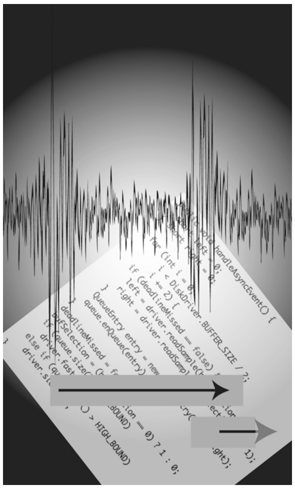
Chapter 9

Tutorial: Getting Started



Getting started with a new platform often involves some frustration with the details. The standard "Hello World!" program from Chapter 8 is a simple program with just enough code to demonstrate that it did compile and run. This tutorial aims to get the first few programs compiled and run, and then to dip far enough into RTSJ to show that it is more than a regular Java platform.

Infrastructure

This section is going to discuss using the RTSJ Reference Implementation (RI). Other implementations of the RTSJ will probably be similar, but have some differences in their installation structure and command line.

Recent reference implementations run on X86 versions of Linux based on version 2.6.18 or newer.

The RI download is free for non-commercial use, and not available for commercial use. Start at www.timesys.com/java, or if that doesn't work look for a link at www.rtsj.org or jcp.org. You'll download a compressed tar file which you can expand into a directory that thinks it's on an installation CD. The installation directory contains a massive shell script named *install*. The shell script is enormous because it includes a uuencoded tar file containing the bulk of the RI.

The RI installer is nothing more than some click-through dialogs wrapped around expanding the encoded tar file into a directory, by default a directory in */opt/timesys* that includes the RI's version number, and an alias for that directory, */opt/timesys/rtsj-ri*.

The RI does not include a Java compiler that can generate Java 1.3-compatible class files. You can download javac from Sun if you don't already have a Java compiler that you like. Even if you like another compiler, you may need to use Sun's. The RI may have problems with class files from other compilers.

A Makefile

A good IDE should be able to target the RI. It is basically an enhanced version of a J2ME JVM, so if the IDE can develop for J2ME it can probably develop for the RI. It is trickiest for something like Eclipse that is coded in Java. A Java IDE needs to be able to look in one place for the JVM that executes the IDE and in another place for the target JVM.

The command line is a reliable and universal way to operate the RI, but the command lines can become long, so a makefile is convenient. Example 9–1 shows a template for a simple makefile designed for gnumake.

A few parts of the makefile template deserve some explanation.

- Specifying immortal memory: The RI is configured for moderate use of immortal memory. The size of its immortal memory pool is fixed at startup, so if the application needs more than the default amount of immortal memory it has to be specified on the command line. You can also shrink the immortal pool here if the default is wasting memory.
- Specifying `target` of 1.3: The RI is based on Java version 1.3, a version of the Sun CVM that is now getting a bit out of date. The major classfile format difference between 1.3 and 1.4 was that 1.4 supports assertions. Version 1.5 has features that are more painful to loose.

Example 9–1 Makefile template

```
RI_DIR  = /opt/timesys/rtsj-ri
RI_CLASSPATH = $(RI_DIR)/lib/foundation.jar
CP = -Djava.class.path=.  -Xbootclasspath=$(RI_CLASSPATH)
IMMORTAL_MEM = -Ximmortal5M
JAVA_OPTS = $(IMMORTAL_MEM) $(CP)
JAVA = tjvm $(JAVA_OPTS)
JAVAC = javac -classpath .:$(RI_CLASSPATH) -source 1.3 -target 1.3

%.class: %.java ;$(JAVAC) $<

all: <list of class files >

#  example dependency line
FaultEvt.class: FaultEvt.java

#  example execution line
run: all
  $(JAVA) FaultEvt
```

Plain Hello World

An implementation of the RTSJ is Java. It has real-time features, but it can also run normal Java applications. So the first program to try is a conventional *Hello World!* Like this:

```
class HelloWorld {
  public static void main(String [] args){
    System.out.println("Hello World!");
  }
}
```

Fill in the makefile template as shown in Example 9–2, type `make run`, and you should see `Hello World!`

Aperiodic Hello World

We can creep into the RTSJ by starting a real-time thread. A real-time thread is a sub-class of `java.lang.Thread`; it extends the base `Thread` class, but you don't need to use the extensions offered by real-time threads. Example 9–3 is a simple version of Hello World that performs its output from a real-time thread.

Several new features are available to the run method in the `HelloRTWorld` class, but at this point it's behaving like a normal `Thread` except that it is executing at a real-time priority. To be painfully precise, it's a real-time thread:

Example 9–2 **Makefile template completed for HelloWorld**

```
RI_DIR  = /opt/timesys/rtsj-ri
RI_CLASSPATH = $(RI_DIR)/lib/foundation.jar
CP = -Djava.class.path=.  -Xbootclasspath=$(RI_CLASSPATH)
IMMORTAL_MEM = -Ximmortal5M
JAVA_OPTS = $(IMMORTAL_MEM) $(CP)
JAVA = tjvm $(JAVA_OPTS)
JAVAC = javac -classpath .:$(RI_CLASSPATH) -source 1.3 -target 1.3

%.class: %.java ;$(JAVAC) $<

all: HelloWorld.class

HelloWorld.class: HelloWorld.java

run: all
  $(JAVA) HelloWorld
```

Example 9–3 **Aperiodic hello world**

```
import javax.realtime.RealtimeThread;

public class HelloRTWorld extends RealtimeThread {
  public void run(){
    System.out.println("Hello RT world!");
  }

  public static void main(String [] args){
    HelloRTWorld rtt = new HelloRTWorld();
    rtt.start();
    try {
      rtt.join();
    } catch (InterruptedException ie) {
      //  Ignore
    }
  }
}
```

- with its current allocation context set to heap memory,
- scheduled by the base priority scheduler,
- with priority of `PriorityScheduler.getNormPriority(null)`,
- and it is aperiodic
  - with a cost of zero,

- a deadline of nearly 300 million years,
- and no cost overrun or deadline miss handler.

Periodic Hello World

Any task that doesn't cycle periodically is aperiodic. Aperiodic tasks are common, but periodic tasks often found in embedded and real-time applications. A periodic thread is a good way to get a first taste of the RTSJ's power. Example 9–4 prints `Hello World!` every 500 milliseconds for 30 seconds. It uses two features

Example 9–4 Periodic hello world

```
import javax.realtime.PeriodicParameters;
import javax.realtime.RealtimeThread;
import javax.realtime.RelativeTime;

public class PeriodicHello extends RealtimeThread {
  PeriodicHello(PeriodicParameters pp){
    super(null, pp);
  }

  public void run(){
    for (int i = 0; i < 30; ++i) {
      System.out.println("Hello periodic world!");
      waitForNextPeriod();
    }
  }

  public static void main(String [] args){
    PeriodicParameters pp = new PeriodicParameters(
      new RelativeTime(500, 0));
    PeriodicHello rtt = new PeriodicHello(pp);
    rtt.start();
    try {
      rtt.join();
    } catch (InterruptedException ie) {
      //  ignore
    }
  }

}
```

of the `RealtimeThread` class:

1. The constructor includes a `PeriodicParameters` value, pp, that gives the thread a period of 500 milliseconds.

2. The real-time thread's `run` method has a loop including a call to the `waitForNextPeriod` method.

If you are lucky enough to hit a slow garbage collection while `PeriodicHello` is running, or something else stalls the program for more than a half second, you'll see an interesting feature of `waitForNextPeriod`. If a release of the periodic loop is delayed, it will "hurry up" until it is back on track.

Watch Those Real-Time Priorities—They are Serious

Normal Java priorities usually reflect the share of processor time each thread should get. In general, threads with higher priorities get more processor time. Real-time priorities, supporting "fixed priority preemptive" scheduling, have more precise meaning. The processor executes the highest priority runnable task, period. (If there is more than one processor, it's more complicated.)

The careful definition of RTSJ priority values is useful to real-time developers, but it can have surprising effects for someone who is used to softer priorities.

`DemonstratePriority` has two real-time threads, call them *master* and *worker*. Master looks like it starts worker, lets it run for half a second, and then stops it.

Example 9-5 **Demonstration of real-time priorities**

```
import javax.realtime.PriorityParameters;
import javax.realtime.RealtimeThread;

class DemonstratePriority extends RealtimeThread {
  public static void main(String [] args){
    RealtimeThread rtt = new DemonstratePriority();
    rtt.start();
    try {
      rtt.join();
    } catch (InterruptedException ie) {
      //  ignore
    }
  }

  public void run(){
    Worker worker = new Worker(getPriority() + 1);
    worker.start();
    try {
      sleep(500); //  sleep for half a second
      worker.quit();
      worker.join();
    } catch (InterruptedException ie) {
      //  ignore
    }
  }
}

class Worker extends RealtimeThread {
  private volatile boolean stop = false;

  Worker(int priority){
    super(new PriorityParameters(priority), null);
  }

  public void run(){
    for (int i = 0; i < 100000000; ++i)
      if (stop)
        return;
    System.out.println("Worker ran to completion");
  }

  void quit(){
    stop = true;
  }
}
```

If you run `DemonstratePriority` (see Example 9-5) on the RI and do not run it from root, it does indeed run for about half a second and terminate without printing anything. If you enable real-time priorities by running the program from root it runs for a long time and then prints `Worker ran to completion`. Why didn't the Master thread's command tell the Worker to quit?

Multi-processor scheduling

The PriorityScheduler has comparatively subtle effects on multi-processor systems. A dual processor will concurrently execute the two real-time threads in DemonstratePriority and result in the worker thread running for a half second—probably closer to 500 milliseconds than the program achieves on a uni-processor using non-real-time priorities.

The worker thread runs at a higher priority than master. Since the scheduler will always run the highest-priority runnable thread, Master gets no CPU time until worker is no longer runnable. Worker starts using CPU time when Master starts it, and since worker remains runnable while it is executing its for loop, it does not give master a chance to return from `start`, much less return from its invocation of sleep. Consequently, master does not ask worker to quit until it has returned from its `run` method and terminated.

This kind of problem is easy to avoid if one remembers that the RTSJ platform strictly respects priority. The easiest solution to the above problem is to run worker at a lower priority than master.

Summary

In this chapter you've installed the RTSJ reference implementation, compiled and run a few RTSJ applications, and seen the RTSJ do something that might have surprised you.