USENIX Association

Proceedings of the 2nd Java™ Virtual Machine Research and Technology Symposium (JVM '02)

San Francisco, California, USA
August 1-2, 2002

JaRTS: A Portable Implementation of Real-Time Core Extensions for Java

Urs Gleim
Siemens AG, Corporate Technology
urs.gleim@mchp.siemens.de, http://www.siemens.com/ct/

Abstract

To implement real-time tasks in Java the Java Virtual Machine has to be enhanced by the ability to run threads in a predictable time and to access hardware directly. One approach is to provide a real-time add-on for standard Java Virtual Machines. Benefits of this approach are a better predictability, a better scalability and more flexibility.

In this paper JaRTS—an implementation of the *Real-Time Core Extensions* specification—is presented. Some implementation details are inspected and first results are shown.

1 Introduction

Complexity of embedded and real-time systems increases from year to year and can only be handled by providing state-of-the-art programming languages. Usually software for those systems is written in C and even assembly language. This implies in-depth knowledge of the underlying hardware and the real-time operating system (RTOS) used. To handle complex software effectively a high abstraction layer of the programming language is needed. The Java Language provides an appropriate abstraction layer but is not designed for resource limited embedded systems and systems with real-time requirements. Thus the *Java Language Specification* [1] and the *Java Virtual Machine Specification* [2] need to be enhanced. In order to add real-time capabilities this is done by several specifications. Implementations of these specifications were not available until December 2001 when TimeSys [13] released the first reference implementation of the Sun's *Real-Time Specification for Java* (RTSJ) [3]. The *Real-Time Core Extensions* (RTCE) [4] are a competing specification by the Real-time Java Working Group.

In this paper we focus on real-time software such as control software for industrial automation. In this area there are systems running real-time software as well as non-real-time parts like graphical user interfaces on the same machine (e.g. the SICOMP industrial microcomputers [15]). On the other hand there are small controllers with very limited resources. Since RTCE fits better for these systems—this will be explained later—we implemented RTCE instead of RTSJ.

Designing real-time systems requires an overall system view. You cannot look at single parts like the Java Virtual Machine separately (assuming a common single processor system is used; hardware related topics are excluded). Therefore the paper starts with a definition of real-time systems, which are targeted. Based on that, current real-time operating system architectures are described in a nutshell. After an overview over the current real-time Java approaches the concept of the JaRTS Java real-time Java compiler will be discussed. Since real-time tasks run in a separate runtime environment the communication between real-time and non-real-time parts is shown in detail. Finally benchmarking results comparing JaRTS to other solutions are presented before we conclude.

2 Real-time systems

The term "real-time" is often used in different contexts with a different meaning. For example online systems with short response times are often named "real-time" systems. This is not meant in this paper. A canonical definition of a real-time system from Donald Gillies [12] is the following:

> A real-time system is one in which the correctness of the computations not only depends upon the logical correctness of the computation but also upon the time at which the result is produced. If the timing constraints of the system are not met, system failure is said to have occurred.

If the timing constraints *must always* be met we speak of a *hard real-time system*, it would be *soft real-time* if missed deadlines only lead to less throughput or an acceptable reduced quality of service. Most real-time systems have additional requirements concerning robustness and availability, which are not discussed in this paper.

Writing hard real-time applications for single-processor systems on top of an multi-threaded operating system needs in-depth knowledge of the scheduling strategy and the worst case times needed for system calls in order to ensure the timeliness of the application. Furthermore worst-case preemption times of every thread and especially system calls have to be known. This leads to one important parameter, the interrupt latency. *Interrupt latency* is the time between occurrence of the hardware interrupt and the entry of the software interrupt handler (interrupt service routine, ISR).

3 Real-time operating systems

The difference between non-real-time operating systems like Windows and Linux and real-time operating systems like VxWorks is that real-time operating systems have short guaranteed thread preemption times and interrupt latencies. A standard Linux for example has interrupt latencies up to a few hundred milliseconds [11] while the latencies of current real-time operating systems are around tens of microseconds which are guaranteed by the system architecture. Worst-case times of a standard Linux can be determined by measurements but there is no guarantee that there are worse cases with even longer latencies.

Real-time operating systems are much more uncomfortable to program than desktop operating systems. They often do not provide a process concept (memory protection), which separates memory spaces of different applications and there is not much abstraction from the underlying hardware. Because applications running on embedded devices become more and more complex additional abstractions are needed. Mostly it is only a small part of a real-time application really having real-time requirements. For the rest of the software one would like to have the convenience of desktop operating systems. Therefore it makes sense to modify desktop operating systems to satisfy embedded and real time needs (or having a programming environment like Java, which we discuss later). To add real-time capabilities to a non-real-time operating system there are two approaches:

- **Preemption Improvement:** Modify the operating system to be preemptible in a defined, short time.
- **Interrupt Abstraction:** is a two kernel model running the unmodified operating system as the idle-task on top of a separate scheduler (also known as *Interrupt Isolation* or *Interrupt Virtualization*).

The fist approach is implemented for example by MontaVista [14], who improved the preemption times of Linux. Two kernel solutions for Linux are RTLinux [7] and RTAI [8]. VxWin [16] and VenturCom's RTX [17] are two-kernel solutions for Windows NT.

Due to the complexity of operating systems like Linux it is a very difficult task to improve preemption times. Because it is really hard to examine every code path in the kernel, worst case preemption times cannot be guaranteed. The interrupt abstraction model allows predicting worse case times in the small real-time layer exactly. A detailed comparison of the two approaches for Linux can be read in the articles [9] [10] [11].

4 Requirements to an implementation

The range of application for real-time Java is quite huge. It can be used in small micro controller systems as well as in large systems having the power of current personal computers and workstations. The most important requirements to a real-time Java implementation in this context are portability and scalability:

- **Portability:** The real-time Java implementation should be available for many hardware platforms and the porting effort has to be minimal.

- **Scalability:** The runtime environment should be used for small resource limited systems up to large systems with graphical user interfaces.

Portability of the Java applications is not explicitly required. It is a Java intrinsic feature that programs are easy to port to different systems. But for the mentioned systems it is necessary to access the hardware directly (access hardware registers, install interrupt handlers, ...). There are frameworks like the *Real-Time Data access* [5] targeting this issue.

5 Real-time Java approaches

Currently there are two leading Specifications adding real-time capabilities to Java. Firstly the *Real-Time Specification* (RTSJ) [3] for Java produced by the Real-Time for Java Expert Group under the auspices of the Java Community Process [18]. In December 2001 the first reference implementation for the Real-Time Specification has been released by TimeSys [13]. Secondly the *Real-Time Core Extensions* (RTCE) [4] produced by the Real-Time Java Working Group supported by HP, Microsoft and other corporations. Both specifications cover the necessary enhancements to enable Java for real-time tasks:

- **Thread scheduling and synchronization:** The Java Language Specification [1] does not define the thread scheduling exactly. In addition the ten priorities provided by Java are not enough for most real-time tasks.

- **Memory management:** Automatic memory management of Java mostly leads to unpredictable timely behavior. There is no definition of worst-case memory allocation times and even concurrent garbage collectors are not preemptible without latencies.

- **Asynchrony:** Hardware interrupts and software events can occur asynchronously in real-time systems and require an *immediate* change of the control flow.

- **Hardware access:** Java does not provide direct access to hardware registers, physical memory and handling of hardware interrupts.

RTSJ implementations require modifications of the Java Virtual Machine (JVM) internals. The application developer can choose the thread scheduling algorithm and there are several strategies for memory management. The approach of RTCE is different. Instead of modifying the whole JVM RTCE can be implemented as a real-time add-on working closely with an arbitrary existing JVM.

6 Real-Time Core Extensions

For our requirements—portability and scalability—the Real-Time Core Extensions are the more suitable approach. The real-time part is small and can be implemented to be easy to port. Beyond that it can be used as a stand-alone solution as well as in combination with an off-the-shelf JVM. The separation of a non-real-time and a real-time runtime environment allows an implementation on operating systems implementing the Interrupt Abstraction approach.

According to the RTCE specification real-time parts of the application run in a runtime environment—called *Core Java*[1]—separated from the standard Java runtime environment—called *Baseline Java* (figure 1). The Core part can also be used as a stand-alone runtime environment. It has its own set of class libraries (`org.rtwg.*`), which are also separated from the standard Java class hierarchy. Root of the Core class tree is not `java.lang.Object` but `org.rtjwg.CoreObject`. This library contains special classes for handing thread scheduling, interrupts, memory, I/O and event handling. The Core runtime environment runs with a higher priority than the Baseline Java. In so doing it is assured that the Baseline threads and garbage collection has no influence on the real-time behavior of Core threads.

The classes of the Core library are shown in figure 2. They provide only basic functionality. The standard Baseline Java can access dedicated methods of Core

[1] The term *Core Java* was not the best choice by the Real-Time Java Working Group since it has a different meaning in the standard Java community.

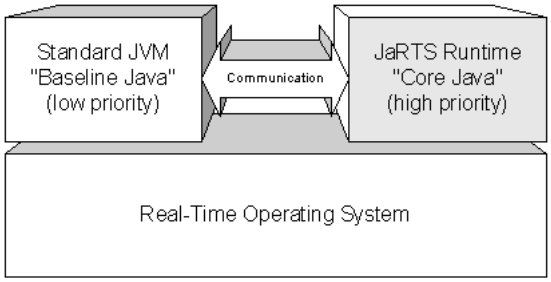

Figure 1: separation of real-time and non-real-time parts

objects, which are marked as *"Core-Baseline Methods"* (more details below). There is a small Baseline library for this communication, shown in figure 3. A detailed documentation of those libraries can be found in [4].

7 JaRTS: basic concept

For performance reasons it makes sense to compile the code to native machine code at compile time (ahead-of-time compilation). CPU time and memory is saved compared to a JIT-compiler and the costs of dynamic class loading are avoided. The real-time Java parts are used for drivers and embedded control algorithms which mostly do not need dynamic class loading (however a solution for dynamic class loading is planned, see section 14).

JaRTS (Java Real-Time by Siemens) is an implementation of a Core Java compiler and the RTCE libraries. Output of the JaRTS compiler is platform independent ANSI-C code as well as Java code for the Baseline-Core communication. The platform dependent parts are placed in separate operating system dependent include files. In addition the communication between Baseline and Core is implemented in platform dependent C files.

The prototype JaRTS compiler and runtime libraries were implemented for RTLinux. The Core parts of an application (real-time parts) are compiled to native code running directly on the real-time scheduler. For RTLinux the real-time code has to be compiled into kernel modules that can be loaded dynamically. Figure 4 shows the whole build process of a JaRTS real-time Java application. Platform dependent files for RTLinux are encircled.

```
CoreObject
     CoreThrowable
          CoreRuntimeException
          CoreException
          ScopedException
     CoreClass
     CoreArray
     AllocationContext
     SpecialAllocation
     CoreString
          DynamicCoreString
     ATCEventHandler
     ATCEvent
     CoreRegistry
     SignalingSemaphore
     CountingSemaphore
     Mutex
     Configuration
     Time
     Unsigned
     CoreTask
          ISR_Task
          SporadicTask
     IOPort

Interfaces:
     PCP
     Atomic
```

Figure 2: Core API

```
java.lang.Object
     lava.lang.Classloader
          BaselineCoreClassloader
     CoreDomain
     java.langException
     ObjectNotFoundException
     CoreBaselineRuntimeException
```

Figure 3: Baseline API

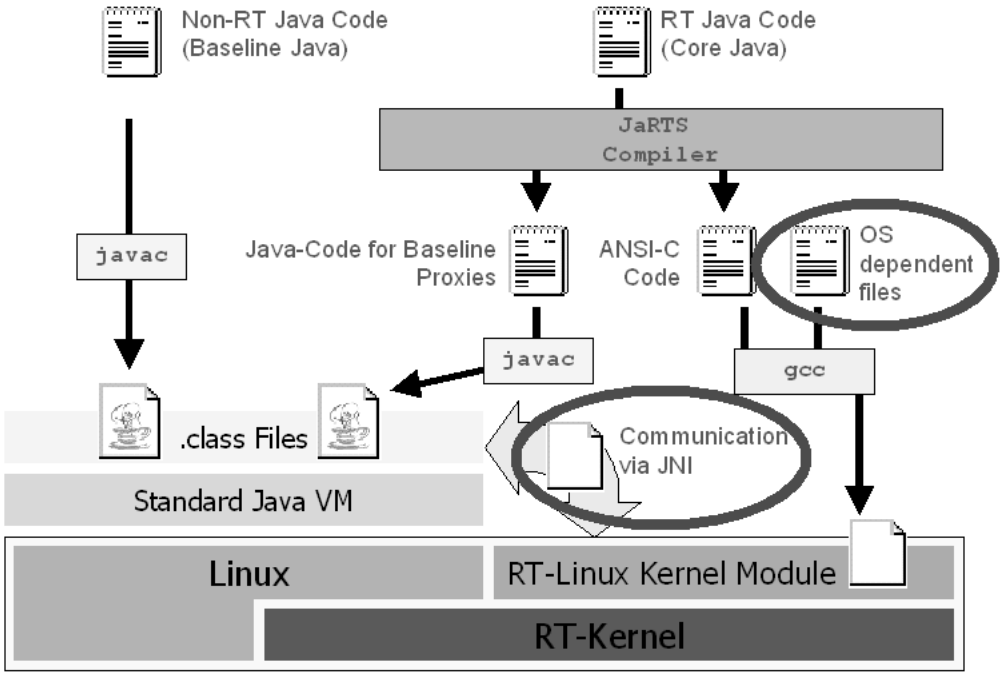

Figure 4: build process and platform dependent files

We implemented the communication of Baseline and Core Java via JNI (*Java Native Interface*) that is supported by most JVMs. Since all the real-time processing should take place in the Core part the interface between the Core and the Baseline part of an application is very small and the poor performance of JNI is not an issue.

8 JaRTS implementation

To get quick results the JaRTS compiler is based on in Open Source Java to C translator called *bock*, written by Charles Briscoe-Smith [6]. This translator did not support threads and therefore no synchronization. This was added to implement `CoreTask`, `ISRTask` and `SporadicTask` of the RTCE specification. Since we needed a periodic task—which is a task executing periodically after a specified time—we added an additional class `PeriodicTask`.

The Core library was implemented straightforward, mapping mutexes, interrupt handlers etc. directly to the functions provided by the underlying system. Native C code used in the libraries is weaved directly into the output of the compiler. So there is no additional runtime overhead for invoking native methods.

In the following section handling of the interrupt service routines, the communication between real-time and non-real-time parts as well as the memory management strategy is described.

9 Interrupt service routines

Because of object orientation Java requires runtime overhead even for conceptually simple tasks like interrupt handling. An interrupt service routine (ISR) in Java (RTCE) is implemented similar to a thread. A `work()` method has to be implemented in a subclass of `ISRTask`. The interrupt number is a member variable of this class. There can be several instances of the same class handling different interrupts.

To access member variables in the C translation of the `ISRTask` a pointer to the corresponding object has to be known inside the translated `work()` method.

In real-time operating systems ISRs are assigned to

a hardware interrupt by a system call. This call has only two parameters: the interrupt number and the address of the handler method. In RTLinux it would look like this:

```
rtl_request_irq(irnumber,
                  handler);
```

Of course it is not possible to pass arguments to the handler method because it will be called by the underlying operating system. Therefore the interrupt handler has to get the object pointer from somewhere else. In the JaRTS runtime environment there is a wrapper function for all interrupt handlers fetching an object pointer of the corresponding `ISRTask` object from a small table whenever an interrupt occurs. This wrapper is used for all interrupts handled by Core Java applications. Roughly the wrapper function looks like this (some details left out):

```
void isr_entry(int irq) {
  isr_object* o=table[irq];
  (o->methods->work)(o);
}
```

This is connected to every handled interrupt by

```
rtl_request_irq(irnumber,
                  isr_entry);
```

The object pointers are written to the table during initialization of the `ISRTask`. Accessing the table (`table[]`) does not require mutual exclusion because it will not be reallocated and values are only read (after the initialization phase where the handler will not be called with the current interrupt number).

So the Java overhead for ISRs is the table access and an additional function call in the `isr_entry()` function. This leads to slightly longer interrupt latencies compared to C but this is a predictable worst-case time. This time can be calculated by looking at the (assembly language) output of the C compiler with the assumption that all data and the code of the `work()` method is not in cache.

10 Baseline-Core communication

The prototype implementation was done for RTLinux [7]. As mentioned, RTLinux uses Interrupt Abstraction to make Linux real-time capable. This is a two-kernel solution where Linux runs on top of a real-time scheduler. Hence the Core Runtime environment has to run directly on the real-time scheduler and communication must be possible from the real-time threads to the non-real-time threads running on a standard JVM on top of the Linux kernel. This is implemented by some communication routines that use the FIFOs provided by RTLinux for communication between real-time threads and non-real-time Linux threads. These FIFOs are accessed by native code via JNI (figure 5).

10.1 The Core part

To access methods in the Core code from Baseline these methods have to be marked as Core-Baseline methods with the keyword *baseline*[2], for example (`CoreTask` is the Core thread class):

```
public class Controller extends
CoreTask {
...
public void baseline setSpeed(...)
...
public void baseline getSpeed()
...
```

By now only the methods which may be accessed by Baseline Code are marked.

Objects instantiated in the Core part can be published to Baseline with a string name:

```
MyCoreObject co =
       new  MyCoreObject();
CoreRegistry.
    publish("myObject01", co);
```

Objects are published by sending the name, the type and a unique object ID via FIFO to the Baseline

[2]There is an notation that doesn't introduce a new keyword: calling `CoreRegistry.registerBaseline(<mthd_signatures>)` in the static initializer

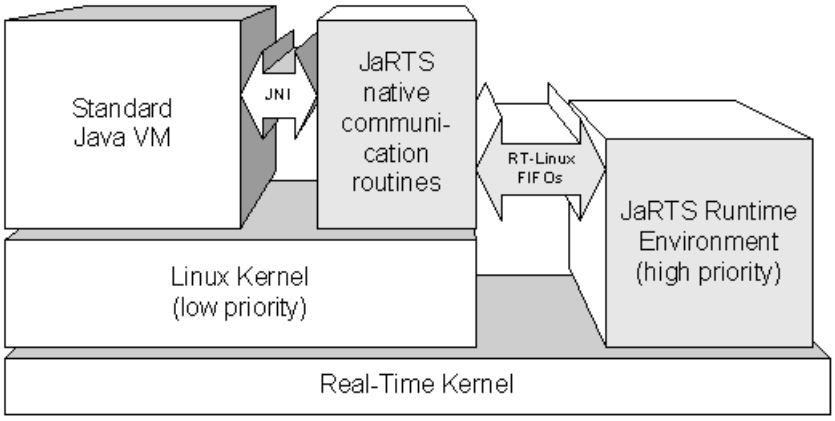

Figure 5: separation of real-time and non-real-time parts

JVM (described below). To receive method calls from Baseline a loop is listening at another FIFO for commands (`CoreListener`).

10.2 The Baseline part

The Baseline instance for Baseline-Core communication is `CoreDomain`. Previously published Core objects can be obtained by calling `CodeDomain.lookup()`, for example:

```
MyCoreObject bco =
CoreDomain.lookup("myObject01");
bco.foo(42, 3.1416);
```

What `CoreDomain.lookup()` actually returns is a proxy object doing the communication with the corresponding Core object. The JaRTS compiler generates the proxy classes for each Core class containing Core-Baseline methods. These proxy classes are subclasses of a Baseline version of `CoreObject` (figure 6).

We added an internal class `BaselineCoreConnector` that handles the connection. It is a singleton and used by the proxy objects. Therefore it is a static member of the proxy superclass `CoreObject`. A `BaselineFifoListener` waits for commands like publishing objects in the Core as well as for returning functions (figure 7). The actual communication is done in native files written in C. These files have to be adapted when porting JaRTS to other platforms.

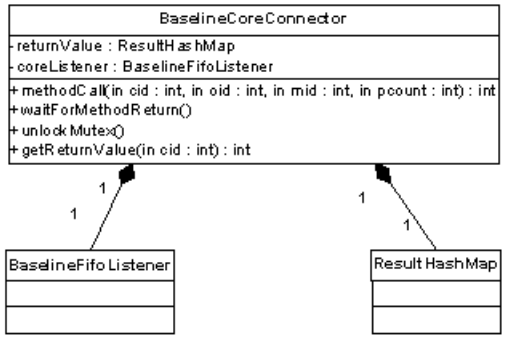

Figure 7: auxiliary classes for the JaRTS implementation

10.3 Calling Core-Baseline methods

Figure 8 shows a sequence diagram of the publication and a Core-Baseline method call. Arrows crossing the horizontal line between Baseline and Core are standing for data sent through the FIFOs. The data sent is described in the balloons.

Name, type and ID (for optimization) of a published object are received by the `FifoListener` and internally stored in a hash map of `CoreDomain` for lookup. The latter is not shown in the sequence chart. For every published object a proxy object on the Baseline side is created. This proxy objects contains wrappers for all the Core-Baseline methods in the corresponding Core object. When called, these wrappers send a "call method" command followed by the object ID, method ID and the parameters to the `CoreListener`. Since multiple threads can call

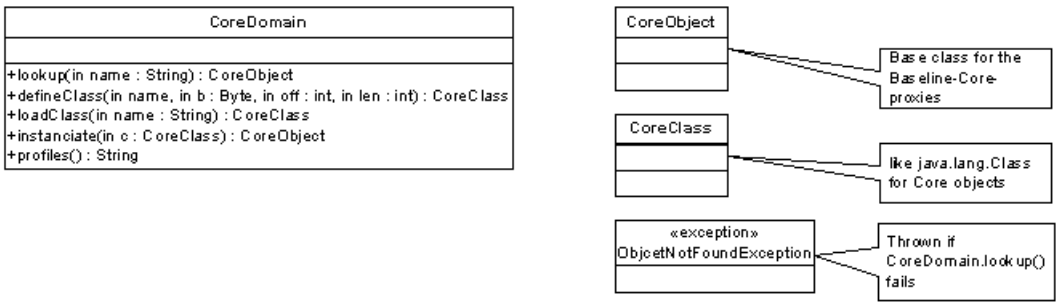

Figure 6: classes for Baseline-Core communication

the same method concurrently an additional call ID is sent. This is important to distinguish the method returns. The call ID can be a thread ID or a unique number. On the Core side a thread is started executing the wanted method.

The calling Baseline thread is blocked by a `wait()`. The calling thread will be resumed by a `notify()` after the calls Core method has sent its return values. Since multiple threads can call Core-Baseline methods concurrently return values are stored in a hash map with their call ID and can be fetched by the wrappers.

11 Memory management

The JaRTS runtime environment does not support garbage collection as known from Standard JVMs. Most real-time software components like interrupt handlers and device drivers would derive no or only a little benefit from having automatic garbage collection. On the other hand garbage collection imposes significant costs in terms of runtime efficiency, predictability and system complexity.

RTCE provides the concept of *Allocation Contexts*. Objects are allocated on an Allocation Context of the current thread. The Allocation Context is released when the thread terminates or can be released explicitly by the programmer. It is also possible to allocate objects on dedicated Allocation Contexts. The second memory strategy of RTCE, allocation objects on the stack is currently not implemented.

	Sieve	*Loop*	*Logic*
JaRTS	180	227	933
CVM	60	61	60
Sun 1.3	285	1098	769

Table 1: benchmarking results

12 Results

Up to now the results are promising:

Latency periods: As expected the interrupt latency periods are nearly the same as the latencies of interrupt handlers implemented in C. The average time between a hardware interrupt (at the parallel port of an 150 MHz Pentium) is 7 µs for interrupt service routines in C. The Java version needs about 9.5 µs. The Java overhead of 2.5 µs is acceptable for most real-time systems.

Performance: Performance was tested with a sieve of Eratosthenes, which calculates prime numbers, a loop test sorting values and a simple test of logical decisions.

Table 1 shows the results (score, higher values are better). Compared to Sun's HotSpot Client VM (1.3) the code generated by JaRTS was about 20% faster for the logic test. The other tests ran about 58% (sieve test) up to 366% (loop test) slower. Compared to Sun's CVM [20] (on which the TimeSys real-time reference implementation is based) JaRTS is 3 times up to more than 15 times faster.

Improvements of the JaRTS performance should still be possible since there was no effort in optimizing performance, yet.

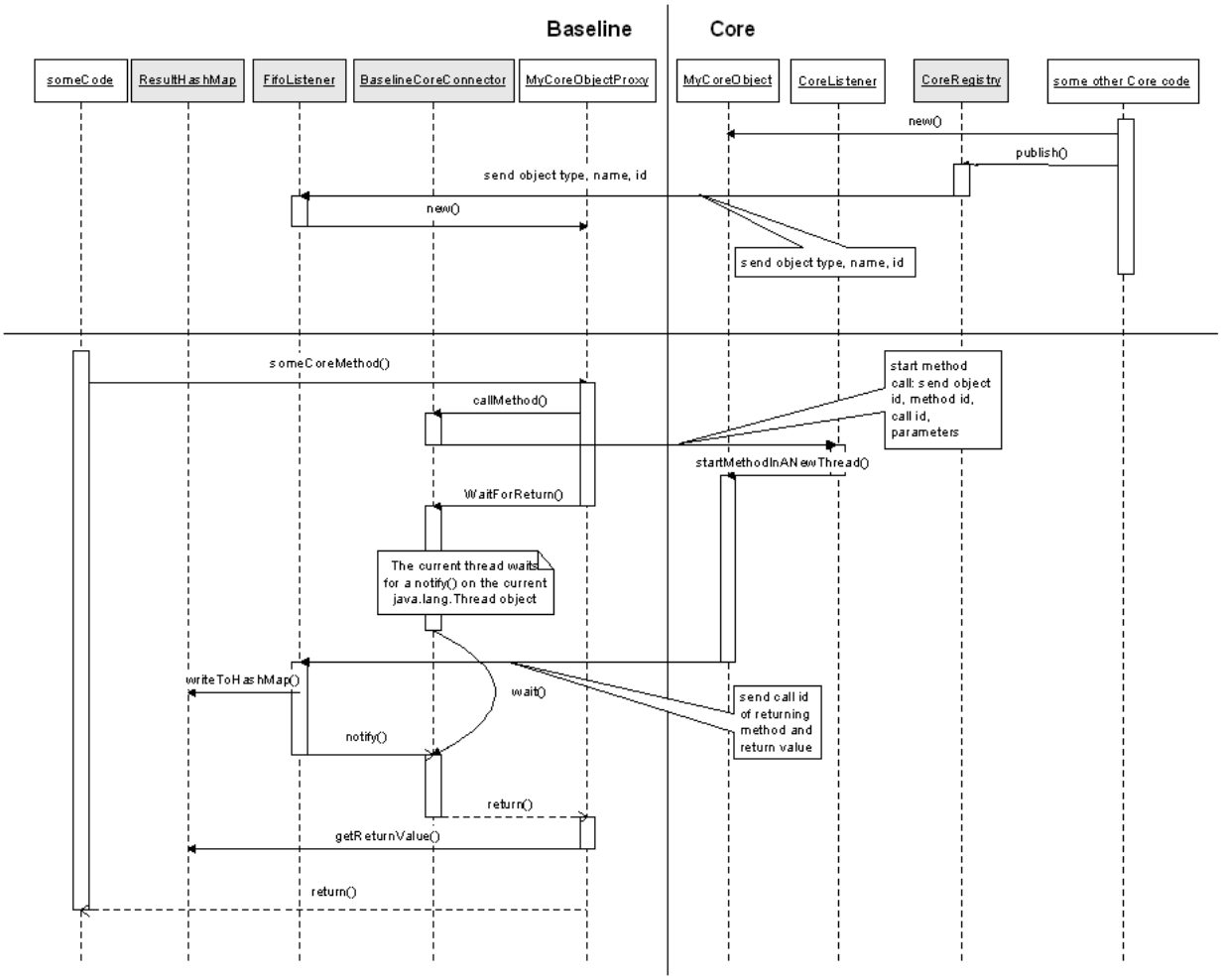

Figure 8: publishing methods and method call

Portability: There are only a few files containing platform dependent code. This code contains wrappers for handling threads, mutexes and semaphores and the code controlling the FIFOs for Baseline-Core communication. Because the JaRTS compiler generates platform independent ANSI-C code it can be ported to any system providing an ANSI-C compiler easily. Due to the separation of real-time and non-real-time parts it can also be run on systems using Interrupt Abstraction.

Code size: The Core run-time currently has a footprint of about 700 kilobytes including the Core libraries. This is about the same size as current MIDP [19] implementations, used for mobile phones. Until now there were no optimization efforts in this area, so there is still space for improvement.

13 Conclusion

We wanted to show that it is possible to implement a portable and scalable real-time Java extension with hard real-time capabilities.

The platform dependent code is relatively small and manageable. It is much easier to port JaRTS to a new platform than a complex JVM implementing RTSJ.

Currently the TimeSys RTSJ reference implementation is based on the CVM and the Foundation Profile libraries [20]. CVM is too big for resource limited systems (more than 2.5 MByte for CVM with Foundation Profile) where only a very small library would be sufficient.

In addition to this JaRTS can be used with any JVM implementing JNI. So the applications can use all available Java libraries. Compared to this CVM is very limited and, for example, does not provide graphics yet.

The following list summarizes the JaRTS features:

- Can be used on operating systems with Interrupt Abstraction (two kernel solution) easily
- Can be used on small embedded systems not requiring a full Java environment
- Easy to port to different systems
- Only some operating system dependent files used by the JaRTS compiler have to be adapted (rest: ANSI-C, compiler available for almost every operating system)
- An off-the-shelf JVM which is already available for most systems is used for the non-real-time parts. This JVM can support all known Java libraries.

Since JaRTS currently is a prototype implementation there is space for improvement in terms of memory and performance. The *bock* compiler was chosen to get to a first implementation very quickly. The translation was not tuned to be efficient and resource saving. Nevertheless with the current implementation it is possible to generate control programs with very low latency times and an acceptable performance.

14 Future work

As mentioned JaRTS is a prototype implementation and of course there are open issues:

Memory Management: Stack allocation has to be implemented.

Tooling: A Java runtime environment can only be used in an efficient way if there are good development tools. A remote debugger, profiling tools an a simulator for the JaRTS runtime have to be developed.

Benchmarks: A more sophisticated benchmarking suite has to be developed. One reason is to determine the performance and memory bottlenecks. The other reason are more convincing comparisons to other real-time Java solutions.

Optimization of the translation: The Java to C translation has to be optimized in terms of performance and memory consumption.

Dynamic class loading: Concepts for dynamic class loading (of Core classes) have to be investigated and implemented. Possible would be to compile loadable classes into loadable native libraries (for RTLinux this would be separate kernel modules, for other operating systems this would be shared libraries) or using the standard Java Bytcode (class files) and common techniques (interpreter, JIT compiler, compile at class-loading time). Compiling at class loading time using the existing compiler (JaRTS in combination with a C compiler) has very high memory and CPU requirements. The JIT solution needs much effort to port it to a new processor. So a feasable solution would be an interpreter or the precompiled loadable libraries.

15 Acknowledgements

I would like to thank my colleague Thomas Henties for a large part of the implementation and the *Siemens CT SE 2 Embedded Team* for many useful suggestions.

Thanks also to Charles Briscoe-Smith who wrote the *bock* compiler and published the source code in the Internet.

References

[1] James Gosling, Bill Joy, Guy Steele, Gilad Bracha: *The Java Language Specification (Second Edition).* Addison-Wesley. 2000
<http://java.sun.com>

[2] Tim Lindholm, Frank Yellin: *The Java Virtual Machine Specification (Second Edition).* Addison-Wesley. 1999
<http://java.sun.com>

[3] Greg Bollella e.al.: *The Real-Time Specification for Java (version 1.0).* Addison-Wesley. Dec. 2001
<http://www.rtj.org>

[4] J-Consortium, Real-Time Java Working Group: *Real-Time Core Extensions (revision 1.0.14).* Sept. 2000
<http://www.j-consortium.org/rtjwg/index.shtml>

[5] J-Consortium, RTDA Working Group: *Real-Time Data Access (version 1.8).* Febr. 2001
<http://rtawg.khe.siemens.de/rtawg.html>

[6] *Bock compiler.* available at:
<http://packages.debian.org/unstable/devel/bock.html>

[7] *RTLinux by FSM labs:*
<http://fsmlabs.com/community/>

[8] *RTAI (Real Time Application Interface):*
<http://www.aero.polimi.it/projects/rtai/>

[9] Tim Bird: *Comparing two approaches to real-time Linux.* Dec. 2000
<http://www.linuxdevices.com/articles/AT7005360270.html>

[10] Tim Bird: *Two Approaches to Real-Time Services in Linux.* RTC Magazine, November 2000

[11] Jim Ready, Kevin Morgan: *Application-Oriented Approach to Real-Time Linux.* RTC Magazine, November 2000

[12] *Comp.realtime: Frequently Asked Questions (FAQs).* (version 3.5)
<http://www.faqs.org/faqs/realtime-computing/faq/>

[13] *TimeSys.*
<http://www.timesys.com/>

[14] *MontaVista.*
<http://www.mvista.com/>

[15] Industrial microcomputers *SICOMP.*
<http://www1.ad.siemens.de/sicomp/index_76.shtml>

[16] *LP-Elektronik.*
<http://www.lp-elektronik.com/>

[17] *VenturCom.* Real-time Extensions for Windows NT
<http://www.vci.com/products/windows_embedded/rtx.asp>

[18] *Java Community Process.*
<http://www.jcp.org/>

[19] *Mobile Information Device Profile.*
<http://java.sun.com/ products/midp/>

[20] *CVM* and the *Foundation Profile.*
<http://java.sun.com/products/cdc/>
<http://java.sun.com/products/foundation/>