User-Defined Clocks in the Real-Time Specification for Java

Wellings, Andy; Schoeberl, Martin

Published in:
Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2011)

Link to article, DOI:
10.1145/2043910.2043923

Publication date:
2011

User-Defined Clocks in the Real-Time Specification for Java

Andy Wellings
Department of Computer Science
University of York, UK
andy@cs.york.ac.uk

Martin Schoeberl
Department of Informatics and Mathematical Modeling
Technical University of Denmark
masca@imm.dtu.dk

ABSTRACT

This paper analyses the new user-defined clock model that is to be supported in Version 1.1 of the Real-Time Specification for Java (RTSJ). The model is a compromise between the current position, where there is no support for user-defined clocks, and a fully integrated model. The paper investigates the implications of supporting a fully generalized model from both the specification's and an implementation's viewpoint. It concludes that for RTSJ the scope of changes required to fully integrate user-defined clocks into the specification would have a major impact on current implementations.

Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Organization and Design—*Real-time systems and embedded systems*

Keywords

Real-time Java, user-defined clocks

1. INTRODUCTION

The introduction of user-defined clocks (and consequently timers that are based on those clocks) into Version 1.1 of the Real-Time Specification for Java (RTSJ) facilitates the release of periodic schedulable objects and timeouts both based on user-detected events. The approach adopted is a compromise between providing no support (as in Version 1.02) and full support throughout the specification. The latter would require that all operations, which either delay a schedulable object for a duration of time or has an associated timeout, would need to allow time values based on user-defined clocks to be supported. Such an approach is considered to be too large a change for the 1.1 version. User-defined clocks have also been discussed in the context of the Ada programming language. Although models have been proposed, they have yet to find their way into the language's definition [7].

In this paper, we first present the requirements that underpin the rationale for RTSJ Version 1.1 time and clock models. Section 3 presents the new model. In section 4, we consider the implications of supporting fully generalized user-defined time and clock models and show how the class hierarchy would need to be changed. Section 5 considers how the revised model can be implemented. Finally, conclusions are drawn.

There are a myriad of definitions and terminology associated with the literature on clocks and time. Throughout our discussions, we assume the following definitions [2].

- A clock has **monotonicity**, if each successive time reading from that clock yields a time that is not before a previous reading.
- Two clocks are **synchronized** when the difference between their time values is less than some specified offset. Synchronization in general degrades with the passage of time, and may be lost, given a specified offset. The error is called **clock drift**.
- **Resolution** is the minimal time value interval that can be distinguished by a clock.
- **Uniformity** refers to the measurement of the progress of time at a consistent rate, with a tolerance on the variability. Uniformity is affected by two other factors, **jitter** and **stability**. Jitter is a short-term and non-cumulative small variation caused by noise sources, like thermal noise. More practically, jitter refers to the distribution of the differences between when events are actually fired or noticed by the software and when they should have really occurred according to time in the real world. (Lack-of) stability accounts for large and often cumulative variations, due to e.g. supply voltage and temperature. In practice a clock is driven by an oscillator. **Accuracy** is the difference between the desired frequency and the actual frequency of the oscillator, and a major reason of synchronization loss.

2. REQUIREMENTS

An embedded real-time computer system needs to coordinate its execution with the 'time' of its environment. The term 'real time' is used to draw a distinction from the computer's time. It is real because it is external.

The introduction of the notion of time into a programming language can best be described in terms of three largely independent topics [1]:

1. Interfacing with 'Time'; for example, accessing clocks so that the passage of time can be measured, delaying tasks until some future time, and programming timeouts so that the

JTRES 2011 September 26-28, 2011, York, UK

non-occurrence of some event can be recognized and dealt with.

2. Representing timing requirements; for example, specifying rates of execution and deadlines.

3. Satisfying timing requirements; for example through real-time scheduling techniques.

A language can also support other forms of 'time', not just the intuitive notion of calendar (or wall clock) time. For example, simulation time, execution time, monotonic time (time that has the monotonicity property given in Section 1) [3]. Hence, depending on the notion of time being used there is a different basis for that time – this can be called the `time base`. For every time base there is an associated clock. The value of the time read from the clock is some transformation from its associated time base. Time values read from clocks should ideally be of an opaque type.

There are three main issues that are of interest.

- The definition of a "time base". The key issue is to be able to differentiate between time bases that can be used to support **active** and **passive** clocks. An active clock allows the association of software timers and hence time values supported by such a clock can also be used to support Java's *sleep* and timed *wait* statements, along with the RTSJ's explicit and implicit use of timers to support the `ReleaseParameter` class hierarchy. The underlying time base for an active clock can be as simple as a hardware timer chip.

  A passive clock simply allows the current time to be read. It does not support timers, and time values supported by such a clock cannot be used to trigger events. An example of an underlying time base for such a clock is a CPU cycle counter or one that takes time from a GPS signal.

- The relation between the absolute and relative time supported by a clock and "real time". The key issue here is whether the epoch of a clock can be (directly) determined in relation to wall clock time. If it can, then does this imply that all time values from a clock can be mapped to time values from wall clock time? They are synchronized in some way. Also whether expressing duration in milliseconds and nanoseconds is appropriate for all clocks. For example, consider a time base that is provided by the rotation of a crankshaft. The full or partial rotation of which represents the tick of the associated clock. This will depend on the speed of rotation and therefore absolute time values will not have a direct correlation with wall clock time and milliseconds and nanoseconds is not a relevant measure of relative time. More practical would be that a tick represents a fraction of the rotation. Such a clock would be monotonic but not have uniform progress.

3. THE RTSJ VERSION 1.1 MODEL

Version 1.1 of the RTSJ provides some support for user-defined passive and active clocks, all of which can act as the base clock in its `HighResolutionTime` class hierarchy. However, `HighResolutionTime` instances that use clocks, other than the real-time clock, are not valid for any purpose that involves sleeping (e.g., `Thread.sleep(long millis) method`) or waiting (e.g., `Object.wait(long timeout)`), including the members of the `RealtimeThread.waitForNextPeriod()` family. They may, however, be used in the fire times and the periods of `OneShotTimer` and `PeriodicTimer`. There are two reasons for this compromise. The first is due to the difficulty of linking user-defined clocks to mechanisms that might be implemented by the underlying operating systems. For example, the timed `Object.wait` method might be implemented by mutexes and condition variables supported by a POSIX-compliant operating system. Allowing the timeout to be based on a user-defined clock would significantly complicate the VM's internal structure. The second reason, is the more pragmatic reason of limiting the impact of any changes on the rest of the specification. For example, any part of the specification that takes a high resolution time type would need to be updated.

Figure 1 shows the main clock and time related classes in the RTSJ Version 1.1. The main new components are additional methods in the clock class and the introduction of a `ClockCallBack` interface. The approach is based on the premise that a user-defined clock should only be responsible for providing the current time and signalling when a single absolute time has been reached. Any queue management associated with timer functions should be supported by the JVM.

Hence, the following new instance methods have been added:

- `drivesEvents` – returns true if the current clock is an active clock and can be used to signal events.

- `registerCallBack(AbsoluteTime, ClockCallBack)` – called by the JVM to request a call back when the absolute time has been reached.

- `resetTargetTime(AbsoluteTime)` – reset the time for the last callback. Returns null if "now" is passed the time to be set.

The `ClockCallBack` interface contains two methods

- `atTime(Clock)` – called by the user-defined clock when the last set absolute time has been reached.

- `discontinuity(Clock, AbsoluteTime)` – called by the user-defined clock if the clock has experienced a discontinuity. This is for the case where the clock is not monotonic. A discontinuity might occur, for example with the calendar clock when it is adjusted for daylight savings time.

The sequence diagram in Figure 2 illustrates how a user-defined clock is used by the application and the JVM. The main flows are described below.

1. The application creates an instance of its user-defined clock.

2. The application creates a new absolute time value based on the created clock.

3. The application creates an asynchronous event handler

4. The application creates a one shot timer that should release the handler at the specified absolute time.

5. The RTSJ infrastructure creates an internal object to handle the clock callback.

6. The RTSJ infrastructure calls the user-defined clock to register the callback for the specified absolute time (assuming no callbacks already registered).

7. When the user-defined clock recognizes that the specified absolute time has arrived it makes the callback to indicate that the time has arrived.

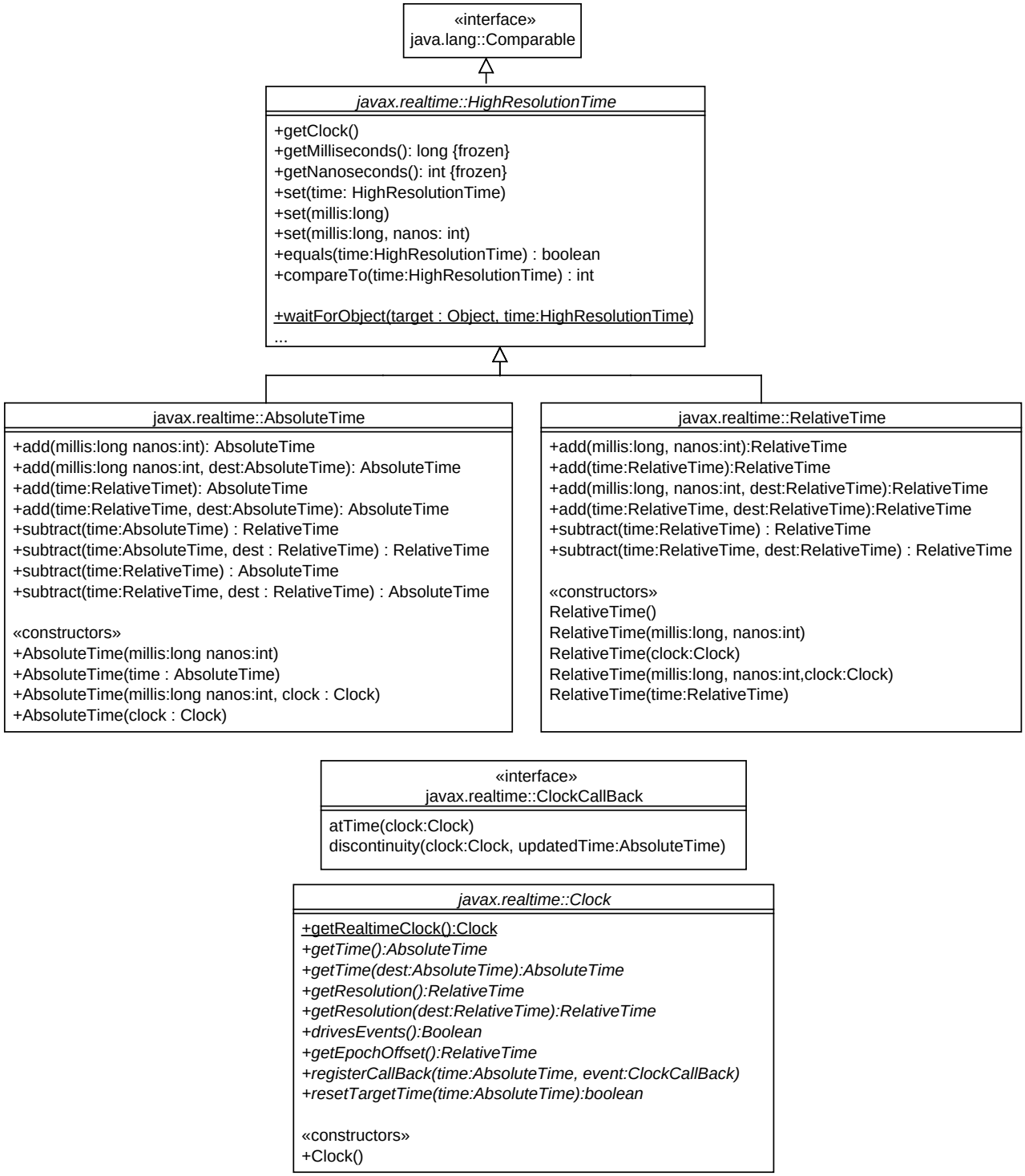

Figure 1: RTSJ Version 1.1 Clock and Time Classes

8. The RTSJ infrastructure then calls infrastructure code to fire the one shot timer.

9. The one shot timer then calls RTSJ infrastructure code to inform the scheduler to release the application event handler.

10. The handler eventually runs and the `handleAsyncEvent` method is called.

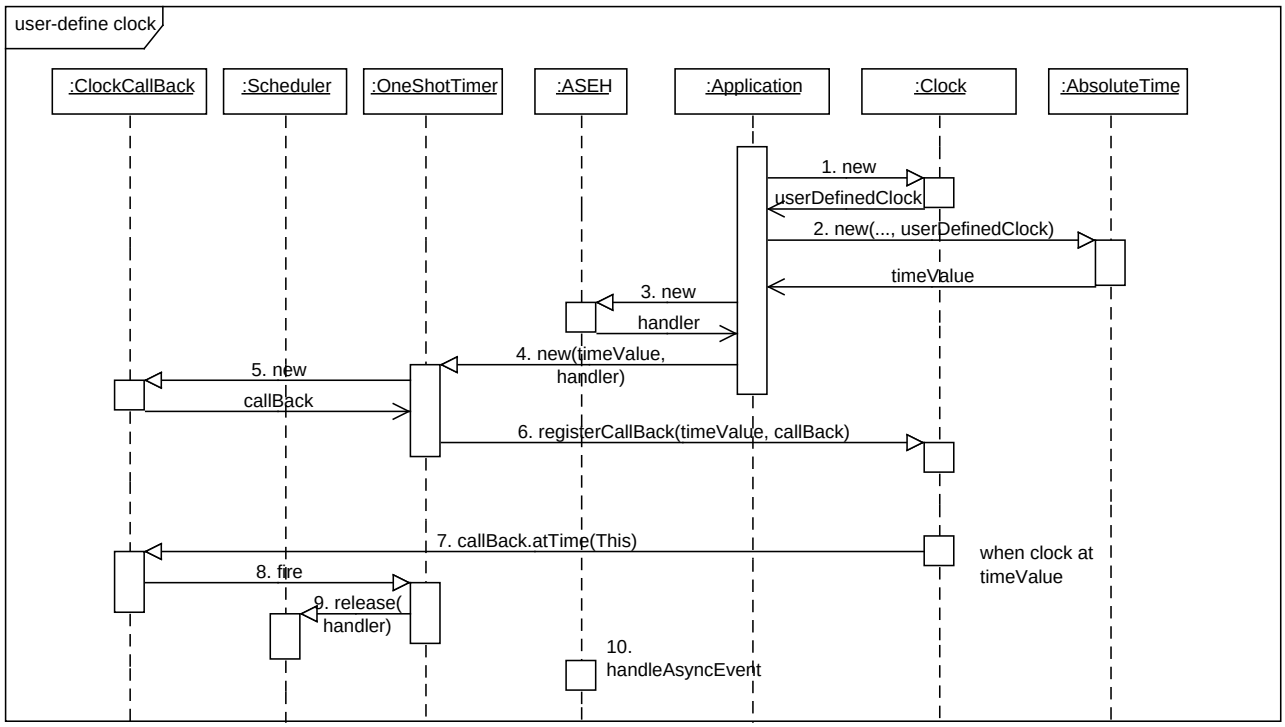

Figure 2: Callbacks

4. EXTENDING THE MODEL

This section considers how the RTSJ user-defined clock mechanisms can be generalized and evaluates the consequences of that generalization.

4.1 Rationale

One of the constraints of the current RTSJ model is that the components of all time values are considered to be milliseconds and nanoseconds. Most notions of time deal in the notion of seconds; for example, calendar time, monotonic time, simulation time, execution time. Even absolute time values are usually stored as relative time values (i.e. milli and nano seconds) passed an epoch. In Section 2, we gave an example of a time base that was based on the rotation of a crankshaft. This clearly cannot be represented as a milliseconds and nanosecond component. Philosophically, any attribute can be used as a time base as long as it has the necessary useful properties. The main property is that the attribute changes its value, and these changes can be detected. This is the role of a clock: to measure the passage of "time". For example, atomic time is currently measured by a clock which counts the vibration of cesium atoms in response to being exposed to microwaves; counting the corresponding cycles is a measure of time. A single oscillation can be considered as a tick of the clock. It is important to distinguish between the physical device that is used to measure the passage of time (hardware clock) and the software clock that keeps track of how much time has passed since the clock's epoch. The hardware clock might have some internal counter that might keep track of a number of "ticks" and this may overflow. The software clock can set an arbitrary epoch and be specified to have whatever range is deemed appropriate. For the real-time clock, a long milliseconds and int nano seconds components with an epoch of the first of January 1970 is considered to provide a significant enough range of values for clock overflows to be safely ignored. Where the software clock does not intervene with the reading provided by the hardware clock (as perhaps in a passive clock based on a hardware counter), then clock overflow may again become an issue.

From an engineering viewpoint, it may be appropriate to consider a physical attribute as a source of a user-defined time base if we want to:

1. release an asynchronous event handler (ASEH) from a timer associated with the time base;
2. release real-time threads from a timer associated with the time base;
3. associate the deadline of some computation with a number of times the physical attribute of the system changes;
4. use the change in the physical attribute as a "timeout" on waiting for another event to occur: e.g. entering into a scope memory area (`joinAndEnter`), a timed `Object.wait`;
5. use it for a minimum inter-arrival "time"; that is, the minimal inter-arrival time of another event should be related to the change in the physical attribute;
6. delay a computation until a certain number of changes have occurred;
7. use "time" values to obtain partial ordering between other events.

Of course, most of the above can be achieved by using the "clock" as a device, and associating asynchronous events with the

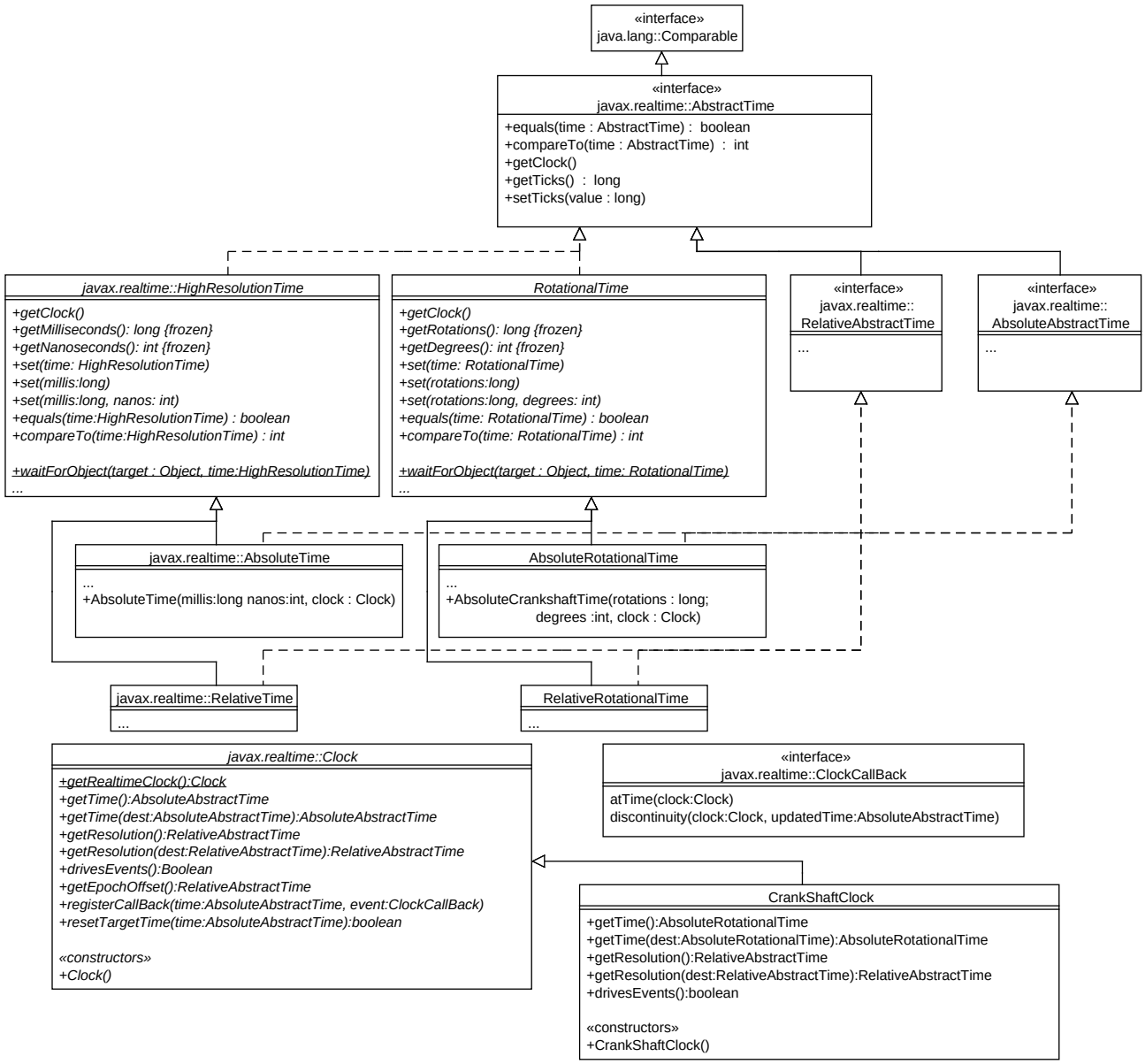

Figure 3: Revised RTSJ Clock and Time Classes with a User-Defined Time Base

changes detected by the devices, and then using an event-based programming model rather than a time-based programming model. For example, considering above points:

1. – the program can use the asynchronous event directly;

2. – the program can use the `release` and `waitForNextRelease` mechanisms available in version 1.1 of the RTSJ;

3. – in theory, the program can set a handler to go off when the "time" has expired and interrupt the thread; in practice this will not be well integrated with the semantics of `waitForNextPeriod` and `waitForNextRelease`[1];

4. – the program can achieve this using `Thread.interrupt` and `Thread.holdsLock(Object)` – assuming the device knows the associate object on which the thread is blocked (this might not be the case, for example, with `joinAndEnter`);

5. – it is difficult to see how this can be achieved without the integration into the time facilities of the RTSJ

6. – the programmer can achieve this using the `Object wait` and `notify` facilities in conjunction with the device;

[1]The RTSJ has supports a cost enforcement model where a schedulable object cannot consume more that a given execution time per release. For periodic releases, the budget must be increased at each release event even though a previous release may not have completed. Integrating this mechanism with user-defined clocks would require an interface to be created which would allow the application to indicate a new cost monitoring period has started.

7. – this can be achieved by reading the "clock" associated with the "time" base.

For the above discussion, it is apparent that there are some advantages to being able to view a variety of attributes as a time base, particularly as it allows greater integration with the RTSJ mechanisms.

A limitation of the RTSJ Version 1.1 facilities is that it only allows user-defined clocks to be used as the basis for explicit timers. Hence, it is not possible, for example, to release a periodic real-time thread from a user-defined clock – even when the time-base associated with that clock is based on milliseconds and nanoseconds.

In the remainder of this section, we explore the implications of a fully general model. In particular, we present a revised class hierarchy for the clocks and time. This is followed, in section 5 with a report of our initial experiences with implementing aspects of the new model.

4.2 The Revised Class Hierarchy

The main challenge in modifying the RTSJ Version 1.1 time and clock APIs comes from the requirement to have a root abstract time which is not tied to representing milliseconds and nanoseconds. Introducing this as an abstract class would not allow a uniform notion of relative and absolute type to be specified. For this reason, the proposed revised API (shown in Figure 3) uses interfaces. This allows the following simple replacement for time values throughout the RTSJ.

```
HighResolutionTime -> AbstractTime
RelativeTime -> RelativeAbstractTime
AbsoluteTime -> AbsoluteAbstractTime
```

These changes are required as currently the root RTSJ types assume milliseconds and nanosecond components. They would be backward compatible with the current version of the RTSJ, assuming that the names `AbstractTime`, `RelativeAbstractTime` and `AbsoluteAbstractTime` are not used in an implementation's support for the current model. There is precedence for this type of change in the RTSJ. For example, the memory classes have been refactored along similar lines to allow the introduction of pinned memory areas.

Figure 3 also includes a user-defined time type and its associated clock that are based on `RotationalTime`. This time base is supported by the crankshaft's rotation discussed in Section 2.

The following Java code illustrates the application implementation of the user-defined clock for rotational time. First, the extended clock class is illustrated. This is a tick-driven clock.

```
import javax.realtime.*;
public class CrankshaftClock extends Clock {
  public CrankshaftClock() {
    super();
  }

  public void tick() {
    now++;
    if(now == nextTime) { cback.atTime(this); }
  }

  @Override
  public AbsoluteAbstractTime getTime(
           AbsoluteAbstractTime dest) {
    if(dest != null) {
      dest.setTicks(1);
      return dest;
    } else return new AbsoluteRotationalTime(now);
  }

  @Override
  public RelativeAbstractTime getResolution(
           RelativeAbstractTime dest) {
    if(dest != null) {
      dest.setTicks(1);
      return dest;
    } else return new RelativeRotationalTime(1);
  }

  @Override
  protected boolean drivesEvents() {
    return true;
  }

  @Override
  protected void registerCallBack(
                AbsoluteAbstractTime time,
                ClockCallBack clockEvent) {
    cback = clockEvent;
    nextTime = time.getTicks();
  }

  @Override
  protected boolean resetTargetTime(
             AbsoluteAbstractTime time) {
    if (now > time.getTicks()) {
      nextTime = time.getTicks();
      return true;
    }
    return false;
  }

  ...

  private long now = 0;
  private long nextTime = 0;
  private ClockCallBack cback;
  private AbstractRelativeTime resolution =
          new RelativeRotationalTime(1);
}
```

Now it is necessary for the crankshaft interrupt to call the tick method in the clock. Here, we assume that RTSJ Version 1.1 suports first-level interrupts using an `InterruptServiceRoutine` class. The name passed to the constructor allows the interrupt to be identified. When the interrupt occurs the `handle` method is called.

```
public class CrankshaftInterruptHandler
       extends InterruptServiceRoutine {

  private CrankshaftClock clock;

  public CrankshaftInterruptHandler(String name,
         CrankshaftClock clock) {
    this.clock = clock;
  }

  @Override
  protected synchronized void handle() {
    // interrupt handling code here
    clock.tick();
  }
}
```

5. IMPLEMENTATION OF THE MODEL ON JOP

To verify that the proposed API for user-defined clocks is sound, we have implemented a prototype on the Java processor JOP [5]. We have chosen JOP, as it is relatively simple to add a hardware device for an additional time base and associate an interrupt with it. Interrupt handlers and the scheduler, which is just the interrupt handler of the default clock, are written in Java and no operating system is in the way.

The runtime system of JOP does not contain a full RTSJ implementation. Instead it targets the upcoming safety-critical Java specification (SCJ) [4]. As SCJ will include the user-defined clocks, this runtime system is a valid platform for experiments. We have refactored the RTSJ 1.1 clock and time classes to the extended model as shown in Figure 3.

JOP contains two counters: one that ticks at 1 MHz and one that ticks with the clock frequency. The first counter can be programmed to drive an interrupt and is used for the scheduler. The default clock, as returned by `Clock.getRealtimeClock()`, is based on the 1 MHz counter. Therefore, this is an active clock. That clock is used for scheduling for the periodic real-time threads. The second counter represents CPU clock cycles and is a passive clock. In the following examples we assume that the CPU clock ticks at 100 MHz, a common operating frequency of JOP in low-cost FPGAs.

5.1 A Passive Clock

For the passive clock example we use the clock cycle counter on JOP. First it is represent with the RTSJ 1.1 model, where the time needs to be mapped to `AbsoluteTime`. Second the clock cycle counter is represented by a user-defined clock from the extended model. In this case the time is also represented by a user-defined type.

5.1.1 The RTSJ Version 1.1

As a first experiment we implemented a passive clock based on the RTSJ version 1.1 model. The following code snippet shows the the core of the implementation.

```
SysDevice sys =
  IOFactory.getFactory().getSysDevice();

public PassiveClock() {
}

public AbsoluteTime getTime(AbsoluteTime dest) {
  int val = sys.cntInt;
  dest.set(val/100000, val%100000);
  return dest;
}
```

On JOP I/O devices are mapped to so-called *hardware objects* [6]. Hardware objects are created by the runtime system and field read and writes are directed to I/O port read and writes. In the example, a hardware object represents the system device, which contains, besides other system services, the CPU cycle counter. The counter value is read by reading field `cntInt` from that object.

As it can be seen, the implementation of a passive user-defined clock for the RTSJ Version 1.1 model is trivial. However, practically we observe two issues with this model: the CPU clock cycle counter in JOP is a 32-bit counter and conversion to the standard RTSJ time format is expensive.

A 32-bit counter clocked at 100 MHz will overflow after around 43 seconds. It can still be used to measure time differences up to 21 seconds. However, this behavior cannot be specified with the current `Clock` class. A passive clock has no option to detect the overflow and provide correcting actions. This can only be performed by an active clock.[2]

The second issue is more subtle. Conversion between a tick number representing ticks at 10 ns and the RTSJ time format needs one division and one reminder operation. Furthermore, these operations have to be performed on `long` values. Division is often expensive on an embedded processor, and even with hardware support for division, the operation on a `long` data type will be expensive on a 32-bit processor.

5.1.2 The Extended Model

A more flexible, and more efficient, representation of a CPU clock cycle counter is possible with the extended model. To represent clock ticks we define two user-defined time types: `AbsoluteUserTick`, which implements `AbsoluteAbstractTime` and `RelativeUserTick`, which implements `RelativeAbstractTime`. The following Java code shows a few methods of the passive user defined clock with user-defined types.

```
public PassiveExtendedClock() {
}

public RelativeAbstractTime getResolution() {
  return new RelativeUserTick(1, this);
}

public AbsoluteAbstractTime getTime() {
  int val = sys.cntInt;
  return new AbsoluteUserTick(val, this);
}

public AbsoluteAbstractTime
  getTime(AbsoluteAbstractTime dest) {

  int val = sys.cntInt;
  dest.setTicks(val);
  return dest;
}

public long getMaxValue() {
  return 0xffffffff;
}
```

The resolution of this clock is individual ticks, represented by `RelativeUserTick`. The actual time is reported as `AbsoluteUserTick`. Note that in this case we need no (expensive) conversion to a different time type. To indicate that this clock represents a 32-bit counter, the method `getMaxValue()` was added to this clock. It might be useful to add this method to the base class `Clock`.

5.2 An Active Clock

An active clock can be used to drive scheduling events for periodic threads that base their period on that clock. We implemented the crankshaft clock, which is driven by a simulation of a motor. The simulation generates interrupts and an interrupt handler in Java triggers the tick of the clock (as shown in the code in Section 4.2). A periodic thread uses the crankshaft clock for the period and start time. The scheduling events of this thread are driven by the active user-defined clock.

[2]Another option is to use a periodic thread that performs overflow corrections. In that case we do not consider the clock strictly passive.

To support user-defined, active clocks the scheduler must be aware of the additional release events. The original scheduler of JOP uses the timer that ticks at 1 MHz. To avoid unnecessary interrupts from the timer, it is programmed on a thread dispatch to interrupt the current thread when a higher priority thread's release is due. The scheduler is also invoked on a `waitForNextPeriod()` to find the highest priority thread, which is ready. Therefore, the ready queue is implicitly encoded in the priority ordered list of threads.

For an extension of the scheduler with additional active clocks, before a dispatch of a threads, the callbacks of all active clocks need to be enabled with `registerCallBack()`). As the clocks are *unrelated* it is not possible to find the single higher priority thread that will be released next.

We have adapted the thread and scheduler implementation to support different clocks for periodic release parameters. The changes have been relative moderate and the implementation was done in less than a day.

The original scheduler of JOP manipulated only integer values, which represent microseconds, for efficient scheduling decisions.With user defined clocks the scheduling check for user-defined clocks is as follows:

```
ref[i].clock.getTime(now);
if ((next[i] - now.getTicks()) <= 0) {
  break;  // found a ready task
}
```

Each thread object (`ref[i]`)) contains a reference to its clock and the actual time is retrieved with `getTime()`. For the comparison with the next release time (`next[i]`) the time is converted to ticks. It would be more efficient when the notion of ticks is associated with the class `Clock` and not with the class `AbstractTime`. When all time values are based on a `long` value that represents ticks, a method `getTicks()` can be added to the abstract class `Clock`. The check for the release time would be simplified to:

```
if ((next[i] - ref[i].clock.getTicks()) <= 0) {
  break;  // found a ready task
}
```

An active clock supports the registration of a callback for a time in the future. However, with the current API the callback time cannot be queried. For the the usage of the callbacks by the JVM scheduler it would be convenient to also query the time of an already registered callback. Therefore, we suggest to add a method `getTargetTime()` to `Clock`.

5.3 Source Access

As the prototype of the extended model is experimental, it is not included in the `master` branch of the JOP repository, but in the branch `clock`. The source can be checked out with:

```
git clone -b clock git://www.soc.tuwien.ac.at/jop.git
```

The file `doc/paper/README.UD_CLOCKS` contains the instructions to build the examples and run them on the software simulation of JOP.

6. CONCLUSIONS

In this paper we have presented the upcoming model of user-defined clocks in RTSJ version 1.1. This model is driven by compromises related to current operating systems engineering practice. We have taken that model a step further to represent user-defined clocks that may not only represent time, but other reoccurring events, which can be represented as a form of monotonic time. Time values, based on user-defined clocks, are allowed for thread release parameters. However, supporting user-defined active clocks for general scheduling, such as release times for periodic threads, needs changes in the scheduler. As we have seen in the implementation on JOP, those changes are relative moderate. Supporting scheduling based on user-defined clocks is possible when thread scheduling is implemented by the JVM, but might be almost impossible when the JVM delegates scheduling to the underlying real-time operating system.

Acknowledgements

The authors gratefully acknowledge the contributions made by Kelvin Nielsen and Alan Burns to some of the ideas expressed in this paper.

7. REFERENCES

[1] A. Burns and A.J. Wellings. *Real-Time Systems and Programming Languages:*. Addison Wesley, 4th edition, 2009.

[2] P. Dibble, R. Belliardi, B. Brosgol, D. Holmes, and A.J. Wellings. The real-time specification for Java (RTSJ 1.02). http://www.rtsj.org.

[3] Object Management Group. Enhanced view of time specification, v2.0. http://www.omg.org/spec/EVoT/2.0.

[4] Doug Locke, B. Scott Andersen, Ben Brosgol, Mike Fulton, Thomas Henties, James J. Hunt, Johan Olmütz Nielsen, Kelvin Nilsen, Martin Schoeberl, Joyce Tokar, Jan Vitek, and Andy Wellings. Safety-critical Java technology specification, public draft, 2011. http://www.jcp.org/en/jsr/detail?id=302.

[5] Martin Schoeberl. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, 54/1–2:265–286, 2008.

[6] Martin Schoeberl, Stephan Korsholm, Tomas Kalibera, and Anders P. Ravn. A hardware abstraction layer in Java. *ACM Trans. Embed. Comput. Syst.*, accepted 2009, 2011.

[7] A. J. Wellings and A. Burns. User-defined clocks is it the right time now? *Ada Lett.*, 30(1):104–115, 2010.