	Filters on class name; for example, `filter=java/lang/String`
unload	A class is unloaded.	
throw	An exception is thrown.	Filters on exception class name; for example, `filter=java/lang/OutOfMem*`
catch	An exception is caught.	Filters on exception class name; for example, `filter=*Memory*`
uncaught	A Java exception is not caught by the application.	Filters on exception class name; for example, `filter=*MemoryError`
systhrow	A Java exception is about to be thrown by the JVM. This is different from the 'throw' event because it is only triggered for error conditions detected internally in the JVM.	Filters on exception class name; for example, `filter=java/lang/OutOfMem*`
thrstart	A new thread is started.	
blocked	A thread becomes blocked.	
thrstop	A thread stops.	
fullgc	A garbage collection cycle is started.	
slow	A thread takes longer than 5ms to respond to an internal JVM request.	Changes the time taken for an event to be considered slow; for example, `filter=#300ms` will trigger when a thread takes longer than 300ms to respond to an internal JVM request.

Event	Triggered when...	Filter operation
allocation	A Java object is allocated with a size matching the given filter specification	Filters on object size; a filter must be supplied. For example, **`filter=#5m`** will trigger on objects larger than 5 Mb. Ranges are also supported; for example, **`filter=#256k..512k`** will trigger on objects between 256 Kb and 512 Kb in size.
traceassert	An internal error occurs in the JVM	Not applicable.
corruptcache	The JVM finds that the shared class cache is corrupt.	Not applicable.

filter option

Some JVM events occur thousands of times during the lifetime of an application. Dump agents can use filters and ranges to avoid excessive dumps being produced.

Wildcards

You can use a wildcard in your exception event filter by placing an asterisk only at the beginning or end of the filter. The following command does not work because the second asterisk is not at the end:

```
-Xdump:java:events=vmstop,filter=*InvalidArgumentException#*.myVirtualMethod
```

In order to make this filter work, it must be changed to:

```
-Xdump:java:events=vmstop,filter=*InvalidArgumentException#MyApplication.*
```

Class loading and exception events

You can filter class loading (load) and exception (throw, catch, uncaught, systhrow) events by Java class name:

```
-Xdump:java:events=throw,filter=java/lang/OutOfMem*
-Xdump:java:events=throw,filter=*MemoryError
-Xdump:java:events=throw,filter=*Memory*
```

You can filter throw, uncaught, and systhrow exception events by Java method name:

```
-Xdump:java:events=throw,filter=ExceptionClassName[#ThrowingClassName.
throwingMethodName[#stackFrameOffset]]
```

Optional portions are shown in brackets.

You can filter the catch exception events by Java method name:

```
-Xdump:java:events=catch,filter=ExceptionClassName[#CatchingClassName.
catchingMethodName]
```

Optional portions are shown in square brackets.

vmstop event

You can filter the JVM shut down event by using one or more exit codes:

```
-Xdump:java:events=vmstop,filter=#129..192#-42#255
```

slow event

You can filter the slow event to change the time threshold from the default of 5ms:

```
-Xdump:java:events=slow,filter=#300ms
```

You cannot set the filter to a time lower than the default time.

allocation event

You must filter the allocation event to specify the size of objects that cause a trigger. You can set the filter size from zero up to the maximum value of a 32-bit pointer on 32-bit platforms, or the maximum value of a 64-bit pointer on 64-bit platforms. Setting the lower filter value to zero triggers a dump on all allocations.

For example, to trigger dumps on allocations greater than 5 Mb in size, use:

```
-Xdump:stack:events=allocation,filter=#5m
```

To trigger dumps on allocations between 256Kb and 512Kb in size, use:

```
-Xdump:stack:events=allocation,filter=#256k..512k
```

Other events

If you apply a filter to an event that does not support filtering, the filter is ignored.

request option

Use the request option to ask the JVM to prepare the state before starting the dump agent. For IBM WebSphere Real Time for RT Linux there is an additional request option; **`multiple`**.

The available options are listed in the following table:

Option value	Description
`exclusive`	Request exclusive access to the JVM.
`compact`	Run garbage collection. This option removes all unreachable objects from the heap before the dump is generated.
`prepwalk`	Prepare the heap for walking. You must also specify **`exclusive`** when using this option.
`serial`	Suspend other dumps until this one has finished.
`multiple`	Produce separate heap dumps for each RTSJ memory area.
`preempt`	Applies to the Java dump agent and controls whether native threads in the process are forcibly preempted in order to collect stack traces. If this option is not specified, only Java stack traces are collected in the Javadump.

For example, the default setting of the request option for Javadumps is `request=exclusive+preempt`. To change the settings so that Javadumps are produced without preempting threads to collect native stack traces, use the following option:

```
-Xdump:java:request=exclusive
```

In general, the default request options are sufficient.

You can specify more than one request option with +. For example:

```
-Xdump:heap:request=exclusive+compact+prepwalk
```

Using Javadump

Javadump produces files that contain diagnostic information related to the JVM and a Java application captured at a point during execution. For example, the information can be about the operating system, the application environment, threads, stacks, locks, and memory.

The IBM SDK for Java V7 User guide contains useful guidance on Javadumps, covering:
- Enabling a Javadump
- Triggering a Javadump
- Interpreting a Javadump
- Environment variables and Javadump

You can find this information here: IBM SDK for Java 7 - Using Javadump.

Supplementary information and sample output for IBM WebSphere Real Time for RT Linux is provided in the following topics.

Storage Management (MEMINFO)

The `MEMINFO` section provides information about the Memory Manager, including heap, immortal, and scoped memory areas.

The `MEMINFO` section of a Javadump shows information about the Memory Manager. See Using the Metronome Garbage Collector for details about how the memory manager component works.

This part of the Javadump provides various storage management values, including:
- amount of free memory
- amount of used memory
- current size of the heap
- current size of immortal memory areas
- current size of scoped memory areas

This section also contains garbage collection history data. The data is shown as a sequence of tracepoints, each with a timestamp, ordered with the most recent tracepoint first.

Javadumps produced by the standard JVM contain a “GC History” section. This information is not contained in Javadumps produced when using the real-time JVM. Use the **`-verbose:gc`** option or the JVM snap trace to obtain information about GC behavior. See “Using verbose:gc information” on page 135 and the dump agents section of the IBM SDK for Java V7 User guide for more details.

If you are running a program which uses scoped memory, and an OutOfMemoryError is thrown, some of the memory areas listed in the Javadump might be empty. When a scope that is nested inside another scope runs out of memory, the inner scope might be deleted by the time the Javadump is generated. To get information which relates to the state of the memory areas at the time the OutOfMemoryError is thrown, run your program with the following command-line option:

```
-Xdump:java:events=throw,filter=java/lang/OutOfMemoryError,range=1..1
```

This command generates an additional Javadump when the OutOfMemoryError is thrown, rather than when the uncaught exception is detected, which happens slightly later. In this Javadump, can see all memory areas that were active at the time the OutOfMemoryError was thrown, including any inner scopes. For further information about using the **-Xdump** option, see the IBM SDK for Java V7 User guide.

In a Javadump, segments are blocks of memory allocated by the Java run time for tasks that use large amounts of memory. Example tasks are:

- maintaining JIT caches
- storing Java classes

The Java runtime environment also allocates other native memory, which is not listed in the MEMINFO section. The total memory used by Java runtime segments does not necessarily represent the complete memory footprint of the Java run time. A Java runtime segment consists of the segment data structure, and an associated block of native memory.

The following example shows some typical output. All the values are provided as hexadecimal values. The column headings in the MEMINFO section have the following meanings:

- Object memory section (HEAPTYPE):

  id The id of the space or region.

  start The start address of this region of the heap.

  end The end address of this region of the heap.

  size The size of this region of the heap.

  space/region
  For a line that contains only an id and a name, this column shows the name of the memory space. Otherwise the column shows the name of the memory space, followed by the name of a particular region that is contained within that memory space.

- Internal memory section (SEGTYPE), including class memory, JIT code cache, and JIT data cache:

  segment
  The address of the segment control data structure.

  start The start address of the native memory segment.

  alloc The current allocation address within the native memory segment.

  end The end address of the native memory segment.

  type An internal bit field describing the characteristics of the native memory segment.

  size The size of the native memory segment.

```
0SECTION       MEMINFO subcomponent dump routine
NULL           =================================
NULL
1STHEAPTYPE    Object Memory
NULL           id         start      end        size       space/region
1STHEAPSPACE   0x00497030     --         --         --     Generational
1STHEAPREGION  0x004A24F0 0x02850000 0x05850000 0x03000000 Generational/Tenured Region
1STHEAPREGION  0x004A2468 0x05850000 0x06050000 0x00800000 Generational/Nursery Region
1STHEAPREGION  0x004A23E0 0x06050000 0x06850000 0x00800000 Generational/Nursery Region
NULL
1STHEAPTOTAL   Total memory:           67108864 (0x04000000)
1STHEAPINUSE   Total memory in use:    33973024 (0x02066320)
1STHEAPFREE    Total memory free:      33135840 (0x01F99CE0)
NULL
1STSEGTYPE     Internal Memory
```

```
NULL           segment    start      alloc      end        type       size
1STSEGMENT     0x073DFC9C 0x0761B090 0x0761B090 0x0762B090 0x01000040 0x00010000
       (lines removed for clarity)
1STSEGMENT     0x00497238 0x004FA220 0x004FA220 0x0050A220 0x00800040 0x00010000
NULL
1STSEGTOTAL    Total memory:           873412 (0x000D53C4)
1STSEGINUSE    Total memory in use:         0 (0x00000000)
1STSEGFREE     Total memory free:      873412 (0x000D53C4)
NULL
1STSEGTYPE     Class Memory
NULL           segment    start      alloc      end        type       size
1STSEGMENT     0x0731C858 0x0745C098 0x07464098 0x07464098 0x00010040 0x00008000
        (lines removed for clarity)
1STSEGMENT     0x00498470 0x070079C8 0x07026DC0 0x070279C8 0x00020040 0x00020000
NULL
1STSEGTOTAL    Total memory:           2067100 (0x001F8A9C)
1STSEGINUSE    Total memory in use:    1839596 (0x001C11EC)
1STSEGFREE     Total memory free:       227504 (0x000378B0)
NULL
1STSEGTYPE     JIT Code Cache
NULL           segment    start      alloc      end        type       size
1STSEGMENT     0x004F9168 0x06960000 0x069E0000 0x069E0000 0x00000068 0x00080000
NULL
1STSEGTOTAL    Total memory:           524288 (0x00080000)
1STSEGINUSE    Total memory in use:    524288 (0x00080000)
1STSEGFREE     Total memory free:           0 (0x00000000)
NULL
1STSEGTYPE     JIT Data Cache
NULL           segment    start      alloc      end        type       size
1STSEGMENT     0x004F92E0 0x06A60038 0x06A6839C 0x06AE0038 0x00000048 0x00080000
NULL
1STSEGTOTAL    Total memory:           524288 (0x00080000)
1STSEGINUSE    Total memory in use:     33636 (0x00008364)
1STSEGFREE     Total memory free:      490652 (0x00077C9C)
NULL
1STGCHTYPE     GC History
3STHSTTYPE     15:18:14:901108829 GMT j9mm.134 -   Allocation failure end: newspace=7356368/8388608
oldspace=32038168/50331648 loa=3523072/3523072
3STHSTTYPE     15:18:14:901104380 GMT j9mm.470 -   Allocation failure cycle end: newspace=7356416/8388608
oldspace=32038168/50331648 loa=3523072/3523072
3STHSTTYPE     15:18:14:901097193 GMT j9mm.65 -   LocalGC end: rememberedsetoverflow=0
causedrememberedsetoverflow=0 scancacheoverflow=0 failedflipcount=0 failedflipbytes=0 failedtenurecount=0
failedtenurebytes=0 flipcount=11454 flipbytes=991056 newspace=7356416/8388608 oldspace=32038168/50331648
loa=3523072/3523072 tenureage=1
3STHSTTYPE     15:18:14:901081108 GMT j9mm.140 -   Tilt ratio: 50
3STHSTTYPE     15:18:14:893358658 GMT j9mm.64 -   LocalGC start: globalcount=3 scavengecount=24 weakrefs=0
soft=0 phantom=0 finalizers=0
3STHSTTYPE     15:18:14:893354551 GMT j9mm.63 -   Set scavenger backout flag=false
3STHSTTYPE     15:18:14:893348733 GMT j9mm.135 -   Exclusive access: exclusiveaccessms=0.002
meanexclusiveaccessms=0.002 threads=0 lastthreadtid=0x00495F00 beatenbyotherthread=0
3STHSTTYPE     15:18:14:893348391 GMT j9mm.469 -   Allocation failure cycle start: newspace=0/8388608
oldspace=38199368/50331648 loa=3523072/3523072 requestedbytes=48
3STHSTTYPE     15:18:14:893347364 GMT j9mm.133 -   Allocation failure start: newspace=0/8388608
oldspace=38199368/50331648 loa=3523072/3523072 requestedbytes=48
3STHSTTYPE     15:18:14:866523613 GMT j9mm.134 -   Allocation failure end: newspace=2359064/8388608
oldspace=38199368/50331648 loa=3523072/3523072
3STHSTTYPE     15:18:14:866519507 GMT j9mm.470 -   Allocation failure cycle end: newspace=2359296/8388608
oldspace=38199368/50331648 loa=3523072/3523072
3STHSTTYPE     15:18:14:866513004 GMT j9mm.65 -   LocalGC end: rememberedsetoverflow=0
causedrememberedsetoverflow=0 scancacheoverflow=0 failedflipcount=5056 failedflipbytes=445632
failedtenurecount=0 failedtenurebytes=0 flipcount=9212 flipbytes=6017148 newspace=2359296/8388608
oldspace=38199368/50331648 loa=3523072/3523072 tenureage=1
3STHSTTYPE     15:18:14:866493839 GMT j9mm.140 -   Tilt ratio: 64
3STHSTTYPE     15:18:14:859814852 GMT j9mm.64 -   LocalGC start: globalcount=3 scavengecount=23 weakrefs=0
soft=0 phantom=0 finalizers=0
3STHSTTYPE     15:18:14:859808692 GMT j9mm.63 -   Set scavenger backout flag=false
3STHSTTYPE     15:18:14:859801848 GMT j9mm.135 -   Exclusive access: exclusiveaccessms=0.004
meanexclusiveaccessms=0.004 threads=0 lastthreadtid=0x00495F00 beatenbyotherthread=0
3STHSTTYPE     15:18:14:859801163 GMT j9mm.469 -   Allocation failure cycle start: newspace=0/10747904
oldspace=38985800/50331648 loa=3523072/3523072 requestedbytes=232
3STHSTTYPE     15:18:14:859800479 GMT j9mm.133 -   Allocation failure start: newspace=0/10747904
oldspace=38985800/50331648 loa=3523072/3523072 requestedbytes=232
3STHSTTYPE     15:18:14:652219028 GMT j9mm.134 -   Allocation failure end: newspace=2868224/10747904
oldspace=38985800/50331648 loa=3523072/3523072
3STHSTTYPE     15:18:14:650796714 GMT j9mm.470 -   Allocation failure cycle end: newspace=2868224/10747904
oldspace=38985800/50331648 loa=3523072/3523072
3STHSTTYPE     15:18:14:650792607 GMT j9mm.475 -   GlobalGC end: workstackoverflow=0 overflowcount=0
memory=41854024/61079552
3STHSTTYPE     15:18:14:650784052 GMT j9mm.90 -   GlobalGC collect complete
```

```
3STHSTTYPE     15:18:14:650780971 GMT j9mm.57 -   Sweep end
3STHSTTYPE     15:18:14:650611567 GMT j9mm.56 -   Sweep start
3STHSTTYPE     15:18:14:650610540 GMT j9mm.55 -   Mark end
3STHSTTYPE     15:18:14:645222792 GMT j9mm.54 -   Mark start
3STHSTTYPE     15:18:14:645216632 GMT j9mm.474 -   GlobalGC start: globalcount=2
            (lines removed for clarity)
NULL
NULL           ------------------------------------------------------------------------
```

Threads and stack trace (THREADS)

For the application programmer, one of the most useful pieces of a Java dump is the THREADS section. This section shows a list of Java threads, native threads, and stack traces. For IBM WebSphere Real Time for RT Linux real-time threads and no heap real time threads are also shown.

A Java thread is implemented by a native thread of the operating system. Each thread is represented by a set of lines such as:

```
"main" J9VMThread:0x41D11D00, j9thread_t:0x003C65D8, java/lang/Thread:0x40BD6070, state:CW, prio=5
 (native thread ID:0xA98, native priority:0x5, native policy:UNKNOWN)
 Java callstack:
 at java/lang/Thread.sleep(Native Method)
 at java/lang/Thread.sleep(Thread.java:862)
 at mySleep.main(mySleep.java:31)
```

Java thread names are visible in the operating system when using the **ps** command. For further information about using the **ps** command, see “General debugging techniques” on page 98.

The properties on the first line are the thread name, addresses of the JVM thread structures and of the Java thread object, thread state, and Java thread priority. The properties on the second line are the native operating system thread ID, native operating system thread priority and native operating system scheduling policy.

Thread names are visible in three ways:

- Listed in javacore files. Not all threads are listed in javacore files.
- When listing threads from the operating system with the **ps** command.
- When using the java.lang.Thread.getName() method.

The following table provides information about IBM WebSphere Real Time for RT Linux thread names.

Table 13. Thread names in IBM WebSphere Real Time for RT Linux

Detail of thread	Thread name
An internal JVM thread used by the garbage collection module to dispatch the finalization of objects by secondary threads.	`Finalizer master`
The alarm thread used by the garbage collector.	`GC Alarm`
The slave threads used for garbage collection.	`GC Slave`
An internal JVM thread used by the just-in-time compiler module to sample the usage of methods in the application.	`IProfiler`
A thread used by the VM to manage signals received by the application, whether externally or internally generated.	`Signal Reporter`

Table 13. Thread names in IBM WebSphere Real Time for RT Linux (continued)

Detail of thread	Thread name
An internal JVM thread used to compile Java code.	`JIT Compilation Thread`
An internal JVM thread used to allow JVMTI agents to attach to a running JVM.	`Attach API wait loop`

The default names of real-time threads (javax.realtime.RealtimeThread) created in Java code are `RTThread-x`, where "`x`" is the thread number.

The default names of no-heap real-time threads are `NHRTThread-x`, where "`x`" is the thread number.

The Java thread priority is mapped to an operating system priority value in a platform-dependent manner. A large value for the Java thread priority means that the thread has a high priority. In other words, the thread runs more frequently than lower priority threads. For further details of how this works for Java threads, real-time threads and no-heap real-time threads, see "Priority mapping and inheritance" on page 11.

The values of state can be:
- R - Runnable - the thread is able to run when given the chance.
- CW - Condition Wait - the thread is waiting. For example, because:
  - A sleep() call is made
  - The thread has been blocked for I/O
  - A wait() method is called to wait on a monitor being notified
  - The thread is synchronizing with another thread with a join() call
- S – Suspended – the thread has been suspended by another thread.
- Z – Zombie – the thread has been killed.
- P – Parked – the thread has been parked by the new concurrency API (java.util.concurrent).
- B – Blocked – the thread is waiting to obtain a lock that something else currently owns.

If a thread is parked or blocked, the output contains a line for that thread, beginning with `3XMTHREADBLOCK`, listing the resource that the thread is waiting for and, if possible, the thread that currently owns that resource. For more information see the topic on blocked threads in the IBM SDK for Java V7 User guide.

When you initiate a Javadump to obtain diagnostic information, the JVM quiesces Java threads before producing the javacore. A preparation state of `exclusive_vm_access` is shown in the `1TIPREPSTATE` line of the `TITLE` section.

```
1TIPREPSTATE Prep State: 0x4 (exclusive_vm_access)
```

Threads that were running Java code when the javacore was triggered are in CW (Condition Wait) state.

```
3XMTHREADINFO      "main" J9VMThread:0x41481900, j9thread_t:0x002A54A4, java/lang/Thread:0x004316B8,
state:CW, prio=5
3XMTHREADINFO1            (native thread ID:0x904, native priority:0x5, native policy:UNKNOWN)
3XMTHREADINFO3           Java callstack:
4XESTACKTRACE                at java/lang/String.getChars(String.java:667)
4XESTACKTRACE                at java/lang/StringBuilder.append(StringBuilder.java:207)
```

The javacore `LOCKS` section shows that these threads are waiting on an internal JVM lock.

```
2LKREGMON          Thread public flags mutex lock (0x002A5234): <unowned>
3LKNOTIFYQ            Waiting to be notified:
3LKWAITNOTIFY            "main" (0x41481900)
```

Using Heapdump

The term Heapdump describes the IBM Virtual Machine for Java mechanism that generates a dump of all the live objects that are on the Java heap; that is, those that are being used by the running Java application.

The IBM SDK for Java V7 User guide contains useful guidance on Heapdumps, covering:

- Getting Heapdumps
- Tools for processing Heapdumps
- Using **`-Xverbose:gc`** to obtain heap information
- Environment variables and Heapdump
- Text (classic) Heapdump file format
- Portable Heap Dump (PHD) file format

You can find this information here: IBM SDK for Java 7 - Using Heapdump.

Supplementary information for IBM WebSphere Real Time for RT Linux:

Enabling multiple Heapdumps for real-time JVMs

The generated Heapdump is by default a single file containing information about all Java objects in all memory areas, Heap memory, Immortal memory and Scoped memory. The main reason to produce multiple dumps is so that each individual heap area can be analyzed using the traditional Heapdump tools without modification.

About this task

By default, Heapdumps contain information about all the objects in the JVM's memory areas, Heap, Immortal and Scoped memory. You can obtain separate Heapdumps containing information about Java objects in each memory area by using the **`request=multiple`** option with **`-Xdump:heap`**. Note that you must repeat the default settings of the request option as well, so you need to specify `request=multiple+exclusive+prepwalk+compact`. This produces a set of Heapdumps with an extra field in the name indicating the specific memory area:

```
heapdump.%id.%Y%m%d.%H%M%S.%pid.phd
```

where *%id* identifies the Heapdump file as containing objects in Heap memory, Immortal memory, or a specific area of Scoped memory.

There are 4 types of heap represented by the following names: “Default”, “Immortal”, “Scope” and “Other”. The Heapdump code replaces the %id in the heap label with one of these names concatenated with an identifier (typically numeric), for example: `heapdump.Immortal12994208.20060807.093653.7684.txt.`

Example

```
java -Xrealtime -Xdump:heap:defaults:request=multiple+exclusive+compact+prepwalk
     <java program>
```

Using this extra option produces multiple Heapdumps in the portable Heapdump (phd) format.

```
java -Xrealtime -Xdump:heap:defaults:request=multiple+exclusive+compact+prepwalk,
     opts=CLASSIC <java program>
```

Using this extra option produces multiple Heapdumps in the CLASSIC text format.

Using the -Xdump:what option shows the dump agents on JVM startup, and is useful for checking the dump options that are in place.

Text (classic) Heapdump file format

The text or classic Heapdump is a list of all object instances in the heap, including object type, size, and references between objects.

Header record

The header record is a single record containing a string of version information.

```
// Version: <version string containing SDK level, platform and JVM build level>
```

Example:

```
// Version: J2RE 7.0 IBM J9 2.6 Linux x86-32 build 20101016_024574_lHdRSr
```

Object records

Object records are multiple records, one for each object instance on the heap, providing object address, size, type, and references from the object.

```
<object address, in hexadecimal> [<length in bytes of object instance, in decimal>]
OBJ <object type> <class block reference, in hexadecimal>
<heap reference, in hexadecimal <heap reference, in hexadecimal> ...
```

The object address and heap references are in the heap, but the class block address is outside the heap. All references found in the object instance are listed, including references that are null values. The object type is either a class name including package or a primitive array or class array type, shown by its standard JVM type signature, see "Java VM type signatures" on page 126. Object records can also contain additional class block references, typically in the case of reflection class instances.

Examples:

An object instance, length 28 bytes, of type java/lang/String:

```
0x00436E90 [28] OBJ java/lang/String
```

A class block address of java/lang/String, followed by a reference to a char array instance:

```
0x415319D8 0x00436EB0
```

An object instance, length 44 bytes, of type char array:

```
0x00436EB0 [44] OBJ [C
```

A class block address of char array:

```
0x41530F20
```

An object of type array of java/util/Hashtable Entry inner class:

```
0x004380C0 [108] OBJ [Ljava/util/Hashtable$Entry;
```

An object of type java/util/Hashtable Entry inner class:

```
0x4158CD80 0x00000000 0x00000000 0x00000000 0x00000000 0x00421660 0x004381C0
0x00438130 0x00438160 0x00421618 0x00421690 0x00000000 0x00000000 0x00000000
0x00438178 0x004381A8 0x004381F0 0x00000000 0x004381D8 0x00000000 0x00438190
0x00000000 0x004216A8 0x00000000 0x00438130 [24] OBJ java/util/Hashtable$Entry
```

A class block address and heap references, including null references:

```
0x4158CB88 0x004219B8 0x004341F0 0x00000000
```

Class records

Class records are multiple records, one for each loaded class, providing class block address, size, type, and references from the class.

```
<class block address, in hexadecimal> [<length in bytes of class block, in decimal>]
CLS <class type>
<class block reference, in hexadecimal> <class block reference, in hexadecimal> ...
<heap reference, in hexadecimal> <heap reference, in hexadecimal>...
```

The class block address and class block references are outside the heap, but the class record can also contain references into the heap, typically for static class data members. All references found in the class block are listed, including those that are null values. The class type is either a class name including package or a primitive array or class array type, shown by its standard JVM type signature, see "Java VM type signatures" on page 126.

Examples:

A class block, length 32 bytes, for class java/lang/Runnable:

```
0x41532E68 [32] CLS java/lang/Runnable
```

References to other class blocks and heap references, including null references:

```
0x4152F018 0x41532E68 0x00000000 0x00000000 0x00499790
```

A class block, length 168 bytes, for class java/lang/Math:

```
0x00000000 0x004206A8 0x00420720 0x00420740 0x00420760 0x00420780 0x004207B0
0x00421208 0x00421270 0x00421290 0x004212B0 0x004213C8 0x00421458 0x00421478
0x00000000 0x41589DE0 0x00000000 0x4158B340 0x00000000 0x00000000 0x00000000
0x4158ACE8 0x00000000 0x4152F018 0x00000000 0x00000000 0x00000000
```

Trailer record 1

Trailer record 1 is a single record containing record counts.

```
// Breakdown - Classes: <class record count, in decimal>,
Objects: <object record count, in decimal>,
ObjectArrays: <object array record count, in decimal>,
PrimitiveArrays: <primitive array record count, in decimal>
```

Example:

```
// Breakdown - Classes: 321, Objects: 3718, ObjectArrays: 169,
PrimitiveArrays: 2141
```

Trailer record 2

Trailer record 2 is a single record containing totals.

```
// EOF:  Total 'Objects',Refs(null) :
<total object count, in decimal>,
<total reference count, in decimal>
(,total null reference count, in decimal>)
```

Example:

```
// EOF:  Total 'Objects',Refs(null) : 6349,23240(7282)
```

Java VM type signatures

The Java VM type signatures are abbreviations of the Java types are shown in the following table:

Java VM type signatures	Java type
Z	boolean
B	byte
C	char
S	short
I	int
J	long
F	float
D	double
L *<fully qualified-class>* ;	*<fully qualified-class>*
[*<type>*	*<type>*[ ] (array of *<type>*)
(*<arg-types>*) *<ret-type>*	method

Using system dumps and the dump viewer

The JVM can generate native system dumps, also known as core dumps, under configurable conditions. System dumps are typically large. Most tools used to analyze system dumps are also platform-specific.Use the **gdb** tool to analyze a system dump on Linux.

The IBM SDK for Java V7 User guide contains useful guidance on using system dumps and the dump viewer, covering:

- Overview of system dumps
- System dump defaults
- Using the dump viewer
  - Using **jextract**
  - Problems to tackle with the dump viewer
  - Commands available in **jdmpview**
  - Example session
  - **jdmpview** commands quick reference

You can find this information here: IBM SDK for Java 7 - Using system dumps and the dump viewer.

Supplementary information for IBM WebSphere Real Time for RT Linux:

Using **`jextract`**

When processing a system dump from a real-time JVM, you must include the **`-Xrealtime`** option. For example:

```
jextract -Xrealtime <core file name> [<zip_file>]
```

When you run **`jextract`** on a JVM that is different from the one for which the dump was produced you see the following error messages:

```
J9RAS.buildID is incorrect (found e8801ed67d21c6be, expecting eb4173107d21c673).
This version of jextract is incompatible with this dump.
Failure detected during jextract, see previous message(s).
```

Similarly, this message is also produced if you were running Java with the standard JVM, but used the **`-Xrealtime`** option when processing the dump with **`jextract`**.

Commands available in **`jdmpview`**

`jdmpview` is an interactive, command-line tool to explore the information from a JVM system dump and perform various analytic functions.

`info jitm`
Lists AOT and JIT compiled methods and their addresses:
- Method name and signature
- Method start address
- Method end address

For all other command options,see the IBM SDK for Java V7 User guide.

Tracing Java applications and the JVM

JVM trace is a trace facility that is provided in IBM WebSphere Real Time for RT Linux with minimal affect on performance. In most cases, the trace data is kept in a compact binary format, that can be formatted with the Java formatter that is supplied.

Tracing is enabled by default, together with a small set of trace points going to memory buffers. You can enable tracepoints at run time by using levels, components, group names, or individual tracepoint identifiers.

The IBM SDK for Java V7 User guide contains detailed information on tracing applications, covering:
- What can be traced
- Types of tracepoint
- Default tracing
- Recording trace data
- Controlling the trace
- Tracing Java applications
- Tracing Java methods

When tracing IBM WebSphere Real Time for RT Linux you must correctly invoke the real-time JVM when including the trace options. For example, when specifying trace options, type:

```
java -Xrealtime -Xtrace:<options>
```

You can find the IBM SDK for Java V7 information here: Tracing Java applications and the JVM.

JIT and AOT problem determination

You can use command-line options to help diagnose JIT and AOT compiler problems and to tune performance.

Although IBM WebSphere Real Time for RT Linux shares some common components with the IBM SDK for Java V7, the behavior of JIT and AOT is different. This section covers troubleshooting for JIT and AOT issues on IBM WebSphere Real Time for RT Linux.

Diagnosing a JIT or AOT problem

Occasionally, valid bytecodes might compile into invalid native code, causing the Java program to fail. By determining whether the JIT or AOT compiler is faulty and, if so, *where* it is faulty, you can provide valuable help to the Java service team.

About this task

To determine what methods are compiled when the shared class cache is populated, use the **`-Xaot:verbose`** option on the admincache command-line. For example:

```
admincache -Xrealtime -Xaot:verbose -populate -aot my.jar -cp <My Class Path>
```

This section describes how you can determine if your problem is compiler-related. This section also suggests some possible workarounds and debugging techniques for solving compiler-related problems.

Disabling the JIT or AOT compiler:

If you suspect that a problem is occurring in the JIT or AOT compiler, disable compilation to see if the problem remains. If the problem still occurs, you know that the compiler is not the cause of it.

About this task

The JIT compiler is enabled by default. The AOT compiler is also enabled, but, is not active unless shared classes have been enabled. For efficiency reasons, not all methods in a Java application are compiled. The JVM maintains a call count for each method in the application; every time a method is called and interpreted, the call count for that method is incremented. When the count reaches the compilation threshold, the method is compiled and executed natively.

The call count mechanism spreads compilation of methods throughout the life of an application, giving higher priority to methods that are used most frequently. Some infrequently used methods might never be compiled at all. As a result, when a Java program fails, the problem might be in the JIT or AOT compiler or it might be elsewhere in the JVM.

The first step in diagnosing the failure is to determine *where* the problem is. To do this, you must first run your Java program in purely interpreted mode (that is, with the JIT and AOT compilers disabled).

Procedure

1. Remove any **`-Xjit`** and **`-Xaot`** options (and accompanying parameters) from your command line.
2. Use the **`-Xint`** command-line option to disable the JIT and AOT compilers. For performance reasons, do not use the **`-Xint`** option in a production environment.

What to do next

Running the Java program with the compilation disabled leads to one of the following situations:

- The failure remains. The problem is not in the JIT or AOT compiler. In some cases, the program might start failing in a different manner; nevertheless, the problem is not related to the compiler.
- The failure disappears. The problem is most likely in the JIT or AOT compiler.

  If you are not using shared classes, the JIT compiler is at fault. If you are using shared classes, you must determine which compiler is at fault by running your application with only JIT compilation enabled. Run your application with the **`-Xnoaot`** option instead of the **`-Xint`** option. This leads to one of the following situations:
  - The failure remains. The problem is in the JIT compiler. You can also use the **`-Xnojit`** instead of the **`-Xnoaot`** option to ensure that only the JIT compiler is at fault.
  - The failure disappears. The problem is in the AOT compiler.

Selectively disabling the JIT compiler:

If your Java program failure points to a problem with the JIT compiler, you can try to narrow down the problem further.

About this task

By default, the JIT compiler optimizes methods at various optimization levels. Different selections of optimizations are applied to different methods, based on their call counts. Methods that are called more frequently are optimized at higher levels. By changing JIT compiler parameters, you can control the optimization level at which methods are optimized. You can determine whether the optimizer is at fault and, if it is, which optimization is problematic.

You specify JIT parameters as a comma-separated list, appended to the **`-Xjit`** option. The syntax is **-Xjit**:*<param1>*,*<param2>*=*<value>*. For example:

```
java -Xjit:verbose,optLevel=noOpt HelloWorld
```

runs the HelloWorld program, enables verbose output from the JIT, and makes the JIT generate native code without performing any optimizations.

Follow these steps to determine which part of the compiler is causing the failure:

Procedure

1. Set the JIT parameter **`count=0`** to change the compilation threshold to zero. This parameter causes each Java method to be compiled before it is run. Use **`count=0`** only when diagnosing problems, because a lot more methods are compiled, including methods that are used infrequently. The extra compilation uses more computing resources and slows down your application. With

count=0, your application fails immediately when the problem area is reached. In some cases, using **count=1** can reproduce the failure more reliably.

2. Add **disableInlining** to the JIT compiler parameters. **disableInlining** disables the generation of larger and more complex code. If the problem no longer occurs, use **disableInlining** as a workaround while the Java service team analyzes and fixes the compiler problem.
3. Decrease the optimization levels by adding the **optLevel** parameter, and run the program again until the failure no longer occurs, or you reach the "noOpt" level. For a JIT compiler problem, start with "scorching" and work down the list. The optimization levels are, in decreasing order:
   a. scorching
   b. veryHot
   c. hot
   d. warm
   e. cold
   f. noOpt

What to do next

If one of these settings causes your failure to disappear, you have a workaround that you can use. This workaround is temporary while the Java service team analyze and fix the compiler problem. If removing **disableInlining** from the JIT parameter list does not cause the failure to reappear, do so to improve performance. Follow the instructions in "Locating the failing method" to improve the performance of the workaround.

If the failure still occurs at the "noOpt" optimization level, you must disable the JIT compiler as a workaround.

Locating the failing method:

When you have determined the lowest optimization level at which the JIT or AOT compiler must compile methods to trigger the failure, you can find out which part of the Java program, when compiled, causes the failure. You can then instruct the compiler to limit the workaround to a specific method, class, or package, allowing the compiler to compile the rest of the program as usual. For JIT compiler failures, if the failure occurs with **-Xjit:optLevel=noOpt**, you can also instruct the compiler to not compile the method or methods that are causing the failure at all.

Before you begin

If you see error output like this example, you can use it to identify the failing method:

```
Unhandled exception
Type=Segmentation error vmState=0x00000000
Target=2_30_20050520_01866_BHdSMr (Linux 2.4.21-27.0.2.EL)
CPU=s390x (2 logical CPUs) (0x7b6a8000 RAM)
J9Generic_Signal_Number=00000004 Signal_Number=0000000b Error_Value=4148bf20 Signal_Code=00000001
Handler1=00000100002ADB14 Handler2=00000100002F480C InaccessibleAddress=0000000000000000
gpr0=0000000000000006 gpr1=0000000000000006 gpr2=0000000000000000 gpr3=0000000000000006
gpr4=0000000000000001 gpr5=0000000080056808 gpr6=0000010002BCCA20 gpr7=0000000000000000
......
Compiled_method=java/security/AccessController.toArrayOfProtectionDomains([Ljava/lang/Object;
Ljava/security/AccessControlContext;)[Ljava/security/ProtectionDomain;
```

The important lines are:

vmState=0x00000000
Indicates that the code that failed was not JVM runtime code.

Module= or Module_base_address=
Not in the output (might be blank or zero) because the code was compiled by the JIT, and outside any DLL or library.

Compiled_method=
Indicates the Java method for which the compiled code was produced.

About this task

If your output does not indicate the failing method, follow these steps to identify the failing method:

Procedure

1. Run the Java program with the JIT parameters **verbose** and **vlog=**<*filename*> added to the **-Xjit** or **-Xaot** option. With these parameters, the compiler lists compiled methods in a log file named `<filename>.<date>.<time>.<pid>`, also called a *limit file*. A typical limit file contains lines that correspond to compiled methods, like:

   ```
   + (hot) java/lang/Math.max(II)I @ 0x10C11DA4-0x10C11DDD
   ```

   Lines that do not start with the plus sign are ignored by the compiler in the following steps and you can remove them from the file. Methods for which AOT code is loaded from the shared class cache start with `+ (AOT load)`.
2. Run the program again with the JIT or AOT parameter **limitFile=**(<*filename*>,<*m*>,<*n*>), where <*filename*> is the path to the limit file, and <*m*> and <*n*> are line numbers indicating the first and the last methods in the limit file that should be compiled. The compiler compiles only the methods listed on lines <*m*> to <*n*> in the limit file. Methods not listed in the limit file and methods listed on lines outside the range are not compiled and no AOT code in the shared data cache for those methods will be loaded. If the program no longer fails, one or more of the methods that you have removed in the last iteration must have been the cause of the failure.
3. Repeat this process using different values for <*m*> and <*n*>, as many times as necessary, to find the minimum set of methods that must be compiled to trigger the failure. By halving the number of selected lines each time, you can perform a binary search for the failing method. Often, you can reduce the file to a single line.

What to do next

When you have located the failing method, you can disable the JIT or AOT compiler for the failing method only. For example, if the method java/lang/Math.max(II)I causes the program to fail when JIT-compiled with **optLevel=hot**, you can run the program with:

```
-Xjit:{java/lang/Math.max(II)I}(optLevel=warm,count=0)
```

to compile only the failing method at an optimization level of “warm”, but compile all other methods as usual.

If a method fails when it is JIT-compiled at “noOpt”, you can exclude it from compilation altogether, using the **exclude**={<*method*>} parameter:

```
-Xjit:exclude={java/lang/Math.max(II)I}
```

If a method causes the program to fail when AOT code is loaded from the shared data cache, exclude the method from AOT loading using the **exclude**={<*method*>} parameter:

```
-Xaot:exclude={java/lang/Math.max(II)I}
```

AOT methods are only compiled into the shared class cache during the **admincache** population step. Preventing AOT loading is the best diagnostic approach for problems with these methods.

Identifying JIT and AOT compilation failures:

For JIT compiler failures, analyze the error output to determine if a failure occurs when the JIT compiler attempts to compile a method.

If the JVM crashes, and you can see that the failure has occurred in the JIT library (`libj9jit26.so`), the JIT compiler might have failed during an attempt to compile a method.

If you see error output like this example, you can use it to identify the failing method:

```
Unhandled exception
Type=Segmentation error vmState=0x00050000
Target=2_30_20051215_04381_BHdSMr (Linux 2.4.21-32.0.1.EL)
CPU=ppc64 (4 logical CPUs) (0xebf4e000 RAM)
J9Generic_Signal_Number=00000004 Signal_Number=0000000b Error_Value=00000000 Signal_Code=00000001
Handler1=0000007FE05645B8 Handler2=0000007FE0615C20
R0=E8D4001870C00001 R1=0000007FF49181E0 R2=0000007FE2FBCEE0 R3=0000007FF4E60D70
R4=E8D4001870C00000 R5=0000007FE2E02D30 R6=0000007FF4C0F188 R7=0000007FE2F8C290
......
Module=/home/test/sdk/jre/bin/libj9jit26.so
Module_base_address=0000007FE29A6000
......
Method_being_compiled=com/sun/tools/javac/comp/Attr.visitMethodDef(Lcom/sun/tools/javac/tree/
JCTree$JCMethodDecl;)
```

The important lines are:

`vmState=0x00050000`
Indicates that the JIT compiler is compiling code. For a list of vmState code numbers, see the Javadump tags table in the IBM SDK for Java V7 User guide, http://publib.boulder.ibm.com/infocenter/java7sdk/v7r0/topic/com.ibm.java.lnx.70.doc/diag/tools/javadump_tags_info.html.

`Module=/home/test/sdk/jre/bin/libj9jit26.so`
Indicates that the error occurred in `libj9jit26.so`, the JIT compiler module.

`Method_being_compiled=`
Indicates the Java method being compiled.

If your output does not indicate the failing method, use the **verbose** option with the following additional settings:

```
-Xjit:verbose={compileStart|compileEnd}
```

These **verbose** settings report when the JIT or AOT compiler starts to compile a method, and when it ends. If the JIT or AOT compiler fails on a particular method (that is, it starts compiling, but crashes before it can end), use the **exclude** parameter on the **-Xjit** or **-Xaot** command-line option to exclude it from JIT or AOT compilation (refer to "Locating the failing method" on page 130). For problems with AOT compilation, destroy your shared class cache before using the

exclude option. If excluding the method prevents the crash, you have a workaround that you can use while the service team corrects your problem.

Identifying AOT compilation failures in non-real-time-mode:

AOT problem determination in non-real-time-mode is very similar to JIT problem determination.

About this task

As with the JIT, first run your application with **-Xnoaot**, which ensures that the AOT'ed code is not used when running the application.

If this fixes the problem, rebuild the AOT jar files using the same technique that is described in "Locating the failing method" on page 130, providing the **-Xaot** option at AOT build time, instead of application run time.

Identifying AOT compilation failures in real-time mode:

AOT problem determination uses the admincache tool to locate the problem.

About this task

In contrast to JIT compilation failures, which occur at application run time, AOT compilation failures occur during the admincache population step.

To find where the problem occurs, run the admincache tool with the **-Xnoaot** option. This ensures that the application does not run with ahead-of-time compiled code.

If using the **-Xnoaot** option fixes the problem, examine the output of the original crash. The output provides information identifying which method is the cause of the problem. Look for a line similar to:

```
Method_being_compiled=myAppClass.main(Ljava/lang/String;)V
```

To avoid the problem, this method must be excluded from ahead-of-time compilation. Do this by adding an option to the admincache command line, similar to the following example:

```
-Xaot:exclude={myAppClass.main(Ljava/lang/String;)V}
```

This exclusion prevents AOT compilation of the problem method.

Performance of short-running applications

The IBM JIT compiler is tuned for long-running applications typically used on a server. You can use the **-Xquickstart** command-line option in non-real-time mode to improve the performance of short-running applications, especially for applications in which processing is not concentrated into a few methods.

-Xquickstart causes the JIT compiler to use a lower optimization level by default and to compile fewer methods. Performing fewer compilations more quickly can improve application startup time. When the AOT compiler is active (both shared classes and AOT compilation enabled), **-Xquickstart** causes all methods selected for compilation to be AOT compiled, which improves the startup time of subsequent runs. **-Xquickstart** might degrade performance if it is used with long-running applications that contain methods using a large amount of processing resource. The implementation of **-Xquickstart** is subject to change in future releases.

You can also try improving startup times by adjusting the JIT threshold (using trial and error). See “Selectively disabling the JIT compiler” on page 129 for more information.

-Xquickstart has no effect on AOT code usage with **-Xrealtime**.

JVM behavior during idle periods

You can reduce the CPU cycles consumed by an idle JVM by using the **-XsamplingExpirationTime** option to turn off the JIT sampling thread.

The JIT sampling thread profiles the running Java application to discover commonly used methods. The memory and processor usage of the sampling thread is negligible, and the frequency of profiling is automatically reduced when the JVM is idle.

In some circumstances, you might want no CPU cycles consumed by an idle JVM. To do so, specify the **-XsamplingExpirationTime**<*time*> option. Set <*time*> to the number of seconds for which you want the sampling thread to run. Use this option with care; after it is turned off, you cannot reactivate the sampling thread. Allow the sampling thread to run for long enough to identify important optimizations.

The Diagnostics Collector

The Diagnostics Collector gathers the Java diagnostic files for a problem event.

Gathering the files that are needed by IBM service can reduce the time taken to solve reported problems. The IBM SDK for Java V7 user guide contains detailed information about using the Diagnostics Collector.

You can find this information here: IBM SDK for Java 7 - The Diagnostics Collector.

Garbage Collector diagnostic data

This section describes how to diagnose garbage collection problems.

The IBM SDK for Java V7 User guide contains useful guidance on diagnosing garbage collector problems, covering:
- Verbose garbage collection logging
- Tracing garbage collection using **-Xtgc**

You can find this information here: IBM SDK for Java 7 - Garbage Collector diagnostic data.

Supplementary information about the IBM WebSphere Real Time for RT Linux Metronome Garbage Collector is provided in the following sections.

Troubleshooting the Metronome Garbage Collector

Using the command-line options, you can control the frequency of Metronome garbage collection, out of memory exceptions, and the Metronome behavior on explicit system calls.

Using verbose:gc information:

You can use the **-verbose:gc** option with the **-Xgc:verboseGCCycleTime=N** option to write information to the console about Metronome Garbage Collector activity. Not all XML properties in the **-verbose:gc** output from the standard JVM are created or apply to the output of Metronome Garbage Collector.

Use the **-verbose:gc** option to view the minimum, maximum, and mean free space in the heap. In this way, you can check the level of activity and use of the heap, and then adjust the values if necessary. The **-verbose:gc** option writes Metronome statistics to the console.

The **-Xgc:verboseGCCycleTime=N** option controls the frequency of retrieval of the information. It determines the time in milliseconds that the summaries are dumped. The default value for N is 1000 milliseconds. The cycle time does not mean that the summary is dumped precisely at that time, but when the last garbage collection event that meets this time criterion passes. The collection and display of these statistics can increase Metronome Garbage Collector pause times and, as N gets smaller, the pause times can become large.

A quantum is a single period of Metronome Garbage Collector activity, causing an interruption or pause time for an application.

Example of verbose:gc output

Enter:

```
java -Xrealtime -verbose:gc -Xgc:verboseGCCycleTime=N myApplication
```

This example shows the initial output of verbose:gc, which contains the version and garbage collection settings:

```
<verbosegc
xmlns="http://www.ibm.com/j9/verbosegc" version="R26_Java726_GA_20110716_0946_B87065">

<initialized id="1" timestamp="2011-07-27T14:17:52.277">
  <attribute name="gcPolicy" value="-Xgcpolicy:metronome"/>
  <attribute name="maxHeapSize" value="0x5800000"/>
  <attribute name="initialHeapSize" value="0x4000000"/>
  <attribute name="compressedRefs" value="false"/>
  <attribute name="pageSize" value="0x1000"/>
  <attribute name="requestedPageSize" value="0x1000"/>
  <attribute name="gcthreads" value="1"/>
  <region>
    <attribute name="regionSize" value="16384"/>
    <attribute name="regionCount" value="4096"/>
    <attribute name="arrayletLeafSize" value="2048"/>
  </region>
  <metronome>
    <attribute name="beatsPerMeasure" value="500"/>
    <attribute name="timeInterval" value="10000"/>
    <attribute name="targetUtilization" value="70"/>
    <attribute name="trigger" value="0x2000000"/>
    <attribute name="headRoom" value="0x100000"/>
  </metronome>
  <system>
    <attribute name="physicalMemory" value="12507463680"/>
    <attribute name="numCPUs" value="8"/>
    <attribute name="architecture" value="x86"/>
    <attribute name="os" value="Linux"/>
    <attribute name="osVersion" value="2.6.24.7-75ibmrt2.18"/>
  </system>
  <vmargs>
```

```
    <vmarg
name="-Xoptionsfile=/my_dir/pxi3270hrt-20110719_02/sdk/jre/lib/i386/realtime/options.default"/>
    <vmarg name="-Xjcl:jclse7b_26"/>
    <vmarg
name="-Dcom.ibm.oti.vm.bootstrap.library.path=/my_dir/pxi3270hrt-20110719_02/sdk/jre/lib/i386/realtime:/
my_dir/pxi3270hrt-2011071..."/>
    <vmarg
name="-Dsun.boot.library.path=/my_dir/pxi3270hrt-20110719_02/sdk/jre/lib/i386/realtime:/my_dir/
pxi3270hrt-20110719_02/sdk/jre/lib..."/>
    <vmarg
name="-Djava.library.path=/my_dir/pxi3270hrt-20110719_02/sdk/jre/lib/i386/realtime:/my_dir/
pxi3270hrt-20110719_02/sdk/jre/lib/i38..."/>
    <vmarg name="-Djava.home=/my_dir/pxi3270hrt-20110719_02/sdk/jre"/>
    <vmarg name="-Djava.ext.dirs=/my_dir/pxi3270hrt-20110719_02/sdk/jre/lib/ext"/>
    <vmarg name="-Duser.dir=/my_dir/pxi3270hrt-20110719_02/sdk/jre/bin"/>
    <vmarg name="_j2se_j9=1120000"
value="F76FF700"/>
    <vmarg name="-Djava.runtime.version=pxi3270hrt-20110719_02"/>
    <vmarg name="-Djava.class.path=."/>
    <vmarg name="-Xrealtime"/>
    <vmarg name="-verbose:gc"/>
    <vmarg name="-Dsun.java.launcher=SUN_STANDARD"/>
    <vmarg name="-Dsun.java.launcher.pid=5543"/>
    <vmarg name="_port_library" value="F7701B80"/>
    <vmarg name="_bfu_java" value="F77029A8"/>
    <vmarg name="_org.apache.harmony.vmi.portlib" value="08051DA0"/>
  </vmargs>
</initialized>
```

When garbage collection is triggered, a `trigger start` event occurs, followed by any number of `heartbeat` events, then a `trigger end` event when the trigger is satisfied. This example shows a triggered garbage collection cycle as verbose:gc output:

```
<trigger-start id="25" timestamp="2011-07-12T09:32:04.503" />

<cycle-start id="26" type="global" contextid="26" timestamp="2011-07-12T09:32:04.503" intervalms="984.285" />

<gc-op id="27" type="heartbeat" contextid="26" timestamp="2011-07-12T09:32:05.209">
  <quanta quantumCount="321" quantumType="mark" minTimeMs="0.367" meanTimeMs="0.524" maxTimeMs="1.878"
    maxTimestampMs="598704.070" />
  <exclusiveaccess-info minTimeMs="0.006" meanTimeMs="0.062" maxTimeMs="0.147" />
  <free-mem type="heap" minBytes="99143592" meanBytes="114374153" maxBytes="134182032" />
  <free-mem type="immortal" minBytes="44234538" meanBytes="60342344" maxBytes="61219900"/>
  <thread-priority maxPriority="11" minPriority="11" />
</gc-op>

<gc-op id="28" type="heartbeat" contextid="26" timestamp="2011-07-12T09:32:05.458">
  <quanta quantumCount="115" quantumType="sweep" minTimeMs="0.430" meanTimeMs="0.471" maxTimeMs="0.511"
    maxTimestampMs="599475.654" />
  <exclusiveaccess-info minTimeMs="0.007" meanTimeMs="0.067" maxTimeMs="0.173" />
  <classunload-info classloadersunloaded=9 classesunloaded=156  />
  <references type="weak" cleared="660" />
  <free-mem type="heap" minBytes="24281568" meanBytes="55456028" maxBytes="87231320" />
  <free-mem type="immortal" minBytes="38234500" meanBytes="41736440" maxBytes="42233458"/>
  <thread-priority maxPriority="11" minPriority="11" />
</gc-op>

<gc-op id="29" type="syncgc" timems="136.945" contextid="26" timestamp="2011-07-12T09:32:06.046">
  <syncgc-info reason="out of memory" exclusiveaccessTimeMs="0.006" threadPriority="11" />
  <free-mem-delta type="heap" bytesBefore="21290752" bytesAfter="171963656" />
  <free-mem-delta type="immortal" bytesBefore="35735400" bytesAfter="35735400"/>
</gc-op>

<cycle-end id="30" type="global" contextid="26" timestamp="2011-07-12T09:32:06.046" />

<trigger-end id="31" timestamp="2011-07-12T09:32:06.046" />
```

The following event types can occur:

<trigger-start ...>
: The start of a garbage collection cycle, when the amount of used memory became higher than the trigger threshold. The default threshold is 50% of the heap. The `intervalms` attribute is the interval between the previous `trigger end` event (with id-1) and this `trigger start` event.

<trigger-end ...>
: A garbage collection cycle successfully lowered the amount of used memory beneath the trigger threshold. If a garbage collection cycle ended, but used memory did not drop beneath the trigger threshold, a new garbage collection cycle is started with the same context ID. For each `trigger start` event, there is a matching `trigger end` event with same context ID. The `intervalms` attribute is the interval between the previous `trigger start` event and the current `trigger end` event. During this time, one or more garbage collection cycles will have completed until used memory has dropped beneath the trigger threshold.

<gc-op id="28" type="heartbeat"...>
: A periodic event that gathers information (on memory and time) about all garbage collection quanta for the time covered. A heartbeat event can occur only between a matching pair of `trigger start` and `trigger end` events; that is, while an active garbage collection cycle is in process. The `intervalms` attribute is the interval between the previous heartbeat event (with `id` -1) and this heartbeat event.

<gc-op id="29" type="syncgc"...>
: A synchronous (nondeterministic) garbage collection event. See "Synchronous garbage collections" on page 138

The XML tags in this example have the following meanings:

<quanta ...>
: A summary of quantum pause times during the heartbeat interval, including the length of the pauses in milliseconds.

<free-mem type="heap" ...>
: A summary of the amount of free heap space during the heartbeat interval, sampled at the end of each garbage collection quantum.

<classunload-info classloadersunloaded=9 classesunloaded=156 />
: The number of classloaders and classes unloaded during the heartbeat interval.

<references type="weak" cleared="660 />
: The number and type of Java reference objects that were cleared during the heartbeat interval.

Note:

- If only one garbage collection quantum occurred in the interval between two heartbeats, the free memory is sampled only at the end of this one quantum. Therefore the minimum, maximum, and mean amounts given in the heartbeat summary are all equal.
- The interval between two heartbeat events might be significantly larger than the cycle time specified if the heap is not full enough to require garbage collection activity. For example, if your program requires garbage collection activity only once every few seconds, you are likely to see a heartbeat only once every few seconds.

- It is possible that the interval might be significantly larger than the cycle time specified because the garbage collection has no work on a heap that is not full enough to warrant garbage collection activity. For example, if your program requires garbage collection activity only once every few seconds, you are likely to see a heartbeat only once every few seconds.

  If an event such as a synchronous garbage collection or a priority change occurs, the details of the event and any pending events, such as heartbeats, are immediately produced as output.
- If the maximum garbage collection quantum for a given period is too large, you might want to reduce the target utilization using the **-Xgc:targetUtilization** option. This action gives the Garbage Collector more time to work. Alternatively, you might want to increase the heap size with the **-Xmx** option. Similarly, if your application can tolerate longer delays than are currently being reported, you can increase the target utilization or decrease the heap size.
- The output can be redirected to a log file instead of the console with the **-Xverbosegclog:<file>** option; for example, **-Xverbosegclog:out** writes the **-verbose:gc** output to the file *out*.
- The priority listed in `thread-priority` is the underlying operating system thread priority, not a Java thread priority.

Synchronous garbage collections

An entry is also written to the **-verbose:gc** log when a synchronous (nondeterministic) garbage collection occurs. This event has three possible causes:
- An explicit `System.gc()` call in the code.
- The JVM runs out of memory then performs a synchronous garbage collection to avoid an OutOfMemoryError condition.
- The JVM shuts down during a continuous garbage collection. The JVM cannot cancel the collection, so it completes the collection synchronously, and then exits.

An example of a `System.gc()` entry is:

```
<gc-op id="9" type="syncgc" timems="12.92" contextid="8" timestamp="2011-07-12T09:41:40.808">
  <syncgc-info reason="system GC" totalBytesRequested="260" exclusiveaccessTimeMs="0.009"
    threadPriority="11" />
  <free-mem-delta type="heap" bytesBefore="22085440" bytesAfter="136023450" />
  <free-mem-delta type="immortal" bytesBefore="62324800" bytesAfter="62324800"/>
  <classunload-info classloadersunloaded="54" classesunloaded="234" />
  <references type="soft" cleared="21" dynamicThreshold="29" maxThreshold="32" />
  <references type="weak" cleared="523" />
  <finalization enqueued="124" />
</gc-op>
```

An example of a synchronous garbage collection entry as a result of the JVM shutting down is:

```
<gc-op id="24" type="syncgc" timems="6.439" contextid="19" timestamp="2011-07-12T09:43:14.524">
  <syncgc-info reason="VM shut down" exclusiveaccessTimeMs="0.009" threadPriority="11" />
  <free-mem-delta type="heap" bytesBefore="56182430" bytesAfter="151356238" />
  <free-mem-delta type="immortal" bytesBefore="23659200" bytesAfter="23659200"/>
  <classunload-info classloadersunloaded="14" classesunloaded="276" />
  <references type="soft" cleared="154" dynamicThreshold="29" maxThreshold="32" />
  <references type="weak" cleared="53" />   <finalization enqueued="34" />
</gc-op>
```

The XML tags and attributes in this example have the following meanings:

<gc-op id="9" type="syncgc" timems="6.439" ...
: This line indicates that the event type is a synchronous garbage collection. The `timems` attribute is the duration of the synchronous garbage collection in milliseconds.

<syncgc-info reason="..."/>
: The cause of the synchronous garbage collection.

<free-mem-delta.../>
: The free Java heap memory before and after the synchronous garbage collection in bytes.

<finalization .../>
: The number of objects awaiting finalization.

<classunload-info .../>
: The number of classloaders and classes unloaded during the heartbeat interval.

<references type="weak" cleared="53" .../>
: The number and type of Java reference objects that were cleared during the heartbeat interval.

Synchronous garbage collection due to out-of-memory conditions or VM shutdown can happen only when the Garbage Collector is active. It has to be preceded by a `trigger start` event, although not necessarily immediately. Some heartbeat events probably occur between a `trigger start` event and the `synchgc` event. Synchronous garbage collection caused by `System.gc()` can happen at any time.

Tracking all GC quanta

Individual GC quanta can be tracked by enabling the `GlobalGCStart` and `GlobalGCEnd` tracepoints. These tracepoints are produced at the beginning and end of all Metronome Garbage Collector activity including synchronous garbage collections. The output for these tracepoints looks similar to:

```
03:44:35.281 0x833cd00 j9mm.52 - GlobalGC start: globalcount=3

03:44:35.284 0x833cd00 j9mm.91 - GlobalGC end: workstackoverflow=0 overflowcount=0
```

Priority changes

In addition to summaries, an entry is written to the **`-verbose:gc`** log when the Garbage Collector thread priority changes (because the application changed thread priorities, or because one or more threads in an application ended). The priority listed is the underlying OS thread priority, not a Java thread priority. An example of a Garbage Collector thread priority change entry is:

```
<gc type="heartbeat" id="73" timestamp="Feb 26 13:11:35 2007" intervalms"1001.754">
  <summary quantumcount="240">
    <quantum minms="0.022" meanms="0.984" maxms="1.011" />
    <classunloading classloaders="11" classes="17" />
    <heap minfree="202833920" meanfree="214184823" maxfree="221102080" />
    <thread-priority maxPriority="11" minPriority="11" />
  </summary>
</gc>
```

Priority changes can be tracked in Real Time by producing the trace point information relating to Garbage Collector thread priorities. This output looks similar to:

```
15:58:25.493*0x8286e00    j9mm.102       - setGCThreadPriority() called with
newGCThreadPriority = 11
```

This output can be enabled by using the ID, as follows:
-Xtrace:iprint=tpnid{j9mm.102}

Out-of-memory entries

When one of the memory areas runs out of free space, an entry is written to the **-verbose:gc** log before the OutOfMemoryError exception is thrown. An example of this output is:

```
<out-of-memory id="71" timestamp="2011-07-23T08:32:51.435" memorySpaceName="Scoped"
  memorySpaceAddress="080EED9C"/>
```

By default a Javadump is produced as a result of an OutOfMemoryError exception. This dump contains information about the memory areas used by your program. Together with the `J9MemorySpace` value given in the **-verbose:gc** output, you can use this information in the dump to identify the particular memory area that ran out of space:

```
NULL           id         start      end        size       space/region
1STHEAPSPACE   0x080EED9C     --         --         --     Scoped
1STHEAPREGION  0x0810C570 0xF1B09028 0xF2B09028 0x01000000 Scoped/Region
NULL
1STHEAPTOTAL   Total memory:          16777216 (0x01000000)
1STHEAPINUSE   Total memory in use:     625952 (0x00098D20)
1STHEAPFREE    Total memory free:     16151264 (0x00F672E0)
```

In the previous example, the memory space ID given in the -verbose:gc output (0x080EED9C) can be matched to the ID of the Scoped memory area of the Java dump. This match can be useful if you have several scopes and need to identify which one has gone out of memory, because the -verbose:gc output only indicates whether the OutOfMemoryError occurred in immortal, scoped, or heap memory.

Metronome Garbage Collector behavior in out-of-memory conditions:

By default, the Metronome Garbage Collector triggers an unlimited, nondeterministic garbage collection when the JVM runs out of memory. To prevent nondeterministic behavior, use the **-Xgc:noSynchronousGCOnOOM** option to throw an OutOfMemoryError when the JVM runs out of memory.

The default unlimited collection runs until all possible garbage is collected in a single operation. The pause time required is usually many milliseconds greater than a normal metronome incremental quantum.

Related information:
Using -Xverbose:gc to analyze synchronous garbage collections

Metronome Garbage Collector behavior on explicit System.gc() calls:

If a garbage collection cycle is in progress, the Metronome Garbage Collector completes the cycle in a synchronous way when `System.gc()` is called. If no garbage collection cycle is in progress, a full synchronous cycle is performed when `System.gc()` is called. Use `System.gc()` to clean up the heap in a controlled manner. It is a nondeterministic operation because it performs a complete garbage collection before returning.

Some applications call vendor software that has System.gc() calls where it is not acceptable to create these nondeterministic delays. To disable all System.gc() calls use the **-Xdisableexplicitgc** option.

The verbose garbage collection output for a System.gc() call has a reason of "system garbage collect" and is likely to have a long duration:

```
<gc-op id="9" type="syncgc" timems="6.439" contextid="8" timestamp="2011-07-12T09:41:40.808">
  <syncgc-info reason="VM shut down" exclusiveaccessTimeMs="0.009" threadPriority="11"/>
  <free-mem-delta type="heap" bytesBefore="126082300" bytesAfter="156085440"/>
  <free-mem-delta type="immortal" bytesBefore="5129096" bytesAfter="5129096"/>
  <classunload-info classloadersunloaded="14" classesunloaded="276"/>
  <references type="soft" cleared="154" dynamicThreshold="29" maxThreshold="32"/>
  <references type="weak" cleared="53"/>
  <finalization enqueued="34"/>
</gc-op>
```

Shared classes diagnostic data

Understanding how to diagnose problems that might occur helps you to use shared classes mode.

For an introduction to shared classes, see Class data sharing between JVMs.

The IBM SDK for Java V7 User guide contains useful guidance on diagnosing problems with shared classes, covering:
- Deploying shared classes
- Dealing with runtime bytecode modification
- Understanding dynamic updates
- Using the Java Helper API
- Understanding shared classes diagnostic output
- Debugging problems with shared classes

You can find this information here: IBM SDK for Java 7 - Shared classes diagnostic data.

Some of the material in the IBM SDK for Java V7 User guide might not be applicable to IBM WebSphere Real Time for RT Linux. In particular:
- In real-time mode, applications have only read access to shared class caches, not read-write access.
- Caches can be modified exclusively using the **admincache** tool.
- Nonpersistent caches are not available in real-time mode.

Using the JVMTI

JVMTI is a two-way interface that allows communication between the JVM and a native agent. It replaces the JVMDI and JVMPI interfaces.

JVMTI allows third parties to develop debugging, profiling, and monitoring tools for the JVM. The interface contains mechanisms for the agent to notify the JVM about the kinds of information it requires. The interface also provides a means of receiving the relevant notifications. Several agents can be attached to a JVM at any one time.

The IBM SDK for Java V7 User guide contains detailed information about using JVMTI, including an API reference section on IBM extensions to JVMTI.

You can find this information here: IBM SDK for Java 7 - Using JVMTI.

Using the Diagnostic Tool Framework for Java

The Diagnostic Tool Framework for Java (DTFJ) is a Java application programming interface (API) from IBM used to support the building of Java diagnostics tools. DTFJ works with data from a system dump or Javadump.

The IBM SDK for Java V7 User guide contains detailed information about DTFJ. Follow this link: Using the Diagnostic Tool Framework for Java

Chapter 10. Reference

This set of topics lists the options and class libraries that can be used with WebSphere Real Time for RT Linux

Command-line options

You can specify options on the command line while you are starting Java. Default options have been chosen for best general use.

Specifying Java options and system properties

There are three ways to specify Java properties and system properties.

About this task

You can specify Java options and system properties in these ways. In order of precedence, they are:

1. By specifying the option or property on the command line. For example:
   ```
   java -Dmysysprop1=tcpip -Dmysysprop2=wait -Xdisablejavadump MyJavaClass
   ```
2. By creating a file that contains the options, and specifying this file on the command line using the **`-Xoptionsfile=`***<filename>* option.

   In the options file, specify each option on a new line; you can use the '\' character as a continuation character if you want a single option to span multiple lines. Use the '#' character to define comment lines. You cannot specify **-classpath** in an options file. Here is an example of an options file:
   ```
   #My options file
   -X<option1>
   -X<option2>=\
   <value1>,\
   <value2>
   -D<sysprop1>=<value1>
   ```
3. By creating an environment variable called **`IBM_JAVA_OPTIONS`** containing the options. For example:
   ```
   export IBM_JAVA_OPTIONS="-Dmysysprop1=tcpip -Dmysysprop2=wait -Xdisablejavadump"
   ```

The last option you specify on the command line has precedence over first option. For example, if you specify the options **`-Xint -Xjit myClass`**, the option **`-Xjit`** takes precedence over **`-Xint`**.

System properties

System properties are available to applications, and help provide information about the runtime environment.

`com.ibm.jvm.realtime`

This property enables Java applications to determine if they are running within a WebSphere Real Time for RT Linux environment.

If your application is running within the IBM WebSphere Real Time for RT Linux runtime environment, and was started with the **`-Xrealtime`** option, the **`com.ibm.jvm.realtime`** property has the value “`hard`”.

If your application is running within the IBM WebSphere Real Time for RT Linux runtime environment, but was not started with the **`-Xrealtime`** option, the **`com.ibm.jvm.realtime`** property is not set.

If your application is running within the IBM WebSphere Real Time runtime environment, the **`com.ibm.jvm.realtime`** property has the value "`soft`".

Standard options

The definitions for the standard options.

-agentlib:*<libname>***[=***<options>***]**
: Loads native agent library *<libname>*; for example **`-agentlib:hprof`**. For more information, specify **`-agentlib:jdwp=help`** and **`-agentlib:hprof=help`** on the command line.

-agentpath:*libname***[=***<options>***]**
: Loads native agent library by full path name.

-assert Prints help on assert-related options.

-cp or -classpath *<directories and .zip or .jar files separated by* **:>**
: Sets the search path for application classes and resources. If **`-classpath`** and **`-cp`** are not used and **`CLASSPATH`** is not set, the user classpath is, by default, the current directory (.).

-D<property_name>=*<value>*
: Sets a system property.

-help or -?
: Prints a usage message.

-javaagent:*<jarpath>***[=***<options>***]**
: Loads Java programming language agent. For more information, see the java.lang.instrument API documentation.

-jre-restrict-search
: Includes user private JREs in the version search.

-no-jre-restrict-search
: Excludes user private JREs in the version search.

-showversion
: Prints product version and continues.

-verbose:[*class***,***gc***,***dynload***,***sizes***,***stack***,***jni***]**
: Enables verbose output.

-verbose:class
: Writes an entry to stderr for each class that is loaded.

-verbose:gc
: See "Using verbose:gc information" on page 135.

-verbose:dynload
: Provides detailed information as each class is loaded by the JVM, including:
- The class name and package
- For class files that were in a `.jar` file, the name and directory path of the `.jar`
- Details of the size of the class and the time taken to load the class

The data is written out to stderr. An example of the output follows:

```
<Loaded java/lang/String from /myjdk/sdk/jre/lib/i386/
softrealtime/jclSC160/vm.jar>
<Class size 17258; ROM size 21080; debug size 0>
<Read time 27368 usec; Load time 782 usec; Translate time 927 usec>
```

Note: Classes loaded from the shared class cache do not appear in **`-verbose:dynload`** output. Use **`-verbose:class`** for information about these classes.

-verbose:sizes
Writes information to stderr describing the amount of memory used for the stacks and heaps in the JVM

-verbose:stack
Writes information to stderr describing Java and C stack usage.

-verbose:jni
Writes information to stderr describing the JNI services called by the application and JVM.

-version
Prints out version information for the non-real-time mode. When used with the -Xrealtime option, it prints out the version information for real-time mode.

-version:<*value*>
Requires the specified version to run.

-X Prints help on nonstandard options.

Non-standard options

Options that are prefixed by **`-X`** are nonstandard and subject to change without notice.

The IBM SDK for Java V7 User guide contains detailed information on non-standard options. You can find this information here: IBM SDK for Java 7 - Command-line options.

Changes from the IBM SDK for Java V7 User guide are documented in the following list:

| • **`-XX:-LazySymbolResolution`** is the default option on the RT Linux operating
| system, not **`-XX:+LazySymbolResolution`**.

Supplementary information for IBM WebSphere Real Time for RT Linux is provided in the following sections.

Real-time options

The definition of the **`-Xrealtime`** option used in WebSphere Real Time for RT Linux.

The following **-X** options are applicable in the WebSphere Real Time for RT Linux environment.

-Xrealtime
Starts the real-time mode. It is required if you want to run the Metronome Garbage Collector and use the Real-Time Specification for Java (RTSJ) services. If you do not specify this option, the JVM starts in non-real-time

mode equivalent to IBM SDK and Runtime Environment for Linux Platforms, Java 2 Technology, version 7.

The **`-Xrealtime`** option is interchangeable with **`-Xgcpolicy:metronome`**. You can specify either one to get real-time mode.

Ahead-of-time options

The definitions for the ahead-of-time options.

Purpose

No option specified:
Runs with the interpreter and dynamically compiled code. If AOT code is discovered, it is not used. Instead, it is dynamically compiled as required. This is particularly useful for non-real time and some real-time applications. This option provides optimal performance and throughput but can suffer non-deterministic delays at run time when compilation occurs.

-Xjit: This option is the same as default.

-Xint: Runs the interpreter only, ignores code written for AOT that might be found in a precompiled jar file, and does not run the dynamic compiler. This mode is not often required, other than for debugging problems that you suspect are related to compilation or for very short batch applications that do not derive benefit from compilation.

-Xnojit:
Runs the interpreter and uses code written for AOT if it is found in a precompiled jar file. It does not run the dynamic compiler. This mode works well for some real-time applications where you want to ensure that no non-deterministic delays occur at run time because of compilation. Code written for AOT can only be used when running with the **`-Xrealtime`** option. It is not supported when running in a standard JVM, that is, **`-Xrealtime`** is not specified.

Example

```
java -Xrealtime -Xnojit outputtest.jar.
```

Metronome Garbage Collector options

The definitions of the Metronome Garbage Collector options.

-Xgc:immortalMemorySize=*size*
Specifies the size of your immortal heap area. The default is 16 MB.

-Xgc:scopedMemoryMaximumSize=*size*
Specifies the size of your scoped memory heap area. The default is 8 MB.

-Xgc:synchronousGCOnOOM | -Xgc:nosynchronousGCOnOOM
One occasion when garbage collection occurs is when the heap runs out of memory. If there is no more free space in the heap, using **`-Xgc:synchronousGCOnOOM`** stops your application while garbage collection removes unused objects. If free space runs out again, consider decreasing the target utilization to allow garbage collection more time to complete. Setting **`-Xgc:nosynchronousGCOnOOM`** implies that when heap memory is full your application stops and issues an out-of-memory message. The default is **`-Xgc:synchronousGCOnOOM`**.

-Xnoclassgc
Disables class garbage collection. This option switches off garbage

collection of storage associated with Java classes that are no longer being used by the JVM. The default behavior is **`-Xnoclassgc`**.

-Xgc:targetUtilization=*N*
: Sets the application utilization to N%; the Garbage Collector attempts to use at most (100-N)% of each time interval. Reasonable values are in the range of 50-80%. Applications with low allocation rates might be able to run at 90%. The default is 70%.

  This example shows the maximum size of the heap memory is 30 MB. The garbage collector attempts to use 25% of each time interval because the target utilization for the application is 75%.

```
java -Xrealtime -Xmx30m -Xgc:targetUtilization=75 Test
```

-Xgc:threads=*N*
: Specifies the number of GC threads to run. The default is 1.

-Xgc:verboseGCCycleTime=N
: N is the time in milliseconds that the summary information should be dumped.

  Note: The cycle time does not mean that the summary information is dumped precisely at that time, but when the last garbage collection event that meets this time criterion passes.

-Xmx*<size>*
: Specifies the Java heap size. Unlike other garbage collection strategies, the real-time Metronome GC does not support heap expansion. There is not an initial or maximum heap size option. You can specify only the maximum heap size.

-Xthr:metronomeAlarm=os*xx*
: Controls the priority that the Metronome Garbage Collector alarm thread runs at.

  where *xx* is a number from 11 to 89 that specifies the priority the metronome alarm thread should run at. Care should be taken in modifying the OS priority that the alarm thread runs at. If you specify an OS priority lower than that of any realtime thread, you will experience OutOfMemory errors because the Garbage Collector ends up running at a lower priority than realtime threads allocating garbage. The default Metronome Garbage Collector alarm thread runs at an OS priority of 89.

Default settings for the JVM

Default settings apply to the Real Time JVM when no changes are made to the environment that the JVM runs in. Common settings are shown for reference.

Default settings can be changed using environment variables or command-line parameters at JVM startup. The table shows some of the common JVM settings. The last column indicates how you can change the behavior, where the following keys apply:

- **e** - setting controlled by environment variable only
- **c** - setting controlled by command-line parameter only
- **ec** - setting controlled by both environment variable and command-line parameter, with command-line parameter taking precedence.

The information is provided as a quick reference and is not comprehensive.

JVM setting	Default	Setting affected by
Javadumps	Enabled	ec
Javadumps on out of memory	Enabled	ec
Heapdumps	Disabled	ec
Heapdumps on out of memory	Enabled	ec
Sysdumps	Enabled	ec
Where dump files are produced	Current directory	ec
Verbose output	Disabled	c
Boot classpath search	Disabled	c
JNI checks	Disabled	c
Remote debugging	Disabled	c
Strict conformancy checks	Disabled	c
Quickstart	Disabled	c
Remote debug info server	Disabled	c
Reduced signalling	Disabled	c
Signal handler chaining	Enabled	c
Classpath	Not set	ec
Class data sharing	Disabled	c
Accessibility support	Enabled	e
JIT compiler	Enabled	ec
AOT compiler (AOT is not used by the JVM unless shared classes are also enabled)	Enabled	c
JIT debug options	Disabled	c
Java2D max size of fonts with algorithmic bold	14 point	e
Java2D use rendered bitmaps in scalable fonts	Enabled	e
Java2D freetype font rasterizing	Enabled	e
Java2D use AWT fonts	Disabled	e
Default locale	None	e
Time to wait before starting plug-in	zero	e
Temporary directory	/tmp	e
Plug-in redirection	None	e
IM switching	Disabled	e
IM modifiers	Disabled	e
Thread model	N/A	e
Initial stack size for Java Threads 32-bit. Use: **-Xiss<size>**	2 KB	c
Maximum stack size for Java Threads 32-bit. Use: **-Xss<size>**	256 KB	c
Stack size for OS Threads 32-bit. Use **-Xmso<size>**	256 KB	c
Initial heap size. Use **-Xms<size>**	64 MB	c

JVM setting	Default	Setting affected by
Maximum Java heap size. Use **-Xmx<size>**	Half the available memory with a minimum of 16 MB and a maximum of 512 MB	c
Target time interval utilization for an application. The Garbage collector attempts to use the remainder. Use **-Xgc:targetUtilization=<percentage>**	70%	c
The number of garbage collector threads to run. Use **-Xgc:threads=<value>**	1	c
Maximum amount of memory that can be allocated to scope memories in **-Xrealtime** mode. Use **-Xgc:scopedMemoryMaximumSize=<size>**.	8 MB	c
Sets the size of the immortal memory area in **-Xrealtime** mode. Use **-Xgc:immortalMemorySize=<size>**	16 MB	c

Note: "available memory" is either the amount of real (physical) memory, or the **RLIMIT_AS** value, whichever is the smallest value.

WebSphere Real Time for RT Linux class libraries

A reference to the Java class libraries that are used by WebSphere Real Time for RT Linux.

The Java class libraries that are used by WebSphere Real Time for RT Linux are described in http://www.rtsj.org/specjavadoc/book_index.html.

Running with TCK

If you are running the Real-Time Specification for Java (RTSJ) Technology Compatibility Kit (TCK) with WebSphere Real Time for RT Linux, you should include `demo/realtime/TCKibm.jar` in the classpath in order for tests to be completed successfully.

`TCKibm.jar` includes the class **VibmcorProcessorLock** which is IBM's extension to the TCK.ProcessorLock class. This class provides uniprocessor behavior that is required in a small set of TCK tests. For more information on the TCK.ProcessorLock class and vendor specific extensions to this class, see the readme file that is included with the TCK distribution.

Notices

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

- JIMMAIL@uk.ibm.com [Hursley Java Technology Center (JTC) contact]

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurement may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

Privacy Policy Considerations

IBM Software products, including software as a service solutions, ("Software Offerings") may use cookies or other technologies to collect product usage information, to help improve the end user experience, to tailor interactions with the end user or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering's use of cookies is set forth below.

This Software Offering does not use cookies or other technologies to collect personally identifiable information.

If the configurations deployed for this Software Offering provide you as customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see: (i) IBM's Privacy Policy at http://www.ibm.com/privacy ; (ii) IBM's Online Privacy Statement at http://www.ibm.com/privacy/details (in particular, the section entitled "Cookies, Web Beacons and Other Technologies");

and (iii) the “IBM Software Products and Software-as-a-Service Privacy Statement” at http://www.ibm.com/software/info/product-privacy.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. If these and other IBM trademarked terms are marked on their first occurrence in this information with a trademark symbol (® or ™), these symbols indicate U.S. registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at http://www.ibm.com/legal/copytrade.shtml.

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Intel and Itanium are trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Other company, product, or service names may be trademarks or service marks of others.

Index

Printed in USA