User-Defined Clocks in RTSJ

Andy Wellings and Martin Schoelberl

With a little help from our friends in JSR 282 particularly Kelvin Nilsen
Introduction

- RTSJ provides a framework for multiple clocks but only requires a monotonic real-time clock
- An implementation could add new clocks using this framework, but the framework is incomplete if the users wants to add their own clocks
- RTSJ V1.1 provided limited added support
Structure

- Requirements
- The RTSJ Version 1.1 model
- Extending the model
- Implementing the model on JOP
- Conclusions
Requirements: the role of time

- Interfacing with *time*
  - Measuring the passage of time
  - Delaying threads until some future time
  - Programming timeouts

- Representing timing requirements
  - Rates of executions and deadlines

- Satisfying timing requirements
  - Schedulability analysis
Requirements: types of time

- Calendar time
- Simulation time
- Monotonic time
- Execution time
- Atomic time
A time base provides the underlying basis for a particular time type.

For every time base there is an associated clock.

- The value read from the clock is a transformation of its time base.

E.g., atomic time is 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.
Active and Passive Clocks

- Active clocks can support timers
  - Underlying time base can be as simple as a hardware timer chip

- Passive clock only allow the current time to be read
  - Underlying time base could be a CPU cycle counter or GPS signal
Relation with `real time`

- Is there a relationship between the user-defined clock’s epoch and calendar time?
- Is milliseconds/nanoseconds the most appropriate measure for duration?
  - Consider a crankshaft: full/partial rotation is a tick
  - Length of tick depends on rotation speed
  - Clock is monotonic but not uniform
RTSJ Version 1.1 Model

- Each user-defined active clock should only be responsible for indicating one timing event
- The RTSJ infrastructure should be responsible for maintaining any delay queues
### RTSJ V 1.1 API

**Interface**

```java
javax.realtime::ClockCallback
```

- `atTime(clock: Clock)`
- `discontinuity(clock: Clock, updatedTime: AbsoluteTime)`

**Class**

```java
javax.realtime::Clock
```

- `...`
- `drivesEvents(): boolean`
- `registerCallBack(time: AbsoluteTime, event: ClockCallback) boolean`
- `resetTargetTime(time: AbsoluteTime): boolean`
User Case: One shot timer

1. new userDefinedClock
2. new(..., userDefinedClock)
3. new handler
4. new(timeValue, handler)
5. new callBack
6. registerCallBack(timeValue, callBack)
7. callBack.atTime(This)
8. fire
9. release(handler)
10. handleAsyncEvent

when dock at timeValue
Limitations

- Active user-defined clocks can only be used with the Timer classes

Reasons
- To avoid the complexity of linking with OS provided time services (e.g. Timed wait on mutexes)
- To limit the scope of changes to the RTSJ spec

Our goal: to explore a more general model
Time Bases and Physical Attributes

1. Release a schedulable object from a timer associated with the time base

2. Associate the deadline of some computation with a number of times the physical attribute of the system changes

3. 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
4. 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

5. Delay a computation until a certain number of changes have occurred

6. Use "time values" to obtain partial ordering between other events
Motivating Example

- A time base that is provided by the rotation of a crankshaft
- The full/partial rotation represents the tick of the associated clock
- A tick depends on the speed of rotation
  - 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
- A tick represents a fraction of the rotation
- Such a clock would be monotonic but not have uniform progress
Alternative Approach

- Treat the “clock” as a device
- Associating asynchronous events with the changes detected by the devices
- Use an event-based programming model rather than a time-based programming model
- Problem: integrating 2, 3, 5 might be difficult
API Refactoring I

```
javax.realtime::HighResolutionTime
+getClock()
+getMilliseconds(): long (frozen)
+getNanoseconds(): int (frozen)
+set(time: HighResolutionTime)
+set(milliseconds:long)
+set(milliseconds:long, nanos: int)
+equals(time:HighResolutionTime) : boolean
+compareTo(time:HighResolutionTime) : int
+waitForObject(target : Object, time:HighResolutionTime)

javax.realtime::AbsoluteTime
...
+AbsoluteTime(milliseconds:nanos:int, clock : Clock)

javax.realtime::RelativeTime
...

RotationalTime
+getClock()
+getRotations(): long (frozen)
+getDegrees(): int (frozen)
+set(time: RotationalTime)
+set(rotations:long)
+set(rotations:long, degrees: int)
+equals(time: RotationalTime) : boolean
+compareTo(time: RotationalTime) : int
+waitForObject(target : Object, time: RotationalTime)

AbsoluteRotationalTime
...
+AbsoluteCrankshaftTime(rotations : long;
  degrees :int, clock : Clock)

RelativeRotationalTime
...
```
The Crankshaft Clock

public class CrankshaftClock extends Clock {

    public CrankshaftClock() {
    }

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

    @Override
    public AbsoluteAbstractTime getTime() {...}

    @Override
    public RelativeAbstractTime getResolution() {...}

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

@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;
    } else return false;
}

private long now = 0;  private long nextTime = 0;
private ClockCallBack cback;
}
public class CrankshaftInterruptHandler extends InterruptServiceRoutine {

private CrankshaftClock clock;

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

@Override
protected synchronized void handle() {
    clock.tick();
}
}
JOP Implementation

- **Experiment 1**
  - Use CPU cycle counter as a passive clock assuming RTSJ Version 1.1 Model

- **Experiment 2:**
  - Experiment 1 with the extended model

- **Experiment 3**
  - Use a simulation of a crankshaft (which generates interrupts) as an active clock
  - Run a periodic thread
Experiment 1

- Implementation trivial
- However:
  1. The counter is 32 bits and overflows after around 43 seconds; this is not catered for in current API but subclass could add a getMaxValue method
  2. Conversion between tick number and RTSJ format needs to operate on longs and requires one division and one remainder operation
Experiment 2

- Introduce two new time types: **AbsoluteUserTick** and **RelativeUserTick**
- Now no need for conversions

- Perhaps: base Clock class needs a `getMaxValue` method?
Experiment 3: Active Clock

- The scheduler must be aware of additional release events
- Current scheduler is highly optimized to avoid unnecessary timer interrupts
  - The ready queue is implicitly encoded in a priority-ordered list of threads
Experiment 3

- The algorithm needed to be changed as it is not possible to find the single higher priority thread that will be released next.
Conclusions

- The RTSJ version 1.1 adds extra capabilities but does not go as far as it could.
- User-defined active clocks can only be used with Timers.
- We have investigated a more general model.
- In the implementation on JOP, these changes are relatively 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.