Lightweight Firm Real-Time Extensions for Low Memory Profile Java Systems

Helge Böhme and Ulrich Golze

Technische Universität Braunschweig, Abteilung Entwurf integrierter Schaltungen,
Mühlenpfordtstraße 23, 38106 Braunschweig, Germany
{boehme,golze}@eis.cs.tu-bs.de

Abstract. This paper shows a novel approach to real-time extensions for Java running on low memory profile systems. Dynamic time-driven firm real-time scheduling is rendered possible by merely using regular Java threads. A three level scheduling scheme using earliest deadline first (EDF), fair-share and round-robin policies is integrated into a Java virtual machine kernel. The application programming interface (API) of this extension is elementary. Just one class representing deadlines must be added to the system library. Time critical code regions are encapsulated using standard synchronization techniques on deadline objects. They are applicable to oneshot, periodic and sporadic tasks. The underlying concepts of this technique are characterized in detail and applications are presented that already make use of these real-time capabilities.

1 Introduction

The vision of the future household with communicating devices and an integrated control of the whole system motivates our researches. The key point of the smart home are low priced embedded systems. Every electrical component will need an integrated controller to communicate over a home network. Node software can be quite complex because the system has to support complex control scenarios. We aim to use object oriented software to handle the physical objects of the house.

1.1 The JControl Project

The JControl project focuses on embedded control especially in the future household. We have developed a Java virtual machine running on 8-bit microcontrollers for home network node realization. To reduce overall system cost, these nodes must be small and cheap, performance is not an issue. Applications range from embedded regulators to human user interfaces using a graphical display and touch screen. All implementations fit in a 16-bit address range. Typical controllers have 60 KBytes on-chip ROM and less than 2 KBytes of RAM, some are equipped with external flash memory.[1]

[1] Our current reference implementation resides on ST-Microelectronics ST7 microcontroller running at 8 MHz core clock. VM kernel and most parts of the class library are natively implemented in assembler [1].

The virtual machine is a clean room implementation of the specification [2] with reduced primitve data type set similar to the CLDC 1.0 configuration of the J2ME [3] (no support of `float`, `double` and `long`). The JavaVM supports multithreading and garbage collection using a concurrent conservative mark-sweep-compact scheme. The VM is using a microkernel architecture with bytecode interpreter, memory manager and thread scheduler. Other components of the Java runtime (e. g. class preparation and garbage collection) are implemented on top of the kernel in a thread.

To handle our application domain under this restriced memory situation we implemented our own class library. The JControl API consists of a reduced set of core Java classes needed by the Java language (some classes of the `java` package) and our own classes for communication and control (in the `jcontrol` package). A typical implementation makes use of 37 classes, interfaces and exceptions.

To improve user interface feedback and in particular implement feasable and accurate regulators we intended to have real-time extensions that fit into our memory profile and class library.

1.2 Related Work

We did not found many projects providing Java virtual machines in the 64 K memory range. Real-time capabilities of these systems are not distinct nor existent. The simpleRTJ [4] supports multithreading but no priorities nor preemptions (possibly simple round-robin scheduling). A simple scheme for asynchronous events is implemented to handle native interrupts. The muvium microVM [5] uses ahead of time compiler techniques to be executed on a PIC microcontroller and doesn't support multithreading at all. The TinyVM [6] for use with Lego Mindstorms RXC module supports multithreading using a preemtive round robin scheduler. No timer is used for preemtion but a bytecode counter. There are no further real-time extensions.

Real-Time Extensions for Java. We considered to use a subset of an established real-time extension for Java. The Real-Time Core Extension (RT-Core, [7]) installs a second class hierarchy for real-time tasks (the *Core*). Core classes use a different verificator, scheduler and memory model than non-real-time classes (the *Baseline*). Using RT-Core would be a good choice for a new designed larger system but not for simple real-time attachments.

The Real-Time Specification for Java (RTSJ, [8]) is a more careful approach mainly using an additional API with virtual machine interface. A thread and event inheritance tree is used for specifying different kinds of real-time behavior. The system is completed by memory scopes and installable and configurable schedulers. The RTSJ is far too large to be used with our memory profile. Most interesting would be a subset of the RTSJ or a similar approach found in [9]. But also these approaches are too large or restrict real-time capabilities.

1.3 Our Approach

This paper shows a different approach for providing small Java systems with real-time capabilities. First we present in section 2 our real-time kernel to give an overview of the principles and functionalities that stand behind our real-time extension. We show how our EDF and fair-share scheduler works and explain some refinements of real-time scheduling. In section 3 we present our minimized real-time API using a time-based approach rather than an event-based. In section 4 we show that this concept is utilizable using some real-life examples. Furthermore we present some measured results how scheduling time depends on various parameters.

2 The Real-Time Kernel

Our virtual machine is not designed to run on top of an operating system. In fact all system management tasks are performed by the VM itself. A main part of the VM is the thread scheduler and dispatcher. Multithreading is implemented without native interrupt usage. After calling the bytecode interpreter to execute code of a thread, the bytecode interpreter is responsible for a reliable return to perform a context switch. The advantage of this behavior is simplicity. The bytecode interpreter will never leave a thread or an object in an undefined state. There is no need for locks on variables, all variables behave as if they were declared `volatile`.

The decision to return from interpreting a thread is taken by some events or directly. A direct return is performed if a thread needs a resource only availabe in another thread (e. g. if a class must be prepared) or by invocation of `Thread.yield()`. Event triggered returns are performed by frequently inspecting a bit field between the execution of bytecodes. Setting a bit works like a native interrupt at bytecode level. *Bytecode interrupts* may be triggered by native interrupts of the controllers interface hardware. A special bytecode interrupt is triggered by a programmable timer implementing a *millisecond counter*. This timer is used by the thread scheduler to apply time slices to thread execution (see below). The millisecond as time base was chosen to be sure that all non-interruptible activities (bytecodes and portions of native code, e. g. memory movements of the garbage collector) will fit into one period (see measurements in section 4). So thread scheduling jitter will always be less than one time increment.

2.1 Thread Dispatching

To simplify thread management the use of multiple run queues (e. g. for different priorities) was abandoned. All `Thread`s reside on the Java heap and are linked to a single list. The list is circular and an internal VM variable points to the currently executing or last executed thread. This list is processed every time the bytecode interpreter returns. The thread scheme presented in [10] was implemented but refined by sub-states (see figure 1). The sub-states of *blocked* are mainly for internal VM block cause distinction. If an event has occured, all

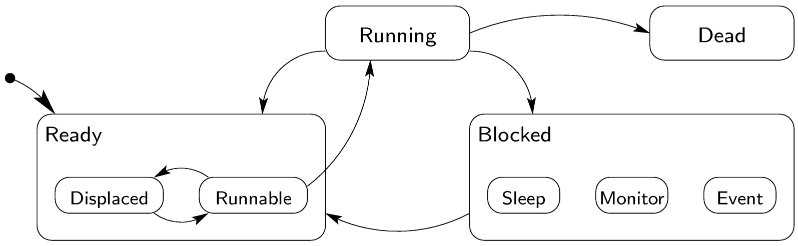

Fig. 1. Thread states

threads with matching state are put to *ready* state and they will check again their blocking condition on execution (and eventually fall back to *blocked* if the condition hasn't changed). Hence, the thread dispatcher doesn't need to know any blocking causes. Threads in the state *blocked by sleep* are handled by the thread dispatcher itself. Thread dispatching follows these steps:

- the scheduler scans the thread list and chooses the thread to be executed next (this is explained in the following section),
- if all threads are in a *blocked* state the system is put into sleep mode and rescheduling is performed after wakeup. If some of these threads are in *blocked by sleep*, the millisecond timer is programmed to wakeup the system just in time when the first of these threads should be scheduled next,
- if one *ready* thread is chosen by the scheduler the bytecode interpreter is called. But first the millisecond timer is programmed to interrupt bytecode execution to reschedule any awaking *blocked by sleep* thread just in time. If no thread has to be awoken by the thread dispatcher in near future, the interruption is done not later than a chosen *time slice*[2].

The thread list scan is always started at the next thread pointed by the last executed thread, so the default behavior (for non-blocked threads of the same priority) is preemptive round-robin scheduling with time slice.

2.2 The Scheduler

Our Java system is designed for interactive and real-time applications. According to [10] mapping the standard Java priorities using one of these two policies were considered:

Priority Scheduling. Only threads with the highest priority are allowed to run. This policy is used in most Java- and real-time systems. The advantage is a high scheduling determinism if all threads have their own dedicated priority and if no unexpected system coherences affect scheduling.

[2] The time slice (CPU quantum) is chosen for balancing scheduling costs and fluent concurrency [10], in our reference implementation it's 64 ms.

Fair-Share Scheduling. A thread's priority defines a rule for its *execution probability*. The advantage is fairness of CPU assignment to all *ready* threads even if some threads are not cooperative to other threads.

Fair-share scheduling was chosen for mainly two reasons. First the system is designed to accept third party software components. Thus prediction how thread-friendly a component will be (by intention or by mistake) is not possible. The second reason for a fair-share scheduler is the designated VM architecture. Some maintenance tasks like garbage collection are running in a thread. This thread must never be blocked by an application thread as this would compromise system reliability. Moreover, this thread (especially the garbage collector) should be unobtrusive to the application so this thread must run at low priority (read more on garbage collection below). The lack of scheduling determinism is compensated by our real-time extension.

2.3 Real-Time Scheduling

Like others our approach should support *periodic* and *sporadic* real-time tasks. Many Java real-time implementations support different tasks using different classes extending `Thread` or some real-time thread class. Every future task is prepared and then represented with an object similar to `Thread` on the heap waiting for its event to fire. To reduce overall object usage in the real-time implementation, the use of already existing threads for realtime tasks was determined. The idea is to add some attributes to the already existing `Thread` object to interact with the scheduler. This interaction is realized by a single class explained in the next main section.

Above, fair-share scheduling for non real-time threads was chosen. So for real-time threads priority based scheduling is not reintroduced but a time based scheduling paradigm is used. All a programmer of a real-time application knows is the designated *time constraint* of some periodic or sporadic task. The system should run out-of-the-box not requiring off-line feasability analysis of static scheduling policies. *Earliest Deadline First* (EDF) scheduling fits best these constraints. It's an efficient and easy to implement dynamic best-effort scheduling approach [11] enabling firm real-time behavior as shown in the following section.

2.4 Scheduler Implementation

To implement EDF scheduling a `deadline` value is associated with any `Thread`. A deadline is created at the point of time a thread enters a time-critical region relative to the current value of the millisecond counter. If no deadline is associated with a thread, it's handled as a thread with an infinite time constraint. So non-real-time threads become real-time threads with always lowest priority and the scheduler doesn't need to distiguish two kinds of threads.

To implement fair-share scheduling a `priorityCounter` is associated with any `Thread`. Every time the scheduler is called by the VM, the counter of all threads is incremented by its current Java `priority` (execution probability). So

Java priorities accumulate over time and threads with lower execution probability will also have a high counter value if not called for a long time.

Scheduling is done by two pass processing of the list of threads:

Pass 1. increment all priority counters, find the deadline that is in the nearest future and store the maximum counter value of the associated threads (additionally the nearest point of time to awake any *blocked by sleep* thread will be noticed here),

Pass 2. if any thread is *ready* find the first thread in the list matching the shortest deadline and maximum priority counter value.

Using this simple scheme, scheduling is done in three layers:

1. according to the EDF policy all *ready* threads with the earliest deadline are *runnable* and are priorized over all other threads which are *displaced*,
2. all *runnable* threads are passed to the fair-share scheduling policy,
3. *runnable* threads of the same execution probability are scheduled using the round-robin policy.

So the Java priority is a second order demand after deadlines. But scheduling using this model does not avoid priority inversion for its own.[3]

2.5 Priority Inversion Avoidance

The priority inheritance policy was implemented and used for deadlines in the first instance. Priority and deadline inheritance is performed on-the-fly by the scheduler. If a thread is entering the *blocked by monitor* state the thread holding the lock on the object is noted in the blocked thread. The scheduler also examines threads in the state *blocked*:

– If a deadline is associated with the blocked thread the scheduler is switched to the blocking thread and uses the minimal deadline of both threads for scheduling, so if any thread is blocking another thread with an earlier deadline it gets its privileges.
– The Java priority of blocked threads is also accumulated to blocking threads, so blocking threads get exactly the CPU time blocked threads release to the system unnoticable by unaffected threads.

In both cases this flow is recursive. As a side-effect the scheduler may detect dependency cycles and abort execution throwing a `DeadlockError`.

In addition a mechanism to improve responsiveness on events was implemented. If a thread changes from *blocked* to *ready* the priority counter is pushed to a high value considering the Java priority and it gets a probability boost. This mechanism doesn't affect deadlines.

[3] For example priority inversion occurs if a thread with an earlier deadline is *blocked* by a non-real-time thread with no associated deadline and there is another deadline later. The non-real-time thread is *displaced* until the later deadline is dismissed and probably the earliest deadline has already passed.

2.6 And What About Garbage Collection?

In many Java based real-time systems the garbage collector draws the attention because it compromises real-time behavior. Main problems with the garbage collector arise with priority scheduled threads. Because the garbage collector must be able to run everytime without blocking, it must have a high priority. Then the garbage collector decides when it runs and how often. But in most cases the garbage collector runs without knowledge of real-time scheduling requirements. So many real-time implementations use an incremental garbage collector (e. g. called after a `new`). But this slows down object allocation rate.

Using fair-share scheduling the garbage collector thread can run concurrently at low execution probability. So not the garbage collector decides when it runs but the real-time scheduler as for any other thread. The garbage collector is also involved in the priority and deadline inheritance scheme. If a thread is *blocked* because it runs out of memory the garbage collector gets its privileges until it completes. Other real-time and non-real-time threads not requiring memory allocations are running continuously. But also using this policy the execution time of real-time tasks may rise unpredictably on memory runout if the `new` operator is used inside a time-critical region. This must be kept in mind of the application programmer (use of prepare and mission phase, see next section).

Currently our garbage collector always runs with the same (lowest) probability. In some applications with huge memory allocation rate (e. g. `String` manipulations) affected threads may run frequently out of memory and block unavoidably until the garbage collector completes memory compaction. To reduce thread block rate in the future the garbage collector execution probability should be at least proportional to the memory allocation rate [12].

3 A Minimized Real-Time API

All real-time scheduling capabilities are only reclaimable if they can easily be used by the application programmer using an appropriate API[4]. As suggested above already existing threads are used and their scheduling attributes are changed at runtime. A naive approch is a pair of `static` methods in a `Thread` extending class to control the `deadline` attribute of the current thread. This concept works in principle but has a number of flaws:

1. it's up to the application programmer *always* to pair the instructions for entering and exiting time-critical regions,
2. it's up to the application programmer to check all invoked code inside the time-critical region for the recursive use of these instructions. If this is the case, the real-time behavior will be not as expected for both, the application code and the invoked code.

[4] The full API specification can be found in [13].

3.1 The Deadline Object

To implement an elegant solution to the stated issues, the `Deadline` object is introduced. It has the facility to access the private attributes of the current `Thread` using native code. To obtain a paired construction for entering and exiting deadlines real-time code is bound to synchronized code blocks.

Inside the virtual machine synchronization is performed by tagged methods or using the two bytecodes `monitorenter` and `monitorexit`. These bytecodes have the same problem as paired methods but in contrast the Java compiler takes care that these bytecodes are always paired. Furthermore, the `synchronize` construct has some security facilities. It is not possible to use the same object for synchronization at the same time by multiple threads. Recursive synchronizations on the same `Deadline` object have no effect because no further monitor is aquired. Recursive time constraints are possible using other `Deadline` instances. Using `Deadline` objects has a security effect itself, it's up to the programmer to publish the object to other classes or use it privately.

Changes inside the virtual machine are minimized to two code blocks representing entering and exiting a monitor (which are also used for synchronized methods). These routines were extended by a type check. If the synchronizing procedure was passed and the synchronized object is an instance of `Deadline` then equivalent code for entering and exiting an EDF region is called. On entering the `deadline` value is calculated from specified time constraint and current millisecond counter value. It's attached to the current thread object, the previous value is stored in the `Deadline` object. At exit the `deadline` is set to its previous value and the thread sleeps the remaining time until the current deadline expires.

The user interface usage of this scheme is outlined in figure 2. All code executed or invoked inside the synchronized block is bound to the specified deadline. One drawback can also be found in this scheme, since the `synchronized` command doesn't provide any possibility to throw an exception on a deadline miss. Instead this is performed by methods defined in `Deadline`. These methods behave similar to `Object#wait()` as an `IllegalMonitorStateException` is thrown in the case if they are not invoked from inside the dedicated synchro-

```
Deadline d=new Deadline(200);          // store time constraint
-> non-real-time code (prepare phase)
try {
    synchronized(d) {                  // set deadline after 200ms
      -> real-time code, max. 200ms (mission phase)
        d.check();                     // checks deadline, may throw dme
    }                                  // sleep until deadline expires
  -> non-real-time code
} catch(DeadlineMissException dme) {
  -> non-real-time code (emergency procedure)
}
```

Fig. 2. Oneshot task using `Deadline`

```
Deadline d=new Deadline(200);
-> non-real-time code (prepare phase)
synchronized(d) {                       // set deadline after 200ms
    for(;;) {
        try {
          -> real-time code, max. 200ms (mission phase)
            d.append(200);              // appends new deadline, may throw dme
        } catch(DeadlineMissException dme) {
          -> real-time code (emergency procedure)
        }
    }                                   // continue, even after miss
}                                       // sleep until last deadline expires
```

Fig. 3. Periodic task using `Deadline`

nized block. This ensures that the deadline is in use. The `check()` command just compares the current millisecond counter value with the deadline and throws a `DeadlineMissException` if required. This enables a simple run-time check of deadlines if this is the last command before leaving the synchronized block. The application programmer can decide where the exception is catched, this can be either outside the time critical section (as shown in the example) or even inside, then the emergency code sequence is still executed under real-time privileges. The programmer may even decide to implement soft real-time tasks and simply omit the invocation of `check()`.

The example shows only an oneshot real-time task. Deadline objects can also be used for periodic (time-driven) and sporadic (event-driven) tasks just using special design patterns:

Periodic tasks. are designed by concatenating further time constraints to an existing deadline using the command `Deadline#append(..)` inside some loop statement (see figure 3). The value of the concatenated time constraints may differ from the one associated with the deadline object for modulated periodicity. `append(..)` works similar to exiting and re-entering a synchronized block, the thread sleeps until the previous deadline expires. Appended deadlines are not relative to the current time found in the millisecond counter but relative to the previous deadline. So there will be no period drifting. As deadlines are adjoined, one missed deadline is no problem for the entire sequence, further periods are able to catch up.

Sporadic tasks. are designed straight forward using the already existing `wait()`/””`notify()` scheme (see figure 4). Invoking `wait()` on some synchronized object, the objects monitor is released and the thread is put into the *waiting for event* state. In the case of a `Deadline` object the deadline is also released. Later if the `Deadline` receives its notification signal, it reacquires the monitor and immediately enters a new deadline using its default time constraint.

Of course time critical regions can also be specified by extending `Deadline` and using synchronized methods.

```
try {
    Deadline d=new Deadline(200);
    synchronized(d) {                  // set deadline after 200ms
        for(;;) {
            d.wait();                  // releases monitor and deadline
          -> real-time code, max. 200ms (mission phase)
        }
    }                                  // sleep until deadline expires
} catch(DeadlineMissException dme) {
  -> non-real-time code (emergency procedure)
}
```

Fig. 4. Sporadic task using `Deadline`

4 Proof of Concept

Applications. We have used our real-time scheduling scheme in our 8 MHz reference system not expecting too much scheduling throughput. So applications requiring only low rate real-time capabilities were implemented. For example one of our GUI components represents an analog clock. The second hand advance is implemented using a periodic task using a `Deadline` object with time constraints of 1000 ms. This clock runs correct compared to a reference clock outside the system. Another GUI component using `Deadline` is a text scroller.

A more audible application is a iMelody player component.[5] Notes are played using the integrated PWM (pulse width modulation) controller of our reference system microcontroller. A periodic task with a `Deadline` object is used for note and pause durations in sequence. So `append(..)` is very useful in this case because it is able to fire many deadlined tasks in short time without memory throughput.

Finally, we are using `Deadline` in our intended system architecture: home-automation. Many devices are communicating using specified protocols.[6] Some protocols define communication time constraints, e. g. handshake exchange time-outs. So these time critical regions are assisted by our software.

Measurements. A critical part of a real-time system is the reaction time and its variation (jitter). To measure these key values our reference implementation kernel was exploited by additional native code. This code slightly affects the measurements by 48 μs per pass so this value is removed from the results. A Java test program was created doing several tests for 3 seconds each. Tests of three groups were done:

[5] iMelody is a text file standard of the Infrared Data Association (IrDA) for specifying melodies for one voice with note sequences [14], it's used by many mobile phones.

[6] We are using the FT1.2 protocol over RS232 serial communication for data exchange with power line modems. Currently we're implementing a high level protocol for use with the CAN bus.

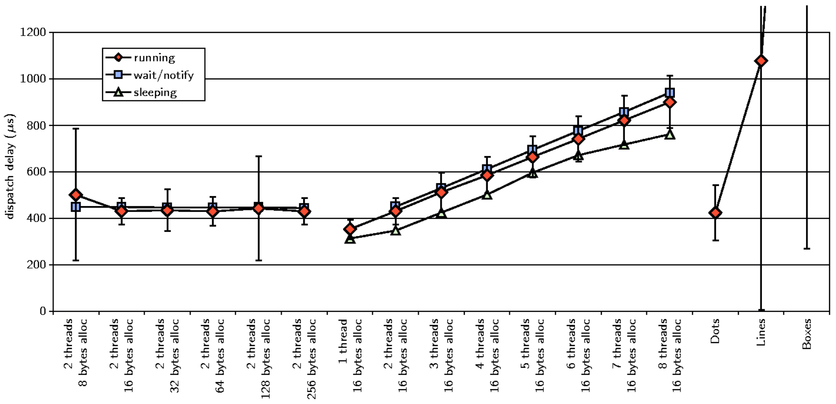

Fig. 5. Event processing delays

1. Memory usage (or garbage collector stress) using a fixed number of threads allocating arrays of different sizes in an endless loop. The garbage collector uses atomic memory block moves, larger blocks (arrays of 256 bytes) take 720 μs and are expected to affect event processing delay.
2. Thread scheduling costs using the same routine but instead changing the number of threads. Threads are processed in a linked list by the scheduler so event processing delays should correlate linearly.
3. Some basic graphical routines using random coordinates. The graphical routines are implemented using atomic native routines.

Figure 5 presents the results. Some tests were replayed using `wait()`/ `notifyAll()` pairs or `sleep()`. Note that this graph doesn't show overall system performance but system reaction time. The results are further influenced by two other threads, the `main()` thread starting the tests and the system thread performing garbage collection. Some results are not as expected. The garbage collector utilization with larger memory blocks doesn't affect the dispatch delay in a predictable way (see deviation marks). Here the block moves are uncorrelated to the events and may be finished just in time before a scheduling initiating event. In contrast the number of threads scales as predicted because thread processing is performed within the scheduler and scheduling is correlated to the events. Most measured delays are within a millisecond so using this value as system timer was a good choice. Programming real-time applications special care must be taken if a graphical display is used then the delay can exceed 5 ms.

5 Conclusion and Future Work

In this paper we presented our real-time implementation for low memory Java systems. In the range of Java systems below 64 K, explicit real-time capabilities are almost unique. The implementation depends on the Java virtual machine because main parts of the system (scheduler, deadline-interface using synchronized blocks) are integrated into the kernel. We introduced our three level dynamic best-effort policy using EDF, fair-share and round-robin scheduling with time slice. The scheduler is implemented as simple as possible using a single linked thread list. The scheduler uses each threads priority counter and deadline field and enables priority and deadline inheritance. The garbage collector is integrated into this concept.

The API of our real-time extension is very compact and consists of just one class (`Deadline`) and one exception. Our idea to bind real-time code regions to synchronized blocks enables compiler and VM assistance for consistency and security. Using special design patterns and methods of `Deadline` and `Object`, periodic and sporadic real-time tasks can be created and fired.

In the future the virtual machine will be redesigned to enable other target architectures than the ST7 microcontroller. The thread dispatcher and scheduler will be separated and the scheduling algorithm will be more configurable. It may be fully or partially implemented in Java and then integrated into the runtime system.

References

1. Böhme, H., Klingauf, W., Telkamp, G.: JControl – Rapid Prototyping und Design Reuse mit Java. In: 11. E.I.S.-Workshop, Erlangen. Volume GMM-Fachbericht 40., VDE-Verlag 2003 79–84 ISBN 3-8007-2760-9.
2. Bracha, G., Gosling, J., Joy, B., Steele, G.: The Java^{TM} Language Specification. Second edn. Addison-Wesley 2000 ISBN 0-201-31008-2 `http://java.sun.com/docs/books/jls`.
3. Sun Microsystems: Java^{TM} 2 Platform, Micro Edition (J2ME). `http://java.sun.com/j2me/`
4. RTJ Computing: simpleRTJ. `http://www.rtjcom.com`
5. Muvium: Muvium (PIC Java VM). `http://www.muvium.com`
6. Solorzano, J.: TinyVM (RCX Java). `http://tinyvm.sourceforge.net`
7. Real-Time Java^{TM} Working Group: International J Consortium Specification, Real-Time Core Extensions. Technical report 2000 `http://www.j-consortium.org/rtjwg/`.
8. Bollella, G., et al.: The Real-Time Specification for Java. Technical report, The Real-Time for Java Expert Group 2002 `http://www.rtj.org`.
9. Schoeberl, M.: Restrictions of Java for Embedded Real-Time Systems. `http://www.jopdesign.com`
10. Tanenbaum, A.S.: Modern Operating Systems. Second edn. Prentice Hall 2001 ISBN 0-13-031358-0.
11. Mathai, J.: Real-time Systems Specification, Verification and Analysis. Prentice Hall 1996 ISBN 0-13-455297-0, `http://www.tcs.com/techbytes/htdocs/book_mj.htm`.
12. Nilsen, K.: Adding real-time capabilities to Java. Communications of the ACM **41** 1998 49–56 `http://www.acm.org/pubs/citations/journals/cacm/1998-41-6/p49-nilsen/`.
13. Böhme, H., Klingauf, W., Telkamp, G.: Jcontrol platform api specification, jcvm8 edition. `http://www.jcontrol.org/html/javadoc/`
14. The Infrared Data Association: imelody v1.2 specification. `http://www.irda.org`