# KESO

## An Open-Source Multi-JVM for Deeply Embedded Systems

Isabella Thomm Michael Stilkerich Christian Wawersich Wolfgang Schröder-Preikschat
{ithomm,stilkerich,wawi,wosch}@cs.fau.de
Friedrich-Alexander University Erlangen-Nuremberg, Germany

## ABSTRACT

Java still is a rather exotic language in the field of real-time and particularly embedded systems, though it could provide productivity and especially safety and dependability benefits over the dominating language C. The reasons for the lack of acceptance of Java in the embedded world are the high resource consumption caused by the Java runtime environment and lacking language features for low-level programming.

KESO is a JVM under LGPL license that was specifically designed for the domain of statically-configured deeply embedded systems. KESO provides a sensible selection of Java features useful to the majority of embedded applications and safe and convenient constructs for low-level programming in Java. A key feature of KESO is its Multi-JVM architecture, which allows the isolated cohabitation of different applications on one hardware platform.

The resource consumption of applications developed on the base of KESO is comparable to C applications, and its mechanisms for communicating among isolated components are efficient and encourage the actual utilization of spatial isolation.

## Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Compilers*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Classes and objects*; D.4.7 [**Operating Systems**]: Organization and Design—*Real-time systems and embedded systems*; D.4.5 [**Operating Systems**]: Reliability—*memory protection*

## General Terms

Reliability, Design, Languages

## Keywords

KESO, Java, memory protection, embedded systems, spatial isolation, OSEK/VDX, AUTOSAR


*JTRES'10* August 19-21, 2010 Prague, Czech Republic


## 1. INTRODUCTION

Embedded software is most commonly developed in low-level languages such as C or even assembler. One reason for this is that in many application domains the embedded system is part of a mass product (e.g., the automotive sector), which results in an immense cost pressure. C comes with a slim runtime system and is thus very resource efficient. In addition, C provides programming constructs that allow to conveniently perform low-level tasks such as programming hardware registers. Unfortunately, the powerful mechanisms that make C attractive to embedded systems programmers are also prone to programming errors. Modern languages such as Java avoid many of these issues and were found to improve the productivity, the development process and the dependability and maintainability of the resulting software product [15, 17]. However, among embedded developers, Java has the reputation of being expensive and unsuited for low-level programming.

Additionally, many embedded systems perform safety critical tasks. Software components are mostly not isolated from each other and possess full access to the memory, wherefore a bug in one software component can easily spread to other components. Memory protection hardware is not available on most low-cost microcontroller units. Additionally, many embedded real-time operating systems do not support hardware-based memory protection and embedded programmers are overstrained with managing their memory in a way that allows region-based hardware protection to be applied.

In this paper, we present an open-source solution to these issues. KESO [24] is a Java Virtual Machine (JVM) that has been designed for the domain of deeply embedded systems with real-time constraints in mind. The focus of KESO is on providing memory protection and in particular spatial isolation of software components or different applications that are cohabited on a microcontroller platform. Besides this focus, KESO provides most of the high-level features that make Java a productive and maintainable language. We investigated existing JVMs for the embedded domain and performed a careful Java feature selection to allow KESO to run on even the smallest low-cost 8-bit microcontrollers. KESO provides safe and convenient abstractions for performing low-level tasks such as programming device drivers, and efficient mechanisms for communicating between spatially isolated software components.

The remainder of this paper is structured as follows: In Section 2 we give an overview on KESO's architecture and discuss the key design decisions. Section 3 presents the native interface that allows performing potentially unsafe operations

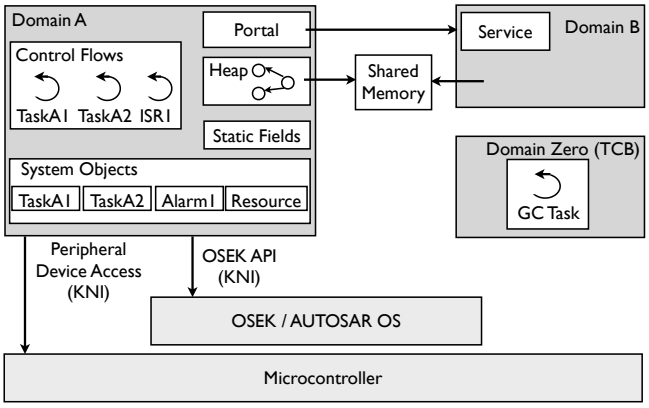



**Figure 1: The KESO architecture**

and safe abstractions provided with KESO that are built on top of this interface. Section 4 presents the available inter-domain communication mechanisms. Section 5 provides an evaluation that is meant to show that the overhead inferred by the use of KESO is a reasonable price for the benefits gained. Appendix A contains instructions on obtaining KESO and running a demo program.

# 2. ARCHITECTURE

KESO allows several Java applications to safely coexist on a microcontroller by providing a Java Virtual Machine (JVM) instance for each application. The KESO architecture is depicted in Figure 1. In this section, we will give an overview on the different architectural components that a KESO system is composed of and present and discuss the key design decisions that make KESO suitable for its target domain.

## 2.1 Overview

The fundamental structural component in a KESO system is the protection *domain*, which defines a realm of protection and enables different applications to peacefully coexist on a microcontroller with communication limited to a set of well-defined and safe communication channels. From the perspective of the application, each domain appears to be a JVM of its own, which is why this architecture is also referred to as a Multi-JVM. Domains cannot be nested.

Domains are containers of control flows (i.e., tasks/threads and interrupt service routines (ISR)) and system objects (i.e., instances of operating system abstractions such as resources/locks or timers/alarms). Actions on these system objects are also limited to control flows within the same domain, except for system objects that are explicitly made available to other domains. We elaborate on KESO's implementation of service protection in more detail in Section 3.2. The special domain Zero is part of the trusted computing base (TCB) and contains privileged control flows of the runtime environment such as the garbage collector.

KESO provides several mechanisms that enable domains to interact with their environment. The KESO Native Interface (KNI) enables unsafe operations such as configuring peripheral hardware or interfacing with native libraries and the operating system API. To communicate with other protection domains, KESO provides an RPC-like, control-flow-oriented mechanism called *portals* and a safe shared memory abstraction for the purpose of working on larger amounts of shared data.

## 2.2 Establishment of Spatial Isolation

Spatial isolation ensures that control flows are only able to access memory of data regions belonging to the domain in the context of which the control flow is being executed. Therefore, each piece of data can be logically assigned to exactly one domain. In Java, type safety ensures that programs can only access memory regions to which they were given an explicit reference; the type of the reference also determines, in which way a program can access the memory region pointed to by the reference. To achieve spatial isolation, KESO ensures that a reference value is never present in more than a single domain. Besides objects on the heap, there are static class fields, which form the second part of global data available in Java. KESO maintains an own instance of these fields within every domain. This constructively ensures that memory accesses are limited to the current protection context, even though there is no hardware protection mechanism that enforces these constraints. To maintain this isolation, all inter-domain communication mechanisms must ensure that no reference values can be propagated to another domain.

## 2.3 Target Domain

KESO targets statically configured applications to be run on even deeply embedded systems, possibly under real-time constraints. On the low end, an example for such a platform is the Atmel 8-bit AVR architecture, a line of microprocessors whose derivates scale very fine-grained in the cost/resources tradeoff. The smallest derivate that we have so far run KESO on is the ATmega8535 device that is equipped with 8 KiB of Flash ROM and 544 bytes internal SRAM. Resource-awareness was therefore a crucial factor in many of the design decisions that we took. For more demanding applications, KESO can also be used on more powerful 32-bit platforms such as Infineon's Tricore architecture.

KESO does not implement thread scheduling and thread synchronization but uses an existing real-time operating system (RTOS) to do so. KESO itself makes few assumptions on the underlying scheduling model. For the garbage collector to work as assumed, it requires priority-based scheduling, which should be fulfilled by any RTOS. The currently available backend for KESO requires an OSEK/VDX [14]-compatible operating system (or its successor AUTOSAR OS [1]) as the underlying base system. OSEK/VDX systems originate from the automotive sector and are the predominant system software in current cars, however, the scheduling model of OSEK/VDX is suitable for other domains of embedded software as well. OSEK/VDX-compatible systems, including emulations for commodity operating systems, are available in both commercial and free variants [2, 20, 11] for a wide variety of architectures.

An OSEK/VDX operating system is little more than a scheduler based on static priorities. The immediate ceiling priority protocol (referred to as priority ceiling emulation by the Real-Time Specification for Java [3]) is used for synchronization; the locks are referred to as *resources* in OSEK/VDX-terminology. A wait-notify mechanism that allows control flows to block during execution called *events* and a time-based notification mechanism called *alarms* complete the functionality of the OSEK/VDX system. A key element of OSEK/VDX systems is their completely static nature: All

system entities (i.e., the control flows (*tasks*), synchronization resources, alarms, etc.) and their relationships (e.g., which task locks which resources at runtime) are specified in an OIL [13] configuration file and are known at compile time. The static nature supports determinism and analyzability and is additionally used by the code generators that are usually shipped with an OSEK/VDX operating system to generate an operating system variant that is tailored towards the application at hand.

## 2.4 Feature Selection

Like other Java platforms for embedded systems, KESO does not support the full spectrum of features provided by the Java 2 Standard platform (J2SE). We decided for the following restrictions either because the concept imposes an unreasonable overhead, or because of the limited use in our static target domain. Looking at C++ embedded software we observed that comparable concepts in C++ (e.g., exceptions, runtime type information) are mostly not used either because of the imposed overhead or impaired predictability.

### 2.4.1 Static Applications

Like OSEK/VDX, KESO requires static applications, which is a common setting in the embedded and particularly the real-time domain. Consequently, KESO does not support dynamic class loading and requires all code to be known at compile time. KESO also does not support the Java reflection mechanism, on the one hand for the overhead imposed by the extended runtime type information that this would require, but on the other hand also because the use of Java reflection severely affects the analyzability of the program.

As a result of these restrictions, KESO is able to perform various static analyses when compiling the program such as dead code elimination or devirtualization of virtual method calls, which enables KESO to generate code small enough to be executed on tiny microcontrollers.

### 2.4.2 Error Handling

KESO does not support the fully featured exception handling mechanism. Instead, we consider the rise of an exception as a fatal error for whose handling a global error handling routine is called. This is an adoption of the *ErrorHook* concept specified in the OSEK/VDX standard. This includes exceptions raised by the runtime environment such as failed `null` reference or array bounds checks. Full exception handling could be provided as an optional feature without affecting the runtime cost of applications that do not need this feature, but this is currently not implemented.

### 2.4.3 Class Library

The class library provided with KESO currently only supports a small subset of the Java standard class library; many features provided by the standard library are of limited use in embedded applications and costly in their implementation. As an example, KESO comes with a simplified `StringBuffer` class—even the code size of this simplified version often accounts for 50% of the total size of the actual application. The current functionality is mostly derived by implementation of standard Java classes as required by ongoing application projects and not proposed as an alternative standard API. Common Java APIs for embedded systems are not fully suited for KESO and could thus only partially be supported. To give two examples, the API of CLDC [10] supports the dynamic creation of new threads, which is by design not possible in KESO, while the Java Card [21] API does not support multiple threads at all and is designed for smart card applications. To ease portability we aim at supporting the CLDC API to the possible extent in the future.

## 2.5 Ahead-of-Time Compilation

The user applications are developed in Java and available as Java bytecode after having been processed by a Java compiler. Normally, this bytecode is interpreted or just-in-time compiled by the JVM at runtime. The interpreter or JIT compiler significantly adds to the complexity and size of the required runtime environment and introduces temporal overhead compared to the execution of native code. There are some embedded JVMs that took this dynamic approach; a prominent representative is SquawkVM [19], whose primary target platform is the Sun SPOT sensor node. Looking at the specifications of this sensor node (180 MHz 32-bit ARM9, 4 MiB Flash ROM, 512 KiB RAM) it becomes clear, that this sensor node is much more powerful than low-cost sensor nodes such as the Crossbow MICA2 (7.37 MHz 8-bit AVR, 128 KiB Flash ROM, 4 KiB RAM). For 8-bit AVR processors, there is also a bytecode interpreting JVM, NanoVM [6]; due to the size of the runtime, NanoVM is designed to take up the entire Flash ROM of its target processor (ATmega8) and loads the actual application bytecode from the EEPROM of the device (which comprises 512 bytes on the ATmega8).

For these reasons, we opted for generating native code ahead of time, which enables us to achieve a very slim runtime environment and performance comparable to that of applications written in languages such as C. Instead of directly compiling the Java bytecode to native code, our compiler *jino* emits ISO-C90 code, which has some advantages over directly generating native code:

- No need for a jino backend for each supported target platform. A standard C compiler is available for almost any of the target platforms.
- The available C compilers allow to create highly optimized code at the function level. We can therefore concentrate on high-level optimizations in jino and leave the low-level optimizations to the C compiler.
- C source code is easier to read than native code, which facilitates debugging.
- A separate compiler backend for each target platform increases jino's complexity and increases the probability of software bugs.
- Existing tools (e.g., WCET analysis) operating on C code remain applicable.

The generated C code does not only contain the compiled class files, but also the KESO runtime data structures. Moreover, additional code is inserted to retain the properties of a JVM, such as `null` reference checks and array bounds checks, and the code of other services of the KESO runtime environment, such as garbage collection and portal services. The generated KESO runtime is tailored towards the application's requirements, so the infrastructure code and data required for features such as floating point arithmetic or the support for multiple domains will only be added to the runtime if used by the application.

There are also other JVMs such as FijiVM [16] that do ahead-of-time compilation instead of the traditional bytecode interpretation at runtime.

## 2.6 Memory Management

Explicit management of dynamically allocated memory (i.e., by use of functions such as `malloc` and `free`) is vulnerable to programming errors such as neglected or false memory release operations causing memory leaks or dangling pointers. This is a severe issue for any type-safe language since memory cannot safely be reallocated as long as there exist dangling references that provide the application with a possibility to access the memory area with its previous type.

Therefore, a JVM that provides dynamic memory management also needs to provide an implicit memory management strategy, also known as garbage collection, which reclaims memory objects automatically but only after having ensured that no references exist to these objects within the application.

Garbage collection frightens off many embedded developers and is under the reputation of being unsuited in the area of deeply embedded systems. One reason for this is that typical garbage collection techniques add noticeable overhead to the system: besides the code required for the garbage collection algorithm itself, the garbage collector (GC) requires runtime data structures that enable the GC to scan all the references existing within the application[1]. Another, more severe issue is that most garbage collection techniques are not suitable for hard real-time systems: Either the GC does not actively fight external fragmentation, which imposes difficulties in giving allocation guarantees, or, particularly for moving (i.e., defragmenting) GCs, needs to stop the application until all references within the application of a moved object have been updated, which may cause unacceptable pause times. Real-time GC is a research area of its own, but there exist GCs, that are suited for being deployed under real-time constraints, such as the GC of JamaicaVM [18]. On the other hand, many real-time applications get along with purely static memory allocation, which also eases the verification required for this kind of application.

KESO supports different heap management strategies. While KESO conceptually only requires the logical separation of different domains' heap areas, we decided to physically separate the heaps rather than allocating objects from a common heap. While this static partitioning of the available memory may seem inappropriate at a first glance, it offers some benefits over a shared heap:

- The static partitioning allows to give heap space guarantees on a per domain basis.
- Domains can opt for different heap management strategies. This allows to mix applications with different dynamic memory requirements in a system and to choose the best-suited strategy for each application.
- Objects of one domain are physically grouped in memory. This allows to additionally apply hardware-based memory protection on targets that are equipped with a memory protection unit and an AUTOSAR OS that supports this type of protection (e.g., in combination with CiAO's memory protection [12]), and increases the robustness with respect to hardware failures.
- Since domains do not share any common references, garbage collection can be performed separately for each domain. This reduces the working set of the GC reducing the time required for one GC run and also reduces the set of control flows that the GC needs to synchronize with to those of the domain that the garbage collection is performed in.

In the remainder of this section, we briefly present the heap management strategies currently available in KESO.

### 2.6.1 Pseudo-Static Allocation

For many embedded applications, dynamic memory management is not required at all. The Java language does, however, not allow a static allocation of objects. For such applications, KESO provides a simple heap strategy that does not provide garbage collection at all. The advantage of this heap implementation is the short, constant and thus easily predictable time required for the allocation of an object. However, since there is no way of releasing the memory of objects that are not required anymore, the application should only allocate memory objects during the initialization phase.

### 2.6.2 Garbage Collection

In addition to the simple strategy, two precise, tracing, non-moving mark-and-sweep garbage collection techniques are available in KESO: A *throughput optimized* GC and an incremental *latency aware* GC. Since both garbage collectors do not address the fragmentation issue, they are not suitable for deployment in hard real-time scenarios without further assumptions. The working principle of the two GCs is the same.

#### *Working Principle.*

The garbage collection is run by a separate control flow, the GC task, that is part of the trusted domain Zero. This single task is responsible for the garbage collection in all domains that use the particular heap management strategy, but only processes the heap of one domain at a time. The GC task is assigned the lowest priority in the system, thus the slack time of the system is used to perform garbage collection runs. This is a good moment to perform a garbage collection, since most tasks will be suspended and only the stacks of blocked tasks need to be scanned.

A GC run is performed in the two traditional phases of mark-and-sweep GCs: In the scan-and-mark phase, the live set of objects is determined by scanning all the reference values present in the application, and the parts of the heap occupied by these live objects are marked. Subsequently, in the sweep phase, the memory of all dead objects is reclaimed (i.e., the parts of the heap that have not been marked in the first phase).

The throughput optimized GC stops all activities within the domain for the entire run of the GC. While this may impose a high pause time, the GC does not need to synchronize with the application and thus yields a high throughput. The incremental GC can be interrupted during a GC run. All critical sections within the incremental GC where interrupt handling needs to be suspended are of constant complexity, so the worst-case interrupt handling latency can easily be predicted. This GC uses OSEK/VDX resources when scanning the stack of a task to synchronize with this task. Tasks with a priority higher than that of the task whose stack is being scanned can continue to run, which prevents unbounded priority inversion.

[1]There are also conservative GCs that use heuristics to distinguish references from primitive data, but these can give no guarantees that all unused memory objects will in fact be reclaimed.

*Reference Scanning.*

The scan phase starts from a root set of references and then transitively proceeds by following all inner references of discovered live objects. All discovered objects are marked (*colored*) to prevent the repeated scanning during one GC run. The root set is comprised by the static reference fields and local reference variables on the stacks of blocked tasks. The GC needs to be able to find these references in memory with as little overhead as possible. We have solved this issue by

- Grouping all static reference fields of a domain into an array that can simply be traversed.
- Using a bidirectional object layout as proposed by SableVM [4] that physically groups reference fields of objects in memory, even when inheritance is being used.
- Grouping all local reference variables in a stackframe, and building a linked list that links the groups of the different stack frames. The list is maintained in the prologues and epilogues of the methods.

# 3. NATIVE INTERFACE

Embedded software often directly interacts with hardware devices, for which in most cases it needs to access memory at specific addresses at which the device registers are mapped into the address space. In addition, particularly to provide a migration path, software may want to make use of existing native libraries or driver packages, or access the operating system's services, which also is a C API in most cases. These actions are not directly possible with the feature set given by the Java language, on the one hand for reasons of safety and on the other hand simply because Java has not been designed as a language for low-level systems programming.

There are several approaches for interacting with native code from Java code. The standard solution is the Java Native Interface [7] (JNI). The JNI provides high compatibility between different JVM implementations and native libraries, but also involves extensive resource efforts. Therefore, the *KESO Native Interface* (KNI) was developed for KESO, which is written in Java and presents a resource saving solution. The KNI can be used to access the OSEK/VDX services, shared memory or peripheral devices.

## 3.1 Concept

The KNI adopts concepts from aspect-oriented programming [9] (AOP). AOP allows transforming a code base by applying *advice* at certain *joinpoints* in the control flow. These joinpoints are most commonly sites of method calls or the bodies of methods. Joinpoints are specified by *pointcut expressions*, which in simple terms are wildcard patterns that match on the function signatures of the methods to be advised. The advice code can then add to or replace the original code at the joinpoint. This code transformation can be performed statically or at runtime by a tool that is referred to as a *weaver*. In the static case, the weaver performs a source-to-source transformation in the original programming language. The compound of pointcut expressions and advice code to give at the resulting joinpoints is called *aspect*.

The KNI does not use an external weaver tool and its functionality is limited compared to real aspect weavers. Aspects are plugged into the compiler. The KNI provides functionality for affecting method calls and the generated code for method bodies as well as *slicing* classes (i.e., adding fields to Java classes). For method invocation joinpoints, the advice code is provided with the context of the call, for example, it can directly access immediate values passed to the method at the specific call. Advice given via the KNI can directly affect the emitted C code for the joinpoint. Additionally, since the aspects are directly plugged into the compiler, the advice code is exposed the full internal API and state of the compiler, which makes the KNI a very powerful mechanism.

The downside is that the KNI has to be used with care as it also weakens the protection concept. Code provided by the KNI must be considered part of the trusted code base. Thus, a KNI invocation complies with switching to privileged mode due to a system call in an operating system kernel with hardware-based memory protection. In this mode, memory protection is deactivated and so only few code fragments shall be implemented in the native section of the application. It is sensible to propagate simple and primitive functions via the KNI such as specific machine instructions like reading and writing device registers. This is—in contrast to JNI usage—associated with a minimal overhead only. Another issue is that code generated by KNI advice code is not subject to the static analyses performed by the compiler. For example, KNI advice must inform the compiler on possibly required methods and types that would otherwise be removed by the dead code elimination step.

In the remainder of this section, we will present KESO's OSEK/VDX API including its service protection mechanism and the mechanism to configure peripheral devices from KESO, both of which are implemented using the KNI mechanism.

## 3.2 OSEK/VDX API with Service Protection

OSEK/VDX system services are provided as static methods of service classes to the Java applications. The services are categorized according to the OSEK/VDX specification, and a service class provides the services for each class (e.g., there is a class `TaskService` that provides all task-related OSEK/VDX services).

For some system services, it is desirable to restrict the access on a per domain basis to guarantee that the domain isolation is not weakened by the ability to abuse system services. As an example, an OSEK/VDX resource could be used to synchronize the concurrent access to a shared data structure by two tasks within the same domain. In this case, a malfunctioning task of another domain could accidentally occupy the resource permanently and prevent the other tasks from running, spreading the error to the other domain. In this case, restricting access to the resource to a specific domain is desirable. On the other hand, OSEK/VDX resources could also be used to synchronize access to a shared memory area used by tasks of different domains. To allow for such a scenario, the access to OSEK/VDX system objects can selectively be granted to other domains in the static configuration.

OSEK/VDX uses scalar values to identify system objects such as tasks or resources. The identifiers are necessary for parameterized system services. OSEK/VDX systems do not support the integration of different applications on one MCU and thus do not provide service protection. In AUTOSAR, service protection is available as an optional feature. In KESO, access restrictions are enforced on the Java language level by encapsulating OSEK/VDX identifiers

```
1  package atmega128;
2  import keso.core.*;
3
4  public class PortA implements MemoryMappedObject {
5    // PortA consists of three 8-bit registers (unsigned)
6    public MT_U8 PINA; // address 0x39
7    public MT_U8 DDRA; // address 0x3a
8    public MT_U8 PORTA; // address 0x3b
9
10   // create a mapping of this class at base address 0x39
11   public final static PortA regs = (PortA)
12     MemoryService.mapStaticDeviceMemory(0x39,
13       "atmega128/PortA");
14
15   // configures one port PIN as output PIN
16   public static void setOutput(byte pinNumber) {
17     regs.DDRA.or(1 << pinNumber);
18   }
19
20   // the class can contain more methods and also regular
21   // fields that do not become part of the mapping
22 }
```

**Figure 2: Memory-Mapped Objects**

into objects. These system objects are statically allocated at system creation time. System objects have to be used as parameters to the system services on the Java level instead of the scalar OSEK/VDX identifiers.

The user application can retrieve a system object by its name using a name service. The name service will only provide system objects that belong to the effective domain of the requesting task or where access from that domain has explicitly been granted in the configuration. Thus, the access to OSEK/VDX system services is effectively restricted by restricting the access to the system objects that are required as parameters. Since the object abstraction imposes some overhead to the system calls that is required to extract the OSEK/VDX scalar identifier, object abstractions have only been created for service classes where access restrictions were found reasonable, that is, for tasks, resources and alarms. Otherwise, the scalar OSEK/VDX identifiers are also used on the Java level. The identifiers are accessible by name similar to OSEK/VDX and provided as constant static values of a class that is automatically generated from the configuration file.

Because system objects are not allocated from any domain's heap memory, are not mutable and do not contain any reference fields, they can safely be accessed from different domains without impairing the spatial isolation.

## 3.3 Peripheral Access

The KESO memory service allows to efficiently access raw untyped memory regions by using the KNI to realize peripheral device access. Registers of peripheral devices are often mapped into the address space of the MCU. Thereby it is possible to control these devices by normal load and store instructions. KESO provides the mechanism of *memory-mapped objects* for accessing such device registers that aims at the same time to be safe and convenient in use.

Memory-mapped objects describe the layout of a specific region in memory and are comparable to C `structs` with a more fine-grained access control. Figure 2 shows a small example as it could be used to access the general purpose

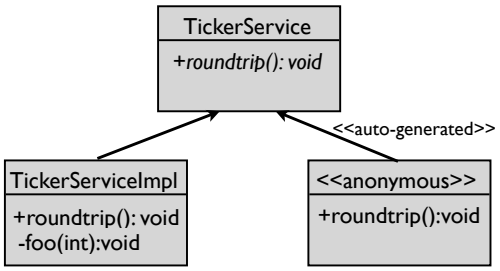



**Figure 3: TickerService Class Hierarchy**

I/O Port A of an ATmega128 MCU.

The layout of the memory region is defined in a regular Java class that contains fields of special memory types (`MT_` prefix). KESO provides memory types for registers of different bit widths and access rights. For example, the type `MT_U8` represents an unsigned 8-bit value that is readable and writable, `MT_U32RO` stands for an unsigned 32-bit read-only value, and `MT_SPACE32` could be used for 32-bits that are reserved in the address space, become part of the mapping (i.e., shift the address of subsequently defined memory type fields) but cannot be accessed by the application. The memory types are classes that contain methods that enable typical accesses such as reading or writing the value and convenience operations such as applying bit-operations on the value. To not affect the soundness of the type system, there are no methods for reading or writing reference values to device memory. In the example in Figure 2, the GPIO register Port A is comprised by three 8-bit registers that are consecutively mapped into the address space at the addresses 0x39–0x3b (lines 6–8). The mapping is then created by calling an API function that creates an instance mapped to the given base address (lines 11–13). The memory-mapped class can also contain regular fields that do not become part of the mapping and methods that provide the external interface to the class if the registers should not be directly accessed from outside the class. For example, a driver class could be mapped to only the registers needed by the driver, not directly exposing the actual registers to other classes (by assigning `private` visibility). In the example, the `setOutput()` method provides an interface for the application to configure a certain PIN of the port as an output pin. The body of the method illustrates how the register is actually accessed.

# 4. INTER-DOMAIN COMMUNICATION

Most sensible applications that are cohabited on a microcontroller do not execute completely independent from one another but need a way to communicate. KESO provides two mechanisms for inter-domain communication; an RPC-like mechanism referred to as portals and shared memory for sharing larger amounts of data. We briefly present both mechanisms in this section.

## 4.1 The Portal Mechanism

Portals are the primary inter-domain communication mechanism in KESO. A domain can export a named *service* that can be imported and subsequently used from other domains. A service definition consists of a *name* that the client domains will use to refer to that service, a Java *interface* and an *implementation* of that interface. The relationships between client and service domains are statically configured in the

```
1 public void foo() {
2   TickerService srv =
3     (TickerService) PortalService.lookup("TickerService");
4   for (int i=0;i<count;i++) { srv.roundtrip (); }
5 }
```

**Figure 4: Retrieving a portal and invoking a service**

configuration file. Only domains that explicitly import the service in the configuration file are able to use the service at runtime. For each service, jino statically allocates a *service object* in the service domain, which is an instance of the service's implementation class. For the client domains, jino creates a *portal* (commonly also known as proxy object). The type of the portal is an automatically generated class that implements the service interface with stub functions that perform the appropriate changes of the protection context.

Figure 3 shows an example for a service interface and implementation. The service implementation may contain additional methods not part of the service interface, which are not made available to client domains.

### 4.1.1 Service Protection

Service protection is available for portals to prevent use by unexpected client domains at runtime. The concept is the same as that used for the OSEK/VDX API. A lightweight name service must be used in the client domain for retrieving the portal object. The name service will return a `null` reference in domains that did not import the service. In the service domain the name service will directly return the actual service object. Figure 4 shows an example for retrieving the portal for a service named *TickerService*. The `lookup` method needs to be provided with a `String` constant, the string does not exist at runtime. The actual lookup is compiled to a simple array lookup.

### 4.1.2 Parameter Passing

In Java, primitive parameters of a method invocation are passed *by-value*, while objects are passed *by-reference*. Applying the by-reference invocation semantic for portal calls would violate domain isolation, since references must not cross domain boundaries. To avoid this issue, a copy of the referenced object along with its transitive closure is created on the heap of the service domain and used in the execution. KESO provides a special marker interface `NonCopyable` to mark classes whose instances should not be copied during a portal call. For these objects, the reference is instead replaced by a `null` reference. The classes of all system objects implement this interface, since system objects must not be copied to other domains to retain service protection.

The duplication of objects leads to a higher resource consumption, makes the portal call potentially costly and normally introduces the need for a garbage collector in the service domain. Moreover, changes to the copied object are not propagated back to the client domain, which is not compliant with the by-reference invocation semantics and may cause unexpected behavior for the programmer.

The developer must be aware of these issues when using the portal mechanism. A simple solution is the exclusive usage of primitive data types when invoking portal services.

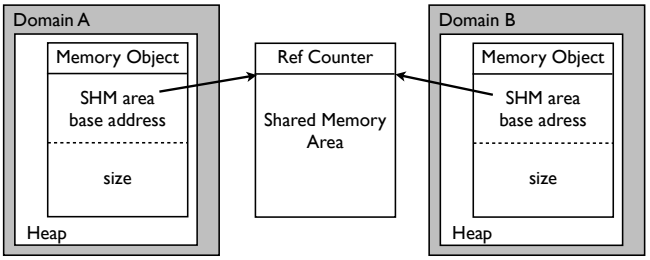



**Figure 5: Memory Objects**

### 4.1.3 Task Migration

KESO portals can be used to allow a task the execution of code within the context of the service domain via a limited set of portal services offered by that domain.

From a conceptual perspective, one would then assume that the code executed in the service domain is executed by a service task belonging to that domain. Implementing such an approach in a way that prevents priority inversion is complex and expensive in an OSEK/VDX system. Therefore, no service tasks are created at all and the portal call is actually handled by the invoking OSEK/VDX task, which is—for the duration of the portal call—*migrated* to the service domain. Instead of a real task switch, the effective domain of the task is changed. In KESO, this means updating two memory words in the simplest case. Migration of a task means that the protection context of the control flow changes. No data is migrated during this process. Thus, as long as a task executes in the context of a service domain, it cannot access data of its original domain. Task migration does therefore not affect the spatial isolation of domains.

## 4.2 Shared Memory

Portals may be inconvenient and costly if two domains need to cooperate on a larger amount of data, since all the data would need to be copied during the portal call. For such situations, KESO provides a low-level shared memory mechanism, where two or more domains can jointly access a common area of memory that is allocated from a special memory pool outside any domain's heap. Figure 5 shows the principle of the shared memory mechanism in KESO. The shared memory area is accessed through *memory objects*, that store address and size of the shared memory area and provide an interface for reading and writing *primitive* data types from and to the shared memory area. Accesses to the shared memory are bounds checked. The application has to handle synchronization of possibly concurrent accesses to the shared memory area itself (e.g., by using a global OSEK/VDX resource). The concept and API of KESO's memory objects is very similar to `RawMemoryAccess` found in the RTSJ.

The shared memory area is a raw data area and not a regular Java object. The address of the shared memory area is a primitive data field from the view of the Java world. To establish a shared memory area, the memory object is initially allocated within one of the participating domains. To enable other domains to access the shared memory area, the memory object can be passed to another domain through a portal. The memory object itself is small and will be duplicated to the heap of the service domain. The cloned memory object internally refers to the same shared memory

area as the original object and enables the service domain to access the shared memory area.

Reference counting is used to release shared memory areas. The reference counter is increased whenever a memory object referring to the area is cloned (e.g., during a portal call). The reference counter is decreased when a referring memory object is reclaimed by the garbage collector.

## 4.3 Comparison to Java Isolates

Application isolation in Java has been standardized in JSR 121 [8], the Java Isolate API. Isolates are conceptually very similar to KESO's domains. The reason that KESO uses a different terminology is due to historical reasons and stems from the JX [5] project, on which KESO is conceptually based. Java Isolates also require the logical separation of different applications' object heaps. The main difference between Isolates and domains is that Isolates are dynamically created while domains are statically configured. KESO does currently not support the Isolate API because there is no application accessible runtime state of a domain and since there is no dynamic creation and destruction of domains, we believe that such state would be of limited use to applications.

Inter-domain communication in the Isolate API is based on message-oriented unidirectional links that can be established among different isolates. The data exchange through such a link is based on the low level of an array of bytes. Serializable objects can be transferred by serializing them to a byte array. As with KESO's portals objects are therefore copied when they need to be passed to a different protection context. This is a direct consequence of the logical heap separation. KESO's portal interface is more convenient for the programmer to use than the link mechanism of the Isolate API where the user has to care for packing data in an array of bytes for sending it and to reassemble the application level data structure from an array of bytes on the receiver side. This explicit conversion of the data may also lead to the data being copied more often than needed. KESO's mechanism is similar in use to Java's Remote Method Invocation mechanism, but the programmer has to be aware of the difference between a portal call where object parameters are copied to the target domain and a regular method call where objects are passed by reference.

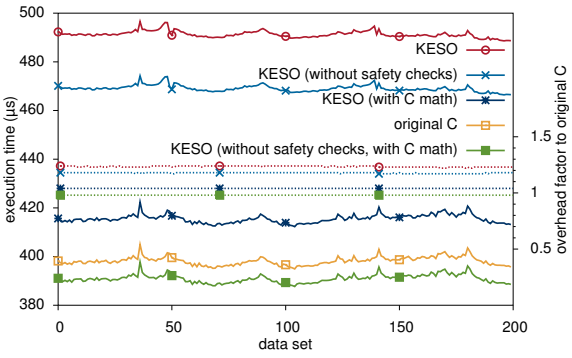
# 5. EVALUATION

In this section, we will give a brief evaluation that is meant to show that the overhead imposed by KESO is tolerable and KESO is applicable for being deployed on the targeted platforms.

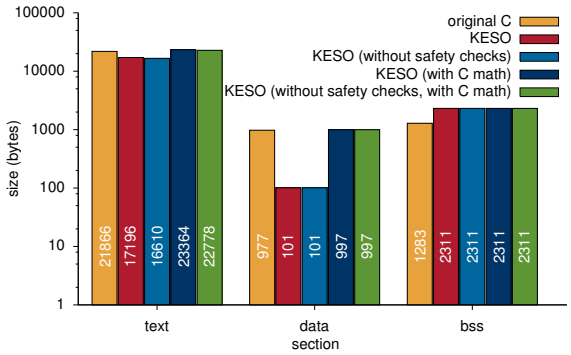
## 5.1 Overhead to C Applications

To get an impression of the cost compared to applications developed in C, we ported the flight attitude control algorithm of the I4Copter [23] quadrocopter to Java. The main control unit on the quadrocopter is an Infineon Tricore TC1796 device (150 MHz CPU clock, 75 MHz system clock, 1 MiB MRAM). The original C variant is generated from a Simulink model using Real-Time Workshop. To ensure comparability, the code of the Java port is as close to the original code as possible with the Java language. An example for a C construct that could not directly be mapped to Java code is stack-allocated arrays, which need to be allocated from the heap in Java.

The control algorithm is responsible for calculating the




(a) Execution Time Overhead




(b) Footprint

**Figure 6: Quadrocopter Attitude Control**

thrust values for the four rotor engines of the quadrocopter. The inputs of the control algorithm are various sensor values from sensors such as gyroscopes and accelerometers. The algorithm is periodically executed every 9 ms and uses single precision floating point arithmetic.

### 5.1.1 *Execution Time Overhead*

To measure the overhead in execution time of KESO, we recorded a trace of the sensor values that serve as input to the control algorithm during a flight. We verified that both the C variant and our Java port output the same actuator values for these samples. For our measurement, we replayed 200 of these data samples and measured the execution time of the algorithm for each sample.

Figure 6(a) shows the execution time in $\mu s$ for each data set and different KESO variants as well as the original C variant. The dotted lines (scaled against the right y axis) show the overhead factor of the KESO variant drawn by the respective continuous line with respect to the C version. The default variant emitted by KESO took 23% longer to compute the actuator values than the original C variant. While we expected some overhead caused by the runtime checks (mostly null and array bound checks) present in the KESO variant, this seemed rather much. To determine the overhead of the runtime checks, we generated a second KESO variant without these checks, which still has a time overhead of 18% compared to the C version. Further investigating the remaining cost difference, we found that our own implementations of the trigonometric functions in the `Math` class were not as efficient as their counterparts in the C library. A KESO variant that

also uses the trigonometric functions of the C library only has an overhead of 4% over the C version. We confirmed that the overhead caused by the runtime checks in this test is about 5–6% by additionally omitting the runtime checks from the variant that uses the C library functions resulting in a variant that provides a throughput that is on par with the C version. The overhead factor is constant within the accuracy of the measurement, since all runtime checks are of constant complexity.

#### 5.1.2 Footprint

Figure 6(b) shows a comparison of the footprint of the original C code and the different KESO configurations, split in the size of the text, data and bss segments. The images that served as basis to the measurement do not include the sample data sets that were included in the execution time measurements. We used KESO's limited error reporting mode, where only the type of an exception is provided to the error handler instead the of the full method signature and bytecode position.

The code size of the default KESO variant is 4.7 KiB smaller than the C version. This is again a result of the trigonometric functions of the C library, which are more efficient than our Java implementations but also noticeably larger in code size. Looking at the KESO variant without runtime checks, we see that the runtime checks contribute 0.6 KiB (3%) to the code size. The KESO variant that uses the C math library is 1.5 KiB (7%) larger in code size than the C implementation, 0.6 KiB of which are caused by the runtime checks and the remaining 900 bytes by the KESO infrastructure code.

The initialized data section shows an overhead of 20 bytes for the KESO versions using the C math library over the C version, which are used for KESO's statically initialized management data (e.g., system objects). The normal KESO variant's data segment is much smaller than that of the C variant. This difference is caused by a large data structure for impure data that is included from the C library, in this case because the trigonometric functions make use of the C library's `errno` variable for error reporting.

Finally, the difference in the bss segment is due to the 1 KiB heap area that was configured for the KESO application and is statically allocated by KESO.

#### 5.1.3 Conclusion

These results are consistent with those of an earlier evaluated prototype application [24]. In both applications, the overhead in code size is less than 10%. Given that KESO could also provide more efficient implementations of the library functions (or simply use the C library's function via the KNI), the overhead in execution time is less than 10% as well.

### 5.2 Inter-Domain Communication Cost

Another important measure is the cost of inter-domain communication, since it determines the extent to which software developers will actually be placing software components in different protection domains. We performed a few microbenchmarks on the cost of different variants of portal calls and compare them to the cost of a regular virtual method call (i.e., the case without spatially isolated components) and also a non-trusted function call in the CiAO OS [12] (i.e., spatial isolation enforced by region-based hardware protection). All measurements were performed on an Infineon Tricore TC1796 MCU clocked at 50 MHz. Table 1 shows the results of these microbenchmarks. The method bodies of all target methods were empty. We thus only measure the cost of the protection domain context switch that is performed on a portal call.

| Call Type | Execution Time |
|---|---|
| portal call | 3.78 $\mu s$ |
| regular virtual method call | 3.12 $\mu s$ |
| CiAO non-trusted function | 33.88 $\mu s$ |
| portal call (2 `int` params) | 4.28 $\mu s$ |
| portal call (3 `int` params) | 4.60 $\mu s$ |
| portal call (1 element linked list) | 30.54 $\mu s$ |
| portal call (2 element linked list) | 57.06 $\mu s$ |
| portal call (3 element linked list) | 85.44 $\mu s$ |

**Table 1: Execution Time for Portal Calls**

#### 5.2.1 Comparison with other Types of Protection

For comparing the cost of a portal call to the cases of no spatial isolation and hardware-based spatial isolation, we use the simplest form of a function that does not take any parameters and not return a value. The portal call introduces an overhead of 21% over the regular virtual method call. This overhead is caused by service protection (i.e., the check, if the calling domain is a valid client to the service) and the change of the running task's effective domain. For comparison, we have also included the cost of a comparable non-trusted function call in a CiAO operating system, which is comparable to a portal call but with domains isolated by hardware-based memory protection rather than constructive software-based memory protection. This measurement shows that the cost of a software-protection context switch is significantly less than that of an MPU reconfiguration that is needed in the case of CiAO.

#### 5.2.2 Portal Calls with Parameters

We also measured the time needed for portal calls with both primitive and reference parameters. In the case of primitive parameters, the added cost is the same as for any regular function that is expanded with parameters. The cost depends on the actions that the C compiler needs to take in order to prepare the parameters according to the ABI. The measured overhead therefore mainly depends on the C compiler and the ABI and is not caused by the portal mechanism.

For portal calls with reference parameters, we passed the head pointer of a linked list of size 1–3. During the portal call, a complex routine that copies the referenced object and all transitively reachable objects to the heap of the target domain is invoked. The cost of the call is dominated by this operation and increased by an order of magnitude compared to the portal calls with only primitive parameters.

## 6. SUMMARY AND CONCLUSION

In this paper, we presented KESO, an LGPL-licensed open-source Multi-JVM for statically configured applications. By exploiting the available ahead-of-time knowledge on the application to generate a tailored JVM variant, KESO can be used on even smallest microcontrollers. The KESO Native Interface provides a lightweight solution to the need of most

embedded applications to program devices and interact with native APIs. KESO comes with safe abstractions built on top of the KNI for accessing operating system services, including language-based service protection, and memory-mapped hardware device registers. For communication between protection domains, KESO provides the high-level RPC-like portal mechanism that allows tasks to perform select actions within a different protection context with their original priority. For larger amounts of data, a shared-memory abstraction provides copy-free cooperation on a raw-memory area.

We believe that KESO enables developers of software for deeply embedded systems to benefit from the use of a modern high-level language, memory protection on low-cost microcontrollers without the need for dedicated protection hardware, and spatial isolation of software components at a reasonable price. Our preliminary evaluation shows that the resource requirements of KESO applications are comparable to that of applications written in C. With restrictions regarding implicit dynamic memory management, KESO is also suited for deployment in real-time settings.

# 7. REFERENCES


[1] AUTOSAR. Specification of operating system (version 2.0.1). Technical report, Automotive Open System Architecture GbR, June 2006.

[2] J.-L. Béchennec, M. Briday, S. Faucou, and Y. Trinquet. Trampoline: An OpenSource implementation of the OSEK/VDX RTOS specification. In *IEEE Conference on Emerging Technologies and Factory Automation, 2006. ETFA '06.*, pages 62–69, September 2006.

[3] G. Bollella, B. Brosgol, J. Gosling, P. Dibble, S. Furr, and M. Turnbull. *The Real-Time Specification for Java*. 1st edition, Jan. 2000.

[4] E. M. Gagnon and L. J. Hendren. SableVM: A research framework for the efficient execution of Java bytecode. In *Java Virtual Machine Research and Technology Symposium (JVM '01)*, pages 27–40, Apr. 2001.

[5] M. Golm, M. Felser, C. Wawersich, and J. Kleinöder. The JX operating system. In *2002 USENIX ATC*, pages 45–58, June 2002.

[6] T. Harbaum. NanoVM - Java for the AVR, July 2006. `http://www.harbaum.org/till/nanovm`.

[7] Java Native Interface Specification 1.1. Sun Microsystems, Aug. 2005. URL: http://tinyurl.com/2wxzf9.

[8] JSR 121: Application Isolation API Specification. Sun Microsystems JCP, June 2006.

[9] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *11th Eur. Conf. on OOP (ECOOP '97)*, volume 1241 of *LNCS*, pages 220–242, June 1997.

[10] J2ME building blocks for mobile devices — white paper on KVM and the connected, limited device configuration (CLDC), May 2000.

[11] D. Lohmann, W. Hofer, W. Schröder-Preikschat, J. Streicher, and O. Spinczyk. CiAO: An aspect-oriented operating-system family for resource-constrained embedded systems. In *2009 USENIX ATC*, pages 215–228, June 2009.

[12] D. Lohmann, J. Streicher, W. Hofer, O. Spinczyk, and W. Schröder-Preikschat. Configurable memory protection by aspects. In *4th W'shop on Progr. Lang. and OSes (PLOS '07)*, pages 1–5, Oct. 2007.

[13] OSEK/VDX Group. OSEK implementation language specification 2.5. Technical report, OSEK/VDX Group, 2004.

[14] OSEK/VDX Group. Operating system specification 2.2.3. Technical report, OSEK/VDX Group, Feb. 2005.

[15] G. Phipps. Comparing observed bug and productivity rates for Java and C++. *Softw. Pract. Exper.*, 29(4):345–358, 1999.

[16] F. Pizlo, L. Ziarek, E. Blanton, P. Maj, and J. Vitek. High-level programming of embedded hard real-time devices. In *ACM SIGOPS/EuroSys Eur. Conf. on Computer Systems 2010 (EuroSys '10)*, pages 69–82, Apr. 2010.

[17] E. Quinn and C. Christiansen. Java Pays – Positively. IDC Bulletin W16212, May 1998.

[18] F. Siebert. Realtime garbage collection in the jamaicavm 3.0. In *JTRES '07: Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems*, pages 94–103, 2007.

[19] D. Simon, C. Cifuentes, D. Cleal, J. Daniels, and D. White. Java$^{TM}$ on the bare metal of wireless sensor devices: the Squawk Java virtual machine. In *2nd USENIX Int. Conf. on Virtual Execution Environments (VEE '06)*, pages 78–88, 2006.

[20] M. Stilkerich and J. Bauer. JOSEK - an open source implementation of the OSEK/VDX API, February 2010. `http://www4.cs.fau.de/Research/KESO/josek`.

[21] Sun Microsystems. Virtual machine specification, Java Card platform version 2.2.2, Mar. 2006.

[22] B. L. Titzer, D. K. Lee, and J. Palsberg. Avrora: scalable sensor network simulation with precise timing. In *IPSN '05: 4th Int. Conf. on Information Processing in Sensor Networks*, page 67, 2005.

[23] P. Ulbrich. The I4Copter project — Research platform for embedded and safety-critical system software. `http://www4.informatik.uni-erlangen.de/Research/I4Copter/`, visited 2010-02-22.

[24] C. Wawersich, M. Stilkerich, and W. Schröder-Preikschat. An OSEK/VDX-based multi-JVM for automotive appliances. In *Embedded System Design: Topics, Techniques and Trends*, IFIP International Federation for Information Processing, pages 85–96, 2007.


# APPENDIX

# A. DEMO INSTRUCTIONS

This section provides information on running a KESO demo program. The aim of this program is to be simple in application logic while still illustrating how to use most of the core functionalities that KESO provides. To be able to try the demo without requiring microcontroller hardware, we chose the Crossbow MICA2 sensor node as the target platform for our demo, for which the free simulator avrora [22] is available. This sensor node is equipped with an 8-bit AVR ATmega128 microprocessor (128 KiB Flash ROM, 4 KiB SRAM). As OSEK/VDX operating system for the AVR we use JOSEK, which is also freely available.

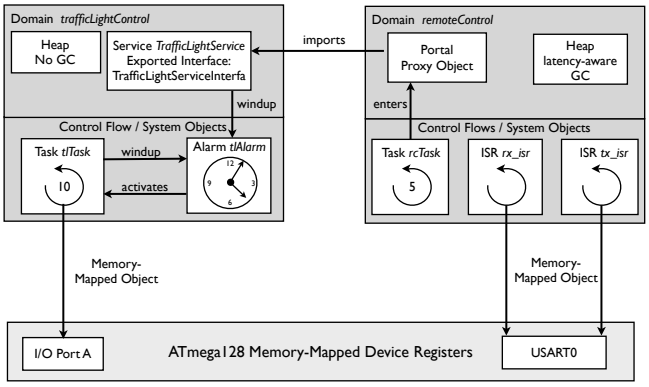



**Figure 7: Traffic Lights Application**

## A.1 TrafficLight Demo Application

The demo application is a (fictive) traffic light control application, that uses the 3 LEDs of the MICA2 to display the current state of the traffic light. The scenario is that of a crosswalk where the light directed towards the pedestrians is normally red and only changes state upon explicit signaling. Since the MICA2 node lacks a button, we use the serial port of the node to signal the external signal change request.

The structure of the demo application is shown in Figure 7. The application consists of two protection domains, where one contains the logic for controlling the traffic light (*trafficLightControl*) and the other (*remoteControl*) the user interaction over the serial port.

The *trafficLightControl* domain contains a task *tlTask* that performs changes to the state of the traffic light that need to be performed at certain times. Whenever the traffic light enters a state that only lasts for a certain time span, the alarm *tlAlarm* is configured to trigger at the respective time. The trigger action of the alarm is to activate *tlTask*.

The *remoteControl* domain interacts with the user over the serial port. The driver for the serial port is also located in this domain. The domain contains two interrupt service routines, which handle the hardware events of received (*rx_isr*) and sent (*tx_isr*) bytes over the serial connection. Whenever *rx_isr* detects a valid incoming command, it activates the task *rcTask* to process the command. *rcTask* cannot directly modify the state of the traffic light controller, since it resides in a different domain, nor can it directly windup *tlAlarm*, as the system object of *tlAlarm* is configured to be accessible only within the *trafficLightControl* domain. Therefore, the *trafficLightControl* domain exports a service, which is described by the Interface `TrafficLightServiceInterface` and implemented in the class `TrafficLightService`. The *remoteControl* domain imports this service, which enables *rcTask* to execute the exported methods in the protection context of the *trafficLightControl* domain.

To configure the state of the LEDs and to interact with the serial port, *tlTask* and the serial port driver use a memory mapped object to access the memory-mapped device registers of the ATmega128 MCU.

## A.2 System Requirements

We assume a Linux system with common development utilities such as GNU make already installed (tested on Ubuntu 10.04 Desktop Ed.). Besides these, KESO needs a Sun-compatible Java development kit, version 1.5 or higher (tested with OpenJDK 1.6) and the Java Compiler Compiler JavaCC, version 4.1 or higher. For the demo, the development toolchain for the AVR architecture (avr-binutils, avr-gcc, avr-libc) is also needed.

## A.3 Installation, Build and Run Instructions

This section contains brief instructions from downloading KESO to running the demo application. More detailed instructions, including sample commands to type, are available online at `http://www4.cs.fau.de/Research/KESO`.

To ease the installation, we have bundled the source code of KESO along with binary versions of JOSEK and avrora for the purpose of this demo. The bundle is available for download at `http://www4.cs.fau.de/Research/KESO/keso_jtres10.tar.gz`. Unpacking the archive will create a new directory `keso`, which we will refer to as the KESO root directory. To setup the needed environment variables inside the active shell, source the script `bin/setup.bash` from the KESO root directory and ensure that the script does not report any errors. In particular, the `JDK` environment variable needs to be set to the installation directory of the JDK to be used *prior* to running the setup script.

To compile the *jino* and run it to translate the Java sources of the demo application to C code change to the `src` subdirectory and run GNU make. The created C sources will be created in the subdirectory `tmp/TrafficLight_Mica2`. Change to that directory and run GNU make to compile the C source code to a binary image that is ready to be loaded onto a MICA2 sensor node. To run the demo application in the avrora simulator, execute the command `avrora -platform=mica2 -monitors=real-time,leds,serial keso_main.od`. Before starting the execution, avrora waits for a TCP connection on port 2390, which will be the remote for the serial port of the MICA2 node. Use a tool such as netcat to interact with the demo application: `nc localhost 2390`. You should now see the greeting of the demo application output by the netcat command and can now send commands to the application by typing to the netcat window. The only supported command is the request for a signal switch, which is issued by typing the character `S`. All other characters will be ignored by the demo application. The switch request is either accepted or rejected by the control application, depending on whether the traffic light is currently in the red state or not. In either case, the application will give feedback on the serial port.

## A.4 Files of Interest

The source and configuration files that belong to the demo application have been commented so that they should be easily understandable. These files are (all paths relative to the `src` directory):

- `rc/trafficlight_mica.kcl`: Configuration file of the demo application. The structure of this file is very similar to the OSEK/VDX OIL [13] format, which is used to configure OSEK/VDX operating systems.
- `libs/mica2_trafficlight`: This directory contains the source files of the application: the implementation of the tasks and the service.
- `libs/avr/.../ATMega128.java`: This class contains the memory-mapped object for the ATmega128's device registers. It was automatically generated from the corresponding avr-libc header file.