OSEK/VDX API for Java

Michael Stilkerich* Christian Wawersich* Andreas Gal†
stilkerich@cs.fau.de wawi@cs.fau.de gal@uci.edu

Wolfgang Schröder-Preikschat* Michael Franz†
wosch@cs.fau.de franz@uci.edu

Donald Bren School of Information and Computer Sciences†
University of California, Irvine
Irvine, CA, 92697, USA

University of Erlangen-Nuremberg*
Computer Science 4, Martensstrasse 1
91058 Erlangen, Germany

ABSTRACT

Modern cars contain a multitude of micro controllers for a wide area of tasks. The diversity of the heterogeneous hardware and software leads to a complicated and expensive integration process.

Integrating multiple tasks on fewer micro controllers reduces diversity and costs of production, but poses new problems with the growing complexity of software on a single micro controller. Therefore a more robust software development process and a safe execution environment is needed in the automotive area and other areas with similar constraints. With the *KESO* system we have implemented a very small and adapted Java execution environment for an OSEK/VDX operating system to address these issues.

In this paper we present our approach for a low overhead OSEK/VDX system interface, which is an integral component of the KESO system. We show how access to the system services can be restricted at low cost to ensure the isolation of tasks by the use of type-safety and modern compiler techniques, while maintaining a familiar programming interface for developers that are used to OSEK application development using the C programming language.

1. Introduction

Modern cars contain a multitude of micro controllers for a wide area of tasks, ranging from convenience features such as the supervision of the car's audio system to safety relevant functions such as assisting the braking system of the car. The diversity of both, hardware and software, likely to stem from a variety of manufacturers, leads to problems, that complicate the integration of heterogeneous hardware and software to form a connected and cooperating system.

Integrating multiple tasks on fewer, but more powerful micro controllers reduces diversity and costs of production. The approach, however, poses new problems. In a network of dedicated micro controllers, the deployed software is physically isolated from each other. This isolation is lacking among tasks running on the same hardware, which enables erroneous tasks to corrupt the memory of other tasks, spreading the error and possibly resulting in a failure of all tasks running on the micro controller.

The impacts of such a failure depend on the duties assigned to the software and can be catastrophic. Software development in the C programming language and Assembler tends to be error-prone, yet they dominate the development of embedded software. This, along with the ever-increasing software complexity, even aggravates the problem and increases the importance of isolation.

Though the approach of software integration leads to the discussed problems of error-prone software development and lacking software isolation, these problems are well-known from the area of personal computing and concepts have been developed to solve these problems. These concepts can be adapted and migrated to embedded systems. Object-oriented programming facilitates coping with complex software and makes the software-development process more robust. Virtual machines such as the Java Virtual Machine (JVM) [5] allow running applications in an isolated environment and provide memory protection without the need for a memory protection unit (MPU) or even a memory management unit (MMU) through type-safety, preventing the use of arbitrary values as memory references.

Migrating these concepts to an embedded JVM provides less error-prone software-development using Java as an object-oriented programming language and isolates tasks (or threads, to speak in terms of Java) to a certain degree. Yet the approach of a single JVM has several shortcomings. First, static class fields allow the wandering of references. Second, resources are shared equally among the threads, which is not tolerable in real-time systems. The Java operating system JX [4] introduces isolated domains addressing these problems.

Our prototype system *KESO* will demonstrate that both, a robust development process for embedded software with OOP and the safe integration of tasks on a single micro controller by means of strong isolation, can be achieved while maintaining real-time capabilities and a small footprint suitable for the limited resources available on embedded systems. Built on top of an OSEK/VDX [6] operating system, which is widely used by the automotive industry, KESO provides a similar system configuration and creation process and a familiar interface to OSEK system services for embedded software engineers. We adopted the isolated domains of the JX OS and redesigned the architecture to fit more closely to the needs of the automotive environment.

The remainder of this paper is structured as follows: In Section 2 we give a short overview of the KESO system architecture

PLOS '06 San Jose, California, USA

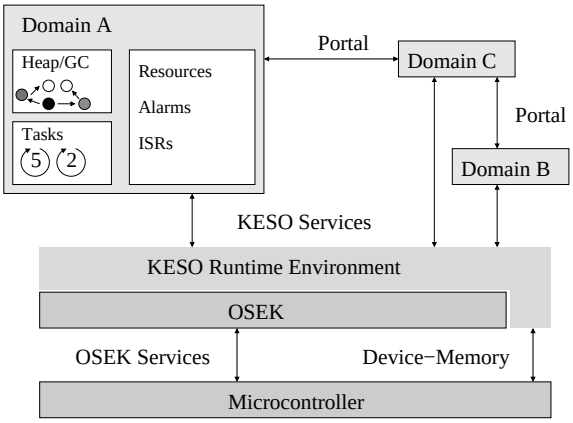

Figure 1. KESO Conceptional Architecture

and build process. Section 3 describes the OSEK/VDX Java API implementation and the overhead introduced by the object oriented design. In Section 4 we discuss the related work in the embedded automotive area.

2. KESO Architecture

2.1 OSEK Overview

OSEK/VDX systems are static systems that provide two different kinds of abstraction for execution, interrupt service routines (ISR) and tasks. Tasks are scheduled based on a fixed priority, whereby either a fully preemptive or a non preemptive execution model can be configured on a per task base. OSEK/VDX provides lock-free synchronization based on a priority ceiling protocol. Events provide a notification mechanism that allows tasks to enter a waiting state until a certain external condition occurs. Alarms allow the triggering of various actions after defined, possibly cyclic delays.

OSEK systems are highly adapted to the needs of the applications. The OSEK system generator (OSEK SG) automatically generates an OSEK kernel that does only contain the parts of the OSEK system required by the application.

2.2 Conceptional Architecture

The architecture of the KESO system is illustrated in Figure 1. The system is statically configured and there is no dynamic class loading or dynamic task creation.

Domains

The system is structured in domains that are strongly isolated from each other. Each domain appears as a self-contained JVM to the application programmer. Therefore each domain contains its own set of static class fields and its own heap, whereby each domain can choose from different garbage collector implementations. Domains are the fundamental units of protection and allocation of the resource memory.

References to objects on the heap of one domain never cross a domain boundary. Thereby, the structuring of the system in domains also produces distinct sets of objects, which eases the work of garbage collectors, that do only need to examine the object set of one domain at a time.

Tasks

In KESO the OSEK task is presented to the application developer as `Task` class instead of the Java `Thread` class. Denoting the scheduleable units as tasks instead of threads expresses the differences in the execution model. OSEK tasks are statically allocated and scheduled based on a fixed priority while Java threads are allocated at runtime and have a wide range of different schedule policies.

In KESO, each `Task` object is assigned a domain and cannot migrate from its assigned domain to another domain.

Portals

Inter-domain communication is possible via portals. A *service domain* can provide a *service* by exporting the interface of a service object. A task of another domain can obtain access to the service as client via a global name service. The service object is represented by an auto generated proxy object and all parameters of a method invocation are copied. This assures that no object references cross the domain boundaries.

KESO Runtime Environment

The two major functions of the KESO runtime environment layer are the provision of a Java runtime environment for the Java applications and a Java class library providing access to KESO services.

KESO services can be divided in three classes:

- Provision of OSEK services on the Java level
- Device-Memory
- Device drivers

The first class allows the user applications to use the system services of the underlying OSEK operating system on the Java level. The OSEK services include synchronization and notification mechanisms as well as limited access to the hardware, e.g. through services, that allow to disable and enable interrupts. The KESO services of this class are part of the OSEK Java API (Section 3).

Further hardware access to memory mapped device registers is possible through *Device-Memory*, that is also a part of the JX operating system. Device-Memory provides methods to access a specific region of memory with methods similar to raw access. The memory region accessible via Device-Memory can be limited to prevent a breakout from the Java protection mechanisms, e.g. by modifying the heap of a domain or the stack of tasks. Device-Memory allows, amongst other things, the implementation of device drivers in Java.

2.3 Code Generation Concept

The user applications are developed in Java and available as Java bytecode after having been processed by a Java compiler. Interpreting or even compiling the bytecode to native code at runtime on the target micro controller is not feasible, because memory and CPU power are very limited on the target platforms.

Instead, the bytecode is compiled to C source code ahead of time by the *KESO builder*. Creating C source code rather than directly compiling the bytecode to native code has the advantage that the application can be easily integrated into the normal OSEK/VDX build process.

The generation process of a KESO system is illustrated in Figure 2. The generated C code does not only contain the compiled class files, but also the KESO runtime data structures, that include

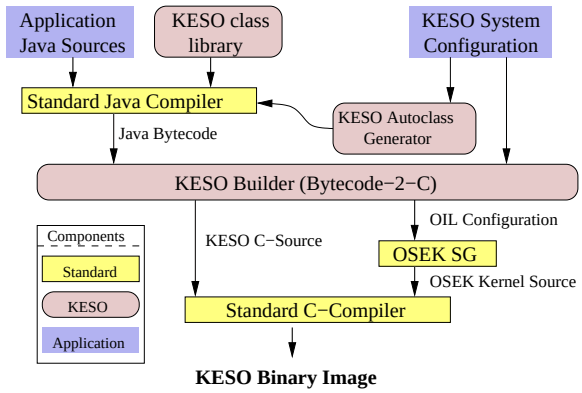

Figure 2. KESO System Generation Process

data such as runtime type information and the virtual method table. Moreover, additional code is inserted to retain the properties of a JVM, such as `null` reference checks and array boundary checks, and the code of other services of the KESO runtime layer, such as the garbage collector and the portal services.

3. OSEK Applications Interface

The OSEK abstraction layer provides a Java class library that allows user applications to access the system services of the underlying OSEK system. Furthermore, the OSEK abstraction layer provides means to restrict the access to the services to guarantee domain isolation also on this level.

3.1 Conceptual Design

OSEK system services are provided as static methods of *service classes* to the Java applications. The services are categorized according to the OSEK specification, and a service class provides the services for each class, e.g. there is a class `TaskService` that provides all task-related OSEK 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 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 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, resources can also be configured with global visibility.

OSEK uses scalar values to identify system objects such as tasks or resources. The OSEK system generator (OSEK SG) automatically generates C macros that allow to use the identifier of a system object by specifying its name. The identifiers are necessary for parametrized system services.

In KESO, access restrictions are enforced on the *Java language level* by encapsulating OSEK identifiers into objects. These so-called *system objects (SO)* are statically allocated by the KESO builder at system creation time and do not belong to any domain heap. SOs have to be used as parameters to the system services on the Java level instead of the scalar OSEK identifiers.

The user application can retrieve a SO by its name using a *name service*, similar as OSEK identifiers are referenced in C code. The name service will only provide SOs that belong to the same domain as the requesting task and global SOs. The OSEK identifiers are stored in a private field of the system objects and cannot be extracted by the application. Furthermore, the user application cannot instantiate additional system objects that could be used to abuse system services. Thus, the access to OSEK 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 scalar identifier, object abstractions have only been created for service classes where access restrictions were found reasonable, i.e. for tasks, resources and alarms. Otherwise, the scalar OSEK identifiers are also used on the Java level. The identifiers are accessible by name similar to OSEK and provided as constant static values of a class that is automatically generated from the KESO configuration file.

Even though the access to system services is restricted through the name service, a task may nevertheless use system services with the scope of another domain through a portal. The available portal services are limited and well-defined by the providing service domain, and the impacts are therefore kept manageable and at a desired amount.

3.2 Magic Methods

The system service methods of the KESO class library need to call the OSEK system services in the generated C code, which is not possible with the use of pure Java code. The Java class library uses so-called *magic methods*, i.e. Java methods that are specially treated by the KESO builder.

Special code for a magic method can either be generated at the call-side, replacing the invocation of the magic method, or in the body of the magic method, leaving the call to the magic method untouched. The first variant corresponds to an inlining of the magic method.

The preferable way mostly depends on the complexity of the generated code. Intercepting magic methods at the call-side can save the overhead of a method call, but is only suitable for short code fragments, whereas leaving the calls to a magic method untouched and generating special code in the method body instead is appropriate for larger portions of code.

3.3 Name Service Implementation

The object abstractions and the name service have been similarly implemented for tasks, resources and alarms. In the following, we will generally talk of resources, but everything is applicable to alarms and tasks, too.

Figure 3 shows an example for the data structures involved in the name service. The example contains a local resource of domain 2 (ResourceA), a globally visible resource (ResourceB) and a globally visible resource (ResourceC) that is shadowed in domain 1 by a local resource with the same name.

The references to the SOs are managed in an array, the *resource index*. The scalar OSEK identifier of a resource can be used as an index into this array to acquire the corresponding SO. Additionally, a special scalar identifier `INVALID_RESOURCE` is introduced, that is assigned the `null` reference on the Java level, i.e. `null` is stored in the location of the resource index that is indexed by the `INVALID_RESOURCE` identifier.

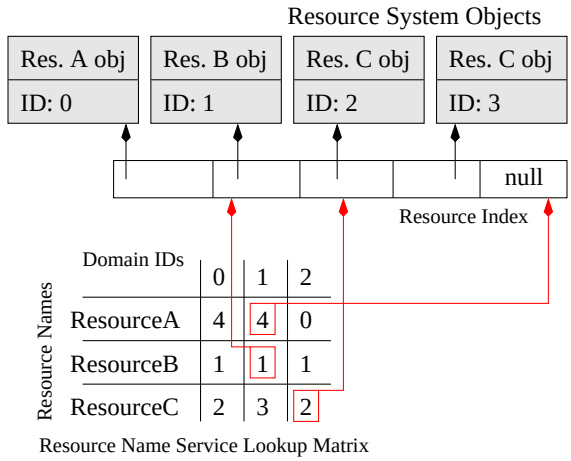

Figure 3. Resource Name Service Data Structures

For the name service, a matrix mapping a resource name and a domain identifier to an OSEK identifier is created, the *resource name service look-up matrix*. The lookup matrix contains a row for each resource name and a column for each domain, plus a column for global SOs if applicable. Each element of the matrix either contains the OSEK identifier of the resource with the respective name, if that resource is visible within the domain, or else the `INVALID_RESOURCE` identifier. In case the same name is used multiple times, only one row is created for the name. The elements in this row then contain the appropriate OSEK identifier for the resource visible within that domain, where a domain local resource shadows a global resource with the same name.

3.3.1 Java Interface to the Name Service

The name service for resources is accessible as a method of the `ResourceService` class, that also contains the other resource-related OSEK system services:

```
public static Resource
      getResourceByName(String resName);
```

The parameter to this method is the name of the resource as specified in the system configuration file. The name of the resource must be provided as a String constant to allow the builder to resolve the name at compile time. The builder determines the number of the row assigned to the name and resolves the lookup for ResourceC to the following:

```
resource_index
 [
   resource_lookup_matrix[2][current_domain]
 ]
```

This first determines the OSEK identifier in the lookup matrix and then acquires a reference to the SO from the resource index, or `null` if the resource is not accessible from the domain. As an optimization, in cases where the lookup can be resolved at compile time, which is the case for global resources, that are not shadowed in any domain, such as ResourceB in the example, the corresponding OSEK identifier is statically inserted in the compiled code.

3.3.2 Related Issues

Global objects violate the domain isolation criterion, that a reference to the same object never crosses a domain boundary. This is, however, not a problem in the case of the `Alarm` and `Resource` SOs, as these

- do not contain any reference fields, and subclasses of the `Alarm` and `Resource` classes must not be created. They can therefore not be used as a container to transport references from one domain to another.
- are immortal objects and not subject to garbage collection. If they were, they would possibly be reclaimed by the garbage collector, that only determines the reachability within one domain.

The first of the above conditions is not satisfied by `Task` SOs, because subclasses of the `Task` class can (and should) be created and these SOs could therefore be abused to transport references across a domain boundary. However, because tasks are objects that represent executable code, tasks outside a domain do not make sense. Therefore, tasks cannot be configured globally and always have to belong to a domain.

The data structures are only created if they are required, e.g. if resources are not used in the system, the resource index and the resource lookup matrix are not created. This reduces the size of the KESO system.

3.4 Problems imposed by Portals

When passing object parameters to a portal, the referenced objects, including the transitive closure, are copied to the service domain. The object copies are allocated from the heap of the service domain.

This approach causes some problems when system objects are passed to a portal call, as these are used to restrict the access to the services. When accessing an OSEK service, a copy of a system object is of the same value as the original object. Therefore, portal calls would allow system objects to spread across domains and escape their scope.

To solve this problem, a marker interface `NonCopyable` was introduced to mark classes that are not to be copied across portal calls. Instead, when a `NonCopyable` object is passed to a portal call, it is replaced by a `null` reference. The system object classes `Alarm`, `Resource` and `Task` implement this interface. Furthermore, classes of the user application may implement this interface to prevent instances from being copied across portal calls.

3.5 Overhead

The KESO runtime environment adds some overhead compared to a plain OSEK system. This overhead is on the one hand imposed by the object abstractions discussed above. On the other hand, there is some additional work that needs to be performed at system startup and some effort is required to maintain the proper runtime environment.

At system startup, the KESO data structures need to be initialized and the constructors of statically allocated objects have to be invoked. These are comprised by

- heap management data structures. The heap area itself is not cleared on startup. Memory of object instances is cleared upon allocation.
- task, resource and alarm index (where applicable)
- class initializers
- constructors of task SOs. Alarm and resource SOs have default constructors that do not need to be invoked.

NO OVERHEAD	ID CONVERSION OVERHEAD	OTHER
EnableAllInterrupts()	ActivateTask()	ChainTask()
DisableAllInterrupts()	GetTaskState()	TerminateTask()
ResumeAllInterrupts()	GetResource()	GetTaskID()
SuspendAllInterrupts()	ReleaseResource()	WaitEvent()
ResumeOSInterrupts()	SetEvent()	GetAlarmBase()
SuspendOSInterrupts()	GetEvent()	
Schedule()	GetAlarm()	
ClearEvent()	SetRelAlarm	
GetActiveApplicationMode()	SetAbsAlarm()	
StartOS()	CancelAlarm()	
ShutdownOS()		

Table 1. System Service Overhead Classes

The time required for this initialization work depends on the system configuration.

Another aspect where KESO adds overhead to the system is the task switch. Upon a task switch, the domain environment needs to be updated for the scheduled task. This requires the invocation of the `GetTaskID()` service to determine the SO of the scheduled task. The effective domain is then read from the SO. Both the SO of the current task and the id of the current domain are kept in global state variables.

The overhead of the OSEK abstraction layer to the OSEK system services can roughly be broke down into two classes. The first class is comprised by system services to which invocations of the magic methods are replaced at the call-side by invocations to the respective OSEK service, which does not impose any overhead.

The second class covers system services that are passed system object parameters. These services require the extraction of the OSEK identifier from the system object, which also requires a `null` reference check on the passed reference. The extraction of the OSEK identifier and the invocation of the OSEK service is generally implemented in a dedicated method. The calls to the magic methods are left intact and add the overhead of a method call for this class of system services. Table 1 shows the classification of the various OSEK services.

Some services cannot be assigned any of the above classes: The `GetAlarmBase()` service is passed a second reference parameter, which adds a `null` reference check. Furthermore, the OSEK `AlarmBase` type needs to be converted to the respective Java type, which requires 3 scalar copies. Otherwise, the overhead of the ID conversion applies.

The SO of the current task is stored in a global field upon task switch. An invocation of the `GetTaskID()` OSEK service can therefore be replaced at the call-side by the global field, which speeds up the use of the service in the user application.

The `TerminateTask()` service is replaced at the call-side with a call to the OSEK service. To notify the garbage collector that the stack of the task is empty, a mark needs to be stored in the stack index before terminating the task.

`ChainTask()` requires the overhead of the ID conversion plus the overhead of the `TerminateTask()` service.

Before blocking a task using the `WaitEvent()` system service, a stack map has to be registered to allow an interleaving garbage collector to scan the stack of the task. The invocation is otherwise replaced at the call-side which incurs no further overhead.

4. Related Work

The AJACS (Applying Java to Automotive Control Systems) [1] project did general research on deploying Java in automotive control systems, i.e. on static embedded systems, and therefore has the same target platforms as the KESO system.

The main objective was developing and defining an open technology that is based on existing standards of the automotive industry, explicitly naming OSEK/VDX operating systems. The expected benefits of using Java on automotive control systems were restricted to a single JVM approach, particularly to software structuring, reusability, dependability, portability and robustness benefits.

The AJACS project concluded with numerous recommendations on how to deal with the limitations of various aspects of Java with respect to real-time support.

While KESO also targets all of the benefits expected from the sole use of Java for the development of embedded applications, it mainly differs from AJACS in the multi JVM approach, that puts the main focus on the isolation of the tasks integrated on the controller.

JPure [2] was a survey of a Java execution environment for controller networks. Contrary to AJACS and KESO the Pure [3] operating system family was used instead of an OSEK/VDX operating system. The main objective was to distribute parts of the JVM to solve the problem of restricted resources.

Both projects did not introduce an OSEK/VDX API and tried to be as Java conform as possible.

References

[1] AJACS: Applying Java to Automotive Control Systems Concluding Paper V2.0, July 2002. http://www.ajacs.org/.

[2] D. Beuche, L. Büttner, D. Mahrenholz, W. Schröder-Preikschat, and F. Schön. JPure - purified java execution environment for controller networks. In *International IFIP TC10 W'shop on Distributed and Parallel Embedded Systems (DIPES '00)*. Kluwer, Oct. 2000.

[3] D. Beuche, A. Guerrouat, H. Papajewski, W. Schröder-Preikschat, O. Spinczyk, and U. Spinczyk. The PURE family of object-oriented operating systems for deeply embedded systems. In *2nd IEEE Int. Symp. on OO Real-Time Distributed Computing (ISORC '99)*, pages 45–53, St Malo, France, May 1999.

[4] M. Golm, M. Felser, C. Wawersich, and J. Kleinöder. The JX operating system. In *2002 USENIX TC*, pages 45–58, Berkeley, CA, USA, June 2002. USENIX.

[5] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. AW, second edition, 1999.

[6] OSEK/VDX Group. *Operating System Specification 2.2.3*. OSEK/VDX Group, Feb. 2005. http://www.osek-vdx.org/.