A Ravenscar-Java Profile Implementation

Hans Søndergaard
Vitus Bering Denmark
University College
DK-8700 Horsens
hso@vitusbering.dk

Bent Thomsen & Anders P. Ravn
Department of Computer Science
Aalborg University
DK-9220 Aalborg Ø
{bt, apr}@cs.aau.dk

ABSTRACT
This paper presents an implementation of the Ravenscar-Java profile. While most implementations of the profile are reference-implementations showing that it is possible to implement the profile, our implementation is aimed at industrial applications. It uses a dedicated real-time Java processor, since we want to investigate if the Ravenscar-Java profile, implemented on a Java processor, is efficient for real applications. During the implementation some ambiguities and weaknesses of the profile were uncovered. However, test examples indicate that the profile is suitable for development of realistic real-time programs.

Categories and Subject Descriptors
D.3.3 [**Programming Languages**]: Language Constructs and Features; D.3.4 [**Programming Languages**]: Processors – *Run-time environments*; D.4.1 [**Operating Systems**]: Process Management **-** *Scheduling, Threads*; J.7 [**Computer Applications**]: Computers in Other Systems – *Real time.*

General Terms
Design, Languages, Performance.

Keywords
Ravenscar-Java profile, Real-time Java, Java processor, Industrial application.

1. INTRODUCTION
Java was originally developed as a programming language for embedded systems [11]; but it was the Internet that propelled Java into mainstream computing, because there was a need for a language that was portable and truly object-oriented, eliminating the error-prone programming of memory allocation and pointer manipulation. However, precisely those features made it less suited for predictable, real-time embedded systems: The virtual machine, that gave portability, was considered inefficient both in terms of time and space. Furthermore, the automatic garbage collection and dynamic class loading made it impossible to analyse and predict execution time and memory consumption.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
JTRES '06, October 11–13, 2006, Paris, France.
Copyright 2006 ACM 1-59593-544-4/06/10…$5.00.

However, since its appearance in 1995, Java has spread tremendously as a software development language; it is used to program all kinds of software from servers to smart cards, and it is now the first (and often the only) language for young programmers joining the industry. Concurrently, software for embedded systems has been a major growth area, because there is consumer demand for more and more sophisticated products based on embedded intelligence.

Programming dependable, real-time systems is already hard for simple functionalities, and when more intelligence is added, it means functionality, which is very costly and error prone to code in assembler or C/C++. Embedded system development needs to benefit from the major advances in object-oriented programming technology that have emerged over the past decade, and one way of doing this is to bring Java back to the application domain for which it was developed.

There is essentially only one way to get a more predictable language, namely to select a set of features which makes it controllable. A major step in that direction was the Real-Time Specification for Java (RTSJ) published in 2000 [18, 5]; many new features were introduced to make it suitable for real-time applications. However, RTSJ is complex, trying to address advanced dynamic scheduling techniques, new types of memory and has a difficult asynchronous transfer of control mechanism. RTSJ is thus targeted at larger systems, e.g. the RTSJ implementation from Sun requires a dual UltraSparc III or higher with 512 MB memory and the Solaris 10 operating system [23].

We are interested in smaller systems, for example Java enabled mobile phones; in fact, there are already more Java-enabled phones than PCs [21]. In phones, the Java 2 Micro Edition (J2ME) is used. It has a virtual machine layer (with or without an OS), a configuration layer, e.g. CDC or CLDC, and a profile layer, e.g. MIDP, which defines the allowable features. Here, and in other small scale systems, the Ravenscar-Java profile [17] for Real-Time Java fits nicely. It defines a subset of RTSJ suitable for a J2ME implementation on top of a real-time virtual machine with the CLDC configuration layer. The key aim of the profile is to define a subset of RTSJ "that meets the temporal requirements of high-integrity real-time systems" [17].

The Ravenscar-Java profile has been further refined and commented [14, 26, 13, 19], but it has been implemented a few times only [10], and the experience of using the Ravenscar-Java profile for the development of industrial embedded systems with real-time requirements is limited.

Ravenscar-Java may not yet have been used for industrial cases, because it needs implementations on platforms that are suitable

for the application area. In order to explore this thesis, we have implemented the Ravenscar–Java profile on a Java processor, the aJ-100, from aJile Systems [12, 1], which aims at such applications.

The contributions of this paper are thus

- an implementation of the Ravenscar-Java profile on the aJ-100 platform,
- a comparison of test examples using the Ravenscar-Java profile and the original aJile API, and
- an assessment of the Ravenscar-Java profile itself, and its suitability for the aJ-100 processor.

The remainder of this paper is structured as follows: A short overview of the Ravenscar-Java profile is given in Section 2. Section 3 identifies some of the most interesting properties of the processor and describes those properties in relation to the Ravenscar-Java profile implementation. Section 4 describes key aspects of the implementation. In Section 5 we compare Ravenscar-Java profile test examples with similar aJile API test examples. Section 6 assesses the Ravenscar-Java profile itself and its suitability for the aJ-100 processor. Conclusion and future work (Section 7) completes the paper.

2. THE RAVENSCAR-JAVA PROFILE

This profile was first proposed in 2001 [17], inspired by the Ravenscar profile for Ada [6], and the profile is still evolving [14, 26, 13]. It is essentially a subset of RTSJ. Where RTSJ has 68 interfaces and classes, the Ravenscar-Java profile has only about 30 interfaces and classes, including some classes not found in RTSJ. Besides making Ravenscar-Java smaller and simpler than RTSJ, a reason for introducing it was to make the programs predictable and analysable wrt. memory utilization and timing.

2.1 The computational model

A Ravenscar-Java application has two phases: an initialization phase and a mission phase, shown in Figure 1.

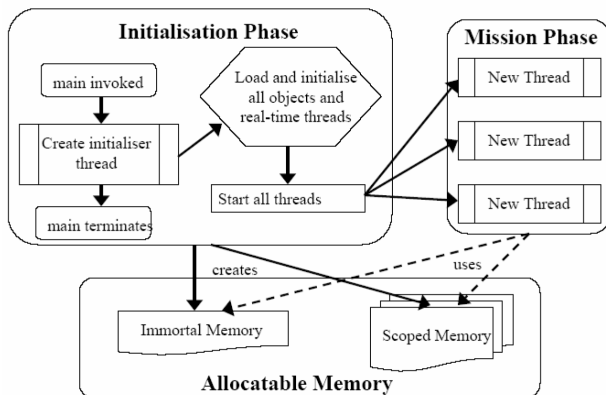

Figure 1. The two execution phases (The figure is adapted from Kwon, Wellings, and King [14])

In the initialization phase all objects needed for the lifetime of the application are created and initialized in immortal memory, including all real-time threads. This phase is not time-critical and is executed by an `Initializer` thread with maximum priority.

In the mission phase the real-time threads and event handlers are running concurrently. This phase is time critical and the priorities of the threads are less than the maximum priority which is reserved for the `Initializer` thread.

A Ravenscar-Java Virtual Machine is not supposed to support garbage collection. In fact a Ravenscar-Java VM does not have to have heap memory. Instead the Ravenscar Java profile defines three types of memory areas: immortal memory, linear time scoped memory and raw memory. When the underlying VM has a heap area, the heap can be used as immortal memory if the garbage collector can be switched off.

2.2 Overview of the Ravenscar-Java classes

The classes in the profile fall into four groups:

- real-time thread classes, which include
  - `Initializer` (for the initialization phase)
  - `PeriodicThread` (for periodic activities)
- sporadic event handler classes, which include
  - `SporadicEvent` (for software triggered events)
  - `SporadicInterrupt` (for hardware triggered events)
  - `SporadicEventHandler`
- memory classes, which include
  - `ImmortalMemory` (for objects with lifetime equal to the lifetime of the application)
  - `LTMemory` (linear time scoped memory for object allocation during the mission phase)
  - `RawMemoryAccess` (for raw memory access)
- time classes, which include
  - `AbsoluteTime`, `RelativeTime`.

We shall focus on the two first groups, since they are most dependent on the processor.

3. THE aJ-100 PROCESSOR

The aJ-100 processor from aJile Systems [1] is based on the 32-bit JEM2 Java chip developed by Rockwell-Collins. It is a 100 MHz direct execution Java processor, “designed for real-time embedded applications that require high-performance” [2]. The aJ-100 is characterized by being:

- a pure Java microcontroller that uses Java bytecode as its native instruction set,
- a real-time processor with an embedded real-time multi-threading kernel microcoded in hardware, including a priority pre-emptive scheduler with 32 priority levels, a priority ceiling protocol and periodic threads,
- supporting two concurrent JVM units, and
- having all common embedded peripherals: I/O Ports, Serial Interface, Timers, etc.

The direct bytecode execution means that the performance of an application on aJ-100 is comparable to a C application for a similar 32-bit microcontroller. The microcoded kernel means that aJile does not require an extra RTOS software layer. Furthermore, the thread switch is very fast, less than 1 µs.

In our implementation of the Ravenscar-Java profile we only use one of the two JVM units, together with its access to the global raw memory.

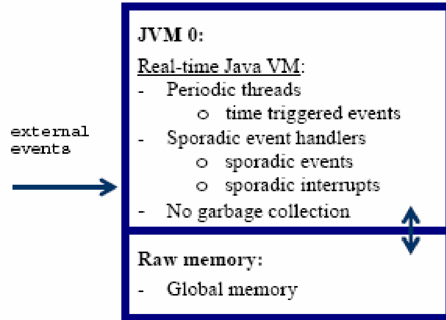

Figure 2. The aJile architecture with use of one JVM

When JVM0 is applied with real-time constraints, the garbage collector can be disabled.

The aJile processor uses a runtime system based on J2ME, CLDC 1.0 [3]. As a supplement to the CLDC library, the aJ-100 processor has a special aJile Java API to access the processor; it includes about 85 interfaces and classes, e.g. `PeriodicThread`, `rawJEM` (low level access to physical memory), and `GpioPin` (controls general purpose IO pins).

The processor must be mounted on a board providing the necessary hardware infrastructure. In our test setup we use the JStik board from Systronix [24], with 2MBytes SRAM.

Processor specific development tools are: JEM Builder, a graphical build tool for static linking and configuration and Charade, a tool for loading and starting the JVMs on aJ-100. Charade has various test facilities. These are the only tools that are aJile specific. In other words, any "standard" Java development tool, such as Eclipse or NetBeans, can be used for programming and generating Java bytecode.

4. IMPLEMENTATION OF RAVENSCAR-JAVA PROFILE

This section describes in more detail, how we implement the Ravenscar-Java profile on the aJ-100 processor using the aJile API. This API is also written in Java, but is often lower level.

4.1 Implementation of the real-time threads

The class hierarchies for the real-time threads are shown in Figure 3. The real-time threads in Ravenscar are subclasses of `java.lang.Thread`.

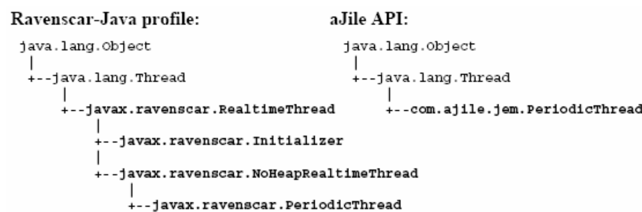

Figure 3. The class hierarchies for the real-time threads

As an example we show the implementation of the `Initializer` thread with max priority:

```
package javax.ravenscar;
public class Initializer extends RealtimeThread
{
  public Initializer()
  {
    super(new PriorityParameters (
      PriorityScheduler.getMaxPriority()),
      null, ImmortalMemory.instance(), null);
  }
}
```

4.1.1 *Periodic threads on aJ-100*

Periodic threads have to be set up for periodic activation. For this aJ-100 has an internal cyclic data structure, called a piano roll. It keeps the activation information for the different periodic threads. The `PianoRoll` class from the aJile API initializes and starts executing the piano roll:

```
package com.ajile.jem;
public class PianoRoll
{
 public PianoRoll(int duration, int beat);
 public PianoRoll(long duration,int durationNanos,
                  long beat, int beatNanos);
 public static void start();
}
```

The aJile API also has a `PeriodicThread` class:

```
package com.ajile.jem;
public class PeriodicThread extends
java.lang.Thread
{
  public PeriodicThread();
  public void makePeriodic(int period,
    int initDelay, int priority, Thread userTCB);
  public static void cycle();
    // equivalent to waitForNextPeriod() in RTSJ
}
```

Here, the `makePeriodic` method sets up a periodic thread, before it is started. The parameter `initDelay` is the time delay between the cycle start of the piano roll and the first activation of the periodic thread.

When all the periodic threads and the piano roll have been initialized with the correct values of beat, duration, periods, initDelays and priorities, and then started, the periodic threads are dispatched by the priority pre-emptive thread scheduler. Upon each tick of the beat timer, the periodic threads at the current index of the piano roll are activated, and the piano roll index is incremented to the next entry in the cyclic piano roll.

Table 1. Example with three periodic threads

Periodic thread	Period	Init delay	Priority
a	3	0	max
b	3	1	max
c	4	0	max-1

This is illustrated in Table 1 with three periodic threads, and Figure 4 shows the corresponding piano roll structure.

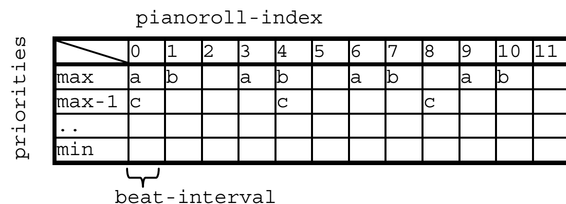

Figure 4. The corresponding piano roll structure

Threads a and b have the same period of 3, thread c has a period of 4. The thread priorities are chosen to be *rate monotonic*.

In each beat interval only one thread of each priority is activated. Hence, multiple periodic threads of different priorities can be readied simultaneously, but if there are multiple periodic threads of the same priority, only one of them will be chosen to run [4]. To overcome this problem, different periodic threads with the same priority (and period) are usually given different initial offsets. If the beat is chosen as the time unit (in msecs or nanosecs), then the following setup conditions must hold:

(P_1) `0 < beat ≤ 65 msec`
(P_2) `duration = n * beat, for some integer n`

These piano roll conditions reflect that the aJ-100 has a 16 bit timer register with a tick of 1µs, and the duration of the roll must be a multiple of the beat.

For every periodic thread, the following must hold:

(T_1) `beat ≤ period ≤ duration`
(T_2) `period * k = duration, for some integer k`
(T_3) `period = m * beat, for some integer m`
(T_4) `initDelay = p * beat, for some integer p`
(T_5) `0 ≤ initDelay < period`
(T_6) `initDelay + period ≤ duration`

These conditions ensure that the piano roll gives correct periodic activation of the threads.

Finally, for each pair of periodic threads with equal periods and equal priorities, we must have:

(E_1) $initDelay_i \neq initDelay_j$

This ensures that both threads are activated.

4.1.2 Implementation of Ravenscar-Java periodic threads

To hide and control all this complexity, some auxiliary classes were developed, including some with methods for rate monotonic priority setup of periodic threads. This is carried out during the initialization phase. These classes are organized as shown on Figure 5.

Class `Setup` is a singleton class. It sets up the piano roll, including the calculation of beat and duration. It attaches the periodic threads, including a calculation of init delays, to the piano roll. Class `SetupInfo` contains the necessary information about a periodic thread (or a sporadic event/interrupt). Class `CheckCyclicSetup` checks all the setup conditions, as described above, including a rate monotonic setup. The decision to use rate monotonic setup is discussed in Section 6.1.

The Ravenscar-Java `PeriodicThread` class encapsulates the aJile periodic thread:

```
package javax.ravenscar;
public class PeriodicThread extends
             NoHeapRealtimeThread
{
  private com.ajile.jem.PeriodicThread aJileTh;
  private SetupInfo info;
  ...
  private class aJilePeriodicThread extends
                com.ajile.jem.PeriodicThread
  {
    public void run()
    {
      // run until start time:
      for(long count = info.startTime/info.period;
               count > 0; count--) {
        PeriodicThread.waitForNextPeriod();
      }

      // run from start time:
      for (;;) {
        logic.run();
        PeriodicThread.waitForNextPeriod();
      }
    }
  }

  public PeriodicThread (PriorityParameters pp,
    PeriodicParameters p, Runnable logic)
  {
    super (pp,p,ImmortalMemory.instance(),logic);
    aJileTh = new aJilePeriodicThread();
  }
```

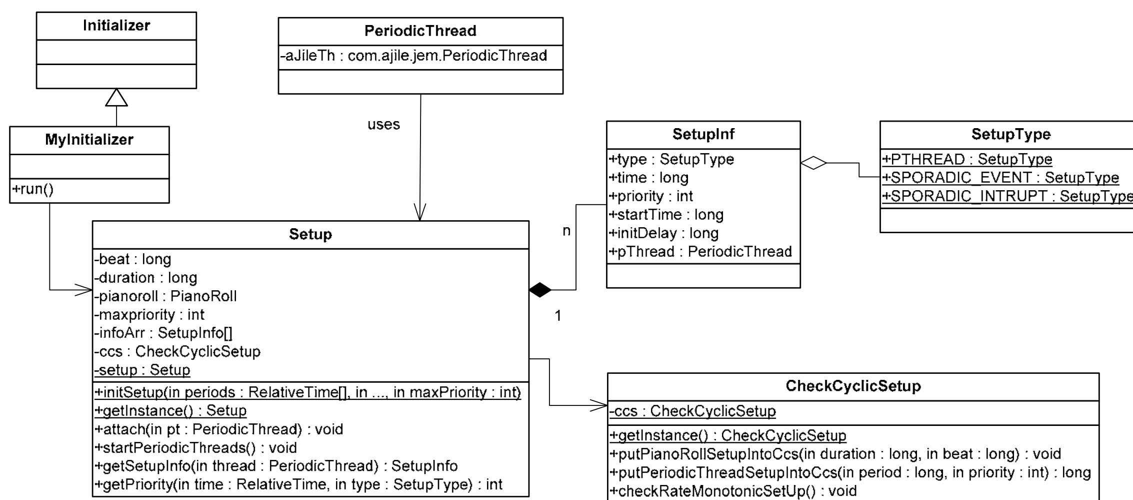

Figure 5. Class diagram for auxiliary classes for setup

```
  public final void run()
  {
    info = Setup.getInstance().getSetupInfo(this);
    aJileTh.makePeriodic (period, priority, …);
    aJileTh.start();
  }
  static boolean waitForNextPeriod()
  {
    com.ajile.jem.PeriodicThread.cycle();
    return true;
  }
}
```

A (somewhat controversial) decision is to let `waitForNextPeriod` always return `true` as suggested in [26]. We discuss this further in section 6.1

4.2 Event and event handler implementation

The Ravenscar-Java profile only specifies sporadic events, and distinguishes between

- software-generated: `class SporadicEvent`
- hardware-generated: `class SporadicInterrupt`.

The class structure is shown in figure 6.

Sporadic events and interrupts have a minimum interarrival time (MIT). The profile does not say anything about the latest permissible completion time (deadline) of the associated schedulable object. Therefore this latest permissible completion time is assumed to be equal to MIT. This is consistent with the periodic threads, where the deadline is assumed to be equal to the period.

If MIT is interpreted in the same way as the period T, then the two types of schedulable objects can be assigned priorities using the rate monotonic priority assignment in both cases. However, should a deadline be added to sporadic events, a deadline monotonic priority assignment is feasible. Note, the sporadic event handlers do not enter into the piano roll structure. We return to this point in Section 6.

4.2.1 *Event and event handling on aJ-100*

To handle hardware-generated events Ravenscar-Java defines the notion of a happening. A happening is a string which has to be mapped to the underlying hardware events.

aJ-100 has 40 General Purpose Input/Output (GPIO) pins. The aJile API provides the `GpioPin` class, an instance of which controls one pin [3] to support them. Each pin is identified by a predefined constant in the `GpioPin` class. This pin identifiier is converted to a string defining the happening for the `SporadicInterrupt` class. A `GpioPin` object has an attached eventhandler which implements the interface `TriggerEventListener`. Its `triggerEvent` metod is called when an external event occurs on the pin. This interface is as follows:

```
package com.ajile.events;
public interface TriggerEventListener
{
  void triggerEvent();
}
```

This aJile specific GPIO event and event handling have been hidden in two classes (also see Figure 7):

`class GpioPinInfo` holds and sets up information about a GPIO pin, and
`class SetupGpioPin` holds the information for up to 40 GPIO pins.

The most interesting in the implementation is the `addHandler` method in class `SetupGpioPin`. This method creates and adds a `TriggerEventListener` object to a GPIO pin, see Figure 7, and it is called by the `addHandler` method in the `SporadicInterrupt` class.

4.2.2 *Implementation of event handlers*

The thread permanently bound to `BoundAsyncEventHandler`, see Figure 6, is implemented as a private inner class `NoHeapRealtimeThread`:

```
private class NoHeapRtThread extends
              NoHeapRealtimeThread
{
  NoHeapRtThread (PriorityParameters priority,
                  Runnable logic)
  {
    super (priority, null, null, logic);
  }

  public void run ()
  {
    for (;;) {
      synchronized (handlerThread) {
        try { wait(); }
        catch (InterruptedException e) {  }
      }
      logic.run(); // the logic to run each time
    }              // an event occurs
  }
}
```

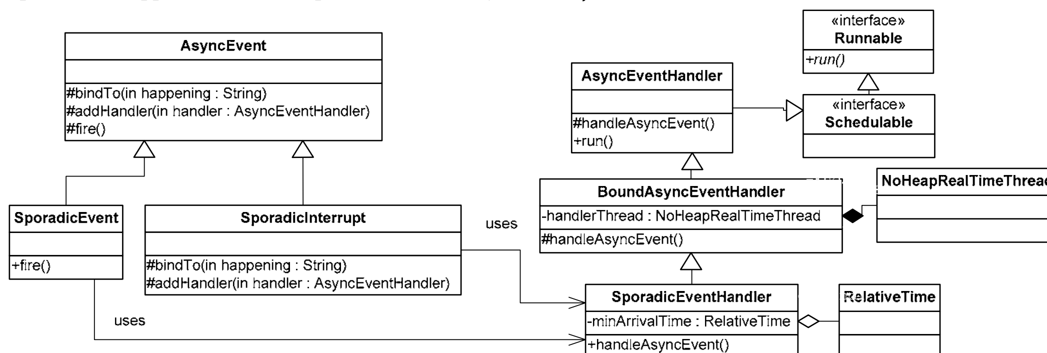

Figure 6. Class diagram for sporadic events and interrupts

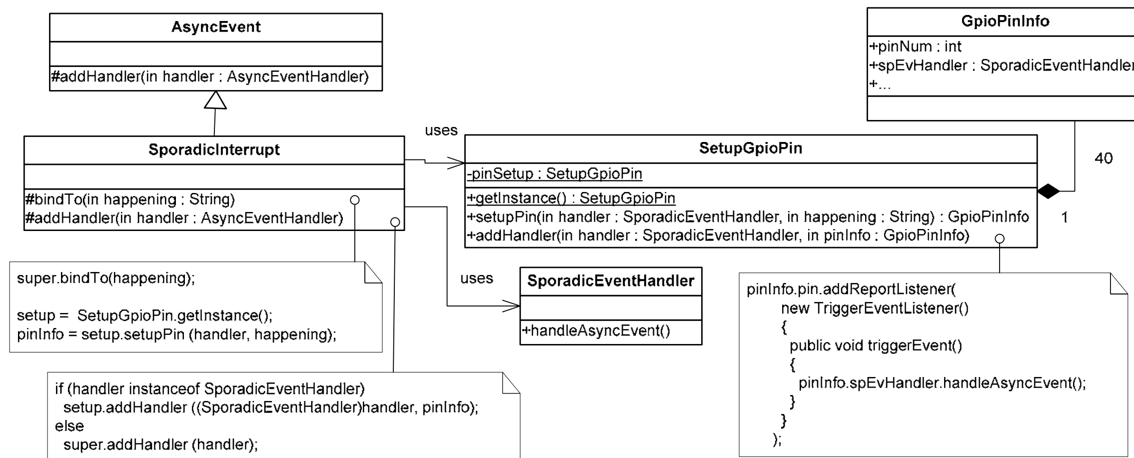

Figure 7. Class diagram for external events – linking up to aJile I/O.

This thread is created and started by the constructor of the `BoundAsyncEventHandler`. The call of `wait` suspends the thread.

When an external event occurs on the GPIO pin, the `triggerEvent` method in `TriggerEventlistener` calls `handleAsyncEvent` in class `SporadicEventHandler`, see Figure 7. This class, initialized with the minimum interarrival time between events, has the method `handleAsyncEvent` implemented as follows:

```
public class SporadicEventHandler extends
             BoundAsyncEventHandler
{
  private long oldTime, newTime, minArrivalTime,
               deltaT;
  ..
  public final void handleAsyncEvent()
  {
    newTime = rawJEM.getTime();
    deltaT  = newTime - oldTime;
    oldTime = newTime;
    if (deltaT >= minArrivalTime)
      super.handleAsyncEvent();
  }
}
```

Just when `minArrivalTime` is less than or equal to `deltaT`, the `handleAsyncEvent` in `BoundAsyncEventHandler` notifies the handler thread:

```
void handleAsyncEvent()
{
  synchronized (handlerThread)
    handlerThread.notify();
}
```

Note that we ignore events that occur too often.

4.3 Implementation of memory classes

Since Ravenscar-Java specifies no garbage collection and aJile allows the garbage collector to be switched off, the heap is used for immortal memory. This means that the implementation of class `ImmortalMemory` is empty.

Linear time scoped memory, `LTMemory`, is intended for object allocation during the mission phase. However, because there is much uncertainty about the semantics of scoped memory, and especially uncertainty on how to use it [13, 19, 8], we only implement immortal memory and raw memory.

To access the raw memory we implement the following two classes from RTSJ:

```
RawMemoryAccess
RawMemoryFloatAccess.
```

The classes contain methods for accessing a raw memory area through simple types (and arrays thereof): `byte`, `short`, `int`, `long`, `float` and `double`. To implement the methods we use the `rawJEM` class from the aJile API [3]. We only show one example:

```
public byte getByte(long offset) throws
            OffsetOutOfBoundsException,
            SizeOutOfBoundsException
{
  check (offset, SIZE_OF_BYTE);
  // atomic get:
  return rawJEM.getByte((int)(base+offset));
}
```

As we see, the methods are nearly the same, but the `rawJEM.getByte` method from the aJile API is low level and has no bounds checks.

4.4 Implementation of time classes

The time classes are implemented in accordance with RTSJ. Time is measured from program start, because aJile has no real-time clock for absolute time with automatic switch to a battery backup supply.

5. COMPARISON OF TEST EXAMPLES

In this section we compare test examples wrt. execution time, initialization and code size, using the Ravenscar-Java profile and the aJile API.

Because periodic threads are central for real-time programs, and in the aJile API implemented with many low level settings [22], we focus on periodic threads.

The loop in the periodic threads is implemented in nearly the same way:

```
for (;;) {       // aJile
  logic.run();
  com.ajile.jem.PeriodicThread.cycle();
}

for (;;) {       // Ravenscar
  logic.run();
  PeriodicThread.waitForNextPeriod();
}
```

They differ in their use of the method `waitForNextPeriod`, implemented by:

```
static boolean waitForNextPeriod()
{
  com.ajile.jem.PeriodicThread.cycle();
  return true;
}
```

In the test we let the method `logic.run` increment a counter. The tests are carried out by letting another periodic thread, with a lower priority and a period typically of 1000 ms, print the value of the counter.

The result is that for a single periodic thread, the period should meet:

- aJile API: `period ≥  9 µsec`
- Ravenscar-Java profile: `period ≥ 11 µsec`

In the overhead of 2 µsec in the Ravenscar implementation, the extra method-call `waitForNextPeriod` takes 1 µsec.

For multiple periodic threads the graphs in Figure 8 show a linear correlation between the number `n` of periodic threads and the minimum period of the threads, T_{min}:

$$T_{min} \approx n * 23 \mu sec$$

The two graphs are overlapping and if `n > 20` there is no measured difference between Ravenscar and aJile.

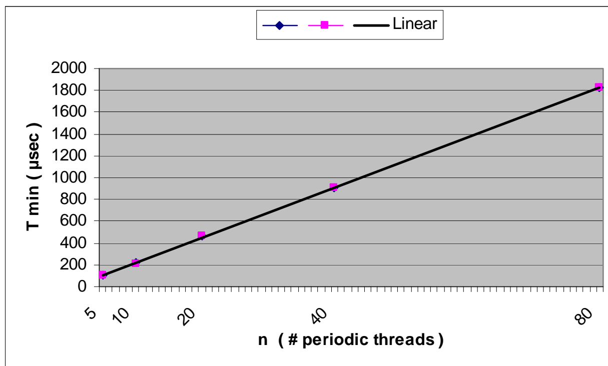

Figure 8. The minimum period, T_{min}, for `n` periodic threads

Next, we compare Ravenscar and aJile regarding initialization and code size.

Using the raw aJile API can be error prone, when there are many periodic threads with different periods, because nothing checks whether the setup conditions in Section 4.1.1 are satisfied; the programmer has to define all the setup constants manually, and then each periodic thread is made periodic with the right constants by calling the `makePeriodic` method, and finally the piano roll has to be setup with the right constants in its constructor.

In our Ravenscar implementation we hide all this, cf. section 4.1.2. The only constants we define are the periods and the max priority. The calculation of the rest of the constants is done in the `Setup` class, together with a validation of their consistency.

The difference in code size is negligible, whereas the difference in initialization is considerable. The aJile API is too low level and error prone.

6. ASSESSMENT

This section contains an assessment of the Ravenscar-Java profile itself and its suitability for the aJ-100 processor. Here we discuss also some further experiments with the implementation.

6.1 Assessment of the Ravenscar-Java profile

Since Puschner and Wellings [17] first proposed the Ravenscar-Java profile, there has been growing interest in evolving and refining the profile [14, 13]. Much of the work is analytical [16], but also critical work appears [20].

When implementing the Ravenscar-Java profile, our main references have been [15, 26, 13], besides the RTSJ [18]. In fact we have in the Java-doc for each class compared [15], [26], and [13] before deciding on an implementation. In [10] the code is documented in a similar manner, but it uses [15] as reference only.

During our implementation of the Ravenscar-Java profile some uncertainties and weaknesses have been uncovered. For example the `RealtimeThread` class is specified as follows in [15]:

```
public class RealtimeThread extends Thread
                             implements Schedulable
{
  RealtimeThread(PriorityParameters pp,
                 PeriodicParameters p);
  ...
  static boolean waitForNextPeriod();
}
```

and in [26]:

```
public class RealtimeThread extends Thread
                             implements Schedulable
{
  RealtimeThread(PriorityParameters pp,
                 PeriodicParameters p,
                 MemoryArea ma, Runnable logic);
  ...
  static boolean waitForNextPeriod();
}
```

and in [13]:

```
public class RealtimeThread extends Thread
                             implements Schedulable
{
  @HRTJProhibited
  public RealtimeThread(
    SchedulingParameters schedule,
    ReleaseParameters release);
  ...
  @HRTJProhibited
  public static boolean waitForNextRelease();
}
```

We note that the three specifications are not compatible. In [15] and [26], `waitForNextPeriod` has no access modifier; but in RTSJ [18] the method is `public`. Note that in [13] the method is

public, but called waitForNextRelease. We also note that one of the constructors has Runnable logic as a parameter.

The semantics of the waitForNextPeriod method in the Ravenscar-Java profile is somewhat underdefined, whereas a very elaborate semantics is given for the method in RTSJ, pages 249-251 in [26].

The description of waitForNextPeriod for Ravenscar-Java on p. 358 in [26] suggests that the method always returns true. However, Kwon, Wellings, and King [14] suggest the following implementation of the PeriodicThread class:

```
package ravenscar;
public class PeriodicThread extends
             NoHeapRealtimeThread
{
  public PeriodicThread(PriorityParameters pp,
    PeriodicParameters p, Runnable logic)
  {
    super(pp, p, ImmortalMemory.instance());
    applicationLogic = logic;
  }

  private java.lang.Runnable applicationLogic;

  public void run()
  {
    boolean noProblems = true;
    while(noProblems) {
      applicationLogic.run();
      noProblems = waitForNextPeriod();
    }
    // A deadline has been missed. If allowed, a
    // recovery routine would be placed here
  }
  ...
}
```

This implementation suggests that a recovery procedure could be implemented if a deadline is missed, i.e. if the call to waitForNextPeriod returns false, but it does not allow such a procedure (e.g. an AsyncEventHandler) to be passed as a parameter and the procedure would thus have to be hard coded into the profile implementation - which we believe is not a very sensible thing to do, especially not in a hard real-time application where off-line analysis is supposed to ensure that deadlines are not missed.

A minor point to criticise about the above suggested implementation is the use of a while loop in the idiom:

```
boolean noProblems = true;
while(noProblems) {
  applicationLogic.run();
  noProblems = waitForNextPeriod();
}
```

A more elegant and efficient solution is a do-while loop:

```
do {
  applicationLogic.run();
} while(waitForNextPeriod());
```

Not only would this save the use of a local variable, it would also lead to more efficient bytecode being emitted by the compiler.

However, as off-line analysis is supposed to ensure that deadlines are not missed, the loop is in effect an infinite loop. Thus waitForNextPeriod ought to just return void, and the loop could be implemented even more efficient as:

```
for(;;) {
  applicationLogic.run();
  waitForNextPeriod();
}
```

Next, we look at the sporadic event handling. It is often used to handle an external event caused by some error-state. The event does not happen very often, - but when it does, it is urgent and hence it has a short deadline D.

The Ravenscar-Java profile has no deadline in the SporadicParameters class [14], but HIJA [13] gives the following specification:

```
public class SporadicParameters extends
             ReleaseParameters
{
  public SporadicParameters(RelativeTime
    minInterarrival, RelativeTime deadline,
    AsyncEventHandler deadlineMissHandler);
  public RelativeTime getMinInterarrival();
}
```

In our implementation we follow the Ravenscar-Java profile. If the minimum interarrival time (MIT) is T, then the profile assumes that D = T. This implies that the rate monotonic setup can be used for setting up the NoHeapRealtimeThread belonging to the SporadicEventHandler, together with the setup of the periodic threads.

However, in some situations it would be useful to define a deadline less than the MIT, D < T. In this case the rate monotonic setup could be replaced by a *deadline monotonic setup* [7, p. 484], defined by:

$D_i < D_j \Rightarrow P_i > P_j$, where P is the priority.

Another uncertainty about the sporadic event handling is what should happen if the MIT is violated. The profile says nothing about this, even though the RTSJ allows the application to specify one of four possible MIT violation policies: EXCEPT, IGNORE, REPLACE or SAVE, see class SporadicParameters in [18]. We have implemented the IGNORE policy (Section 4.2.2) because it seems to be most in line with the Ravenscar-Java philosophy, as a violation should never occur due to off-line analysis.

The Ravenscar Java profile follows the RTSJ philosophy of an application being defined as a set of classes with at least one class having a public static main method, i.e. following the Java application philosophy. However, as Ravenscar-Java is targeted towards smaller and embedded systems implemented on top of J2ME and CLDC, it would perhaps have been more natural to define a Ravenscar-Java application as a "-let", e.g. a RAVENSCARlet, just as applications for mobile devices are defined as MIDlets. Following this idea a Ravenscar-Java application would extend a class RAVENSCARlet. The RAVENSCARlet class would have an InitializeApp method replace the initialize phase in Ravenscar-Java today. This method would set up the system resources and create the periodic threads. The RAVENSCARlet class would also have a startApp method to be called when the system is ready to run. It may also be useful to have a destroyApp method which could be used to take down the system gracefully. Although the Ravenscar-Java philosophy seems to be that the mission phase runs forever, practical systems, even embedded real-time systems may have to be shut down, e.g. for maintenance.

A further argument for pursuing this approach is that the vision of extensive off-line analysis of Ravenscar-Java programs is very much in line with the off-line verification of security and resource properties already in use in J2ME/CLDC/MIDP development. It would be natural for developers already familiar with J2ME to use such tools as part of the software development environment

6.2 Assessment of the aJ-100 processor

It has been relatively straightforward to implement the Ravenscar-Java profile on the aJ-100 processor, because aJ-100 has much of the functionality specified in RTSJ, - either in the microcoded real-time kernel or in its API.

Table 2 shows that the full implementation of the Ravenscar-Java profile requires only a total of 1550 lines of code (comments and brackets are not included). Of these, the utility classes for setup and check are about 900 lines.

Table 2. #classes and #code-lines in Ravenscar-Java profile

	# classes	# code lines	avg. code lines/class
Ravenscar classes	35	650	19
Utility classes	15	900	60
Total	50	1550	31

To hide the aJile-specific implementation details, two main strategies have been used:

- the Singleton design pattern, ensuring that a class has only one instance, and providing access to the singleton object by a `public static getInstance` method. Examples are the `Setup` and `CheckCyclicSetup` classes (see Figure 5) and the `SetupGpioPin` class (see Figure 7),
- private inner classes, ensuring both privacy of the class and access to data structures in the surrounding scope. An example is the aJile periodic thread in the Ravenscar `PeriodicThread` class (Section 4.1.2), another is the anonymous `TriggerEventListener` class (Figure 7).

In other implementations, the processor-specific details are typically hidden in native functions, often written in C, and specified as `native` Java methods. In this case two different system development methodologies are necessary: structured system development for the C-part and object-oriented system development for the Java-part. In our implementation on the aJ-100 processor everything is written in Java. Thus we only need to use one system development methodology: the object-oriented method.

Initial experiments suggest that it is an efficient platform for real-time systems using the Ravenscar-Java profile:

- changing from thread-to-thread: < 1 μsec,
- call of a method: 1.2 μsec,
- up to 500 periodic threads, each with a stack size of 1000 bytes,
- execution time nearly the same as JOP (Java Optimized Processor) [19],
- execution time comparable with C, because the bytecodes are executed directly [12].

We have omitted implementing the scoped memory concepts of the Ravenscar-Java profile. Scoped memory is a controversial topic and hard to use, see [19, 8]. Furthermore, we have so far not seen any applications that could not be programmed using immortal memory instead.

7. CONCLUSION AND FUTURE WORK

We have described our work of implementing the Ravenscar-Java profile on the aJ-100 processor. The implementation has been relatively straightforward as the aJ-100 processor has a rich API with much of the functionality specified in RTSJ. As this API is written in Java only, the object-oriented development method is supported, as opposed to other implementations where such APIs typically are implemented in C, thus calling for traditional structured development as well.

Clearly one may ask: why implement the Ravenscar-Java profile on an (almost) RTSJ compliant processor in the first place? There are three arguments for doing it:

1. The aJile API has a number of proprietary features that make portability of applications hard.
2. RTSJ is complex and the aJile API is too low level, while the Ravenscar-Java profile is simpler and more amiable to off-line analysis, for instance with tools like UPPAAL [25].
3. The programming model presented by the Ravenscar-Java profile will be relatively easy to use for developers familiar with applications for small devices using J2ME, CLDC, and profiles like MIDP.

The last point could be further strengthened by introducing a `RAVENSCARlet` concept analogous to the `MIDlet` concept for mobile applications.

During our implementation of the Ravenscar-Java profile some uncertainties, inconsistencies and weaknesses have been uncovered, especially the semantics of the `waitForNextPeriod` method is somewhat underdefined and could in our opinion beneficially be simplified to return `void` instead of a `boolean` value signalling a deadline miss, something that off-line analysis should prevent anyway.

Preliminary benchmarks show that the implementation is fast enough to be competitive with similar implementations in C/C++. However, execution speed is not overly important for real-time systems as long as the implementation is fast enough to satisfy the real-time constraints that a given application demands. Predictability of the implementation is much more important and this is addressed very well by Ravenscar-Java. When these two issues have been resolved, the remaining issues are all software engineering issues: how easy is it to write application code, how easy is it to maintain the code, how reusable and how portable is the code?

Our intial experiments indicate that applications written for the Ravenscar-Java profile will be easier to write, maintain and port. To further substantiate this belief we plan to use our implementation of the Ravenscar-Java profile in a larger industrial application scenario together with FOSS Analytical A/S, Denmark [9], who constructs advanced equipment for chemical and micro-biological analysis for use in the food industry and chemical industry. We plan to compare the Java implementation with a C/C++ implementation, in terms of functionality, reliability, analysability, maintainability and development time.

8. REFERENCES

[1] aJile Systems. www.ajile.com/, as of May 2006.

[2] aJile Systems. Products. Available at www.ajile.com/products/aj100.htm, as of May 2006.

[3] aJile Systems. *aJile Systems CLDC Runtime Version 3.16.09.* Available at www.ajile.com/support/, as of May 2006.

[4] aJile Systems. *Periodic Threads*. Internal white paper, July 2004, personal communication.

[5] Greg Bollella et al. *The Real-Time Specification for Java™*. Add-Wesley, 2000.

[6] Alan Burns, Brian Dobbing, and George Romanski. The Ravenscar Tasking Profile for High Integrity Real-Time Programs. In *Proc. Ada Europe Conf.*, pages 263 – 275, Uppsala, Sweden, 1998.

[7] Alan Burns and Andy Wellings. *Real-Time Systems and Programming Languages. Ada 95, Real-Time Java and Real-Time POSIX.* Add-Wesley. 3rd Edition. 2001.

[8] Peter C. Dibble. *Real-Time Java Platform Program-ming.* Sun Microsystems Press/Prentice Hall, 2002.

[9] FOSS Analytical A/S, Denmark. www.foss.dk/, as of May 2006.

[10] Ludovic Gauthier and Marc Richard-Foy. *Expresso. Realtime Java for Safety and Mission Critical Embedded Systems.* 2003. Available at www.irisa.fr/rntl-expresso/, as of May 2006.

[11] James Gosling & Henry McGilton. *The Java Language Environment.* White paper. May 1996. Available at java.sun.com/docs/white/langenv/, as of May 2006.

[12] David S. Hardin. *aJile Systems: Low-Power Direct-Execution Java™ Microprocessors for Real-Time and Networked Embedded Applications.* White paper. Available at www.ajile.com/papers/, as of May 2006.

[13] HIJA. *High-Integrity Java Application*. Project Number IST-511718, Version 2.4. 8, June 2005, available from Andy Wellings.

[14] Jagun Kwon, Andy Wellings, and Steve King. Ravenscar-Java: A High Integrity Profile for Real-Time Java. In *Proceedings of JGI'02*, pages 131-140, Seattle, Washington, 2002.

[15] Jagun Kwon, Andy Wellings, and Steve King. *Ravenscar-Java: A High Integrity Profile for Real-Time Java.* Department of Computer Science, University of York, UK. York Technical Report YCS 342. May 2002.

[16] Jagun Kwon, Andy Wellings, and Steve King. Predictable Memory Utilization in the Ravenscar-Java Profile. In *Proc. 6th IEEE Internat. Symp. on Object-Oriented Real-Time Distributed Computing*. May 2003.

[17] Peter Puschner and Andy Wellings. A Profile for High-Integrity Real-Time Java Programs. In *Proc. 4th IEEE Symposium on Object-Oriented Real-Time Distributed Computing*. 2001.

[18] Real-Time Specification for Java (RTSJ). www.rtsj.org/, as of May 2006.

[19] Martin Schoeberl. *JOP: A Java Optimized Processor for Embedded Real-Time Systems*. Dissertation. Vienna University of Technology, January 2005.

[20] Martin Schoeberl. Restrictions of Java for Embedded Real-Time Systems. In *Proc. 7th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 93–100, Vienna, Austria. May 2004.

[21] J. Schwartz. *Welcome to the 2006 JavaOne Conference.* java.sun.com/javaone/sf/Jonathans_welcome.jsp, as of June 2006.

[22] Hans Sondergaard. *Periodic threads on aJ-100.* White paper. 2004. Available at www.cs.aau.dk/ravenscar/output.htm, as of May 2006.

[23] Sun Java Real-Time System. java.sun.com/j2se/realtime/, as of May 2006.

[24] Systronix. www.systronix.com/, as of May 2006.

[25] UPPAAL. www.uppaal.com/, as of May 2006.

[26] Andy Wellings. *Concurrent and Real-Time Programming in Java.* Wiley. 2004.