Restrictions of Java for Embedded Real-Time Systems

Martin Schoeberl
Strausseng. 2-10/2/55, A-1050 Vienna,
AUSTRIA
martin@jopdesign.com

Abstract

Java, with its pragmatic approach to object orientation and enhancements over C, got very popular for desktop and server application development. The productivity increment of up to 40% compared with C++ [1] attracts also embedded systems programmers. However, standard Java is not practical on these usually small devices. This paper presents the status of restricted Java environments for embedded and real-time systems. For missing definitions, additional profiles are proposed. Results of the implementation on a Java processor show that it is possible to develop applications in pure Java on resource constraint devices.

1 Introduction

Java was first used in an embedded system. In the early '90s Java, which was originally known as Oak, was created as a programming tool for a wireless PDA. The device (known as *7) was a small SPARC based hardware device with a tiny embedded OS. However, *7 was not issued as a product and Java was officially released in 1995 as a new language for the internet (to be integrated into Netscape's browser). Over the years, Java technology has become a programming tool for desktop applications and web services. With every new release, the library (defined as part of the language) grows bigger and bigger.

Java for embedded systems was clearly out of Sun's focus. With the arrival of mobile phones, Sun again became interested in this embedded market. Sun defined different subsets of Java, which are analyzed in this paper.

As the language became popular, with easier object oriented programming than C++ and threads defined as part of the language, its usage in real-time systems was considered. Two competing groups began to define how to convert Java for use in these systems.

Nilsen published the first paper on this subject in November 1995 [2] and formed the Real-Time Working Group. The other group, known as the Real-Time Expert Group, published the RTSJ (Real-Time Specification for Java) [3]. RTSJ was the first specification request under Sun's Java Community Process and gained much attention from academic and industrial research.

This paper will give:

- An extended overview of actual specifications for Java for embedded and real-time systems
- Proposed definitions to fill the gap for small embedded systems, implemented on JOP (a Java Optimized Processor) [4]

This paper is structured as follows. Section 2 summarizes the problems with standard Java on embedded systems. The various definitions for small devices given by Sun are described in Section 3. Section 4 gives an overview of the two real-time extensions of Java and approaches for restricting RTSJ for high-integrity applications. If, and how, these specifications are sufficient for small embedded systems in general and specifically for JOP is analyzed. In Section 5 the missing definition for small embedded real-time systems is proposed. Implementation results of this definition and conclusions are presented in Section 6 and 7 respectively.

2 Java Support for Embedded Systems

When not using the cyclic executive approach, programming of embedded (real-time) systems is all about concurrent programming with time constraints. The basic functions can be summarized as:

- Threads
- Communication
- Activation
- Low level hardware access

2.1 Threads and Communication

Java has a built in model for concurrency, the class Thread. All threads share the same heap resulting in a shared memory communication model. Mutual exclusion can be defined on methods or code blocks with the keyword synchronized. Synchronized methods acquire a lock on the object of the method. For synchronized code blocks, the object to be locked is explicitly stated.

2.2 Activation

Every object inherits the methods wait(), notify() and notifyAll() from Object. These methods in conjunction with synchronization on the object support activation.

The classes java.util.TimerTask and java.util.Timer (since JDK 1.3) can be used to schedule tasks for future execution in a background thread.

2.3 Problems

Although Java has language features that simplify concurrent programming the definition of these features is too vague for real-time systems.

2.3.1 Threads and Synchronization

Java, as described in [5], defines a very loose behavior of threads and scheduling. For example, the specification allows even low priority threads to preempt high priority threads. This protects threads from starvation in general purpose applications, but is not acceptable in real-time programming. Even an implementation without preemption is allowed.

Wakeup of a single thread with notify() is not precisely defined: *the choice is arbitrary and occurs at the discretion of the implementation.*

It is not mandatory for a JVM to deal with the priority inversion problem.

2.3.2 Garbage Collector

Garbage collection greatly simplifies programming and helps to avoid classic programming errors (e.g. memory leaks), but is not suitable for real-time systems and problematic in embedded systems. A more conservative approach to memory allocation is necessary.

2.3.3 WCET on Interfaces (OOP)

Interfaces (and method overriding), the key concepts in Java to support object oriented programming, are problematic for WCET (Worst Case Execution Time) analysis such as function pointers in C. Implementation of interface look up usually requires a search of the class hierarchy or very large dispatch tables.

2.3.4 Dynamic Class Loading

Dynamic class loading requires the resolution and verification of classes. This function is usually too complex (and too memory consuming) for embedded devices. Upper bound of execution time for this function is almost impossible to predict (or too large). This results in complete avoidance of dynamic class loading in real-time systems.

2.3.5 Standard Library

For an implementation to be Java-conformant, it must include the full library (JDK). The JAR files for this library constitutes about 15 MB (in JDK 1.3, without native libraries), far too large for many embedded systems.

Since Java was designed to be a save language with a save execution environment no classes are defined for low-level access to hardware features.

The standard library was not defined and coded with real-time applications in mind.

3 Java Micro Edition

The definition of Java also includes the definition of the class library (JDK). This is a huge library and too large for some systems. To compensate for this Sun has defined the *Java 2 Platform, Micro Edition* (J2ME) [6]. As Sun has changed the focus of Java targets several times, the specifications reflect this through their slightly chaotic manner. J2ME reduces the function of the JVM (e.g. no floating point support) to make implementation easier on smaller processors. It also reduces the library (API).

J2ME defines three layers of software built upon the host operating system of the device:

Java Virtual Machine: This layer is just the JVM as in every Java implementation. Sun has assumed that the JVM will be implemented on top of a host operating system. There are no additional definitions for the J2ME in this layer.

Configuration: The configuration defines the minimum set of JVM features and Java class libraries available on a particular category of devices. In a way, a configuration defines the lowest common denominator of the Java platform features and libraries that the developers can assume to be available on all devices.

Profile: The profile defines the minimum set of Application Programming Interfaces (APIs) available on a particular family of devices. Profiles are implemented upon a particular configuration. Applications are written for a particular profile and are thus portable to any device that supports that profile. A device can support multiple profiles.

There is an overlap of the layers *configuration* and *profile*: Both define/restrict Java class libraries.

Sun states: '*A profile is an additional way of specifying the subset of Java APIs, class libraries, and virtual machine features that targets a specific family of devices.'* However, in the current available definitions JVM features are only specified in *configurations*.

3.1 Connected Limited Device Configuration (CLDC)

CLDC is a configuration for connected devices with at least 192 kB of total memory and a 16-bit or 32-bit processor. As the main target devices are cellular phones, this configuration has become very popular (Sun: '*CLDC was designed to meet the rigorous memory footprint requirements of cellular phones.*'). The CLDC is composed of the K Virtual Machine (KVM) and core class libraries.

The following features have been removed from the Java language definition:

- No floating point support
- No finalization

Error handling has been altered so that the JVM halts in an implementation-specific manner.

The following features have been removed from the JVM:

- Floating point support
- Java Native Interface (JNI)
- Reflection
- Finalization
- Weak references
- User-defined class loaders
- Thread groups and daemon threads
- Asynchronous exceptions
- Data type long is optional

These restrictions are defined in the final version 1.0 of CLDC. A newer version (1.1) again adds floating-point support. All currently available devices (as listed by Sun) support version 1.0.

The CLDC defines a subset of the following Java class libraries: java.io, java.lang, java.lang.ref and java.util.

An additional library (javax.microedition.io) defines a simpler interface for communication than java.io and java.net. Examples of connections are: HTTP, datagrams, sockets and communication ports.

A small-footprint JVM known as K Virtual Machine (KVM) is part of the CLDC distribution. KVM is suitable for 16/32-bit microprocessors with a total memory budget of about 128 kB.

When implementing CLDC, one may choose to pre-load/prelink some classes. A utility (*JavaCodeCompact*) combines one or more Java class files and produces a C files that can be compiled and linked directly with the KVM.

There is only one profile defined under CLDC: the Mobile Information Device Profile (MIDP) defines a user interface for LC displays, a media player and a game API.

3.2 Connected Device Configuration (CDC)

The CDC defines a configuration for devices with network connection and assumes a minimum of a 32-bit processor and 2 MB memory. CDC defines no restrictions for the JVM. A virtual machine, the CVM, is part of the distribution. The CVM expects the following functionality from the underlying OS:

- Threads
- Synchronization (mutexes and condition variables)
- Dynamic linking
- malloc (POSIX memory allocation utility) or equivalent
- Input/output (I/O) functions
- Berkeley Standard Distribution (BSD) sockets
- File system support
- Function libraries must be thread-safe. A thread blocking in a library should not block any other VM threads.

The tools *JavaCodeCompact* and *JavaMemberDepend* are part of the distribution. *JavaMemberDepend* generates lists of dependencies at the class member level. The existence of *JavaCodeCompact* implies that preloading of classes is allowed in CDC.

Three profiles are defined for CDC:

Foundation Profile is a set of Java APIs that support resource-constrained devices without a standards-based GUI system. The basic class libraries from the Java standard edition (like java.io, java.lang and java.net) are supported and a connection framework (javax.microedition.io) is added.

Personal Basis Profile is a set of Java APIs that support resource-constrained devices with a standards-based GUI framework based on lightweight components. It adds some parts of the Abstract Window Toolkit (AWT) support (relative to JDK 1.1 AWT).

Personal Profile completes the AWT libraries and includes support for the applet interface.

Although a device can support multiple profiles additional libraries for RMI and ODBC are known as *optional packages*.

3.3 Additional Specifications

The following specifications do not fit into the layer scheme of J2ME. However, they are defined in the same way as the above subsets of the JVM and subsets/extensions of Java classes (API):

Java Card is a definition for the resource-constraint world of smart cards. The execution lifetime of the JVM is the lifetime of the card. The JVM is highly restricted (e.g. no threads, data type int is optional) and defines a different instructions set (i.e. new bytecodes to support smaller integer types).

Java Embedded Server is an API definition for services such as HTTP.

Personal Java was intended as a Java platform on Windows CE and is now marked as end of life.

Java TV is an extension to produce interactive television content and manage digital media. The description states that the JVM runs on top of a RTOS, but no real-time specific extensions are defined.

Other than Sun's, the only view specifications that exist for embedded Java are:

leJOS [7] is a JVM for Lego Mindstorm with stronger restrictions on the core classes than the CLDC.

RTDA [8] although named 'Real-Time Data Access' the definition consists of two parts:

- An I/O data access API specification applicable for real-time and non real-time applications.
- A minimal set of real-time extensions to enable the I/O data access also to cover hard real-time capable response handling.

3.4 Discussion

Many of the specifications (i.e. *configurations* and *profiles*) are developed using the Java Community Process (JCP). JCP is not an open standard nor is it part of the open source concept. Although the acronym J2ME implies Java version 2 (i.e. JDK 1.2 and later) almost all technologies under J2ME are still based on JDK 1.1.

Besides Java Card, CLDC is the 'smallest' definition from Sun. It assumes an operating system and is quite large (the JAR file for the classes is about 450 kB). There are no API definitions for low-level hardware access. CLDC is not suitable for small embedded devices. Java Card defines a different JVM instruction set and thus compromises basic ideas of Java.

A more restricted definition with following features is needed:

- JVM restrictions, such as in CLDC 1.0
- A package for low-level hardware access
- A minimum subset of core libraries
- Additional profiles for application domains

4 Real-Time Extensions

In 1999, a document defining the requirements for real-time Java was published by NIST [9]. Based on these requirements, two groups defined specifications for real-time Java. A comparison of these two specifications and as compared with Ada 95's Real-Time Annex can be found in [10]. The following section gives an overview of these specifications and additional defined restrictions of RTSJ.

4.1 Real-Time Core Extension

The Real-Time Core Extension [11] is a specification published under the J Consortium. It is still in a draft version.

Two execution environments are defined: the *Core* environment is the special real-time component. It can be combined with a traditional JVM, the *Baseline*. For communication between these two domains, every Core object has two APIs, one for the Core domain and one for the Baseline domain. Baseline components can synchronize with Core components via semaphores.

Two forms of source code are supported to annotate attributes: *stylized* code with calls of static methods of special classes and *syntactic* code with new keywords. Syntactic code has to be processed by a special compiler or preprocessor.

4.1.1 Memory

A new object hierarchy with CoreObject as root is introduced. To override final methods from Object the semantics of the class loader is changed. It replaces these methods with special named methods from CoreObject. A Core task is only allowed to allocate instances of CoreObject and its subclasses. These objects are allocated in a special allocation context or on the stack. The objects are not garbage collected. However, an allocation context can be explicit freed by the application.

4.1.2 Tasks and Asynchrony

Core tasks represent the analog of java.lang.Threads. All real-time tasks must extend CoreTask or one of its subclasses. No interface such as java.lang.Runnable is defined. Tasks are scheduled preemptive priority-based (128 levels) with FIFO within priorities. Time slicing can be supported, but is not required.

Although stop() is depreciated in Java 2 it is allowed in the CoreTask for the asynchronous transfer of control (besides a class ATCEvent). To prevent the problem of inconsistent objects after stopping a task an *atomic synchronized* region defers abortion.

A special task class is defined to implement interrupt service routines. The code for this handler is executed *atomically* and must be WCET analyzable. SporadicTask is used to implement responses to sporadic events, triggered by invoking the trigger() method of the task. No enforcement of minimum time between arrivals of events is available.

No special events or task types are defined for periodic work. The methods sleep() and sleepUntil() of CoreTask can be used to program periodic activity.

4.1.3 Exceptions

References from the java.lang.Throwable class hierarchy are silently replaced by the class loader with references to Core classes. A new scoped exception, which needs special support from the JVM, is defined.

4.1.4 Synchronization

snychronized is only allowed on this. To compensate for this restriction additional synchronization objects like semaphores and mutexes are defined. Queues on monitors, locks and semaphores are priority and FIFO ordered. Priority inversion is avoided by using the priority ceiling protocol. To allow locks to be implemented without waiting queues, a Core task is not allowed to execute a blocking operation while it holds a lock.

4.1.5 Helper Classes

The standard representation of time is a long (64-bit) integer with nanosecond resolution. A Time class with static methods is provided for conversions. A helper class

supports treating signed integers as unsigned values. Low-level hardware ports can be accessed via IOPort.

4.2 Discussion on the RT Core

A new introduced object hierarchy and new language keywords lead to changes in the class verifier and loader semantics. The behavior of the JVM has changed, so it would make sense to change the methods of Object to fit to the Core definition. This would result in a single object hierarchy. The restriction on synchronized disables the elegant style of expressing general synchronization problems in Java.

Although Nilsen lead the group, NewMonics PERC systems [12] support a different API.

4.3 RTSJ

The Real-Time Specification for Java (RTSJ) defines a new API with support from the JVM [3]. The following guiding principles led to the definition:

- No restriction of the Java runtime environment
- Backward compatibility for non-real-time Java programs
- No syntactic extension to the Java language or new keywords
- Predictable execution
- Current practice and allow future implementations to add advanced features

A Reference Implementation (RI) of RTSJ forms part of the specification. RTSJ is backward compatible with existing non-real-time Java programs, which implies that RTSJ is intended to run on top of J2SE (and not on J2ME). The following section presents an overview of the RTSJ.

4.3.1 Threads and Scheduling

A priority-based, preemptive scheduler with at least 28 real-time priorities is defined as base scheduler. Additional levels (ten) for the traditional Java threads need to be available. Threads with the same priority are queued in FITO order. Additional schedulers (e.g. EDF) can be dynamically loaded. The class Scheduler and associated classes provide optional support for feasibility analysis.

Any instances of classes that implement the interface Schedulable are scheduled. In RTSJ RealtimeThread, NoHeapRealtimeThread, and AsyncEventHandler are *schedulable objects*. NoHeapRealtimeThread has and AsyncEventHandler can have a priority higher than the garbage collector. As the available release-parameters indicate, threads are ether periodic or asynchronous events. Threads can be grouped together to bind the execution cost and deadline for a period.

4.3.2 Memory

As garbage collection is problematic in real-time applications, RTSJ defines new memory areas:

Scoped memory is a memory area with bound lifetime. When a scope is entered (with a new thread or through enter()), all new objects are allocated in this memory area. Scoped memory areas can be nested and shared among threads. On exit of the last thread from a scope, all finalizers of the allocated objects are invoked and the memory area is freed.

Physical memory is used to control allocation in memories with different access time.

Raw memory allows byte-level access to physical memory or memory-mapped I/O.

Immortal memory is a memory shared between all threads without a garbage collector. All objects created in this memory area have the same live time as the application (a new definition of *immortal*).

Heap memory is the traditional garbage collected memory.

Maximum memory usage and the maximum allocation rate per thread can be limited. Strict assignment rules between the different memory areas have to be checked by the implementation.

4.3.3 Synchronization

The implementation of synchronized has to include an algorithm to prevent priority inversion. Priority inheritance protocol is the default and priority ceiling can be used on request. Threads waiting to enter a synchronized block are priority and FIFO within priority ordered. Wait free queues are provided for communication between instances of java.lang.Thread and RealtimeThread.

4.3.4 Time and Timers

Classes to represent relative and absolute time with nanosecond accuracy are defined. All time parameters are split to a long for milliseconds and an int for nanoseconds within those milliseconds. Each time object has an associated Clock object. Multiple clocks can represent different sources of time and resolution. This allows for the reduction of queue management overheads for tasks with different tolerance for jitter. A new type, rationale time, can be used to describe periods with a requested resolution over a longer period (i.e. allowing release jitter between the points of the *outer* period). Timer classes can generate time-triggered events (one shot and periodic).

4.3.5 Asynchrony

Program logic representing external world events is scheduled and dispatched by the scheduler. An AsyncEvent object represents an external event (such as a POSIX signal or a hardware interrupt) or an internal

event (through call of fire()). Event handlers are associated to these events and can be bound to a regular real-time thread or represent something *similar* to a thread. The relationship between events and handlers can be many-to-many. Release of handlers can be restricted to a minimum interarrival time.

Java's exception handling is extended to represent asynchronous transfer of control (ATC). RealtimeThread overloads interrupt() to generate an AsynchronousInterruptedException (AIE). The AIE is deferred until the execution of a method that is willing to accept ATC. The method indicates this by including AIE in its throw clause. The semantics of catch is changed so that, even when it catches an AIE, the AIE is still propagated until the happened() method of the AIE is invoked. Timed, a subclass of AIE, simplifies the programming of timeouts.

4.3.6 Support for RTSJ

Implementations of RTSJ are still rare and under development:

RI is the freely available reference implementation for a Linux system [13].

jRate is an open-source implementation based on ahead-of-time compilation with GNU compiler for Java [14].

FLEX is a compiler infrastructure for embedded systems developed at MIT [15]. Real-Time Java is implemented with region-based memory management and a scheduler framework.

aJile supports RTSJ with CLDC 1.0 on top of the aJ-80 and aJ-100 chips.

4.4 Discussion of the RTSJ

RTSJ is a complex specification leading to a big memory footprint. Size of the RI on Linux:

- Classes in javax/realtime: 343 kB
- All classes in library foundation.jar: 2 MB
- Timesys JVM executable: 2.6 MB

RTSJ assumes a RTOS and the RI runs on a heavyweight RT-Linux system. RTSJ is too complex for low-end embedded systems. This complexity also hampers programming of high-integrity applications.

Runtime memory allocation of the RTSJ classes is not documented.

4.4.1 Threads and Scheduling

If a real-time thread is preempted by a higher priority thread, it is not defined if the preempted thread is placed in front or back of the waiting queue. It is not specified whether the default scheduler performs, or has to perform, time slicing between threads of equal priority.

4.4.2 Memory

It would be ideal if real-time systems were able to allocate all memory at the initialization phase and forbid dynamic memory in the mission phase. However, this restricts many of Java's library functions.

The solution to this problem in RTSJ is ScopedMemory, a memory space with limited lifetime. However, it can only be used as a parameter for thread creation or with enter(Runnable r). In a system without dynamic thread creation, using scoped memory at creation time of the thread leads to the same behavior as using immortal memory. The syntax with enter() leads to a cumbersome programming style: for each code part where limited lifetime memory is needed a new class has to be defined and a single instance of this class allocated at initialization time. Trying to solve this problem elegantly with anonymous classes, as in Figure 1 (example from [16] p. 623), leads to an error.

```
import javax.realtime.*;
public class ThreadCode implements Runnable
{
    private void computation()
    {
        final int min = 1*1024;
        final int max = 1*1024;
        final LTMemory myMem = new
                    LTMemeory(min, max);
        myMem.enter(new Runnable()
        {
            public void run()
            {
                // access to temporary memory
            {
        } );
    }

    public void run()
    {
        ...
        computation();
        ...
    }
}
```

Figure 1: Example of scoped memory usage

On every call of computation(), an object of the anonymous class (and a LTMemory object) is allocated in immortal memory, leading to a memory leak.

A simpler syntax is shown in Figure 2. The main drawback of this syntax is that the programmer is responsible for its correct usage.

```
LTMemory myMem;

// create memory once in constructor
MyThread() {
    myMem = new LTMemeory(min, max);
    ...
}

public void run() {

    ...
    myMem.enter();
    {   // new code block disables access
        // to new objects in outer scope.
        // access to temporary memory
        Abc a = new Abc();
        ...
    }
    myMem.exit();
    ...
}
```

Figure 2: Simpler syntax for scoped memory

New objects and arrays of objects have to be initialized to their default value after allocation [17]. This usually results in zeroing the memory at the JVM level and leads to variable (but linear) allocation time. This is the reason for the type LTMemoryArea in the RTSJ. As suggested in [14], this initialization could be lumped together with the creation time and exit time of the scoped memory. This results in constant time for allocation (and usually faster zeroing of the memory).

With RTSJ memory areas, it is difficult to move data from one area to another [18]. This results in a completely different programming model from that of standard Java. This can result in the programmer developing his/her own memory management.

4.4.3 Time and Timers

Why is the time split into milliseconds and nanoseconds? In the RI, it is converted to ns for add/subtract. After all mapping and converting (AbsoluteTime, HighResolutionTime, Clock and RealTimeClock) the System.currentTimeMillis() time is used.

Since time triggered release of tasks can be modeled with periodic threads, the additional concept of timers is superfluous.

4.4.4 Asynchrony

An unbound AsyncEventHandler is not allowed to enter() a scoped memory. However, it is not clear if scoped memory is allowed as a parameter in the construction of a handler. An unbound AsyncEventHandler leads to the implicit start of a thread on an event. This can (and, in the RI, does - see [14]) lead to substantial overheads. From the application perspective, bound and unbound event handlers behave in the same way. This is an implementation hint expressed through different classes. A consistent way to express the *importance* of events would be a scheduling parameter for the minimum allowed latency of the handler.

The syntax that is used in the throws clause of a method to state that ATC will be accepted is misleading. Exceptions in throws clauses are usually generated in a method and not *accepted*.

4.4.5 J2SE Library

It is not specified which classes are safe to be used in RealTimeThread and NoHeapRealTimeThread. Several operating system functions can cause unbound blocking and their usage should be avoided. The memory allocation in standard JDK methods is not documented and the use in immortal memory can lead to memory leaks.

4.4.6 Missing Features

There is no concept such as start mission. Changing scheduling parameters during runtime can lead to inconsistent scheduling behavior.

There is no provision for low-level blocking such as disabling interrupts. This is a common technique in device drivers where some hardware operations have to be atomic without affecting the priority level of the requesting thread (e.g. a low priority thread for a flash file system shall not get preempted during sector write as the chip internal write starts after a timeout).

4.4.7 On Small Systems

Many embedded systems are still built with 8 or 16-bit CPUs. 32-bit processors are seldom used. Java's default integer type is 32-bit, still large enough for almost all data types needed in embedded systems. Why are (often expensive) longs (64 bit integer) used in the RTSJ?

4.5 Subsets of RTSJ

RTSJ is complex to implement and applications developed with RTSJ are difficult to analyze (because of some of the sophisticated features of the RTSJ). Various profiles have been suggested for high-integrity real-time applications that result in restrictions of the RTSJ.

4.5.1 A Profile for High-Integrity Real-Time Java Programs

In [19], a subset of the RTSJ for high-integrity application domain with hard real-time constraints is proposed. It is inspired by the Ravenscar profile for Ada [20] and focuses is on exact temporal predictability.

Application structure: The application is divided in two different phases: *initialization* and *mission*. All non time-critical initialization, global object allocations, thread creation and startup are performed in the initialization phase. All classes need to be loaded and initialized in this phase. The mission phase starts after returning from main(), which is assumed to execute with maximum priority. The number of threads is fixed and the assigned priorities remain unchanged.

Threads: Two types of tasks are defined: *Periodic time-triggered activities* execute an infinite loop with at least one call of waitForNextPeriod(). *Sporadic activities* are modeled with a new class SporadicEvent. A SporadicEvent is bound to a thread and an external event on creation. Unbound event handlers are not allowed. It is not clear if programmatic trigger of the event is allowed (invocation of fire()). A restriction for a minimum interarrival time of events is not defined. Timers are not supported as time-triggered activities are well supported by periodic threads. Asynchronous transfers of control, overrun and miss handles and calls to sleep() are not allowed.

Concurrency: Synchronized methods with priority ceiling protocol provide mutual exclusion to shared resources. Threads are dispatched in FIFO order within each priority level. Sporadic events are used instead of wait, notify and notifyAll for signaling.

Memory: Since garbage collection is still not time predictable, it is not supported. This implicit converts the traditional heap to immortal memory. Scoped memory (LTMemory) is provided for object allocation during the mission phase. LTMemory has to be created during the initialization phase with initial size equal maximum size.

Implementation: For each thread and for the operations of the JVM WCET must be computable. Code is restricted to bound loops and bound recursions. Annotations for WCET analysis are suggested. The JVM needs to check the timing of events and thread execution. It is not stated how the JVM should react to a timing error.

4.5.2 Ravenscar-Java

The Ravenscar-Java profile [21] is a restricted subset of RTSJ and is based on work mentioned above (4.5.1). As the name implies it resembles Ravenscar Ada [20] concepts in Java.

To simplify the initialization phase RJ defines Initializer, a class that has to be extended by the application class which contains main(). The use of time scoped memory is further restricted. LTMemory areas are not allowed to be nested nor shared between threads. Traditional Java threads are disallowed by changing the class java.lang.Thread. The same is true for all schedulable objects from the RTSJ. Two new classes are defined:

- PeriodicThread where run() gets called periodically, removing the loop construct with waitForNextPeriod().
- SporadicEventHandler binds a single thread with a single event. The event can be an interrupt or a software event.

4.5.3 Criticisms of Subsets of the RTSJ

If a subset of RTSJ is implemented and allowed it is harder for programmers to find out what is available and what not. This form of *compatibility* causes confusion. The use of different classes for a different specification is clearer and less error prone.

Ravenscar-Java, as a subset of the RTSJ, claims to be compatible with the RTSJ, in the sense that programs written according to the profile are valid RTSJ programs. However, mandatory usages of new classes such as PeriodicThread need an emulation layer to run on a RTSJ system. In this case, it is better to define complete new classes for a subset and provide the mapping to RTSJ. This allows a clearer distinction to be made between the two definitions.

It is not necessary to distinguish between heap and immortal memory. Without a garbage collector, heap implicitly equates to immortal memory.

Objects are allocated in immortal memory in the initialization phase. In the mission phase, no objects should be allocated in immortal memory. Scoped memory can be entered and subsequent new objects are allocated in the scoped memory area. Since there are no circumstances in which allocation in these two memory areas are mixed, no newInstance() such as those in the RTSJ or Ravenscar-Java are necessary.

4.6 Extensions to RTSJ

The Distributed Real-Time Specification for Java [22] extends RMI within the RTSJ. In 2000, it was accepted in the Sun Community Process as JSR-50. This specification is still under development. According to [23], three levels of integration between RTSJ and RMI are defined:

Level 0: No changes in RMI and RTSJ are necessary. The proxy thread on the server acts as an ordinary Java thread. Real-time threads cannot assume timely delivery of the RMI request.

Level 1: RMI is extended to Real-Time RMI. The server thread is a real-time thread that inherits scheduling parameters from the calling client.

Level 2: RMI and RTSJ are extended to form the concept of *distributed real-time threads*. These threads have a unique system-wide identifier and can move freely in the distributed system.

5 Proposed Definitions for Small Embedded Systems

Restrictions on Java, such as the available configurations and profiles of J2ME and the RTSJ, are still too large and complex for small embedded real-time systems. Simpler definitions are therefore proposed: a *configuration* restricts the function of the JVM and the necessary library. Class definitions for low-level I/O are added. On top of this configuration, a *profile* for high-integrity real-time applications is defined.

5.1 Small Embedded Devices Configuration (SEDC)

SEDC is a configuration that fits into Sun's J2ME layer model. It is intended for small embedded devices with a 16-bit (or even 8-bit) microprocessor and a low memory budget (below 128 kB). The JVM restrictions are similar to CLDC 1.0. The subset of the Java classes is smaller.

5.1.1 Restrictions of the JVM

The Following features have been removed from the JVM:

- Floating point support
- Data type long is optional
- Java Native Interface (JNI)
- Reflection
- Finalization
- Class loading is optional
- Threads are optional
- Thread groups and daemon threads
- Asynchronous exceptions

To simplify the JVM the application is preverified and preloaded (and sometimes also linked with the JVM).

Since small embedded devices have no or only a very simple user interface (like switches and LEDs), no stream input/output facilities are usually necessary. As memory is very limited, the Java class library is reduced to the absolute minimum. Only the following classes need to be available:

```
java.lang.Object
java.lang.String
```

Should threads be part of SEDC? There are no threads in, for example, Java Card and we in any case define new thread behavior (with real-time semantics) in an extra profile.

An additional library is provided for low-level I/O-access:

5.1.2 Class IOPort

Static methods of this class allow the application low-level access to I/O ports.

`private IOPort()`
No instantiation of this class is allowed.

`public static int read(int address)`
Read from I/O port.

`public static void write(int value, int address)`
Write to I/O port.

5.1.3 Class Clock

Almost all microcontrollers have some kind of timer or counter. The class Clock provides a standard way of querying counter values.

`public static int count()`
Returns the current time in clock ticks. Tick frequency is device dependent and can be queried.

`public static int ticksPerSecond()`
Returns the resolution of the clock.

`public static int ticksPerMs()`
Returns the resolution of the clock per millisecond. This method is necessary if the clock frequency leads to an overflow of int, i.e. it is higher than 2.15 GHz.

`public static int resolutionInBits()`
Returns the length of the internal clock in bits.

5.1.4 Class Interrupt

The interrupt handler must extend the class Interrupt and register itself on an interrupt. For simple blocking, all interrupts can be disabled.

`public static void enable()`
This method general enables interrupts. The individual masking of interrupts is device specific and is handled via IOPort.

`public static void disable()`
This method disables all interrupts (like cli on x86).

`public static final Interrupt register(int number)`
An object registers itself using this method. The interrupt number is device dependent. A previous installed handler is returned (or null).

`public abstract void handle()`
An interrupt causes the object's handle method to be called.

5.2 A Real-Time Profile for Embedded Java

A simple real-time profile is defined in the concept of the ADA Ravenscar profile [20]. It resembles the concepts from [19] and [21] but is not compatible with RTSJ. Since the application domain for RTSJ is different from high-integrity systems, it makes sense for it *not* to be compatible with RTSJ. Restrictions can be enforced defining new classes (e.g. setting thread priority in the constructor of a real-time thread alone, enforcing minimum interarrival times for sporadic events).

This profile addresses the same devices as SEDC (it is not, of course, restricted to small devices). With an emulation layer, it should be possible to run programs written for this specification on top of RTSJ.

The guidelines:

- High-integrity profile.
- Easy syntax, simplicity.
- Easy to implement.
- Low runtime overhead.
- No syntactic extension of Java.
- Minimum change of Java semantics.
- Support for time measurement if WCET analysis tools are not available.
- Known overhead (documentation of runtime behavior and memory requirements of every JVM operation and all methods have to be provided).

- Implementation possible on top of SEDC.

5.2.1 Application Structure

The following restrictions apply to the application:

- Initialization and mission phase.
- Fixed number of threads.
- Threads are created at initialization phase.
- All shared objects are allocated at initialization.

5.2.2 Threads

Three schedulable objects are defined:

RtThread represents a periodic task. As usual task work is coded in run() which gets called on missionStart(). A scoped memory object can be attached to a RtThread at creation.

HwEvent represents an interrupt with a minimum interarrival time. If the hardware generates more interrupts, they get lost.

SwEvent represents a software-generated event. It is triggered by fire() and needs to override handle().

Figure 3 shows the definition of the basic classes.

```
public class RtThread {

    public RtThread(int priority, int period)
    public RtThread(int priority, int period,
                    int offset)
    public RtThread(int priority, int period,
                    Memory mem)
    public RtThread(int priority, int period,
                    int offset, Memory mem)

    public void enterMemory()
    public void exitMemory()

    public void run()
    public boolean waitForNextPeriod()

    public static void startMission()
}

public class HwEvent extends RtThread {

    public HwEvent(int priority, int minTime,
                   int number)
    public HwEvent(int priority, int minTime,
                   Memory mem, int number)

    public void handle()
}

public class SwEvent extends RtThread {

    public SwEvent(int priority, int minTime)
    public SwEvent(int priority, int minTime,
                   Memory mem)

    public final void fire()
    public void handle()
}
```

Figure 3: Schedulable objects

5.2.3 Scheduling

Threads and events are scheduled with fixed priority. No real-time threads or events are scheduled during the initialization phase. Most real-time systems use a fixed-priority preemptive scheduler. Tasks with the same priority are usually scheduled in a FIFO order. Two common ways to assign priorities are rate monotonic or, in a more general form, deadline monotonic assignment. When two tasks are given the same priority, we can choose one of them and assign a higher priority to that task and the task set will still be schedulable. This results in a strictly monotonic priority order and we do not need to deal with FIFO order. This eliminates queues for each priority level and results in a single, priority ordered task list with unlimited priority levels.

Synchronized blocks are executed with priority ceiling protocol. With objects for which the priority is not set, a top priority is assumed. This avoids priority inversions on objects that are not accessible from the application (e.g. objects inside a library).

In addition, the scheduler contains methods for worst-case time measurement for both the periodic work and handler methods. These measured execution times can be used during development when no WCET analysis tool is available.

5.2.4 Memory

The profile does not support a garbage collector. All memory allocation should be done at initialization phase. For new objects during the mission phase, a scoped memory is provided. Every scoped memory area can be assigned to one RtThread. A scoped memory cannot be shared between threads. No references from the heap to scoped memory are allowed. Scoped memory is explicitly entered and left with calls from the application logic. Memory areas are cleared on creation and when leaving the scope (call of exitMemory()) leading to a memory area with constant allocation time.

5.2.5 Restriction of Java

A list of some of the language features that should be avoided for WCET analyzable real-time threads and bound memory usage:

WCET: Only analyzable language constructs are allowed.

Static class initialization: Since the definition when to call the static class initializer is problematic, they are disallowed. Move this code to a static method (e.g. init()) and call it explicit in the initialization phase.

Inheritance: Reduce usage of interfaces and overridden methods.

String concatenation: In immortal memory scope only string concatenation with string literals is allowed.

Finalization: finalize() has a weak definition in Java. Because real-time systems run *forever*, objects in the heap will never be finalized. Objects in scoped memory are released on exitMemory(). However, finalizations on these objects complicate WCET analysis of exitMemory().

Dynamic Class Loading: Due to the implementation and WCET analysis complexity dynamic class loading is avoided.

A program analysis tool can greatly help in enforcing these restrictions.

5.2.6 An Example

Figure 4 shows the principle coding of a worker thread. An example for creation of two real-time threads and an event handler can be seen in Figure 5.

```
public class Worker extends RtThread {

    private SwEvent event;

    public Worker(int p, int t,
                  SwEvent ev) {

        super(p, t,
            // create a scoped memory area
            new Memory(10000)
        );
        event = ev;
        init();
    }

    private void init() {
        // all initialzation stuff
        // has to be placed here
    }

    public void run() {

        for (;;) {
            work();       // do some work
            event.fire(); // and fire an event

            // some work in scoped memory
            enterMemory();
            workWithMem();
            exitMemory();

            // wait for next period
            if (!waitForNextPeriod()) {
                missedDeadline();
            }
        }
        // should never reach this point
    }
}
```

Figure 4: A periodic real-time thread

```
// create an Event
Handler h = new Handler(3, 1000);

// create two worker threads with
// priorities according to their periods
FastWorker fw = new FastWorker(2, 2000);
Worker w = new Worker(1, 10000, h);

// change to mission phase for all
// periodic threads and event handler
RtThread.startMission();

// do some non real-time work
// and call sleep() or yield()
for (;;) {
    watchdogBlink();
    Thread.sleep(500);
}
```

Figure 5: Start of the application

6 Implementation Results

The proposed profile is implemented on JOP. In this section, the implementation of the simple real-time profile is compared with the RI (Reference Implementation) of RTSJ on top of Linux. JOP is implemented in a low cost FPGA (Cyclone EP1C6) from Altera clocked with 20 MHz. The clock frequency is defined by the board used and not by the design. According to the place & route tool, the design can be clocked up to 100 MHz in this FPGA. The test results for the RI were obtained on an Intel Pentium MMX 266 MHz, running Linux SuSE 8.2 with a generic kernel version 2.4.20 and TimeSys GPL Linux 3.1 [24] as recommended by the RI. For each test 500 measurements were made.

6.1 Periodic Threads

Many activities in real-time systems must be performed periodically. Low release jitter is of major importance for tasks such as control loops. The test setting is similar to the periodic thread test in [25]. A single real-time thread only calls waitForNextPeriod() in a loop and records the time between calls that follow. Table 1 shows the average, standard deviation and extreme values for different period times on JOP. In Table 2 the same values are shown for the RI.

	Avg.	*Std. Dev.*	*Min.*	*Max.*
T=300 us	300 us	91 us	256 us	499 us
T=500 us	500 us	0 us	500 us	500 us
T=1 ms	1 ms	0 ms	1 ms	1 ms
T=5 ms	5 ms	0 ms	5 ms	5 ms
T=10 ms	10 ms	0 ms	10 ms	10 ms
T=30 ms	30 ms	0 ms	30 ms	30 ms
T=50 ms	50 ms	0 ms	50 ms	50 ms

Table 1: Jitter of Periodic Threads with JOP.

	Avg.	*Std. Dev.*	*Min.*	*Max.*
T=500 us	315 us	93 us	18 us	569 us
T=1 ms	1.00 ms	0.01 ms	0.946 ms	1.055 ms
T=5 ms	4.00 ms	7.92 ms	0.017 ms	19.90 ms
T=10 ms	6.64 ms	9.34 ms	0.019 ms	19.94 ms
T=20 ms	20.0 ms	0.015 ms	19.87 ms	20.14 ms
T=30 ms	30.0 ms	0.031 ms	29.69 ms	30.31 ms
T=35 ms	35.0 ms	5.001 ms	29.75 ms	40.25 ms
T=50 ms	50.0 ms	0.018 ms	49.95 ms	50.06 ms

Table 2: Jitter of Periodic Threads with RI/RTSJ.

Using of a microsecond accurate timer interrupt, programmed by the scheduler, results in excellent performance of periodic threads in JOP. No jitter can be observed with a single thread at periods above 300 us.

The measurements for the RI do not include the first values. The first values are slightly misleading, as the RI behaves unpredictable at *startup*. The RI performs inaccurately at periods below 20 ms. This effect has also been observed in [25]. Larger periods that are multiples of 10 ms have very low jitter. However, using a period such as 35 ms shows a standard deviation of five ms. A detailed look at the collected samples shows only values of 30 and 40 ms. This implies a timer tick of 10 ms in the underlying operating system. No real difference can be seen when running this test on the generic Linux kernel and the TimeSys kernel. Table 2 gives the measurements on the generic kernel.

6.2 Context Switch

The test setting consists of two threads. A low priority thread continuously stores the current time in a shared variable. A high priority periodic thread measures the time difference between this value and the time immediately after waitForNextPeriod(). Table 3 gives the time for the context switch in processor clock cycles.

	Avg.	*Std. Dev.*	*Min.*	*Max.*
JOP	4088	10.29	4083	4116
RI Linux	4253	1239	3232	19628
RI TS Linux	12923	1145	11529	21090

Table 3: Time for a Thread Switch in Clock Cycles.

This test did not produce the expected behavior from the RI on the generic Linux kernel. When the low priority thread runs in this tight loop, the high priority thread was not scheduled. After inserting a Thread.yield() and an operating system call, such as System.out.print(), in this loop, the test performed as expected. This indicates a major problem in the RI or the operating system scheduler. This problem did not occur when the RI was run on the TimeSys Linux kernel. However, the context switch time was three times longer than on the standard kernel.

6.3 Asynchronous Event Handler

In this test setting, a high priority event handler is triggered by a low priority periodic thread. As AsynchEventHandler performs poorly [14], a BoundAsynchEventHandler is used for the RI test program. The time elapsed between the invocation of fire() and the first statement of the event handler is measured. Table 4 shows the elapsed time in clock cycles for JOP and the RTSJ RI.

	Avg.	*Std. Dev.*	*Min.*	*Max.*
JOP	4283	3.0	4283	4350
RI Linux	53685	7014	47400	87196
RI TS Linux	69273	7832	63060	101292

Table 4: Dispatch Latency of Event Handlers.

The time taken to dispatch an asynchronous event is similar to the context switch time in JOP. This is to be expected as events are scheduled and dispatched as threads. The maximum value occurred only for the first event, all following events being dispatched in the minimum time.

In the RI, the dispatch time is about 10 times larger than a context switch with a significant variation in time. This indicates that the implementation of fire() and the communication of the event to the underlying operating system is not optimal. The time factor between context switch and event handling on the TimeSys kernel is lower than on the standard kernel, but is still significant.

6.4 Ravenscar Java

To verify that the profile is expressive enough for high-integrity applications, Ravenscar Java was implemented on top of it. The only restriction observed is the absence of support for long in the JVM. This type is man-

mandatory in the RTSJ for absolute and relative time definitions.

6.5 Resource Usage

JOP is implemented on a low cost FPGA (EP1C6) consuming about 30% of the chip area. The unused area can be used for custom peripherals or a multi processor solution on a single chip. This resource usage is similar to Alteras NIOS, a 32-bit load/store RISC processor.

The basic JVM with the proposed real-time extension consumes 9 KB of memory, leaving enough room for the application code and data even in very small devices. The real-time scheduler needs 0.5 KB per thread at runtime.

7 Conclusion

Some definitions of Java for embedded and real-time systems do exist. CLDC, as a restriction for embedded systems, suits small systems best, but is still large. The most common specification for real-time Java is RTSJ. However, this specification is complex and large, making it not the primary choice for small embedded systems or high-integrity systems. Restrictions on the RTSJ can transform the definition into a high-integrity profile, but they inherit the complex API. A very small *configuration* for embedded systems and a high-integrity real-time *profile* that fits this configuration are proposed. The configuration and the profile are implemented on top of JOP and used in a number of real-world applications. Future work will explore additional hardware support for real-time systems in JOP. The implementation of JOP and the real-time profile is available at: http://www.jopdesign.com.

References

[1] E. Quinn and C. Christiansen. Java Technology Pays Positively. IDC Bulletin #W16212, May 1998.

[2] K. Nilsen. Issues in the Design and Implementation of Real-Time Java, July 1996. Published June 1996 in *Java Developers Journal*, republished in Q1 1998 *Real-Time Magazine*

[3] Bollela, Gosling, Brosgol, Dibble, Furr, Hardin and Trunbull. *The Real-Time Specification for Java*, Addison Wesley, 1st edition, 2000.

[4] M. Schoeberl. JOP: a Java Optimized Processor. In *Workshop on Java Technologies for Realtime and Embedded Systems (JTRES 2003)*, Catania, Sicily, Italy, November 2003.

[5] J. Gosling, B. Joy, G. Steele and G. Bracha. *The Java Language Specification*. Addison Wesley, 2nd edition, 1997.

[6] Sun Microsystems. *Java 2 Platform, Micro Edition (J2ME)*, available at: http://java.sun.com/j2me/docs/

[7] leJOS, Java for the RCX, available at: http://lejos.sourceforge.net/

[8] International J Consortium Specification. Real-Time Data Access, Release 1.0, November 2001. Available at http://www.j-consortium.org/

[9] K. Nilsen, L. Carnahan and M. Ruark , editors. Requirements for Real-Time Extensions for the Java Platform. Published by *National Institute of Standards and Technology*. September 1999. Available at http://www.nist.gov/rt-java.

[10] B. Brosgol and B. Dobbing. Real-time convergence of Ada and Java. In *Proc. of the 2001 annual ACM SIGAda international conference on Ada*, pp.11-26, Bloomington, MN, 2001

[11] International J Consortium Specification. Real-Time Core Extensions, Draft 1.0.14, September 2nd 2000. Available at http://www.j-consortium.org/

[12] K. Nilsen and S. Lee. *PERC Real-Time API (Draft 1.3)*. NewMonics, July 1998

[13] TimeSys. Real-Time Specification for Java Reference Implementation. http://www.timesys.com/

[14] A. Corsaro, D. Schmidt. The Design and Performance of the jRate Real-Time Java Implementation. Appeared at *the 4th International Symposium on Distributed Objects and Applications*, 2002

[15] FLEX, a compiler infrastructure written in Java for Java. Available at http://www.flex-compiler.csail.mit.edu/

[16] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages*, Addison Wesley, 3rd edition, 2001.

[17] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 2nd edition, 2000.

[18] A. Niessner, E. Benowitz. RTSJ Memory Areas and Their Affects on the Performance of Flight-Like Attitude Control. In *Workshop on Java Technologies for Realtime and Embedded Systems (JTRES 2003)*, Catania, Sicily, Italy, November 2003.

[19] P. Puschner and A. J. Wellings. A Profile for High Integrity Real-Time Java Programs. In *Proceedings of the 4th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC)*, 2001

[20] A. Burns and B. Dobbing. The Ravenscar Tasking Profile for High Integrity Real-Time Programs. In *Proc. of the 1998 annual ACM SIGAda international conference on Ada*, pp. 1-6, Washington, D.C., United States, 1998

[21] J. Kwon, A. Wellings and S. King. Ravenscar-Java: a high integrity profile for real-time Java, In *Proc. of the 2002 joint ACM-ISCOPE conference on Java Grande*, pp. 131-140, Seattle, Washington, USA, 2002

[22] E.D. Jensen. A proposed initial approach to distributed real-time Java. In *Proceedings Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2000)*, pp. 2-6, March 2000.

[23] A. Wellings, R. Clark, D. Jensen and D. Wells. A framework for integrating the real-time specification for Java and Java's remote method invocation. In *Proceedings Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2002)*, pp. 13-22, April 2002.

[24] TimeSys. Linux RTOS Standard Edition available at: http://www.timesys.com/

[25] A. Corsaro, D. Schmidt. Evaluating Real-Time Java Features and Performance for Real-Time Embedded Systems. Appeared at *The 8th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2002.