An OSEK Operating System Interface and Memory Management for Java

Diploma Thesis

by

Michael Stilkerich

born April 25, 1982, in Forchheim

Department of Computer Science

Distributed Systems and Operating Systems

University of Erlangen-Nuremberg

Tutors:	Dipl. Inf. Christian Wawersich
	Dipl. Inf. Andreas Gal (UC Irvine, USA)
	Prof. Dr.-Ing. W. Schröder-Preikschat
	Prof. Michael Franz (UC Irvine, USA)

Begin: April 1, 2006
Submission: August 10, 2006

Eine OSEK Betriebssystemschnittstelle und Speicherverwaltung für Java

Diplomarbeit im Fach Informatik

vorgelegt von

Michael Stilkerich

geb. am 25. April 1982 in Forchheim

Angefertigt am

Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme)

Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer:	Dipl. Inf. Christian Wawersich
	Dipl. Inf. Andreas Gal (UC Irvine, USA)
	Prof. Dr.-Ing. W. Schröder-Preikschat
	Prof. Michael Franz (UC Irvine, USA)

Beginn der Arbeit:	1. April 2006
Abgabe der Arbeit:	10. August 2006

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe, und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den 9. August 2006 __________________________

Abstract

KESO is a Java runtime environment for embedded systems based on a standard OSEK operating system. KESO provides a strong isolation of the different applications running in the system, that is solely based on software.

Software-based memory protection can be advantageous, because hardware without a memory management unit (MMU) is cheaper to obtain than microcontrollers with an MMU and can provide a higher flexibility for allocating system resources.

The use of the object-oriented programming language Java for developing embedded software furthermore provides a more robust software development process than the low-level programming languages C and Assembler, that still dominate the embedded field and tend to be error-prone with the ever increasing complexity of software.

In this thesis, a Java abstraction layer for the OSEK operating system interface was developed, which has the task of providing access to the OSEK system services to Java applications, while at the same time restricting access to the system services in order to retain the strong isolation among the different applications.

Furthermore, a Java interface was created, that provides direct access to memory at specific addresses and therefore access to memory mapped device registers. The memory areas accessible by this means can be restricted to guarantee, that the type-safety of Java is not violated. This interface allows, amongst other things, the implementation of device drivers in Java.

The main part of this thesis is a heap implementation for KESO that provides automatic memory management. The garbage collector assumes a cooperative application developer, that incorporates the garbage collector in the design of the whole system and ensures, that enough time for the garbage collection will be available.

The developed heap implementation does presently not fulfill hard real-time requirements, mostly because the fragmentation problem has yet to be solved. The garbage collector is, however, interruptible at any stage with very low latencies, and does therefore not affect the real-time capabilities of real-time parts of the system, that do not use automatic memory management.

The concluding measurements show, that the overhead caused by the measures taken to ensure the interruptibility of the garbage collector is tolerable for most applications, and justifiable by the benefits that the application has of automatic memory management.

Zusammenfassung

KESO ist eine Java Laufzeitumgebung für eingebettete Systeme, die auf einem traditionellen OSEK Betriebssystem basiert. KESO ermöglicht eine rein software-basierte, starke Isolierung der verschiedenen im System laufenden Anwendungen.

Ein software-basierter Speicherschutz bietet gegenüber einem auf Hardware basierten Speicherschutz die Vorteile, dass zum einen Mikrocontroller ohne Speicherverwaltungseinheit in der Regel günstiger in der Anschaffung sind und zum anderen eine flexiblere Allokation von Systemressourcen möglich ist.

Die Verwendung der objektorientierten Programmiersprache Java zur Entwicklung von eingebetteter Software bietet zudem einen robusteren Entwicklungsprozess als die in diesem Feld dominierenden Sprachen C und Assembler, welche in Verbindung mit ständig komplexer werdender Software die Wahrscheinlichkeit von Fehlern in der Programmentwicklung erhöhen.

In dieser Arbeit wurde eine Java Abstraktionsschicht für die Systemschnittstelle des OSEK Systems entwickelt. Diese hat unter anderem die Aufgabe, OSEK Systemdienste für Java Anwendungen bereitzustellen, aber auch den Zugriff auf diese durch die verschiedenen Anwendungen im System zu regeln, so dass die starke Isolation weiterhin gewährleistet bleibt.

Außerdem wurde eine Java Schnittstelle geschaffen, die den direkten Zugriff auf Speicher an festen Adressen und damit auf in den Speicher abgebildete Geräteregister ermöglicht. Die so zugänglichen Adressbereiche können eingeschränkt werden und gewährleisten so, dass die Typsicherheit von Java nicht verletzt wird. Diese Schnittstelle ermöglicht unter anderem die Entwicklung von Gerätetreiber in Java.

Den Hauptteil dieser Arbeit stellt eine Heap Implementierung für KESO mit automatischer Speicherverwaltung dar. Die Speicherbereinigung geht von einem kooperativen Anwendungsentwickler aus, der sie in den Zeitplan des Gesamtsystems einplant und gewährleistet, dass ausreichend Zeit für die Speicherbereinigung zur Verfügung steht.

Vor allem aufgrund der noch ungelösten Fragmentierungsproblematik kann die entwickelte Heap Implementierung derzeit allerdings noch keine harten Echtzeitanforderungen erfüllen. Der Speicherbereinigungsmechanismus ist jedoch mit geringer Latenz an jeder Stelle unterbrechbar und beeinträchtigt somit nicht die Echtzeitfähigkeit anderer Teile des Systems, die auf eine automatische Speicherbereinigung verzichten.

Die abschließenden Messungen zeigen, dass der Mehraufwand, der durch die zur Unterbrechbarkeit der Speicherbereinigung notwendigen Maßnahmen entsteht, für die meisten Anwendungen tolerierbar ist, und durch die Vorteile, die den Anwendungen durch die automatische Speicherbereinigung entstehen, gerechtfertigt wird.

Contents

Chapter 1

Introduction

1.1 Motivation

1.1.1 Problems Imposed by Heterogeneity in the Embedded Field

While the development speed in the field of universal computers is abating, the embedded computing industry is booming. There is a vast number of visions in this field and promising concepts like the idea of *pervasive computing* are still in their infancy. An end of this fast-paced development is not in sight.

The automotive industry constitutes one major application area of embedded computing. Modern cars may contain up to well over 100 microcontrollers for all kinds of tasks, ranging from convenience functions, such as the automatic regulation of the air conditioning to maintain a set temperature, over infotainment components, e.g. a car navigation system, to safety-critical functions, like controlling the braking system of the car.

More often than not, the multitude of microcontrollers in a car as well as the software running on these controllers are produced by a number of different manufacturers. The resulting heterogeneity in the deployed hardware and software imposes integration problems.

1.1.2 Solving Integration Problems by Reducing Diversity

With microcontrollers becoming more and more powerful, the heterogeneity of the hardware can be reduced by integrating multiple tasks on a single, more powerful microcontroller.

This approach introduces new problems though. When deploying dedicated microcontrollers, physical separation provides isolation of the tasks running on these controllers. An erroneous task running on a dedicated controller cannot spread and impact tasks running on other microcontrollers. When multiple tasks share a microcontroller, the physical isolation ceases. Usually, there are no memory protection mechanisms preventing one task from accessing or modifying another task's memory. Thus, a malfunctioning task can affect the other tasks on the controller and possibly cause a malfunction of the whole system.

Because the software integrated on a microcontroller is often developed by different manufacturers, this problem can hardly be regulated by means of a diligent quality assurance. In case of the failure of a component in the car, the microcontroller, that was controlling the component that failed, is easy to determine, but the software task, that was the cause for the failure, and that may be a different task than the one that was controlling the failed component, can hardly be identified. In case of claims for damages, however, it is necessary to determine the manufacturer of the malfunctioning component, which would require a complex and costly reconstruction of the failure.

The problem is even aggravated by the way most embedded software is developed. The dominating programming languages in the embedded field are C and Assembler, with which development tends to be error-prone, because they provide few concepts for supporting robust code development. Increasing functionality of software is accompanied by increasing code size and complexity, both factors that also raise the chance of faulty software development.

1.1.3 Software Isolation in the Area of Personal Computing

Though these problems are new to the embedded field, they are well-known from the field of personal computers and mature concepts for solving these problems have already been developed.

Object-oriented programming (OOP) languages such as C++ and Java allow developing applications in a more maintainable and less error-prone manner.

Encapsulation is one of the core concepts carried out by OOP, that restricts access to data to a well-defined set of established channels. Though this concept reduces the error-proneness of application development and is encouraged by OOP, it is commonly not enforced by object-oriented programming languages.

The Java platform goes one step farther by running software in a controlled environment isolated from the rest of the system, the Java Virtual Machine (JVM) [LY99]. Java achieves software-based memory protection without a memory management unit (MMU) through its strict type system, that inhibits the use of arbitrary values as addresses. Type-safety is, in the last instance, ensured by the Java bytecode verifier, but also at compile time by a trustworthy Java compiler.

The memory protection provided by the Java platform, however, is still not sufficient to solve the isolation problem. Running multiple tasks, or threads in terms of Java, in a single JVM still possibly allows one task to gain access to the memory of another task through static class fields, that can be used by all tasks running in the JVM simultaneously. Furthermore, there is no means of task specific resource allocation, which is

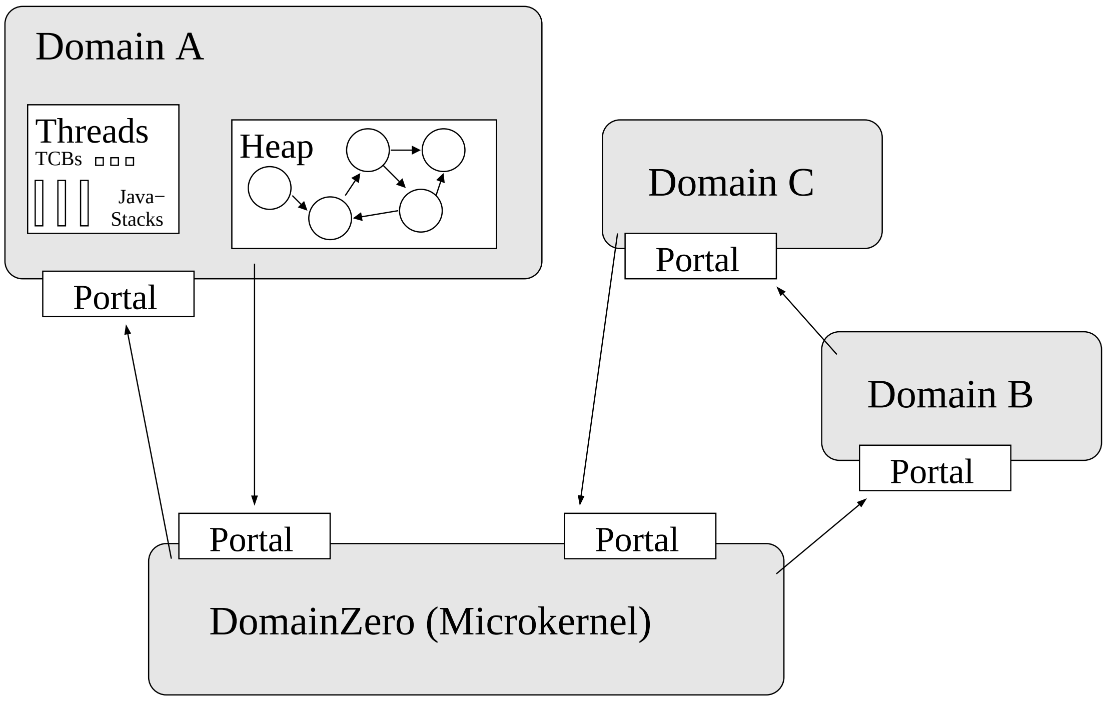


Figure 1.1: Architecture of the JX operating system. JX is structured in domains with strong isolation among each other. Each domain is running pure Java code, has a heap and a garbage collector of its own, whereby different domains may deploy different algorithms for garbage collection. DomainZero is an exception from this and represents the microkernel of the system, a minimal trusted codebase written in C and Assembler.

required for a real-time system. Memory and computing time are shared equally among the running tasks, without a way to create a priority relation between the different tasks.

1.1.4 Migration of Operating System Concepts

Java is not really to blame for the above shortcomings with respect to thread isolation. Resource allocation and provision of a process concept—which eventually provides the isolation in modern operating systems for personal computers—are rather the duties of an operating system than that of a (virtual) machine.

The Java operating system JX [GFWK02] solves the above problems for the domain of personal computers. The JX system architecture is shown in figure 1.1. Domains are the fundamental units of resource allocation and memory protection in the JX operating system. From the application developer's point of view, each domain represents a JVM of its own. In spite of being strongly isolated, domains are yet able to communicate and cooperate via the *portal* mechanism, that allows domains to offer service methods to other domains and can be used in a manner similar to Java RMI (Remote Method Invocation) [Mic97]. Whenever an object is parameter to a service method invocation at a portal, a copy of the object along with its transitive closure is created in the service domain and used in the execution. Thus, a reference to an object of a domain's heap never crosses a domain boundary.

An attempt [Dom04] to port JX to the Lego Mindstorm RCX architecture has shown, that JX in its current form is not suitable for being deployed on microcontrollers. The JX microkernel alone has a size of 70–100 kB, which is small in the area of personal computers, but too big for embedded microcontrollers. This is due to features, that are desirable and required in a Java operating system for personal computers, such as the dynamic loading of classes. Embedded systems in contrast are statically configured. A facility such as a dynamic class loader is not required, because all classes ever used are known at system creation time.

Inspired by JX concepts, KESO, a Java runtime environment specially suited for embedded environments, built on top of a standard operating system for microcontrollers, has been developed. An OSEK [OSE05] operating system was chosen as the underlying operating system, because of its widespread use in the automotive industry.

In this thesis, a Java abstraction layer for the underlying OSEK operating system was developed, that provides a controlled way for Java applications to access the OSEK system services. Furthermore, a heap implementation that provides automatic memory management was developed.

This thesis is structured as follows: In chapter 2, other work related to KESO is reviewed. In chapter 3 and chapter 4, the architecture and some relevant runtime data structures of KESO are explained to establish the KESO-related knowledge required for the remainder of this work. The design and implementation of the OSEK abstraction layer is covered in chapter 5, and chapter 6 describes the heap implementation that was developed in this work. Finally, chapter 7 summarizes the results and outstanding issues of the developed KESO components, and gives a glimpse of the current and future work on the KESO system.

1.2 Typographic Conventions

Throughout this thesis, the following typographic conventions are used:

OSEK concepts and task states OSEK concepts and task states are written with capitals to differentiate from common language use. The affected terms describing OSEK concepts are Tasks, Resources, Events, Alarms and Counters. The existing OSEK task states are Ready, Suspended, Waiting and Running.

Code elements Code elements, that are part of the continuous text, are written in type-writer font. Function and method names are additionally appended brackets, names of classes are usually written with a capital and names of fields or variables are usually uncapitalized. Macros and constants are per convention written in all capital letters. Examples: `pushObject()` designates a function or method with the name pushObject, `Task` indicates the class Task, `domain_id` designates a variable of the name domain_id and `INVALID_DOMAIN` marks an identifier with the name INVALID_DOMAIN.

New terms Whenever a new term is introduced, it is written in italic font.

Names Names are written sans-serif font, e.g. DomainZero.

Chapter 2

Related Work

2.1 The AJACS Project

The AJACS (Applying Java to Automotive Control Systems) project [con02] 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.

2.2 The JX Java Operating System

The multi JVM architecture of the Java operating JX [GFWK02] posed the paradigm for the architecture of the KESO system. While the architecture of KESO is similar to the architecture of JX, both systems have not much in common in their implementation, which is mainly a consequence of the diversities of the application domains that both systems target, which are personal computers for the JX operating system and microcontrollers for the KESO system, and the concepts inherited by the underlying OSEK operating system for the KESO system, while JX uses its own microkernel.

2.3 Embedded JVMs

This section reviews some embedded JVMs, that are related to KESO in that they also have to face resource limitations of the target hardware.

2.3.1 NanoVM

NanoVM [Har06] is a small embedded JVM for the AVR architecture. It is limited to a small subset of the Java class library and the Java programming language. NanoVM interprets Java bytecode at runtime, but requires standard class files to be converted to a NanoVM specific format before uploading the classes to the target.

2.3.2 TinyVM

TinyVM [Sol00] is a Java-based firmware replacement for the Lego Mindstorm^{TM} RCX microcontroller with a footprint of about 10 kB. Similar to NanoVM, TinyVM does not provide a complete Java runtime environment, notably garbage collection and floating point support are missing in TinyVM.

2.3.3 The Squawk Virtual Machine

The Squawk VM [SCC$^+$06] is a small JVM written in Java that runs without an operating system. Squawk complies to the Connected Limited Device Configuration (CLDC) 1.1 Java Micro Edition (Java ME) configuration [Sun04]. Java bytecode is converted to a Squawk specific *Suite File* format, that incorporates space, execution and garbage collection simplification optimizations.

Squawk implements an isolation mechanism similar to that of Java Specification Request (JSR) 121 [Cza00, jsr06]. Squawk encapsulates applications into so-called *Isolates*, whereby each Isolate maintains an own copy of mutable data such as static class variables. An Isolate may contain multiple threads, therefore the Isolate concept shows some similarities to the domain concept used in KESO. Contrary to domains, all Isolates allocate new objects from the same heap, therefore Isolates are no separate units of memory allocation.

Squawk further differs from KESO in the thread and scheduling concept, and the non-preemptible system code including the garbage collector, that can have a negative impact on the interrupt handling latency.

2.4 Java and Real-Time

2.4.1 The Real-Time Specification for Java

The real-time specification for Java (RTSJ) [BBG^{+}00] is an extension to the Java language definition and the Java standard libraries, that adds support for real-time threads, while remaining backwards compatible with existing (non real-time) Java applications. Among the important changes are the necessity of a priority inheritance or a priority ceiling mechanism on Java monitors and an extended thread concept. Two types of real-time threads are introduced, that can be assigned priorities higher than the priority of the garbage collector. The system is then split in a real-time part and a non real-time part, where the real-time threads use memory areas that are not under the control of the garbage collector. The communication and synchronization between real-time and non real-time parts is heavily restricted to avoid the deferment of real-time threads by the garbage collector, that can occur by synchronizing a non real-time thread and a real-time thread in combination with priority ceiling or priority inheritance.

2.4.2 JamaicaVM

The commercial JVM JamaicaVM [Sie04] provides support for the RTSJ and contains a garbage collector, that is suitable for hard real-time constraints. The heap of JamaicaVM is divided in fixed-size blocks, that also represent the increment unit of the preemptible garbage collector. Garbage collection is scheduled upon allocation by the allocating thread. The amount of work done by the thread depends on the size of the allocated object, i.e. threads have to *pay* for the allocation by performing garbage collector work. The fragmentation problem is solved by composing objects of a linked list of fixed size blocks. These blocks do not need to be sequential on the heap, thus completely avoiding the need to compact memory.

Chapter 3

Architecture of the KESO System

3.1 Conceptional Architecture

The architecture of the KESO system is illustrated in figure 3.1. From a conceptional point of view, it is very similar to the architecture of the JX operating system. Contrary to JX, however, KESO is based on top of an OSEK operating system rather than its own microkernel. The environment of an OSEK operating system affects the KESO design in several aspects, differentiating it from the design of the JX operating system:

1. OSEK operating systems use the concept of Tasks as the basic schedulable unit. OSEK scheduling is based on priorities statically assigned to Tasks. Consequently, KESO also uses the notion of priority-assigned Tasks to represent threads of control rather than Java threads, whereby each Task is assigned to a KESO domain.

2. Allocation of the system resource CPU is controlled by the priority-based scheduler of the OSEK operating system. Memory, however, can be assigned on a per-domain base by configuring the size of each domain's heap.

Domains

Similar to JX, each domain appears as a self-contained JVM to the user application. Each domain contains a set of static class fields and a heap of its own, whereby each domain can choose from different heap implementations.

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.

In addition to the domains specified by the user application there is a special domain *DomainZero*. This domain is used to execute code, that does not belong to any of the

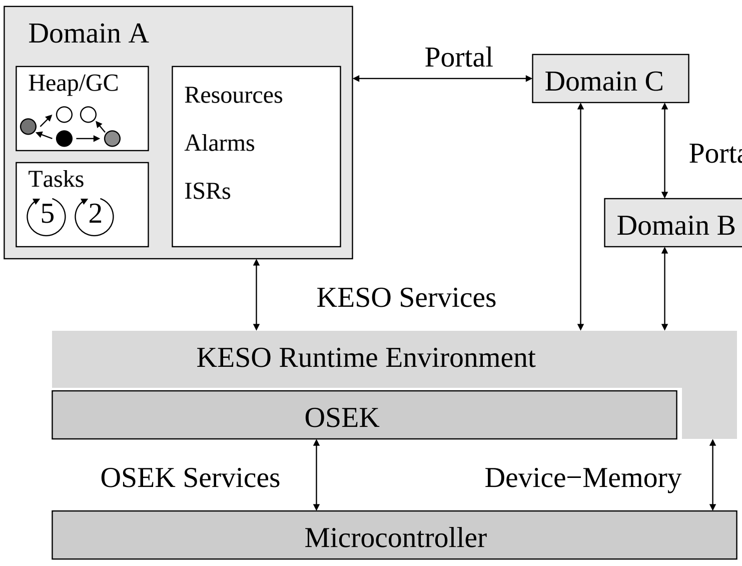


Figure 3.1: KESO Conceptional Architecture

regular domains, which can, for instance, be the case for interrupt service routines. Furthermore, it contains system objects that are accessible from all other domains, i.e. *globally visible* immortal system objects[1]. DomainZero does not contain any Tasks.

DomainZero also exists in JX, but the semantics is entirely different. It is planned, that DomainZero will be removed from KESO and replaced by a user configurable system domain, that is only optionally present in the system in case global code or objects are required by the user application.

Portals

Inter-domain communication is possible via portals in a manner similar to JX. A *service domain* can provide a *portal service*, that allows Tasks of other domains to execute code in the service domain. A portal service consists of a well-defined interface that offers *service methods* to other domains. Tasks of other domains may invoke methods of the portal. The execution of a service method takes place in the environment of the service domain.

The implementation of KESO portals is described as part of the Task Management in section 5.3.1.

[1]A reference to a global system object can be present in multiple domains. However, these objects are not allocated from a domain heap and not subject to garbage collection, therefore this does not pose any problems. System objects are discussed more detailed in chapter 5.

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. These 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 abstraction layer (chapter 5).

Further hardware accesses to memory mapped device registers are 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.

Device Drivers are not available yet, but a CAN (Controller Area Network) driver for the Tricore architecture is currently in development. Device drivers will allow access to hardware devices on the Java level without the need to program those devices. Applications can then use the higher level interface provided by a device driver, which greatly increases the portability of user applications.

3.2 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 microcontroller 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 a few advantages:

- Directly compiling to native code would require a compiler back-end for the KESO builder for each supported target platform. However, a standard C compiler is available for almost all of the target platforms.

- The available C compilers allow to create highly optimized code at the function level.
- C source code is easier to read than native code, which eases debugging.
- A separate compiler back-end for each target platform increases the complexity of the KESO builder and increases the probability of software bugs.

Components and Processes

The generation process of a KESO system is illustrated in figure 3.2. The components that participate in the process are of three kinds:

- Components provided by the developers of the user application. These are comprised by the Java source code of the user application and the KESO system configuration file.
- Standard components, that are comprised by a Java compiler, that compiles Java source code to Java bytecode, the OSEK system generator (OSEK SG), that creates the source code of the OSEK kernel from the OSEK OIL [OSE04] (OSEK implementation language) configuration file, and a C compiler and linker, that is used to compile the C sources of the user application and the OSEK kernel, and links the resulting objects files to the KESO binary image.
- KESO components, comprised by the KESO class library, that provides the KESO services to Java applications, the *KESO autoclass generator*, which automatically generates parts of the KESO class library from the system configuration file, e.g. for providing OSEK identifiers on the Java level, and the KESO builder, that compiles the Java bytecode of the user application to C source code and generates the OSEK OIL configuration file from the KESO system configuration file.

The generated C code does not only contain the compiled class files, but also the KESO runtime data structures (chapter 4), that include data structures such as the *class store*, that contains type information required at runtime for tasks such as checking a cast, and the virtual method table, that is required to resolve virtual methods calls at runtime. 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.

Optimizations with Respect to Code Size

KESO is a static system, that does not allow the dynamic loading of classes at runtime, which opens some optimization potential for reducing the size of the generated system.

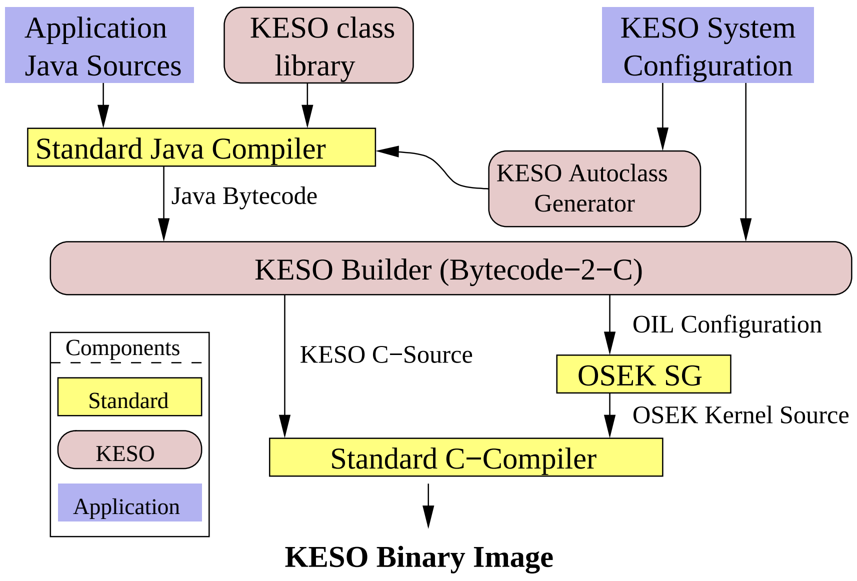
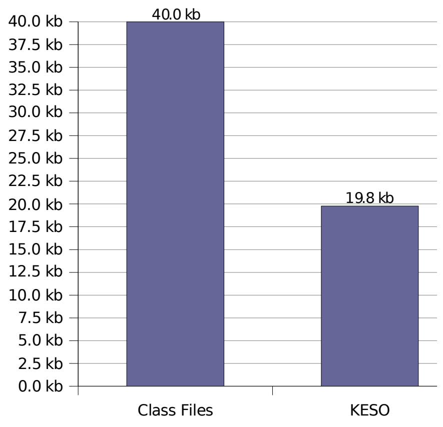


Figure 3.2: KESO System Generation Process

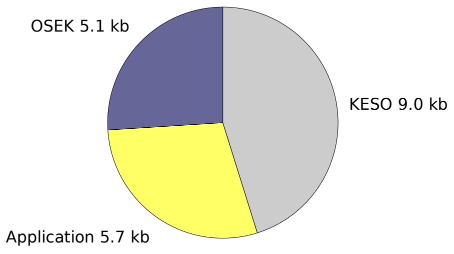
The KESO compiler performs a reachability analysis on the bytecode and eliminates classes, methods and fields, that are not accessed by the user application. This reduces both, the code size of the generated system and the size of the KESO data structures. Additionally, only the parts of the KESO abstraction layer, that are actually used by the application, are added to the generated code, e.g. if an application does not make use of OSEK Resources, the data structures and code for the Resource-related services are not added to the generated system.

Figure 3.3 shows a code size comparison for a test application, that was used to test the garbage collector discussed in chapter 6. The test application processes an infinite loop and maintains a FIFO of fixed size. In each loop cycle, a new element is be allocated and added to the FIFO, and another element is removed. Additionally, the application outputs a string, that is constructed using numerous `StringBuffer` objects from several integers and constant strings. At the end of a cycle, the Task waits for an OSEK Event, triggered by a cyclic OSEK Alarm, and continue with the execution once the Event has been set. In the meantime, the garbage collector reclaims the memory of unreachable objects. OSEK Resources are also used to synchronize the test application with the garbage collector (chapter 6).

Figure 3.3(a) opposes the size of the class files actually used by the test application and the size of the resulting KESO image, containing all of the above optimizations. If one chose to actually interpret or just-in-time compile bytecode on the microcontroller, the size of the deployed JVM would additionally need to be added to the size of the class



(a) Comparison of total system sizes



(b) Components of the KESO system

Figure 3.3: Code size comparison for a garbage collector test application. The test application makes use of OSEK Events, Alarms and Resources and contains the garbage collector described in chapter 6.

files, but the resulting system would still neither contain an OSEK OS nor the KESO runtime layer. The figure shows, that the resulting KESO system, including the OSEK OS and the KESO runtime layer, is only half the size of the unoptimized class files. The application component of the KESO system, that corresponds to the translated and optimized class files, is even only 14.25% the size of the original class files.

Figure 3.3(b) shows the composition of the KESO system. The bulk of the system is posed by the KESO layer, that consists almost half (4 kB) of the code of the garbage collector, that will remain constant for larger applications. The other major part of the KESO layer is posed by the runtime data structures. These data structures grow with the number of classes and methods used in the system. The size of the OSEK system also depends on the needs of the user applications, however, the test application already makes use of OSEK Resources, Alarms and Events. Thereby, even for larger applications, only a decent increase of the OSEK component has to be expected. The most variable component is posed by the user application. With larger applications, the application component will increase in size most, and the fraction of the other two components in the size of the entire system will shrink.

Chapter 4

KESO Runtime Data Structures

This chapter describes some of the KESO runtime data structures. Only the data structures relevant to this work are discussed, i.e. others, such as the virtual table, that is used to resolve virtual method calls at runtime, are omitted, because they are not needed in the following.

4.1 Class Store

The *class store* is an array of *class descriptor* structures, that contain runtime information on each class. Each class descriptor structure contains the following elements[1]:

- Type range (*unsigned 16-bit*): This field is required to check type information of a class and is used in operations such as `checkcast` or `instanceof`. It is not relevant in this work and not further discussed here.
- Size (*unsigned 16-bit*): The size in bytes of an instance of the class.
- Interfaces (*unsigned 8-bit*): Information about the interfaces implemented by the class. This is also not relevant in this work.
- Reference offset (*unsigned 8-bit*): The number of reference fields that an instance of the class contains. For the bidirectional object layout (see section 4.3.3), this can be used to determine the offset between the begin of an instance and the object header.
- Extensions (*unsigned 16-bit*): Reserved for extensions and not used at the moment. This ensures the 32-bit alignment of the class descriptor structures.

[1]The shown contents apply for the Tricore architecture. For other architectures, such as the AVR architecture, that is currently in development, some elements may be missing or differ in size from the elements shown here.

Each class is assigned a *class identifier*, that can be used as an index into the class store array to access the class descriptor belonging to the class. The class identifier is stored in the object header of an instance.

4.2 Domain Descriptor Table and Domain Identifiers

The *domain descriptor table* contains *domain descriptor* structures with runtime information on each domain.

Currently, the domain descriptor contains only data describing the domain's heap implementation. The actual fields of the descriptor therefore depend on the heap implementation used by the domain. The only common field present in every domain descriptor is the `allocator` field, that contains the address of the heap specific allocator function. For the *IdleRoundRobin (IRR)* heap implementation (see chapter 6), the domain descriptor furthermore contains the following fields:

- Freeslots: The number of slots available for allocation on an IRR heap.
- Slotsize: The size of a slot on an IRR heap in bytes.
- Heapsize: The total size of the IRR heap in bytes.
- Freemem: Pointer to the first element of the free memory list of an IRR heap.
- NewSlotsAllocated: Recorded number of slots allocated for new objects since the last run of the garbage collector in a domain using the IRR heap.
- Heaptop: Address of the lower boundary of the IRR heap. Can be used in conjunction with the Heapsize to determine the address range of the heap.
- Colorbit: Contains the current value of the colorbit, that represents the color gray or black. The value is either 2 or 0, and is toggled after each garbage collector cycle.

The *domain identifier* assigned to each domain can be used as an index into this array to access the domain descriptor of the respective domain. The global indicator *current domain* contains the domain identifier of the domain, that the currently executed code belongs to, which is (in most cases) the domain of the currently running Task.

The special domain identifier `INVALID_DOMAIN` is used in some places where no valid domain identifier is applicable in the current state, e.g. the garbage collector uses it to signal, that it is currently not active in any domain.

The domain identifier is also needed in various other places, e.g. to access static fields of a class, as each domain maintains a set of static fields of its own.

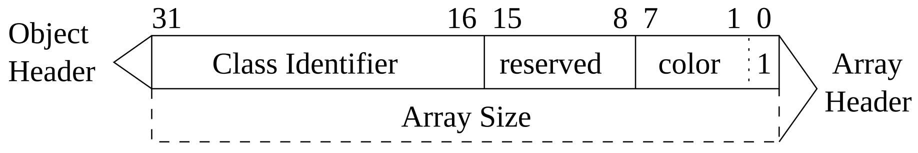


Figure 4.1: Layout of the object header and array object header

4.3 Object Layout

The object layout describes the internal runtime representation of an allocated object in a Java virtual machine. It commonly consists of some meta data, the so-called *object header*, and the actual fields of the object.

To provide an efficient access to object fields, the offset of an object field from the object header should be computable at compile time rather than having to be computed at runtime, regardless of inheritance. This can be easily accomplished for Java, because Java only supports single inheritance and does not allow the declaration of fields in interfaces.

The object layout is a crucial aspect for the performance of tracing garbage collectors, such as the one implemented in this thesis (chapter 6). Tracing garbage collectors determine the set of reachable (*living*) objects by recursively scanning the inner reference fields of discovered objects, starting from a root set. The object layout affects the performance of this scanning phase, depending on how much effort is necessary to find all reference fields in an object.

4.3.1 Object Header

The object header contains meta data on the referenced object. An object reference always points to the object header. The 32-bit object header currently used for objects in the KESO system is shown in figure 4.1.

The header contains the unsigned 16-bit class identifier of the object, that can be used as an index into the class store and represents the type of the object. The least significant byte of the object header, `color`, is reserved for information specific to a heap implementation. Currently, only the IRR heap implementation (chapter 6) uses this field. The remaining byte of the object header is currently not used and reserved for future development.

Java specifies some special array types. There is an array class for each primitive Java data type and for object references. These array types have an extended object header, that additionally contains the number of elements of the array. The header of an array class will also be called the *array header*.

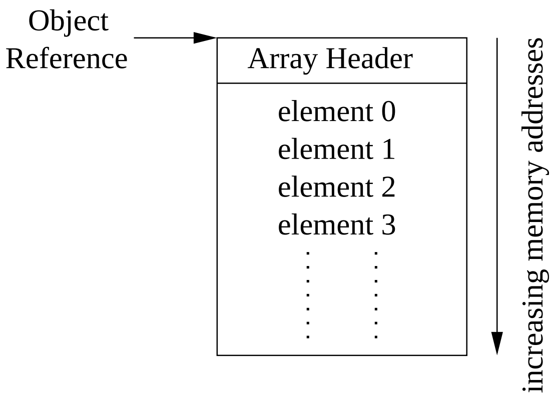


Figure 4.2: Layout of Array Types

4.3.2 Array Classes

Array classes not only differ in their header from non-array classes, they also use a different object layout, which is shown in figure 4.2. The elements are simply put behind the array header, starting with the element at index 0 to the higher indices.

To scan an array of object references, the garbage collector only needs a simple loop, where the length of the array can be used as loop counter and array index.

4.3.3 Non-Array Classes

Traditional Object Layout

The traditional approach for the object layout is illustrated in figure 4.3, and was initially also used in the KESO system. The example shows the object layout for a class C, that is a subclass of class B, which in turn is a subclass of class A. In the traditional object layout, non-reference fields and reference fields are grouped together for each class in the class hierarchy. The order in which the fields of each class are put after the object header is starting with the most general class (A in the example) to the most specialized class (C in the example).

To scan the reference fields of an object, for each class in the type hierarchy of the object, the number of reference fields of the class and the offset of the reference fields (or, alternatively, the number of bytes occupied by non-reference fields) need to be stored. This can be done in two ways:

First, for each class, the offset and number of reference fields of the class and each of its superclasses can be stored in the class descriptor of the class, which would increase

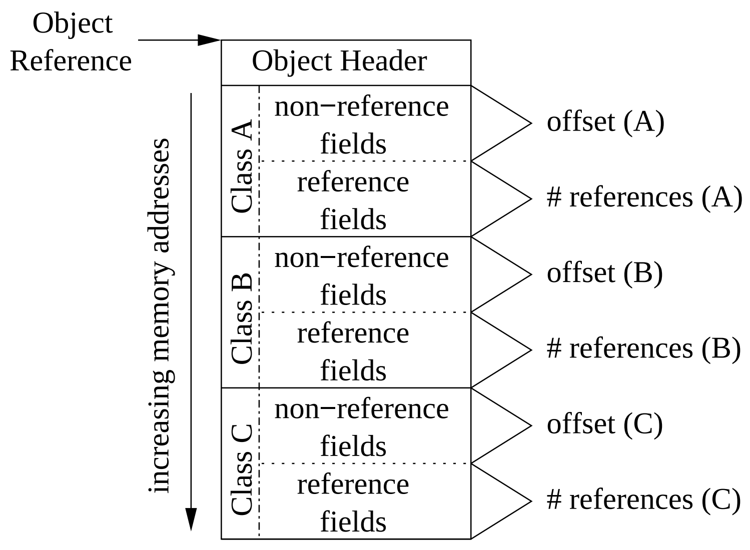


Figure 4.3: Traditional Object Layout (Class C extends B extends A)

the memory required by the class store, depending on the depth of the type hierarchy[2].

Alternatively, the garbage collector would have to determine each super class at runtime and read offset and number of reference fields from the class descriptors of all superclasses, which would significantly complicate the scan process.

Generally, to scan an object, two nested loops would be required, the outer one for climbing the type hierarchy and the inner loop for reading offset and number of references and scanning the references for each class on the type hierarchy.

Bidirectional Object Layout

To facilitate the scanning phase of the garbage collection, the traditional object layout was replaced by a bidirectional object layout similar to the one proposed in the SableVM project [Gag02, GH01]. The layout used in KESO is shown in figure 4.4 for the same class that the traditional layout was illustrated with. Contrary to the traditional layout, an object now grows in both directions from the object header, where reference fields are put before and non-reference fields are put after the object header. The ordering remains the same, i.e. starting with the most generic class (A) to the most specialized class (C). For an object with reference fields, the object header is now not placed in the beginning of the object, but behind the reference fields, thus the object reference and the starting address of the object usually differ in the bidirectional object layout. For each

[2]Because the class descriptors in the KESO class store need to be of equal size, the array containing offset and number of reference fields for each class would either have to be stored outside of the class store, or the maximum depth of the type hierarchy would have to be limited.

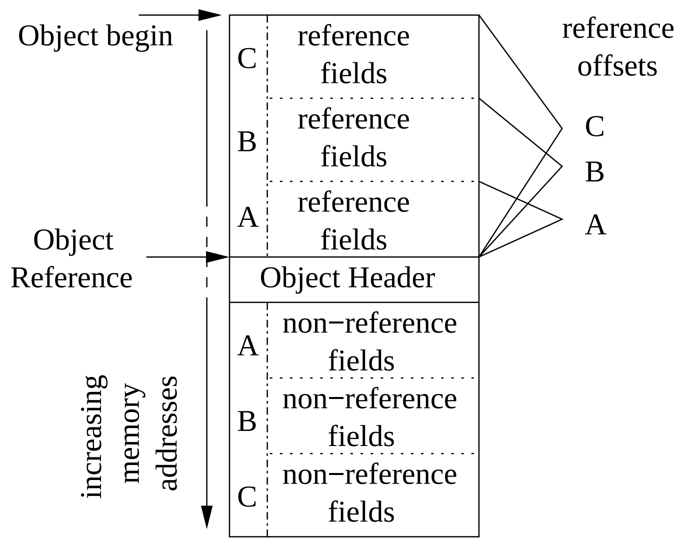


Figure 4.4: Bidirectional Object Layout (Class C extends B extends A)

class, a *reference offset* can be identified, that specifies the number of reference fields of a class (including inherited reference fields). Knowledge of the reference offset is also sufficient to compute the difference between the beginning of an object and the object header. Thus, for each class, only the reference offset needs to be stored in the class store.

The offset of a field from the object header is still constant and computable at compile time, so field accesses remain efficient with the bidirectional layout.

When scanning an object, the garbage collector now only needs to read the reference offset from the class descriptor and read the references ahead of the object header. Only a regular loop is required where the reference offset can directly be used for the loop counter. The bidirectional offset thus saves both, computing time and memory accesses for the scanning phase of the garbage collector, plus it reduces the memory required for the class descriptor of each class.

Chapter 5

OSEK Abstraction Layer

Since KESO is built on top of an OSEK [OSE05] operating system, the OSEK concepts need to be made available to application developers on the Java level. A Java class library and Java abstractions to OSEK system objects have been created for this task.

OSEK systems are configured with a configuration file described in the OSEK implementation language (OIL) [OSE04]. Most of the parameters and defined system objects of the generated OSEK system are also required in the build process of the KESO system.

To eliminate the need for specifying this information in two separate configurations, the OSEK system configuration parameters have been integrated with the KESO configuration parameters in a single KESO system configuration file. The OIL configuration file for the underlying OSEK system is then automatically generated by the KESO builder from the KESO configuration file. This provides the application developer with a single point for configuring the entire system.

Several goals can be identified for the OSEK abstraction layer:

- Create an object-oriented view on OSEK concepts such as Tasks and Resources
- While the object-oriented abstraction is always desirable from the perspective of software engineering, it is not always a viable choice. Because resources are usually very limited on the KESO target platforms, a tradeoff between abstraction and overhead has to be made. In some cases, e.g. OSEK Events, the advantages of an object-oriented abstraction are outweighed by the overhead going along with the abstraction and different solutions have to be considered.
- As a major goal of the whole KESO system, strong isolation of domains is also a goal of the OSEK abstraction layer. Accesses to OSEK system services need to be restricted to provide this isolation. As an example, it must be possible to restrict access to a certain OSEK Resource to a domain, if this is eligible for the user application.

- The Java class library providing access to the OSEK system services should be modeled closely to the original OSEK interface. This helps providing a familiar programming interface to OSEK application developers, easing the transition to application development for the KESO system.

Knowledge of the OSEK OS specification [OSE05] and the OSEK implementation language specification [OSE04] is presumed throughout this chapter.

This chapter is structured as follows: Section 5.1 gives an overview on the general design decisions made for the OSEK abstraction layer. The object abstractions were similarly implemented for the different OSEK concepts and the general implementation is described in section 5.2. Sections 5.3–5.7 cover implementation details on the various OSEK topics. Finally, section 5.8 discusses some problems, that arise by the use of portals, and the used solutions.

5.1 Conceptual Design

For each class of OSEK system services, as classified by the OSEK specification, a service class has been created, that provides a static method for each OSEK system service function, plus some additional functions, such as a *name service*, that enforces access restrictions to the system services on the *Java language level*.

5.1.1 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 makes heavy use of 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 inserted at the call-side, removing the invocation of the magic method, or in the body of the magic method, leaving the call of 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.

5.1.2 Access Restrictions to OSEK Services

The OSEK abstraction layer must ensure, that the isolation of domains is not weakened by the inappropriate use of the OSEK system services. To ensure this, a mechanism needs to be provided that allows to restrict the access to certain system services.

For instance, a Resource might be created to synchronize two Tasks of the same domain. In such cases, it is desirable, that Tasks of other domains are not able to use the `GetResource()` service on that Resource. On the other hand, a Resource could also be used to synchronize Tasks of two different domains, that, for instance, both access a periphery device. Therefore, the OSEK abstraction layer supports both *global* and *domain local* object abstractions.

The services classes, that an access restriction was found to be useful to, are

1. Resource Service: Provide domain local Resources, that are only visible inside of a domain and can be used for synchronization of Tasks within the same domain (see example above).

2. Task Management: The activation of other Tasks through the use of the `ChainTask()` and the `ActivateTask()` services is restricted to Tasks within the same domain.

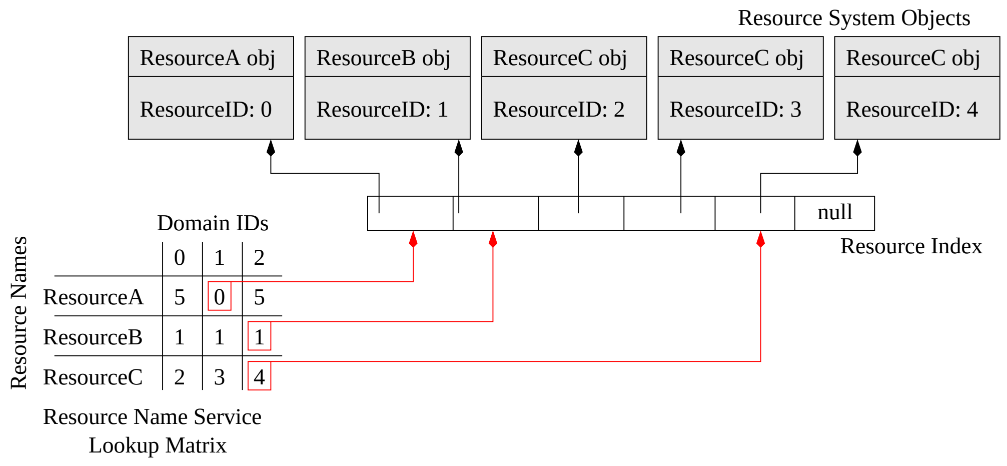
3. Alarm Service: Services of this class allow the modification, activation and cancellation of Alarms. Alarms are therefore assigned to a domain and the use of the system services is restricted to Alarms within the same domain as the calling Task.

Access restrictions are enforced on the language level. Object abstractions have been created in the classes `Resource`, `Alarm` and `Task`. Instances of these classes will be called *system objects* (SO). The SOs are automatically created by the KESO builder using the information from the KESO configuration file. Affected system services do not further check the domain of the calling Task and the domain of the system object (that may be another Task, a Resource or an Alarm), but require a reference to the respective system object as a parameter where OSEK uses plain identifiers on the C level. A name service is provided, that allows Tasks to acquire references to KESO system objects, by referencing the objects by their name as configured in the system configuration file[1]. The name service does only return references of local SOs within the same domain as the caller and references to global SOs to the caller, thereby restricting the use of system services by limiting the access to SOs.

5.1.3 Provision of OSEK Constants and Identifiers

Where no access restrictions are required and object abstractions are not required due to other reasons, the OSEK Identifiers are directly made available on the Java language level. This affects Counters, Events and Application-Modes. For each of these, a Java

[1]These are the same identifiers, that would be used when programming C code for an OSEK application.



(a) Name service example for Resources

DOMAIN NAME	ID
DomainA	0
DomainB	1
DomainZero	2

(b) Domains

RESOURCE NAME	ID	SCOPE
ResourceA	0	DomainB
ResourceB	1	global
ResourceC$_1$	2	DomainA
ResourceC$_2$	3	DomainB
ResourceC$_3$	4	global
Invalid Resource	5	-

(c) Resources

Figure 5.1: Name Service Implementation

class is automatically generated by the KESO autoclass generator, that determines the identifiers and the assigned values from the KESO configuration file. These Java classes contain a final static field for each identifier with the assigned value, that can be used in the Java applications similar to the use of OSEK identifiers in C applications.

5.2 Object Abstractions and Name Service

The object abstractions and the name service have been similarly implemented for Tasks, Resources and Alarms. Differences are covered in the sections dedicated to each OSEK topic below.

An example for the name service data structures and object abstractions for Resources is shown in figure 5.1. In the following, only Resources are mentioned, but everything applies for Tasks and Alarms, too.

5.2.1 General Description

For each Resource, a Resource SO is created at compile time by the KESO builder. The SO contains a private field with the OSEK identifier of the Resource represented by the SO. When the `GetResource()` service is called on the Java level with the SO, the builder compiles it to a call to the OSEK `GetResource()` service on the C level and uses the OSEK identifier contained in the SO.

The references to the SOs are managed in an array, the *Resource index*. The OSEK identifier of a Resource can be used as an index into this array to acquire the reference of the associated SO. Additionally, a special value `INVALID_RESOURCE` is introduced, that is assigned the `null` reference[2].

For the name service, a matrix mapping a Resource name and a domain identifier to an OSEK identifier is created, the *Resource name service lookup matrix*. The lookup matrix contains a row for each Resource name and a column for each domain, including DomainZero. 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.

5.2.2 Explanation of the Example

Figure 5.1 shows an example scenario for a Resource name service. The available domains and the associated domain identifiers are shown in figure 5.1(b), including the system domain DomainZero. The configured Resources with their OSEK identifiers and scope are shown in figure 5.1(c). The scope specifies the domain a Resource is assigned to, or global if the Resource is accessible in all domains.

ResourceA shows the case of a domain local Resource, which is only accessible from DomainB. The row of ResourceA in the lookup matrix contains the OSEK identifier of ResourceA in the column of DomainB. In the columns of the other two domains, the special identifier `INVALID_RESOURCE` is stored. A lookup for ResourceA therefore resolves to a reference to the appropriate SO if invoked in DomainB. In the other two domains, the lookup resolves to the `null` reference, which represents the `INVALID_RESOURCE`.

ResourceB illustrates the case of a global Resource. The row of ResourceB in the lookup matrix contains the OSEK identifier of the Resource for all domains. Therefore, a lookup of ResourceB resolves to a reference to the valid SO in every domain.

[2]OSEK already specifies an identifier `INVALID_TASK`. For Tasks, this identifier is used, but new identifiers have been created for Resources and Alarms. In either case, the invalid objects are always represented by the `null` reference on the Java level.

The name ResourceC is used multiple times, in each DomainA and DomainB to specify a domain local Resource and additionally for a global Resource. Therefore, three different OSEK Resources are created, and for each of them a SO is statically allocated. In the row of ResourceC in the lookup matrix, the column of DomainZero contains the identifier of the globally configured ResourceC. For the other two domains, the domain local Resources shadow the global Resource of the same name, and the identifiers of the domain local Resources are stored in the respective columns. A lookup for ResourceC resolves to the domain local Resources in DomainA and DomainB, and to the globally configured Resource in DomainZero.

5.2.3 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

```
getResourceByName("ResourceC")
```

to the following (assumed the values from the example):

```
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 (as ResourceB in the example), the method call

```
getResourceByName("ResourceB")
```

is resolved to a direct lookup in the resource index:

```
resource_index[1]
```

As an alternate solution, the lookup matrix could directly contain the references, which would save the additional lookup in the resource index at the cost of a larger lookup matrix.

5.2.4 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 be created and these SOs could therefore be abused to transport references across a domain boundary. 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.

5.3 Task Management

OSEK allows to query the identifier of the current Task. Similar to the current domain, a reference to the SO of the current Task is stored in a global field `current_task`.

Besides the OSEK identifier of a Task, the associated SO additionally contains the domain identifier of the domain that the Task was configured in (field `domain_id`) and the identifier of the domain that the Task is currently running in, the *effective domain* (field `e_domain_id`). The configured domain and the effective domain may differ in case the Task makes use of portal services, see section 5.3.1.

Upon a Task switch, the current Task and the current domain need to be set for the scheduled Task. They also need to be updated for interrupt service routines and Alarm callback functions, but these two are described in section 5.4 and section 5.7. Upon a Task switch, immediately before scheduling the new Task, the `PreTaskHook()` is invoked by the OSEK system. KESO uses this hook to perform the update operations for the current Task and the current domain. First, the OSEK ID of the current Task is acquired using the `GetTaskID()` OSEK service. The Task ID is then used to lookup a reference to the SO of the Task in the Task index. This reference is set as the current Task, if the effective domain equals the configured domain of the Task, or to `null`, which represents the `INVALID_TASK` on the Java level. The effective domain of the current Task is set as the current domain. This is sufficient to setup the proper runtime environment after each rescheduling.

5.3.1 Implementation of Portals

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

From a conceptional perspective, one would then assume, that the code executed in the other domain is executed by a *service Task* belonging to that domain. This approach is, however, difficult to implement on an OSEK system. The service Task would have to be configured with a fixed priority. Switching from the Task invoking the portal to the service Task could only be handled by close-by or equal priorities combined with a blocking of the invoking Task, which is only possible using Events. However, the same service can be used by multiple Tasks, and for each Task the service should be executed with the priority of the invoking Task. This could only be solved by adding a dedicated service Task with an appropriate priority for each Task that could *possibly* use the service, along with Events for each pairing. This is not a feasible solution.

Instead, no service Tasks are created at all and the portal call is actually handled by the invoking OSEK Task, which is—for the duration of the portal call—migrated to the service domain. Therefore, for each Task, the *configured domain* and the *effective domain* can be distinguished, where the former represents the domain that the Task was assigned to in the configuration, and the latter represents the domain that the Task is currently running in. In case of a portal call, the effective domain is set to the domain identifier of the service domain. This effectively changes the runtime environment of the Task to the service domain, i.e. the Tasks have access to the SOs of the service domain through the name service, to the static variables of the service domain, and allocate objects from the heap of the service domain.

While a Task is executing code in a service domain, the `GetTaskID()` service always returns `INVALID_TASK`. This is necessary, because SOs must not cross a domain boundary, not even in copied form (see section 5.8).

Figure 5.2 shows an example for a Task (TaskA) that uses portal services of another domain. The configured domain of TaskA is DomainA with the identifier 0. The Task first uses a portal of DomainB, and while executing code in DomainB invokes a portal of its configured domain. The changes of the effective domain identifier in the SO of the Task as well as the respective values of the current Task and the current domain are shown in the figure. Notably, when a Task migrates back to its configured domain through another portal call as the example Task, the current Task contains a valid reference to the SO of the Task, even though the presence in the configured domain is through the use of a portal.

An important aspect in this context is the layout of the Task stacks used by KESO, which is also illustrated in figure 5.2. The stack of a Task that migrates among domains

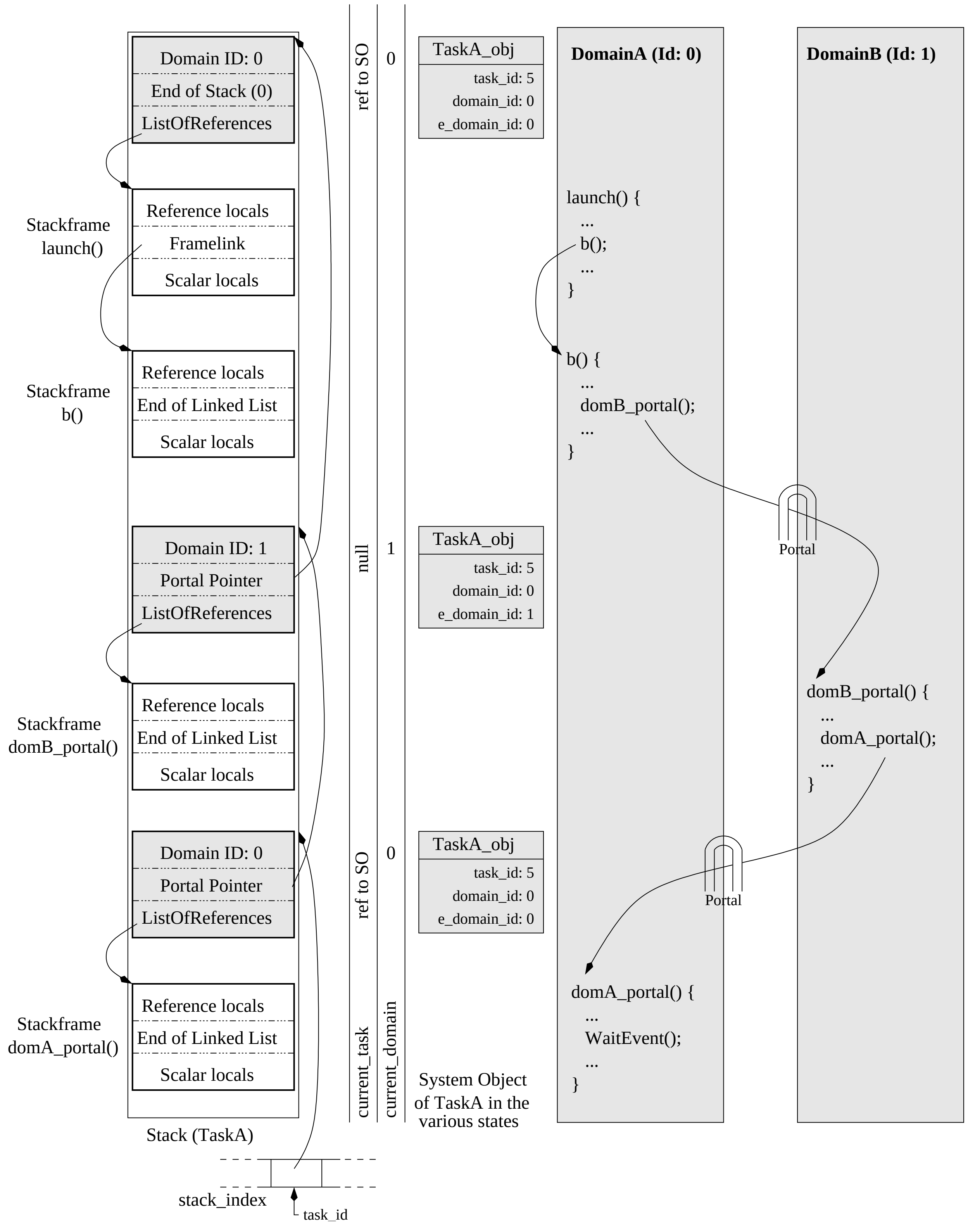


Figure 5.2: Portal implementation and Layout of a KESO Task Stack

using portal calls can be divided into several *stack chunks*, each of which can be assigned a domain. A stack chunk represents a part of the stack of a Task that belongs to methods executed in a specific domain. Each object reference present on such a chunk either refers to an object on the heap of the respective domain or to a SO. This is important to the garbage collector, that needs to scan all references of a domain (chapter 6), and therefore must find the relevant chunks of the stack associated with the scanned domain.

Each stack chunk is described by a structure that contains the domain identifier of the chunk's domain, a *portal pointer* to the next chunk in case the Task migrated to another domain through a portal, or `NULL` for the last chunk in the chain, and a pointer to the head of a *linked list of references (LLREF)*. The *stack index* contains the address of the structure describing the last[3] chunk on the stack for each Task, or `NULL` if the stack is empty. Thus, one can traverse the different chunks on the stack by starting with the address in the stack index, and successively following the portal pointers of the chunk describing structures until a `NULL` value designates the first chunk.

Within a chunk, a garbage collector needs to find all object references stored in local variables of the stack frames of each method. To render this possible, a linked list of references contains all the references stored in the stack frames of a chunk. Each stack frame contains one element of the LLREF, that is structured as follows: first, all local reference variables of the method are combined in an array. Immediately after the last reference, a pointer to the reference array of the next stack frame is stored, or the special value `KESO_EOLL` that designates the last stack frame of the chunk. This pointer is labeled *framelink* in the figure. `KESO_EOLL` can be any value, that is not a valid value for a reference field, and is currently defined as `-1`. The remaining non-reference local variables of the method are stored after the framelink pointer and are not of interest to a garbage collector. The describing structure of a chunk contains the pointer to the reference array of the first stack frame of the chunk, or `NULL` if there are no stack frames on the chunk.

The maintenance of the LLREF adds some overhead to each method invocation. The garbage collector described in chapter 6 only runs if no other Tasks in the system are in the ready state. This means, whenever the garbage collector Task is running, the other Tasks are either in the Suspended state, in which the stack is empty, or the Waiting state. The LLREF therefore only needs to be created for *blocking methods*. A method is considered a blocking method, if the method itself or—recursively—any of the methods invoked by this method use the `WaitEvent()` system service. This analysis is performed by the builder and each method is assigned an attribute that specifies if the method is a blocking method. The additional code required to link and unlink stack frames of subsequently invoked methods is only generated for blocking meth-

[3]It is easier to implement to always link to the last chunk of the stack rather than the first one from the stack index. This way, when adding a new chunk to the stack, the address of the previous chunk can always be determined from the stack index.

ods. If, for instance, a blocking method invokes a non-blocking method, the LLREF is built to the level of the blocking method, but not further on the code path of the non-blocking method. This analysis is performed across portal calls, i.e. a new chunk with the corresponding describing structure is only created for portal calls that use the `WaitEvent()` service on any of the possible code paths.

5.3.2 Task-Related System Services

A class `TaskService` provides the Task-related system services, that allow the activation of other Tasks, the termination of the current Task and the explicit triggering of a reschedule for non-preemptible Tasks. Furthermore, the current Task and the state of a Task may be queried.

All services, that require a Task identifier to be passed on the C level, require a Task SO on the Java level. Therefore, these services are restricted to Tasks within the same domain[4]. The following summarizes in short how the different services were implemented:

- `GetTaskID()`: A call to this service is replaced at the call-side with the global current Task field.
- `TerminateTask()` and `Schedule()`: These services are replaced at the call-side with calls to the respective OSEK services, without any additional overhead. For `TerminateTask()`, additional code is inserted to clear the stack by storing `NULL` in the appropriate field of the stack index.
- `ActivateTask()` and `ChainTask()`: Both of these services require a single Task SO as parameter. In order to call the respective OSEK services, the OSEK identifier has to be read from the Task object, and a `null` reference check has to be added. The necessary wrapper code is inserted in the body of the magic method (see figure 5.3)
- `GetTaskState()`: This service allows to query the state of a Task and expects a Task SO as parameter. The method body is replaced similar as above, however, the state of the queried Task is returned as an `int` to the application. The various values are defined as constant values of the `Task` class, e.g. the `int` value associated with the Ready state can be accessed via the static field `Task.READY`.

All of the above services, except `GetTaskState()` and `GetTaskID()`, pass the state of the called OSEK service on. The different state values are defined as constant

[4]A Task can nevertheless use the Task services on a Task in a different domain through the use of a portal, however, the application developer has to explicitly enable it in the design of the user application by providing a portal service.

```
int activateTask(TaskClass_t *taskSO) {
    if( taskSO == NULL) {
        throw_NullPointerException();
    }

    return ActivateTask(taskSO->task_identifier);
}
```

Figure 5.3: `ActivateTask()` Code Example. The above code fraction shows the code generated for the magic method `TaskService.activateTask()`. The reference to the Task SO is checked for `null`. Afterwards, the OSEK identifier of the Task is read from the SO and passed to the OSEK system service. The return state of the OSEK service is passed through to the Java application. Similar code is created for all KESO system service wrapper functions that expect a SO as parameter.

values of the `TaskService` class with the same name as the corresponding OSEK macros, e.g. `TaskService.E_OK` represents the state returned upon a successful service invocation.

5.3.3 Wrapper Functions

OSEK specifies the `TASK` macro to define the initial function of a Task. In KESO, the OSEK standard parameters for the Task configuration are extended by the parameters `MainClass` and `MainMethod`, that specify the initial class and method of a Task. The initial class must be a subclass of the system class `Task`. The builder creates an instance of that class as SO for the Task, and the initial method of the Task, in the remainder of this thesis also referred to as the *launch()* method of a Task, is passed a reference to this object as the `this` reference. The `launch()` method should not return anything and must not expect any parameters.

The constructors of the Task objects are called from the `StartupHook()` after initializing the KESO runtime data structures. The application developer must obey the restricted set of system services that may be used in the `StartupHook()` as specified in the OSEK specification [OSE05]. In particular, the `GetTaskID()` service must not be used from the `StartupHook()`.

The builder creates a wrapper method using the `TASK` macro for each Task, that invokes the user specified `launch()` method. Additionally, the builder adds code to initialize the describing structure for the first chunk on the stack and to store the address of that structure in the stack index, if the `launch()` method is a blocking method.

5.3.4 Issues of the Object Abstraction

Since the application developer has to create subclasses of the `Task` class, there is no Java language construct to prohibit the arbitrary creation of instances of that subclass. These objects can, however, be used with the system services, which is not desired. To solve this problem, the constructor of the `Task` class initializes the internal identifier with the OSEK `INVALID_TASK` identifier. The Task SOs created legally by the builder have the correct task identifier inserted in the `StartupHook()` after invoking the constructors.

Because the `INVALID_TASK` identifier is known to the OSEK OS, invoking OSEK services with this identifier does not pose any problems, and the services will fail with an appropriate return state.

5.4 Interrupt Handling

OSEK distinguishes two different types of interrupt service routines (ISR): category 1 interrupt service routines (ISR1) bypass the OSEK scheduler and are directly executed on the stack of the currently running Task. ISR1 must not use any system services, with the exception of the interrupt-related system services. Category 2 interrupt service routines (ISR2) in contrast are scheduled by the OSEK scheduler and provided with an ISR frame on the stack. Upon leaving an ISR2 rescheduling takes place.

ISRs are defined in the OIL configuration file. The OSEK OS takes charge of registering the handlers in the interrupt vector table and enables the interrupts. OSEK defines macros `ISR` and `ISR1` for ISR2 and ISR1 to define the ISRs in C code.

For KESO, ISRs are configured with the same options as they are configured in an OIL configuration. Additionally, for each ISR a `HandlerClass` and `HandlerMethod` needs to be defined, that name class and signature of the Java method that ought to be installed as the service routine. An ISR handler method must be a static method with the following signature:

```
public static void handlerMethod();
```

An ISR is either assigned a domain or defined globally to specify the execution environment of the ISR. An ISR, that is assigned a domain, runs within that domain when executed, i.e. it has access to the system objects and static fields of that domain. Globally defined ISRs run in DomainZero, and are only able to access global system objects.

To switch to the proper execution environment upon entering an ISR, a wrapper routine is added to each ISR that uses the appropriate OSEK macro and then invokes the user specified handler method (figure 5.4). For ISR1, a backup of the current domain is made. ISR2 do not require the restoration of the current domain or the current Task, because a rescheduling takes place when the ISR2 is left and the `PreTaskHook()` is

```
ISR(isr_asc0_rx) {
  unsigned char db = current_domain; /* ISR1 only */
  current_domain = 0;      /* 0 == Domain of ISR */

  if ( current_domain != 0 ) {     /* ISR2 only */
    current_task = NULL;           /* ISR2 only */
  }
  rxIRQHandler();      /* user specified handler */

  current_domain = db;             /* ISR1 only */
}
```

Figure 5.4: ISR wrapper function. Source lines marked with ISR1 or ISR2 only apply to wrapper routines for category 1 or respectively category 2 ISRs. `isr_asc0_rx` is the OSEK identifier of the ISR as configured in the system configuration file, `rxIRQHandler()` is the handler method specified by the application developers.

invoked, that sets the correct values for both. The domain of the ISR is set as the current domain afterwards for any ISR category.

For category 2 ISRs, the current Task is additionally set to `INVALID_TASK`, if the ISR2 belongs to a different domain than the currently running Task. This is necessary because otherwise a Task object could migrate to a different domain through an ISR2. For ISR1, this is not required, because ISR1 must not use the `GetTaskID()` system service.

The user defined handler routine is then invoked and, for ISR1, the saved value of the current domain is restored before leaving the ISR1.

The OSEK system services related to interrupt handling allow the disabling and enabling of interrupts. These are provided as static methods by a class `Interrupt-Service`. Calls to these methods are intercepted at the call-side and replaced with calls to the OSEK services. These services may therefore be invoked without any overhead.

5.5 Event Mechanism

The Event mechanism allows extended Tasks to enter the Waiting state until one of the Events the Task is waiting for is set. This is also the only way that an OSEK Task can possibly block during its execution, which is an important aspect for garbage collection.

An Event is represented by its mask. The mask of an Event needs to contain a bit unique for each Task, that accesses the Event, because bit operations can be used to create masks containing multiple Events. It is possible, that two Events use the same mask, if there is no Task in the system that accesses both of the Events. The mask of an Event is either configured manually in the configuration file or automatically generated by the OSEK configurator. OSEK specifies, that the mask of each Event is accessible via the identifier of the Event in the code, and the identifiers can be used with any bit operation.

In KESO, Events are also represented by their mask. There is no need to create an object abstraction for Events, since the setting and querying of Events for a Task is already limited by the requirement for a Task SO. Furthermore, an object abstraction would add a significant overhead to provide the creation of masks using bit operations. The mask of an Event is provided as a constant value with the name of the Event identifier in the class `Events`. This class is automatically generated from the information of the KESO configuration file. The builder computes an appropriate Event mask for each Event where automatic generation of the mask is configured, and the generated OIL file contains the computed values of the mask. This prevents, that the OSEK configurator computes different masks than the KESO builder.

The Event-related services are provided by the `EventService` class and implemented as follows:

- `ClearEvent()` and `WaitEvent()` both expect an Event mask as parameter. The mask can be constructed using the identifiers provided by the `Events` class and bit operations. Calls to these methods are replaced at the call-side with calls to the corresponding OSEK services.

- `SetEvent()` and `GetEvent()` are passed the SO of the Task that the service is to be performed on. `SetEvent()` additionally expects an Event mask. The method body of these services is replaced, whereby additional code to check the reference to the Task SO for `null` and to extract the Task identifier from the SO is added, similar to the wrapper code created for the Task services.

5.6 Resource Management

Resources allow the coordination of concurrent accesses of several Tasks to shared resources. KESO supports both, domain local Resources for the coordination of Tasks within the same domain and global Resources for the inter domain coordination of Tasks. The implementation of the object abstraction for Resource has already been extensively described in section 5.2.

There are only two system services related to Resource management, the `GetResource()` service, that allows to occupy an OSEK Resource, and the `ReleaseResource()` service, that allows to release an occupied Resource. The reader is assumed to be familiar with the ceiling protocol used by OSEK systems in conjunction with the Resource management, that plays a role in the synchronization of the garbage collector implemented in chapter 6.

Both of the above services expect a Resource identifier on the C level and require a Resource SO on the Java level. Unlike Task objects, Resource SOs cannot be created by the user application. A Resource SO therefore always contains a valid Resource identifier.

Additional code that checks the SO reference for `null` and extracts the Resource identifier from the SO, needs to be inserted in the body of the magic methods, similar to the wrapper code added for the Task services.

5.7 Alarms and Counters

OSEK provides a two-stage concept for processing recurring events, Alarms based on Counters.

5.7.1 Counters

A Counter is represented by its value that is measured in *ticks*. A Counter can either be increased by hardware or by software. The OSEK specification does not further specify this, but requires that OSEK implementations provide at least one Counter that is derived from a hardware or software timer. Alarms are based on a Counter and expire when the Counter reaches a predefined counter value. In reaction to the expiration of an Alarm, a Task can be activated, an Event for a Task can be set or an alarm callback routine can be invoked. Multiple Alarms can be based on the same Counter.

Since the number of available timers, that a Counter can be based on, is usually very limited, Counters are only configured globally in KESO. OSEK associates a few constants to each Counter that are accessible via macros, such as the maximum allowed value of a Counter. On the Java level, these are provided as constants of the `Counters` class. The constants have the same name as the macros provided by OSEK, e.g. the maximum allowed value for the Counter with the identifier `c1` is available as `Counters.OSMAXALLOWEDVALUE_c1`.

5.7.2 Alarms

Alarms, on the other hand, can be configured as both, global and domain local objects. Object abstractions were introduced for Alarms as described in section 5.2. Multiple Alarms can use the same Counter, even if the Alarms are configured in different domains.

Scope of an Alarm

The scope of an Alarm restricts the use of the system services (e.g. a domain local Alarm can only be canceled by a Task within the same domain) and limits the reactions to an Alarm:

- Activate a Task: This action can only activate Tasks within the same domain as the Alarm. Global Alarms cannot use this action.
- Set an Event for a Task: The Event can only be set for Tasks within the same domain as the Alarm. Global Alarms cannot use this action.
- Invoke a callback function: This is the only action possible for global Alarms. The callback function is executed in the domain of the Alarm, or DomainZero for global Alarms.

Wrapper Function for Alarm Callback Functions

OSEK specifies, that Alarm callback functions must be defined using the `ALARMCALLBACK` macro. Very similar to ISRs, a wrapper function is created by the OSEK builder that uses the macro and calls the user defined callback function. The callback function is specified in the configuration file by the class and the signature of the callback method. The callback method must be static and can neither have parameters nor a return value.

The wrapper code created around the invocation of the user specified alarm callback method is the same as for an ISR1, i.e. a backup of the current domain is taken and the current domain is set to the domain of the Alarm associated with the callback function before invoking the user defined callback method, and the saved domain is restored afterwards.

Alarm Services

The Alarm-related system services are provided by the `AlarmService` class on the Java level. The Alarm services allow to query the properties of the underlying counter of an Alarm using the `GetAlarmBase()` service. A class `AlarmBase` has been implemented on the Java level that contains all the elements of the `AlarmBaseType` structure on the C level specified by the OSEK specification and can be used in the same manner.

All functions that expect an Alarm identifier as parameter on the C level require an Alarm SO on the Java level. Wrapper code has been added in these functions that extracts the OSEK identifier of the SO to invoke the OSEK service and checks if the reference is `null`, similar to the wrapper code created for the Task services. Contrary to Task objects, Alarm SOs cannot be created by the user application. Thus, an Alarm SO always contains a valid Alarm identifier.

5.8 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, the interface may be implemented by classes of the user application to prevent instances to be copied across portal calls.

Chapter 6

Memory Management

KESO was designed to allow the coexistence of different heap implementations. Each domain in a KESO system can choose a heap implementation for managing its heap.

Besides the heap implementation developed in this work, KESO already contained a very simple heap implementation, that does not provide any garbage collection. 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 memory requirements of the application must be exactly predictable and must not grow with the runtime of the application. This heap implementation is suitable for hard real-time systems where no overhead for garbage collection can be tolerated.

The heap implementation developed in this work features a precise, incremental, tracing, non-moving mark-and-sweep garbage collector (GC). The reader is supposed to be familiar with garbage collection techniques, as they are not covered in this work. A survey on garbage collection techniques on uniprocessors is available in [Wil92]. This heap implementation was named the *IdleRoundRobin (IRR)* heap implementation, because—from the viewing point of the application developer—the garbage collection is performed at the *idle* time of the system, and because a single OSEK Task, the *garbage collector Task (GCT)*, manages multiple domains in a manner similar to *Round Robin* (see section 6.6.1, *Domain Selection*). The OSEK Task of the garbage collector is configured with the very lowest priority of all Tasks in the system, and is therefore only active when there is no other Task in the system in the Ready state, i.e. all other Tasks are either in the Suspended state or the Waiting state. This is a very advantageous point in time for garbage collection, because the stacks of Tasks in the Suspended state are empty, and therefore the effort required to scan the root set of reachable objects is expected to be minimal at this point.

The application developers have to ensure, that there is enough time for the garbage collector to run, by integrating the garbage collector in the design process of the whole system just as any other user application. This approach can be termed *cooperative* *garbage collection*, as the cooperation of the application developer is required.

As an incremental garbage collector, the GC process can be interrupted at any stage. To ensure a low latency on interrupts (i.e. external events), which is crucial on real-time systems, the garbage collector was designed in a way, that restricts all critical sections[1] to constant complexity. The worst case reaction time to an incoming interrupt is therefore low and easily predictable.

The remainder of this chapter is structured as follows: At first, the data structures of the IRR heap implementation is described in section 6.1, followed by a description of the coloring mechanism used by the IRR heap in section 6.2. Then, the implementation of the free memory list and the special list processing function `listwalker()`, that allows traversal of the list with all critical sections being of *O(1)* (constant) complexity, is discussed (section 6.3). The `listwalker()` function is the base of both, the allocator function (section 6.4) and the garbage collector (section 6.5). The discussion of the garbage collector poses the main part of the chapter. The different phases of a garbage collector cycle are explained in section 6.6. An interruption of the garbage collector during the scan phase imposes particular problems, that are identified and solved in section 6.7. Finally, the overhead and interrupt latency caused by the garbage collector are presented in section 6.8.

6.1 Data Structures of the IdleRoundRobin Heap

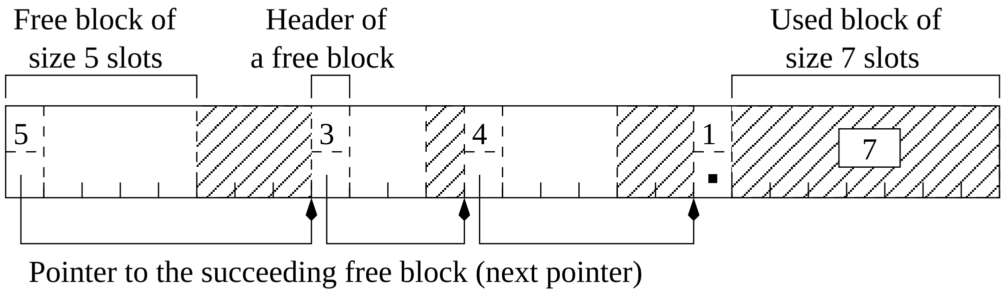
This section describes the data structures deployed by the IRR heap in order to establish fundamental knowledge required to understand the allocator and garbage collector implementations. The figures used for illustration also introduce the figure conventions used later on in the more sophisticated illustrations.

6.1.1 Slot Division and Bitmap

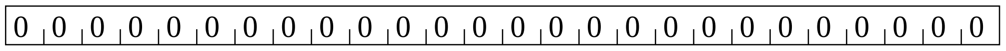
As every heap in the KESO system, an IRR heap is of a statically configured size that cannot be changed at runtime. During the scan phase the garbage collector needs to keep track of the parts of the domain heap that are used by living[2] objects. The garbage collector uses a *bitmap* to mark the occupied memory by setting corresponding bits in the bitmap. Allocating one bit in the bitmap for each byte of the heap would be wasteful, since each object has a minimum size due to the object header, which is currently four bytes. The heap is therefore divided in *slots* of a fixed, configurable length, the *slot size*. Each object occupies one or more consecutive slots on the heap. A slot is never shared

[1]The term *critical section* is used throughout this chapter to refer to a section of code, that is protected by a blockade of interrupts when executed, i.e. interrupts are disabled straight before entering the critical section of code, and immediately reactivated after leaving the critical section of code.

[2]An object is said to be *living* if it is still reachable by the user applications.


Free block of size 5 slots
Header of a free block
Used block of size 7 slots
5
3
4
1
7
Pointer to the succeeding free block (next pointer)

(a) Example of an IRR-heap with embedded free memory list



(b) Empty bitmap with one bit associated with each slot of the heap

Figure 6.1: Example of an IdleRoundRobin heap with the associated bitmap

among objects, so there is a certain clipping in cases where objects do only partially occupy a slot.

Figure 6.1(a) shows an example heap with the corresponding bitmap in figure 6.1(b).

The slot size is configurable for each domain that uses the IRR heap implementation, however, there are some restrictions. First, the slot size always needs to be a multiple of the word size of the underlying hardware, i.e. a multiple of 4 bytes on the 32-bit Tricore architecture, because an object always needs to be word-aligned in order to address word-sized object fields like references. Second, the size of the object header (currently 4 bytes) sets a minimum limit for the reasonable slot size, as it determines the minimum size of any object. Third, the header of a free block in the free memory list (see section 6.1.2), which is 8 bytes on the used 32-bit Tricore architecture, is stored in the first slot of a first memory block, thus a slot must be large enough to accommodate the entire free block header. The minimum size for a slot in our test configuration is thus limited by the latter to 8 bytes.

To keep the used memory as low as possible the slot size must be carefully selected. The smaller the slot size, the larger the size of the bitmap. The larger the slot size, the higher the clipping of objects that do only partially occupy their last slot.

There is only one bitmap shared among all domains using the IRR heap, the size of which is determined by the heap with the most slots. The sharing of the bitmap is possible, because the garbage collector never processes multiple domains at the same time. This saves memory compared to a separate bitmap for each heap.

6.1.2 Free Memory List

The IRR heap implementation uses a linked list to keep track of the free memory on the domain heap, referred to as the *free memory list* in the following. The free memory list consists of *list elements*, each containing meta information on a *block of contiguous free memory*. A block of contiguous free memory consists of consecutive unused slots on the domain heap, and a block's size is therefore always a multiple of the slot size. The term *free memory block*, or in this context also just *block* in short, refers to the free memory block as a whole, composed of consecutive free slots, while the term *list element* is used to refer to the data structure containing meta data of a block such as its size.

The list element is stored in the first slot of the block it describes. This way, no extra memory is consumed by the free memory list. The size of a list element therefore sets the lower limit for the size of a slot to 8 bytes on the Tricore architecture (as discussed above in section 6.1.1). The address of the first list element is stored in the domain descriptor.

The meta data contained within a list element are the 16-bit `size` of the associated block, expressed as the count of its slots, the `colorbit` initially assigned to newly allocated objects from this block, the locking `mode` of the element, each of which is assigned an 8-bit portion of the element, and a pointer to the `next` element, that has a size of 32 bits on the Tricore architecture.

The 16-bit unsigned integer, the `size` is stored in, limits the maximum size of a free memory block to 65536 slots. In the current implementation, this is also the limit for the total size of the heap, since an empty heap is described by a single list element. The implementation could, however, be extended to use multiple list elements for describing memory blocks exceeding this limit.

The locking mode is explained along with the interrupt friendly list implementation in section 6.3.

The pointer to the next list element contains the address of the list element following the current one in the list, or the special value `KESO_EOFML`, that equals `0xffffffff` (on a 32-bit architecture) to denote the end of the list. The next pointer must never point to a list element at a lower address than the current one, i.e. the list elements are sorted and the list is always processed from the element at the lowest address to the element at the highest address. This is a prerequisite for some algorithms deployed in the garbage collector. It should be noted, that the special value used to mark the end of the list is always considered a higher address in a pointer comparison with the address of any list element.

An example for a partially occupied heap and an embedded free memory list managing the unused parts of the heap is shown in figure 6.1(a).

6.1.3 Working Stack for the GC Scanning Phase

The *working stack* is an array of object references that is used by the garbage collector during the scanning phase to store references to objects that still need to be scanned. The *stack pointer* is an index into the working stack array and contains the index of the next unused array element. Thus, pushing to the stack is a post-increment operation on the stack pointer while popping references from the stack is a pre-decrement operation. The stack is empty when the stack pointer is 0.

The maximum size of the working stack has to be predictable, because the stack must not overflow during the scanning. The scan algorithm implemented in the IRR-GC guarantees, that an object reference is never present on the working stack multiple times. Thus, the worst-case size of the stack can easily be calculated as the maximum number of objects, that can be allocated from the heap, plus the number of immortal objects that exist in the system.

The maximum number of objects, that can be allocated from an IRR heap, equals the number of slots on the heap. Immortal objects are allocated by the KESO builder, thus the number of immortal objects is known at compile time.

To further pursue the idea of cooperative garbage collection, the worst case size of the stack could be reduced with hints by the application developer, who could possibly provide a closer estimate on the maximum number of reachable objects in the system at the time of garbage collection.

6.1.4 Managed Domains Array

As aforementioned, the IRR heap implementation creates a single Task for garbage collection that manages all domains deploying this heap type. The *managed domains array* contains the domain identifiers of all domains managed by the GC and is used by the domain selector function to determine the domain for a garbage collection cycle.

6.1.5 Garbage Collector Domain

The garbage collector of this heap can be interrupted during a garbage collection cycle. During certain phases of the garbage collection, it is important for the interrupting code to know, if a garbage collection is currently being performed on the domain the code runs in. This is explained in detail in section 6.7, where the interruptibility of the garbage collector is discussed. The *garbage collector domain (GC domain)* contains the identifier of the domain, that the IRR-GC is currently operating on, or the special identifier `INVALID_DOMAIN`, if the garbage collector is currently not running in a critical phase.

6.2 Coloring of Objects

During the scan phase, the garbage collector performs a coloring of objects according to the tricolor marking scheme [Bak92]. The `color` field of an object header (section 4.3.1) is used to store the color of an object. Bit 0 of the `color` byte is always set to 1. This allows to distinguish the object header from object references, in which the least significant bit is always cleared because of the alignment of the object header[3].

To store the color, only one bit in the `color` byte of the object header is actually used, indicating whether the object is marked as black, gray or white. The distinction between black and gray is made by the objects presence on the working stack of the garbage collector. If the object is present on the working stack, it has not been scanned yet and its color corresponds to gray, else the object is not scanned in this cycle and the color corresponds to black[4].

After each garbage collector cycle, the color of all objects that survived the cycle needs to be reset to white for the next cycle. To achieve this with the least necessary effort, the meaning of the color bit is simply inverted after each garbage collection cycle. The domain descriptor of each domain, that uses the IRR heap, contains a field that stores the current value for colored (black, gray) objects.

6.3 Interrupt Friendly List Implementation

A crucial aspect of embedded real-time operating systems is minimizing the interrupt latency, which determines the worst-case reaction time to external events. For all operations that require a traversal of the free memory list, the IRR heap implementation provides a generic and reentrant function `listwalker()`, that allows traversing the list and applying a callback function to each list element. All critical sections are of constant complexity. The interrupt latency is thereby kept low and predictable.

To achieve this, each list element contains an 8-bit locking field, of which only one bit is actually used. The semantics of this bit is, that as long as it is set, the list element must not be removed from the list. It may, however, be resized. The locking bit is also called the *mode* of the list element in the following, and always has a value of either 0 (unlocked) or 1 (locked), indicating whether the element may be removed from the list or not.

The `listwalker()` function performs a single traversal of the list, calling a specified callback function for each list element. The callback function may cause the termination of the list traversal if no further processing is required.

[3]This is only true for architectures with an address capacity of 16 bits or higher. For 8-bit architectures, a different solution would have to be found.

[4]An object is either black because it has already been scanned in the garbage collector cycle or because it was allocated black during a garbage collector cycle.

The implementation of the `listwalker()` function is shown in figure 6.2, where `domain_t` and `listel_t` are the types of a domain descriptor structure and a list element structure respectively. The callback function is of the type `callback_fct_t`. The function was slightly simplified by removing the special handling necessary for the first element.

6.3.1 Interface to the Caller

The `listwalker()` is passed a pointer to the domain descriptor structure in the parameter `domaindesc`, which contains the beginning of the free memory list. This pointer is also handed through to the callback function. The parameter `cbparam` can be used by the application to pass an optional parameter to the callback function and is not used by the `listwalker()` itself. The parameter `callback` contains the address of the callback function. The callback function needs to conform to the interface described in section 6.3.2.

6.3.2 Interface to the Callback Function

The callback function is invoked for each list element in the free memory list and finally with the special value `KESO_EOFML`. Besides the addresses of the domain descriptor structure and the optional parameter `cbparam`, it is passed the address of the currently processed list element, the location of the pointer linking to the current element (`prevNextPointer`), which is either the next pointer of the preceding list element or the head pointer in the domain descriptor structure for the first list element, and the original locking mode of the current element. The callback function may cancel the list traversal at any time by returning a 0 value, e.g. when the allocator has found a suitable memory block to satisfy the request. If further processing of the list is required, a value of 1 has to be returned.

The `listwalker()` takes charge of acquiring and releasing the locks on the elements. If the structure of the list is changed, however, the mode of the current element cannot safely be restored by the `listwalker()`. An allocator might have removed the current element and handed it to the application for instance—in that case, restoring the locking mode would corrupt the memory occupied by an object of the application. The `listwalker()` detects a change to the list structure by checking if the linking pointer still refers to the current element. If this is not the case, the callback function needs to restore the original locking mode, if the element is still a member of the free memory list, which is e.g. the case when new elements are inserted by the garbage collector between the previous and the current element.

Interrupts are enabled while calling the callback function, however, it is guaranteed, that the current element, as well as the location with the pointer referring to the current element, is not removed asynchronously while the callback function processes it. If

```
void listwalker( domain_t       *domaindesc,
                 callback_fct_t  callback,
                 void           *cbparam )
{
  listel_t **prevNextPointer;
  listel_t *curElement, *prevElement;
  int cbreturn=1;
  unsigned char prevmode=0, mode;

  /* address of head pointer of free memory list */
  prevNextPointer = &( domain->freemem_head );

  while( cbreturn != 0 ) {
    DisableAllInterrupts();
    curElement = (listel_t *) *prevNextPointer;

    if( curElement == KESO_EOFML ) {
      /* end of the free memory list */
      EnableAllInterrupts();
      callback(curElement, 1, prevNextPointer,
               domain, cbparam);
      break;
    }

    mode = curElement->mode;  /* backup original mode */
    curElement->mode = mode | 1; /* lock element */
    EnableAllInterrupts();

    cbreturn = callback(curElement, mode,
                        prevNextPointer,
                        domain, cbparam);

    /* advance to next element */
    if( *prevNextPointer == curElement ) {
      prevElement->mode = prevmode;
      prevElement = curElement;
      prevNextPointer = &curElement->next;
      prevmode = mode;
    }
  }

  /* restore mode of previous element */
  prevElement->mode = prevmode;
}
```

Figure 6.2: Implementation of the `listwalker()` (simplified)

the callback function performs any critical operations on the list, however, it has to synchronize the critical operations by itself. This topic is covered with the description of the particular operations in section 6.4 for the allocator and section 6.6 for the various phases of the garbage collector.

6.3.3 Synchronization of Recursive Invocations

Because the list walking operation can be interrupted by higher priority allocator functions that change the structure of the list, appropriate measures have to be taken to synchronize overlapping invocations of the `listwalker()`. The two possible scenarios are

1. An allocator function is interrupted by another allocator function invoked by a higher priority OSEK Task.
2. The garbage collector is interrupted by an allocator function invoked by a higher priority OSEK Task.

It is impossible for the garbage collector to interrupt an allocator function, because the GCT always runs on the very lowest priority[5]. The only structural change that an allocator function can possibly do to the free memory list is the removal of a free memory block. The insertion of new blocks and the merging of close-by blocks is only performed in the garbage collector.

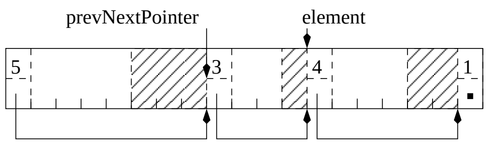
During a run of the `listwalker()`, neither the current element nor the memory location of the pointer linking to the current element may be removed from the list (Figure 6.3). The straightforward solution to avoid this problem is the disabling of interrupts for the entire run of the `listwalker()`. This critical section is, however, at least of a complexity linear to the number of list elements[6], causing a possibly unacceptable high interrupt latency. To avoid the problem, a locking mechanism was implemented that allows the execution of the callback functions with interrupts enabled.

One bit in the header of a free block is used to protect list elements from being removed. During a list traversal, the element that is currently being processed needs to be protected and—if the current element is not the first element—also the previous element. This is necessary, because the pointer linking to the current element needs to be updated upon removal of the element. If the previous element was removed in the meantime, the operation cannot be performed anymore (Figure 6.4).

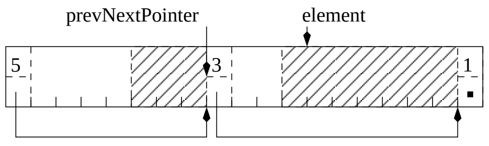
The critical operation that needs to be performed by the `listwalker()` is advancing to the next list element, consisting of the following steps:

[5]An OSEK Task can only overlap the execution of a higher priority OSEK Task, if the latter is in the Waiting state. Because an allocator function never enters the Waiting state, this scenario cannot happen.

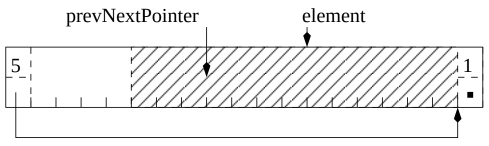
[6]Because the interrupts would also be disabled when running the callback function, the complexity might be higher, depending on the complexity of the callback function



(a) Before the interruption. The list traversal is currently at the third list element. The third element is linked by the next pointer in the second element, thus `prevNextPointer` points to this location.

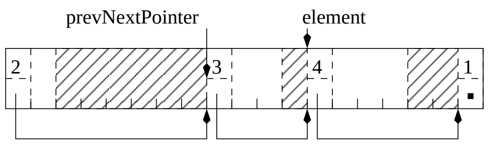


(b) The current element was removed by an overlapping allocator function. The callback function does not immediately notice this, and either misleadingly interprets memory of the allocated object as a list element or, even worse, modify the data thereby corrupting the state of the object.

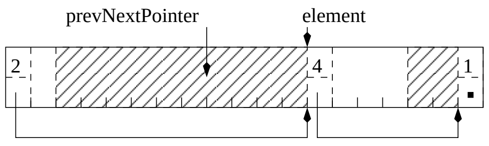


(c) The `listwalker()` could detect the removal of the current element by checking the contents of the `prevNextPointer`, however, the list element containing this pointer might also be removed by a second interrupting allocator function, leaving the `listwalker()` in an unrecoverable state.

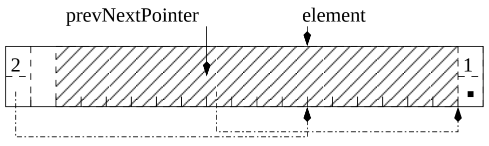
Figure 6.3: Problem scenario of a list traversal that is overlapped by one or more allocators that remove list elements at critical positions, when there is no synchronization mechanism.



(a) An allocation request on 4 slots before the interruption. The list traversal is at a point where an element exactly fitting the requested size has been found and ready to remove that element.



(b) An overlapping allocation of a block of 3 slots removes the previous list element. The 4 slots sized block is now linked from the first block in the list and the `prevNextPointer` of the interrupted allocator has become invalid.



(c) Removing the 4 slot sized block leaves the list in an inconsistent state and corrupts the object allocated in the 3 slots block, because the correct location of the linking pointer is not known to the allocator function anymore.

Figure 6.4: Race condition where two overlapping allocators both remove a list element and leave the list in an inconsistent state.

1. Read the next pointer of the current element.
2. Read and backup the locking mode of the next element.
3. Acquire a lock on the next element by setting the locking bit.

These steps need to be performed atomically, because

- An interrupting allocator between steps 1 and 2 might remove the following element. After removal of the element the assumed list element header now actually contains either the header of the object allocated in the removed block or a reference field of that object. Reading the locking mode would be pointless and setting the locking bit would corrupt the state of the object.
- The removal of the next element between steps 2 and 3 likewise corrupts the object's state when setting the locking bit. A special machine instruction such as bit-test-and-set would allow the atomic performance of steps 2 and 3 without the need to disable interrupts, but is not available on the Tricore architecture[7].

After acquiring the lock, the callback function is invoked, and upon return of the callback function the lock on the previous element can be released.

6.4 Allocator Function

The design of KESO allows to choose a specific heap implementation on a per domain base. Each heap implementation provides an allocator function of its own, but since code is shared among domains, a common entry point for allocating new objects needs to be available.

6.4.1 Generic Allocator Function Interface

The generic function for allocating new objects is the `keso_allocObject()` function. This function determines the size in bytes required by the new instance and invokes the appropriate heap implementation specific allocator function. The allocation of instances of array classes works slightly different: For each array class type, a generic

[7]The initial approach was reserving a bit in the object header for the locking bit at the location corresponding to the locking bit in the header of a list element. This would allow the locking bit to be written even to an already allocated object without corrupting that object. This would, however, render the use of the bidirectional object layout impossible, because in case of objects that contain inner reference fields the former header of the free list element would be occupied by such a reference field rather than the object's header. Additionally, the Tricore architecture does not provide an instruction for setting single bits using an indirect addressing mode. One would therefore have to reserve an entire byte in the object header. This would clash with the goal of a minimal memory consumption.

allocator function is provided that works similar to `keso_allocObject()`. The differences are the calculation of the object size, that incorporates the length of the array and the byte size of each array element, and that the instance has an extra `length` field in the object header. For the allocation of the required memory, the generic allocator functions for the array class types use the same heap specific allocator function as the `keso_allocObject()` function. In the following description of the interface, only the `keso_allocObject()` function is described. The interface is illustrated in figure 6.5. The generic `keso_allocObject()` function is invoked with the class identifier (15) of the instance that is to be created. The required size in bytes (12 bytes) for the new instance is then determined from the class store using the provided class identifier. The address of the heap specific allocator function is looked up in the domain descriptor of the current domain, and the allocator function is then invoked with the size in bytes of the memory needed for the instance. The heap specific allocator function is expected to return a chunk of memory of a size larger than or equal to the size requested, with every word except the first be cleared. If the heap specific allocator function fails to allocate a suitable chunk of memory, it is expected to throw an `OutOfMemoryException`. In the first byte of the memory chunk, the initial `color` (allocation color) of the object is passed. The generic `keso_allocObject()` reads the `color` byte, clears the first word of the memory chunk and initializes the header of the object at the appropriate reference offset. The `color` byte is written as given by the heap specific allocator function to the object header. The semantics of the `color` byte may differ for each heap implementation and are not known to the generic allocator function.

6.4.2 IdleRoundRobin Allocator Function

The basic steps performed by the allocator function of the IRR heap are shown in figure 6.5. First, the size in bytes (12 bytes) provided by the generic allocator function is rounded up (16 bytes) to a multiple of the slot size (8 bytes) and converted to slots (2 slots). The `listwalker()` function is then invoked with an appropriate allocator callback function to find a suitable free block (see section 6.4.3). The found chunk of memory is cleared, except the first word, that contains the initial `color` byte of the object or, if no suitable block is available, and `OutOfMemoryException` is thrown. Finally, the allocated chunk of memory is returned to the generic allocator function.

6.4.3 Allocation of a Memory Block

To find a suitable block of free memory, the free memory list needs to be traversed. The `listwalker()` is used for this task and a callback function for allocating a suitable block is provided. The callback function checks the size of each element it is invoked with. The first element that has a size larger than or equal to the required size is then

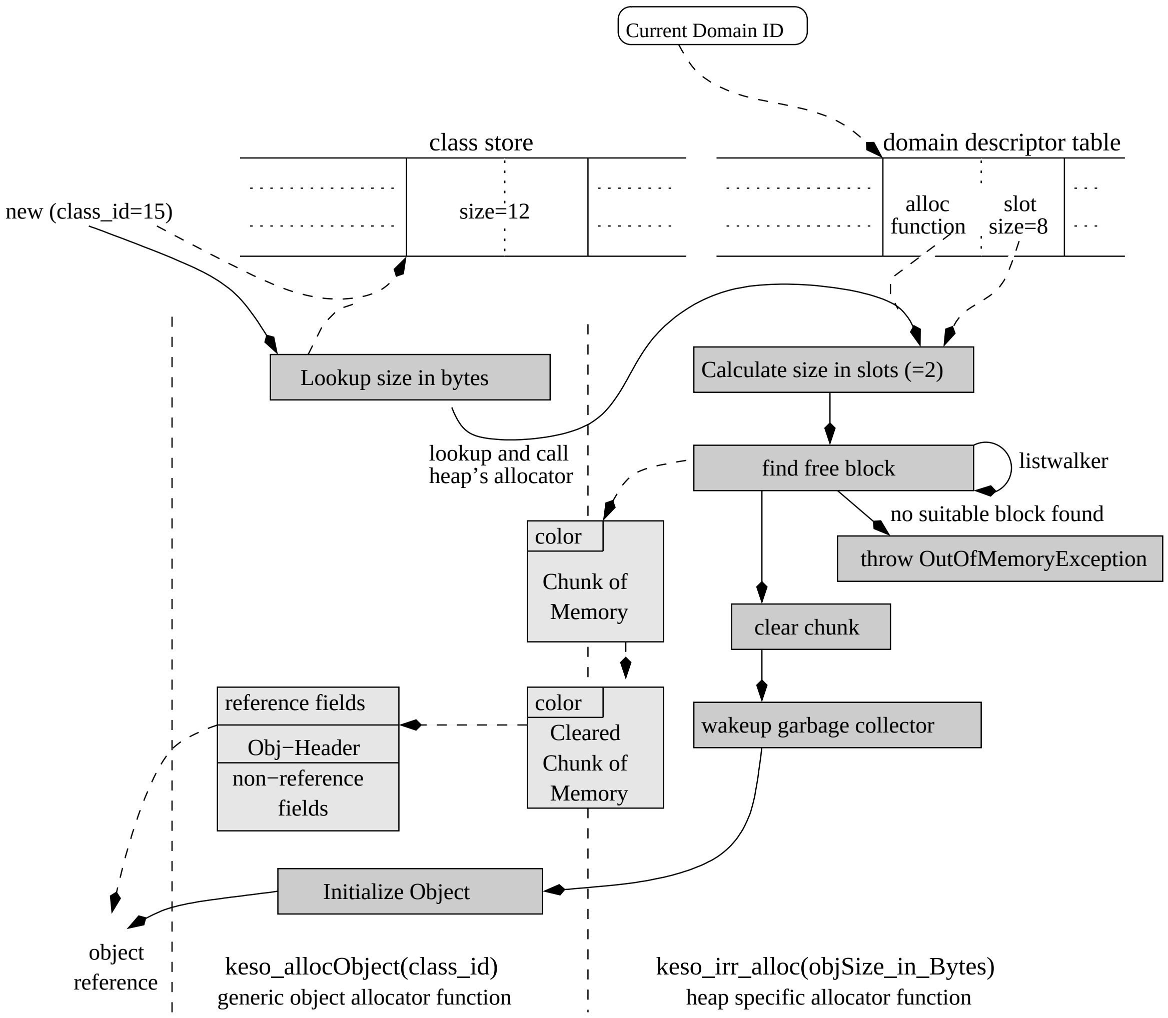


Figure 6.5: Interface between generic and heap specific allocator function

either downsized or, in case of an exact fit, removed, and the list traversal is terminated. The pointer to the allocated block is returned via the `cbparam` parameter to the callback function, that contains the address of a pointer variable of the IRR heap's allocator function. If no suitable block is found, `NULL` is returned and the IRR heap's allocator function throws an `OutOfMemoryException`.

Checking the size of free memory block is not a critical operation, as it does only read the value from the list element. The downsizing and removal operations of a free block, however, are critical sections and need to be secured.

Downsizing of a free list element. When the callback function encounters a free block with a size *larger* than the required size, the following critical section is entered[8]:

1. Read the size of the free memory block again. It might have been downsized after checking the size but before disabling interrupts by a higher priority allocator function.
2. Check if the size is still larger than the requested size. If it is not, leave the critical section and continue with the check for an exactly fitting block.
3. Calculate the difference between the free memory block's size and the requested size, which is the remaining size of the free memory block.
4. Write the remaining size to the list element.

The allocated memory is then available behind the downsized memory block, and its location can be calculated with the knowledge of the address of the free memory block and the remaining size of the free memory block, both of which are available in local variables.

Removal of an exactly fitting element. When encountering a block of free memory matching exactly the required size, the free memory block can be entirely allocated and removed from the free memory list, provided that the element was not locked. If the element was locked by a lower priority Task, it has to be ignored. In case the element was not locked before, the following critical section is entered:

1. Read the size of the block again. Although the element is protected from being removed it might have been downsized by a higher priority allocator function.
2. If the size still matches, read the value of the next pointer of the current list element, or else leave the critical section and continue.

[8]A compare-and-swap or similar instruction would allow the atomic execution of these steps, but is not available on the Tricore architecture.

3. Store the read next pointer in the pointer linking to the current element, pointed to by `prevNextPointer`, effectively removing the current element from the free memory list.

The removed free memory block can now entirely be used to store the new object. Steps 2 and 3 need to be performed atomically, because the next list element could be removed from the list, thereby updating the next pointer of the current list element. Storing the previously read value in the linking pointer would then link to memory block that has been removed from the free memory list and thereby corrupt the list.

The `listwalker()` function detects the removal of the element, because the value of the linking pointer was updated, and does not modify the allocated memory attempting to restore the original locking mode.

Allocation Color

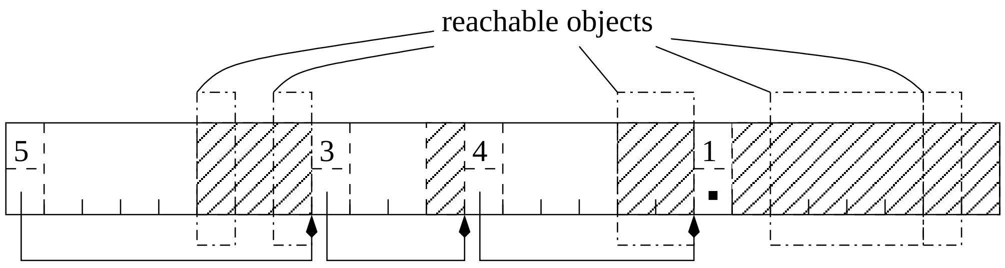
The initial color of the object needs to be passed in the `color` byte to the generic allocator function in the first byte of the allocated memory. Objects are generally allocated white, except when the garbage collector is currently active in the domain. The transition from white to black allocation happens per list element (see section 6.6.2), thus the color needs to be stored with each list element. Therefore, to pass the initial color of the object in the first byte of the memory, the value only needs to be copied from the list element header of the free memory block that the object was allocated from.

6.5 The Garbage Collector of the IRR Heap

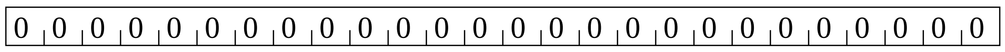
The garbage collection is performed by a single lowest priority garbage collector OSEK Task. This Task manages the heaps of all domains that deploy the IRR heap implementation as their heap type. Whenever the garbage collector starts a cycle in a specific domain, the GCT *migrates* to this domain. This is done by changing the domain id of the associated KESO Task object, that is then set as the current domain by the `PreTaskHook()` whenever the GCT is scheduled. This is necessary for any user application code (e.g. the `finalize()` methods of reclaimed objects) to run in the proper domain environment.

6.6 Phases of the IRR Garbage Collector

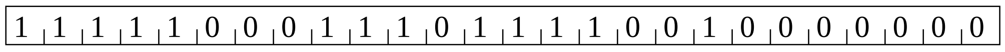
A garbage collection cycle can be divided into several phases. As a mark-and-sweep garbage collector, the IRR garbage collector goes through a scan-and-mark phase, in



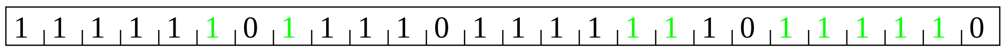
(a) Heap with reachable and unreachable objects before the Garbage Collection



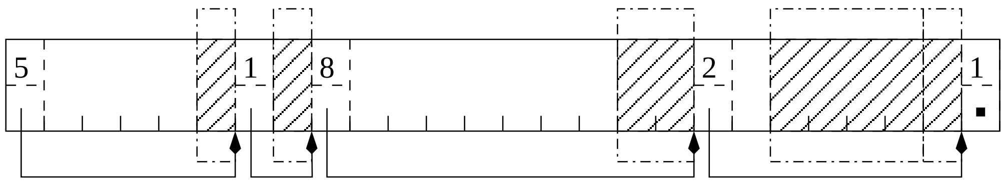
(b) Initial cleared bitmap, every bit associated with one slot of the heap



(c) Bitmap after marking the slots associated with free memory blocks



(d) Bitmap after marking the slots occupied by reachable (*living*) objects



(e) Heap after the garbage collection cycle. The memory of unreachable objects was added to the free memory list.

Figure 6.6: Sample Garbage Collection divided by phases

that the reachable objects are scanned and colored, and a sweep phase, in which the memory of unreachable objects is reclaimed and added to the free memory list. Additionally, as the IRR garbage collector manages multiple domains, there is an additional initial phase in which the domain for the garbage collection cycle is chosen, and because the garbage collector does not keep track of the total set of allocated objects, there is a phase in that the free memory is marked in the bitmap in order to distinguish it from the memory of unreachable objects in the sweep phase.

In the following, the different phases of the IRR garbage collector are described in the order of processing in a garbage collector cycle. Figure 6.6 illustrates the impacts of each phase by means of a simple example.

6.6.1 Domain Selection

In the very beginning of the garbage collection cycle, the domain, whose heap will be garbage collected, has to be selected. Originally a simple Round Robin scheme was used to select the domain, which has the disadvantage, that garbage collection is often performed on domains that still have enough free memory, and no or only little garbage can be collected, whereas other domains are running out of memory.

To avoid this problem, the *Need* of a domain for garbage collection is calculated as

$$Need = FreeSlots - NewSlotsAllocated - \frac{HeapSize}{5}$$

where *FreeSlots* is the total number of slots available in the free memory list, *NewSlots-Allocated* is the recorded number of slots allocated from the domain's heap since the last garbage collection cycle, and *Heapsize* is the total number of slots of the entire heap, 20% of which is additionally subtracted as a grace value. *FreeSlots* and *NewSlots-Allocated* are recorded in the domain descriptor of the domain and updated by the IRR allocator function and the IRR garbage collector. A domain is considered to require a garbage collection if a negative Need value is calculated for the domain.

The idea behind the Need function is that embedded systems typically run cyclic Tasks that retain a similar allocation behavior during the runtime of the system. Thus, if there is still enough memory to allocate the number of slots since the last cycle plus a deviation of 20%, the domain likely has enough memory until the next cycle.

In order to keep a certain fairness, the selection of the domain is not solely based on the Need value, and a Round Robin element is kept in the algorithm: All domains managed by the IRR garbage collector are ordered in a circular buffer. The garbage collector remembers the last processed domain across cycles. Starting from this domain, it searches through the circular buffer and selects the first domain with a negative Need value that is found, which is then selected for this cycle. In case there is no domain with a negative Need value at the time of the domain selection, the behavior of the garbage collector depends on its working mode, that is statically configured in the KESO system configuration:

Lazy Mode (Default) In this mode, the garbage collector enters the Suspended state (`TerminateTask()`) and sets a global flag, indicating that the garbage collector is sleeping. The IRR allocator function checks this flag upon each invocation and reactivates the garbage collector (`ActivateTask()`) if necessary.

This mode is the preferable in most scenarios and hence the default. First, it enables the OSEK operating system to put the microcontroller in a low power mode when the system is idle. Second, a running garbage collector needs to disable interrupts at certain stages, which delays the reaction to external events. Putting the garbage collector in the Suspended state allows immediate reaction to interrupts when a garbage collection is not required (except if interrupts are blocked by the user application). Third, write barriers (section 6.7.1) make certain types of write accesses to reference fields more expensive. When the garbage collector is not active, the fastest path through a write barrier can be taken (figure 6.7).

Workaholic Mode In this mode, the garbage collector always chooses the domain with the smallest Need value in the case that every domain has a positive Need value, therefore the garbage collector is always active.

The Round Robin element of the selection algorithm prevents one domain, that maintains a large set of reachable objects using a large portion of the domain heap and therefore keeps a low Need value, from starving other domains[9].

6.6.2 Marking of Free Memory Blocks

In the second phase of the cycle, the free memory list is traversed one time and the bits in the bitmap of the garbage collector, that are associated with slots of free memory blocks, are set. This ensures, that after completing the scan phase, only the bits of slots that belong to unreachable objects are still clear. After marking the bits of a free memory block, new objects allocated from that block cannot be reclaimed in the running cycle anymore, and do not need to be scanned because of the active write barriers (see section 6.7.1). When marking the slots of a free memory block, the allocation color for new objects allocated from that block is changed to black at this point.

This phase is implemented as a callback function to the `listwalker()`, that performs the following operations for each block on the free memory list:

[9]If domains allocate a large set of objects in the startup phase and keep references to those objects, and do only allocate few objects in the regular runtime, the *NewSlotsAllocated* value is small after the first cycle and the Need function computes a higher Need value, which additionally compensates the starving effects.

1. Read the size of the list element
2. Change the allocation color in the list element header to black (after the cycle the meaning of the color is inverted and objects allocated in the meantime automatically become white)
3. Mark the bits in the bitmap corresponding to the block, using the previously read size. The first corresponding slot can be calculated as

$$FirstSlotIndex(elementaddress) = \frac{elementaddress - heapstartaddress}{slotsize}$$

These steps do not need to be performed atomically, though it may happen, that an object is allocated white between steps 1 and 2 and the bits associated with the slots in the bitmap are marked in step 3. If the object is reachable during the scan phase, it would unnecessarily be scanned and colored, and the bits in the bitmap would be marked again, which causes some additional work, but does not impose any further problems.

The order of steps 1 and 2 is important, however. If the color would be changed before reading the size of the element, it could happen that an element is allocated black after changing the allocation color, but before reading the size of the free memory block. In this case, the bits in the bitmap corresponding to the slots of the object would not be marked in step 3 and would neither be marked in the scan phase as the object is already colored black. Therefore, it might happen that the object is reclaimed although it is still reachable.

The bitmap modifications performed in this phase are illustrated in figure 6.6(b) to figure 6.6(c), for the example scenario shown in figure 6.6(a).

6.6.3 Scan and Mark of Reachable Objects

In the scan-and-mark phase, the garbage collector scans, starting with a root set of objects, the reference fields of each discovered object, adds the referenced objects to the working set, and marks the discovered objects, i.e. colors them and marks the slots used by these objects in the bitmap. The root set consists of immortal system objects[10], static reference fields of the respective domain and the stacks of Tasks in the Waiting state. The CPU registers of the saved contexts of Tasks in the Waiting state do not need to be scanned, because every reference held in a CPU register is also stored in some reference field in the memory.

[10]These are generally comprised by the object abstractions of the OSEK abstraction layer, i.e. Task, Alarm and Resource objects, however, Alarm and Resource objects do not contain any reference fields and no subclasses of these classes may be created. Therefore the set of immortal system objects that need to be scanned by the garbage collector is solely comprised by the Task objects.

Overlapped execution of the scan-and-mark phase and the user applications imposes certain problems beyond the consistency of global data structures of the garbage collector, and appropriate synchronization mechanisms need to be deployed. These problems and the used techniques to solve these problems are described in section 6.7.

The working set of the garbage collector is maintained on the working stack of object references. Because an object reference is never added to the working set twice in a cycle, the maximum size of the working stack depends on the maximum number of reachable objects in the system at a time, which is, in the worst case, the maximum number of objects that can be allocated from the heap plus the number of immortal objects. The former is equal to the number of slots on the heap, the latter is fixed and known at system creation time. This number could, however, be significantly reduced with hints by the application developer.

Initially, the whole root set is pushed to the working stack. Pushing an object reference, that is not a `null` reference, consists of the following steps:

1. Check if the object is already colored (if it is, do not push the reference)
2. Color the object by setting the color bit in the object header to the value that currently represents black or gray color
3. Read the stack pointer
4. Increase the stack pointer by one
5. Write back the increased stack pointer

These operations are performed in the function `pushObject()` (figure 6.7), which is also used by write barriers (see section 6.7.1). This critical section needs to be protected by disabling interrupts. An overlapping `pushObject()` of the same reference, performed by a write barrier, between steps 1 and 2, can cause the object to be pushed on the stack multiple times. This is not tolerable as it might impact an overflow of the stack. Steps 3 through 5 need to be performed atomically, because an overlapping `pushObject()` between these steps would use the same stack slot, and the reference pushed by the overlapping write barrier would be overwritten and lost[11].

The critical section above actually can be divided into two distinct critical sections between steps 2 and 3, but has been combined to avoid the doubled overhead for enabling and disabling the interrupts in between[12].

[11]In theory, `pushObject()` could overlap n times in this critical section, loosing n-1 pushed references, where n is the number of OSEK Tasks configured in the system (including the GCT) plus the number of ISRs plus the number of Alarm Callback routines.

[12]Steps 1 and 2 can be performed atomically on architectures with bit-test-and-set and bit-test-and-reset instructions as available on 80386 architectures, steps 3 through 5 with a compare-and-swap or similar instruction.

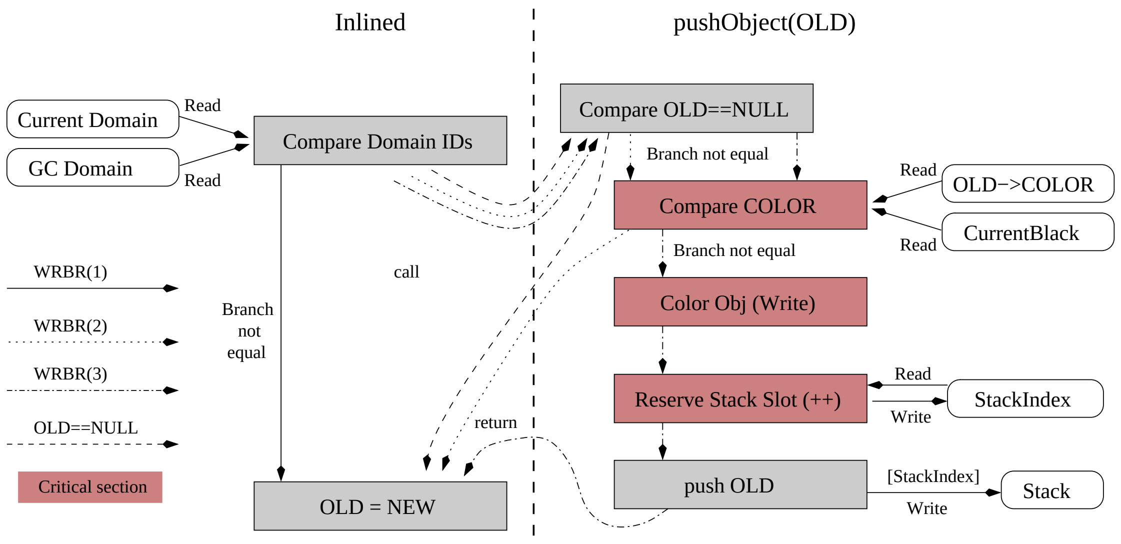


Figure 6.7: Code Paths through a Write barrier and the pushObject function

After leaving the critical section, the reference is written to the reserved stack slot. This operation is uncritical, because references are only popped from the stack by the garbage collector, which never overlaps a `pushObject()`.

After pushing all references of the root set to the working stack, the garbage collector begins scanning the stack by successively popping elements from the stack until the stack is empty. For each popped reference, all inner reference fields of the referenced object are pushed to the working stack using the above procedure.

Because of the bidirectional object layout (see section 4.3.3), scanning the inner reference fields of an object is easy. First, it needs to be determined whether the object is an array of references, which can be done by comparing the object's class identifier against the class identifier of the reference array class. For all other classes, only one lookup in the class store is required to determine the reference offset of the object.

After working off the entire working set, all slots used by living objects have been marked in the bitmap. Slots whose bits in the bitmap are still clear now mark the space occupied by unreachable objects that can be reclaimed. Figure 6.6(d) shows the example bitmap after the scan phase, for the reachable objects as displayed in the example heap (figure 6.6(a)).

Because the Tricore architecture does not support instructions to directly address single bits, the coloring was optimized to not only use one bit in the object header but the entire `color` member of the object header. To check the color of an object, the whole byte can be read and compared, and for setting the color the whole byte can be written. This renders the additional CPU operations required for extracting or

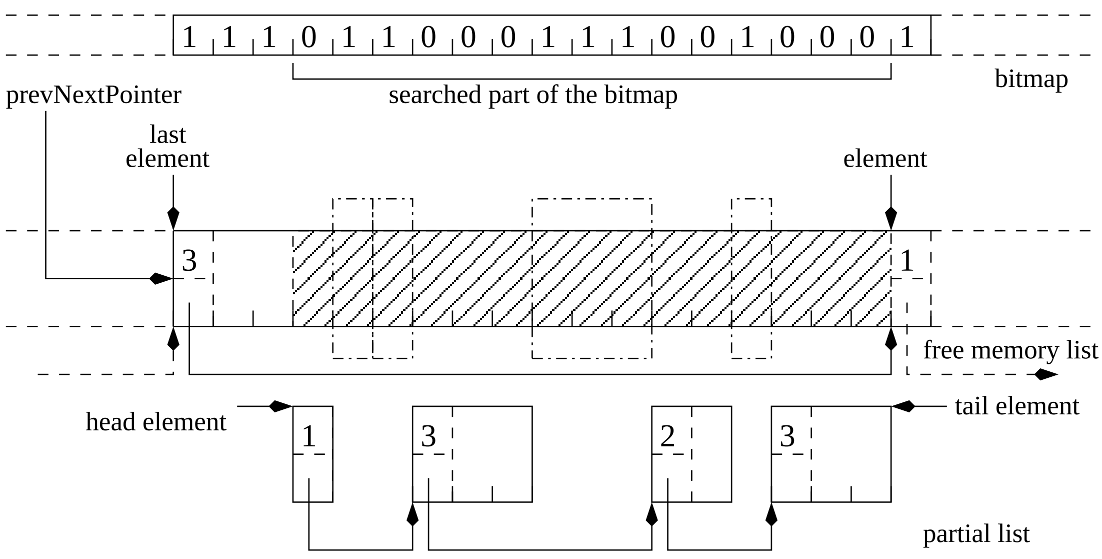


Figure 6.8: Memory Reclaiming in the Sweep phase

setting single bits of the byte unnecessary, and noticeably increases the performance of the color-related operations (for benchmarks see section 6.8). On the other hand, more memory is required in the object header, because the `color` byte of the header is completely used.

6.6.4 Sweep: Recover Memory of Unreachable Objects

In the sweep phase, the memory of objects that were not marked in the scan-and-mark phase is reclaimed and added to the free memory list. Furthermore, the JVM specification [LY99] and the Java Language specification [GJSB05] require, that the finalizer of an object is invoked before it is reclaimed by the garbage collector, which is also done in this phase.

The sweep and allocation algorithms ensure that the elements of the free memory list always remain sorted by address, i.e. an element, that is not the last element in the free memory list, always links to another element at a higher address. The sweep mechanism is implemented as a callback function to the `listwalker()`, that searches for reclaimable memory between two list elements (the *last element* and the *current element*) at a time, creates a linked list of the new free blocks found (the *partial list*) and inserts this list between the currently processed two elements, possibly merging the first (the *head element*) and last element (the *tail element*) of the partial list with the surrounding elements of the free memory list (figure 6.8). The sorted property of the free memory list allows the merging of elements without searching the entire free memory list for candidates. Merging of reclaimed objects that occupied near-by slots happens automatically as they appear as a sequence of cleared bits in the bitmap. Building the partial list instead of inserting each discovered free block immediately into the free memory list is advantageous, because it does not need to be synchronized with the user application.

Because the `listwalker()` only passes the current element to the callback function, the callback function must remember the last element it processed. The optional parameter of the callback function `cbparam` is used for this purpose and contains a pointer to a structure that contains the address of the last element as well as the current position in the bitmap. The pointer is initialized with `NULL` and the bitmap position with 0 before starting the `listwalker()`. The three possible invocation scenarios of the callback function are

- For the part of the heap before the first element, the last element pointer is `NULL` and the element pointer points to the first list element.
- For any part of the heap between two elements of the free memory list, the last element points to the last processed element and the element pointer points to the current element.
- For the part of the heap after the last element, the last element pointer points to the last element of the free memory list and the element pointer contains the special value `KESO_EOFML`.

Only the parts of the bitmap that correspond to the scanned part of the heap per invocation of the callback function need to be scanned, i.e. the parts corresponding to free memory blocks are skipped. The phase in which the free memory blocks are marked in the bitmap is nevertheless necessary because the size of a free memory block might have changed since the beginning of the cycle, or free memory blocks may have been removed from the list. These slots belong to objects that were allocated during the garbage collector cycle and must not be freed during this cycle.

Before a new free memory block is inserted in the partial list, the finalizers of the unreachable objects contained within that block need to be invoked. Since the garbage collector does not keep track of allocated objects, the object headers need to be found within the free memory block. In front of the object header, there may be several reference fields, depending on the class of the object. Because the least significant bit of the object header is always set, it can be distinguished from references, which have their least significant bit cleared for alignment reasons[13]. When the object header is found,

[13]On 8-bit microcontrollers, this approach does not work and an alternate solution needs to be found. A possible solution would be aligning object headers on slot boundaries. A slot boundary is always at an address with a cleared least significant bit due to the minimum slot size of the heap.

the finalizer of the object can be invoked. Afterwards, the non-reference fields of the object can easily be skipped as the class identifier can now be read from the object header, and be used to lookup the size of the object in the class store.

Synchronization with the User Applications

The insertion of the partial list in the free memory list needs to be synchronized with interrupting allocator functions. Besides the case that there was no reclaimable memory between the currently processed elements, there are four cases that are differently processed.

Neither head nor tail are mergeable If neither the head nor the tail of the partial list can be merged with the surrounding elements of the free memory list, an uncritical insertion operation can be performed[14]. First, the next pointer of the tail element of the partial list is assigned the address of the current element. Because the current element is locked by the `listwalker()`, it cannot be removed by an allocator and the reference stays valid. Afterwards, the address of the head element of the partial list is stored in `prevNextPointer`. After the address was written back, the new elements are available to allocator functions in the free memory list. The current element needs to be unlocked by the callback function, except if the current element has the special value `KESO_EOFML`.

Only head is mergeable If only the head element of the partial list is mergeable with the last element, the next pointer of the tail element is assigned the address of the current element, as above. The interrupts need to be disabled for the following critical section:

1. Check if the head is still mergeable with the last element (If not, proceed as if neither head nor tail were mergeable).
2. Add the size of the head element to the size of the last element.

All operations from checking if the elements are still mergeable until storing the increased size must not be interrupted by an allocator, as downsizing of the last element would revoke the mergeable property. After the critical section, the next pointer of the last element is updated with a local copy of the next pointer of the head element, that was made before entering the critical section. This is not critical, because the free memory list is in a consistent state where neither the last nor the current element may be removed, and storing the next pointer is an atomic

[14]On architectures where a store operation for an address value needs to be split into two or more store operations, such as the AVR architecture, storing an address value is not an atomic operation and updating the next pointer needs to be included in the critical section.

operation[14]. Therefore, the remainder of the partial list is simply not visible to allocator functions until the store operation for the next pointer has been performed. The current element needs to be unlocked by the callback function, except if the current element has the special value `KESO_EOFML`.

Tail element mergeable (two cases). If the tail element is mergeable with the current element, this cannot be changed by interrupting allocators, because the current element must not be removed. A critical section is entered that performs:

1. Add the size of the current element to the size of the tail element
2. Copy the next pointer of the current element to the next pointer of the tail element. The tail element and the current element have now been merged.
3. Check if the head element can be merged with the last element
   - If the head element cannot be merged, store the address of the head element in the linking location, pointed to by `prevNextPointer`.
   - If the head element can be merged, add the size of the head element to the size of the last element, and copy the next pointer of the head element to the next pointer of the last element.

Finalize Issue

Before adding a new block to the partial list, the finalizers of all contained objects need to be invoked. Through the operations of the finalizer of an object, objects that were previously unreachable by the application may become reachable again, if they were reachable by the finalizable object (*f-reachable*), e.g. by writing a reference to an f-reachable object to a static reference field. The issue is discussed in detail in the Java Language Specification [GJSB05]. In the simplest case, the finalizer can store the `this` reference to a static reference field thereby making the object the finalizer is invoked on reachable again.

Basically, the Java specification requires an additional scan phase after calling the finalizer that finds that the object is still unreachable. If the object becomes reachable again, the application will work with an object that has already been finalized. If a finalized object is found to be unreachable again, the finalizer is not invoked again on that object and it is reclaimed.

The IRR heap does currently not handle this issue. Instead, finalizers that make f-reachable objects reachable again are prohibited. This has the advantage that all objects found unreachable can immediately be reclaimed.

The downside is, however, that the distinction between f-reachable and reachable objects might not always be obvious to the application developer. Therefore, as alternate solutions, it is considered to either

- not invoke the finalizers at all, which would violate the Java specification, or
- introduce a finalized state for objects, by using an additional bit in the object header. This is only necessary for class instances that override the default finalizer. On those instances, the `finalize()` method is invoked when an unfinalized object is found unreachable in the scan phase, the finalized bit is set but the object is not reclaimed. If a finalized object or an object whose class does not override the default `finalize()` method is found unreachable, it is reclaimed. This adds the downside, that objects that override the `finalize()` method can at the earliest be freed in the second garbage collector cycle after becoming unreachable.

The finalizers are invoked by the GCT at the lowest priority, therefore they may be interrupted by other Tasks. This behavior is conforming to the Java specification that allows finalizers the be executed in any thread and even the concurrent execution of multiple finalizers in different threads. The invocation of a `finalize()` method does, however, prolongate the garbage collector cycle, which might impose problems for real-time systems, as it must be calculated by the application developer.

6.7 Interruptibility of the Scan Phase

During the scan phase of the garbage collection, the garbage collector can be interrupted by the user applications at any stage, with the exception of the short protected critical sections. By overwriting reference fields, the user application (also called *mutator* in this context) modifies the *graph of reachable objects*. If the mutator creates a new reference from a black object to a white object, this object is not discovered by the garbage collector on this path during the scan phase anymore. This is not a problem if the object is still reachable via another discoverable path, which is the case in the beginning of the garbage collector cycle. If the mutator removes the last discoverable path to a living object, the object is not marked by the garbage collector and reclaimed. To avoid this problem, one of the following two conditions must be satisfied:

- a black reference field must never reference a white object
- the last discoverable path to a living, white object may never be removed

There are two different methods, that guarantee at least one of the two above conditions. Read barriers color white objects gray whenever the mutator reads a reference. Thus the mutator never gets to see a white object, and therefore cannot create a path from a black to a white object.

Write barriers, on the other hand, color objects upon write accesses. Different kinds of write barriers [GLS75, DLM$^+$76, JL96] have been developed that differ in which object's color is changed. Because write barriers commonly offer a better performance

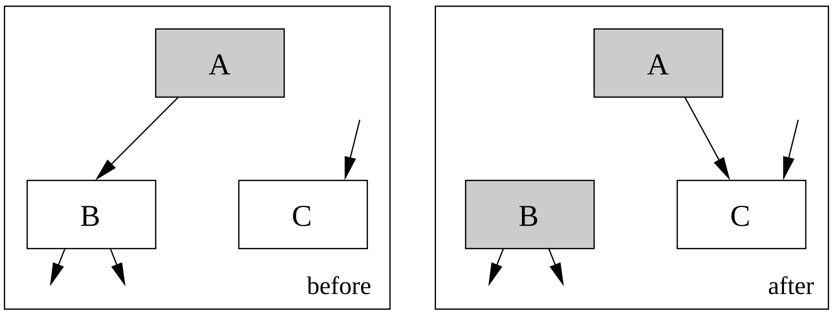


Figure 6.9: Yuasa's write barrier implementation

than read barriers, basically based on the assumption that read accesses occur more often than write accesses, write barriers were used in KESO. A comparison of barrier methods for garbage collection including overhead measurements is available in a technical report by B. Zorn [Zor90].

6.7.1 Write Barriers

For KESO, the write barrier variant of Yuasa [JL96] was chosen (figure 6.9). In this implementation, whenever a reference to a white object is overwritten, no matter if the reference field was already scanned or not, the object referenced by the overwritten reference (B) is colored gray. This guarantees the second of the above conditions, as it prevents the last discoverable path to a living white object from being removed. Write barriers are enabled before starting the phase where the free memory blocks are marked in the bitmap.

Write barriers add a significant amount of overhead to write accesses on reference fields. In order to keep write accesses to local reference variables of a Java method as fast as possible, write barriers are not used on local variables. Write barriers are enabled for write accesses to static reference fields, reference fields in object instances and field writes in an array of object references. All objects, that were reachable at this point, are not reclaimed in the same garbage collection cycle, because the write barrier marks them, except if the object was only referenced on the stack of a waiting Task and this reference is removed, in which case the object can safely be reclaimed.

6.7.2 Atomic Scanning of the Task Stacks

Because write barriers are not active on the local reference variables of Java methods, the Tasks' stacks must be scanned atomically. Only stacks of Tasks in the Waiting state need to be scanned, the stacks of all other Tasks in the Suspended state are empty at the time the garbage collector starts.

Scanning a stack is of a complexity linear to the size of the stack. In most cases, the size of the stack of a Task is easily predictable at the time the stack is scanned by the garbage collector, because the stack is only scanned when a Task is waiting for an Event, which is only the case in few well-known places where `WaitEvent()` is called. Recursive invocation of blocking methods may, however, complicate the prediction of the stack size.

The straight-forward solution of disabling the interrupts for the entire scan of the stacks imposes a high interrupt latency, that depends on the size of the stacks which in turn depends on the user application, and would foil the low interrupt latency achieved by the aggressively optimized critical sections in the other parts of the garbage collector. However, it is not required to scan all Tasks atomically, but only the stack of one Task at once[15], and only the Task whose stack is scanned actually needs to be delayed until the scan operation is finished. Stefan Gabriel solved a similar problem in his study thesis [Gab05] when implementing a real-time garbage collector for the JX operating system [GFWK02]. In his implementation, he completely disabled the scheduling to create copies of each stack. These copies could then be scanned without additional synchronization. This allows the processing of first level interrupt handlers while a stack is copied.

Copying the stacks is not an attractive solution for KESO because of memory constraints, furthermore the way KESO's stacks are organized allows a very fast scanning of the stacks and an exact and fast detection of references on the stack, that is nearly of the same efforts as copying the stack[16]. When scanning a stack, all discovered objects are merely colored gray, i.e. the reference is passed to `pushObject()`. The scanning of the objects themselves is performed after completing the stack scan.

Synchronization using OSEK Resources

The basic idea of Stefan Gabriel's garbage collector was adapted for the IRR garbage collector, however, the granularity was refined. Entirely disabling the scheduling still poses the problem that high priority Tasks may be delayed by the garbage collection, even though the stack of those Tasks is not scanned. For the IRR heap, a synchronization mechanism based on OSEK Resources was developed that allows a maximum flexibility for the granularity.

The basic idea is to assign a *garbage collector resource (GCR)* to each Task that uses the `WaitEvent()` system service. These Tasks can be easily identified, because the

[15]The only problem one might think of is the propagation of a reference that is only held in a local reference variable on an unscanned (white) stack to an already scanned (black) stack. This is, however, only possible via a common object or static reference fields, and write barriers are active on both of these.

[16]In JX, reference fields on the stack need to be identified via a complex heuristic, which posed the major reason for copying the stacks instead of immediately coloring the objects.

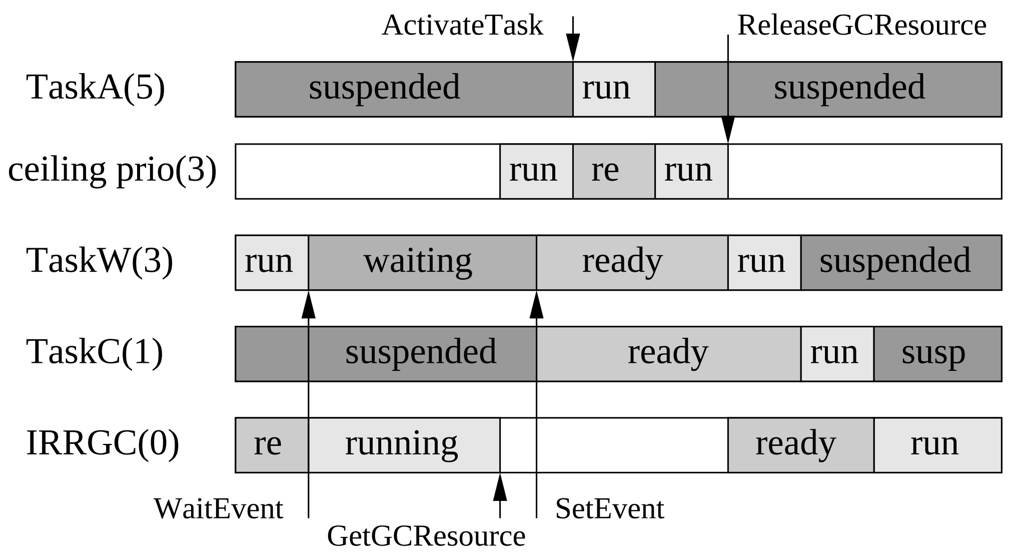


Figure 6.10: Synchronization of Task stack scanning using OSEK Resources. The figure shows the GCT (IRRGC), a low-priority Task (TaskC), a medium priority Task (TaskW), that enters the Waiting state while the garbage collector is performing the scan phase, and a high priority Task (TaskA). The priority is parenthesized behind the Task names. The ceiling priority of the GCR is equal to the priority of TaskW.
In the example, while the GCT is scanning the stack of TaskW, both TaskW and TaskC become Ready, however, they are not scheduled, because the GCT is running with the ceiling priority that is greater than or equal to the priority of TaskC and TaskW. Later on, while the GCT is still scanning the stack of TaskW, the high-priority TaskA becomes Ready and is immediately scheduled. After TaskA terminates, the GCT finishes scanning the stack and release the GCR. TaskW and TaskA are then scheduled according to their priorities.

OSEK OIL specification [OSE04] requires Tasks to declare references for every Event they may react to in the system configuration file. While scanning the stack of a waiting Task, the GCR of the Task is occupied (figure 6.10[17]). The OSEK priority ceiling raises the priority of the GCT to the ceiling priority of the GCR, which is in this case the priority of the waiting Task. This still allows any Task with a higher priority than the waiting Task, first and second category ISRs and alarm callback routines to interrupt the garbage collector during the scanning of the stack. Tasks with a lower priority than the

[17]Note that there is no direct transition from the Suspended state to the Running state as implied by the figure. Actually, the suspended Task does first enter the Ready state and is then scheduled according to its priority and the other Tasks in the Ready state. In the figure, this intermediate state is omitted for simplification in cases where the priority of the Task causes its immediate transition to the Running state

ceiling priority of the GCR are delayed, which is desired to avoid priority inversion.

Creating a separate GCR for every extended Task that makes use of the `WaitEvent()` system service is the finest granularity of the stack scanning synchronization and allows a minimum number of Tasks being delayed by the operation. However, this finest granularity might not be required for every application scenario. Therefore, the IRRGC allows the definition of *synchronization groups*, that allow choosing coarser granularities to save system resources. Every Task belonging to a domain using the IRR heap that uses the `WaitEvent()` system service is assigned a synchronization group, and a GCR is then created for each defined synchronization group and shared by all members of that group. The resulting ceiling priority of a GCR is determined by the highest priority Task in the synchronization group. Locking this GCR therefore affects a higher number of Tasks than a GCR chosen on a per-Task base. As an example, a synchronization group could be defined per domain.

6.7.3 Scan Order

In the scan phase, it is imperative that the stacks of waiting Tasks are scanned before scanning the remaining parts of the root set. Otherwise, the only reference to a white object on a Task stack could, for instance, be written to a static reference field that was already scanned and removed from the stack. Since write barriers do only color the object to that a reference is overwritten, they do not color the object in this case because the reference is overwritten on the stack where write barriers are not active.

6.8 Benchmarks

This section presents measurements of the overhead posed by the used write barriers as well as comparisons of different implementation approaches.

All benchmarks are run on an Infineon TC1796b microcontroller, containing a 32-bit Tricore v1.3 CPU clocked at 150 MHz during the test runs, 2 megabytes of embedded flash memory, 192 kilobytes on-chip SRAM and a 16 kilobytes instruction cache. The microcontroller is attached to a Lauterbach hardware debugger, and the measured run times were recorded by the debugger.

KESO is under fast-paced development, and changes to the KESO code will likely affect the runtime of the tests. The tests that show the benefits of write barrier inlining and the coloring optimization were made with revision 357, all other tests with revision 390 of the KESO subversion repository, and the test results should be reproducible with that revision.

For each of the three types (static reference fields, object instance reference fields, components of an array of references) of write barrier protected field writes, a test program was written, that performs 20,000 consecutive writes of the respective field type.

To determine the loop overhead of each test, the actual field writes were removed from each test and the time was measured. In all shown numbers, the loop overhead is already subtracted. The shown values thus represent the actual time required for the 20,000 reference writes of the respective type.

The overhead posed by a write barrier depends on the actual reference that is pushed.

The three possible code paths through a write barrier where a non-null reference is overwritten are illustrated in figure 6.7. Each test was performed for all three code paths. To force the write barrier to take a specific path through the write barrier during the benchmark, the garbage collector was modified. The modifications for each code path are as follows.

Without Write barriers (Without WRBR) Instead of passing through the write barrier, only the actual write access to the field was left in the code. This is exactly the same code as it would be used without write barriers.

Write barriers not active (WRBR(1)) To model the path where the GCT is not performing a scan phase in the domain of the test Task, the GC domain was set to a fixed value that differs from the test Task's domain. Thus, the WRBR(1) path is always taken.

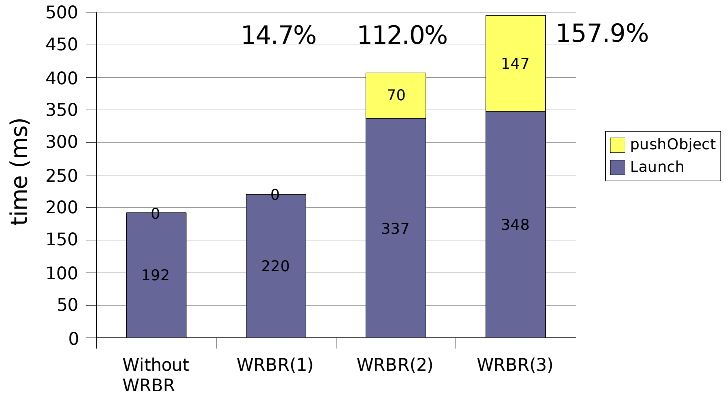
Write barriers active, reference to a colored object overwritten (WRBR(2)) The GC domain is set to a fixed value of the test Task's domain. Furthermore the condition of the test, that checks if the object is already colored, is inverted. The comparison then indicates that the object is already colored and never colors the object.

Write barriers active, reference to a white object overwritten (WRBR(3)) The GC domain is set to a fixed value of the test Task's domain. The function `pushObject()` is modified to always color the referenced object white, effectively disabling the coloring of the object. Therefore, the WRBR(3) code path is always taken, but instead of coloring the object gray it is always recolored white.

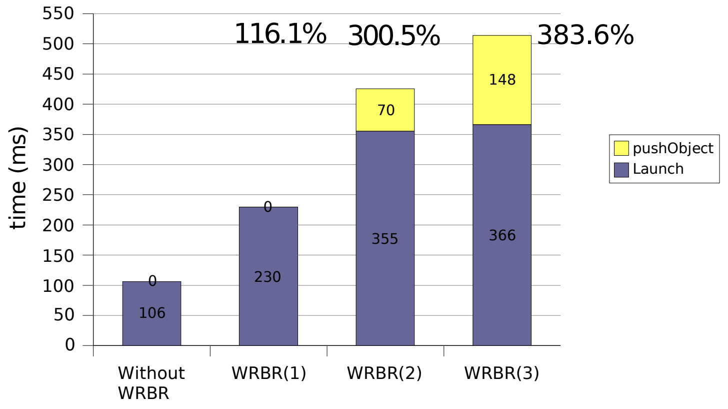
6.8.1 Overhead Measurements

The three tests were run with coloring optimization and inlined write barriers, and compared with the reference time without write barriers. The displayed times are the absolute measured run times of the functions minus the loop overhead. `Null` reference checks for instance reference writes and `null` reference and array boundary checks for reference array component writes are not considered loop overhead and included in the printed numbers.

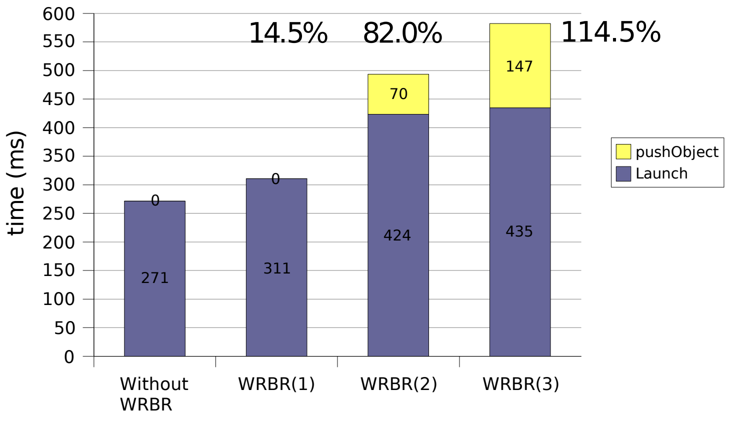
Write access to an instance reference field without write barriers is the fastest of the three test types. Before accessing the field the reference only needs to be checked



(a) Static reference fields



(b) Instance reference fields



(c) Reference array components

Figure 6.11: Write barrier Overheads on each reference field type. For each test, the height of the bar shows the entire time taken by the test. Each bar is split in the runtime spent at the call-side (the `launch()` method of the test Task in this case), and the time spent in the `pushObject()` function where applicable. The values above the bars show the overhead percentage of a write barrier path. The loop overhead was separately identified for each test and subtracted, the shown run times represent the isolated time required for the field accesses.

for `null`. The address of the reference is at a constant offset from the object reference which is known at compile time. Static reference field writes are slightly slower, because the current domain value needs to be loaded and the address of the reference needs to be calculated. Writing a component of an array of references is the most expensive write type in the test field, as it requires a `null` reference check, a check of the array boundary and the calculation of the address.

For the WRBR(1) path, which is considered the most frequent case for a GCT working in lazy mode, write barriers incur the least overhead (28 ms, 14.7%) on the static references test, because the current domain, that is required to test whether the GCT is currently performing a scan phase in the current domain, needs to be loaded anyway to access the correct set of static fields. Therefore, only the GC domain needs to be loaded and compared with the current domain. Writing components of a reference array is more expensive (40 ms, 14.5%) because the current domain does not need to be loaded without write barriers, but due to the already high costs for such an access the overhead percentage is even less than with static reference fields. In the instance field test, the overhead is much more noticeable (114 ms, 116.1%), because instance field writes need to load the current domain and are the fastest of the three access types, yielding a high overhead percentage.

For paths WRBR(2) and WRBR(3), the invocation of the `pushObject()` function causes additional overhead at the call-side of approx. 115 ms for all tests. The time spent in the `pushObject()` function is an additional 70 ms for checking if the overwritten reference is `null`, and checking if the referenced object is already gray or black.

The WRBR(3) paths need to perform all operations needed on the WRBR(2) plus additionally color the object and push it to the working stack, adding approx. 77 ms to the runtime of the `pushObject()` invocations.

6.8.2 Benefits of the Coloring Optimization

Figure 6.12 shows the benefits achieved by using a whole byte to represent the color value instead of a single bit, exemplified by the test case for object instance reference field writes. The optimization saves the instruction required to extract the colorbit from a byte value for checking the color, and the instructions needed to toggle a bit in the color byte. Using the optimization, the comparison can be realized as a direct comparison of two byte values, and coloring an object as writing a per garbage collector cycle fixed value to the object header. For the WRBR(2) code path, where the color is only checked but not set, the optimization reduces the time spent in the pushObject function by approximately 33%. For the WRBR(3), that double benefits because it needs to check the color as well as set it afterwards, a speed increase of almost 41% could be observed.

The downside of the optimization is an increased memory requirement in the object header, but the other bits of the color byte are not used by the IRR heap anyway.

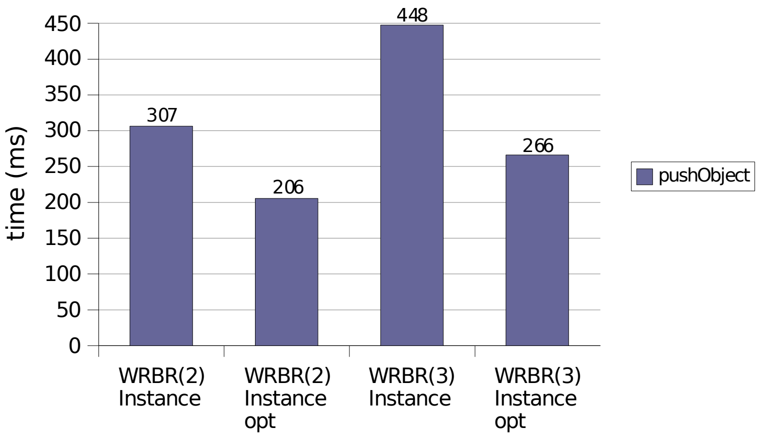


Figure 6.12: Benefits of the optimized coloring method. The test application for object instance reference fields was taken for illustration. Only paths WRBR(2) and WRBR(3) are relevant to this test, because WRBR(1) does not enter the pushObject function. The bars show the time spent in the pushObject function during the test, on top of the bars the exact measured times are printed.

6.8.3 Differences between inlined and not-inlined Write barriers

When implementing write barriers, an implementation decision had to be made among inlining parts of the write barrier or alternatively providing a write barrier function performing all operations of the write barrier, including the actual write operation. A dedicated function has the advantage that less code needs to be generated at the call-side. The inlined variant performs the comparison of the current domain and the GC domain and the actual write operation at the call-side, but still calls the `pushObject()` function for all further operations of the write barrier (figure 6.7), which generates additional code at the call-side but does not require the overhead of a function call in most cases, when the garbage collector is currently not performing a scan phase on the domain. The test for static reference field writes has been performed with a dedicated write barrier function and compared with the run times with inlined write barriers (figure 6.13).

The comparison shows a speed increase of 37% for the codepath WRBR(1) where the `pushObject()` function does not need to be invoked. The time spent at the call-side to perform the comparison of the domain ids and the actual write operation for inlined barriers is less than the time spent at the call-side to invoke the write barrier function. For the other paths, the benefit percentage is less due to the additional operations required on those paths, but still measurable.

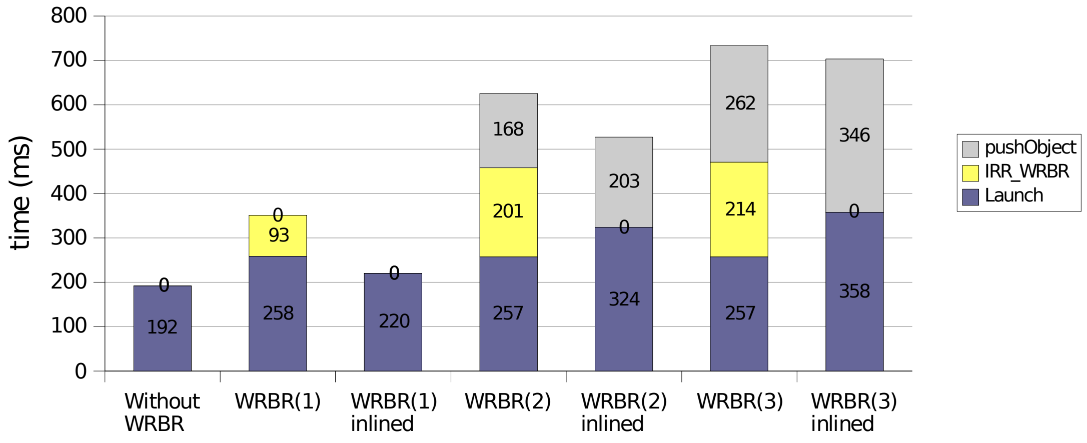


Figure 6.13: Comparison of inlined write barriers and dedicated write barrier function (IRR_WRBR). The coloring optimization was activated for the test runs.

6.8.4 Runtime Spectrum of the `pushObject()` Function

The run times of the `pushObject()` function for the different paths have measured for each test and are shown in figure 6.14. As expected, the runtimes do not differ for the different tests.

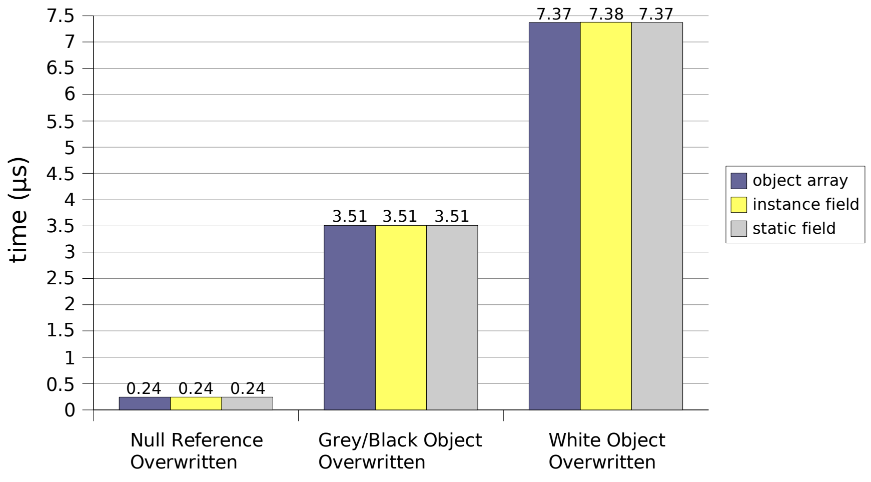


Figure 6.14: Run times of the `pushObject()` function

Chapter 7

Conclusion and Future Work

In this thesis, the design and implementation of the OSEK abstraction layer and a heap implementation, that provides automatic memory management, both core parts of the KESO system, were presented.

7.1 OSEK Abstraction Layer

The OSEK abstraction layer provides access to OSEK system services, whereby access restrictions guarantee, that the strong isolation of domains is retained. The access restrictions are enforced on the Java language level through a name service, carrying on the type-safety-based isolation concept.

Different aspects of the OSEK system have been analyzed to identify the possible impacts on the domain isolation. Object abstractions, implemented in the form of immortal system objects, have only been introduced where necessary to provide access control to system services. The overhead imposed by the system object abstraction compared to the use of plain OSEK data types is therefore only introduced in places where it was found useful and required.

System services that do not require system objects as parameters, e.g. all interrupt-related system services, have been implemented with zero overhead by replacing invocations of the respective Java methods at the call-side with immediate calls to the corresponding OSEK services. The system services, that are subject to access control and therefore are passed system objects additionally contain wrapper code in the method body around the call of the respective OSEK service, including a `null` reference check for the system object. This adds the overhead of the wrapper code plus the call to the Java service method. The exact impact still needs to be evaluated. In some cases, e.g. for the `ActiveTask()` service, where the wrapper code mostly consists of the `null` reference check, it might be better to replace the service call at the call-side, as the reference possibly was already checked in the calling method and can thus be omitted. This would additionally save the code and time required for the method call, thus possibly creating a win-win situation saving both, code size and CPU time.

The name service that provides the user application with system objects uses static data structures. A lookup is resolved to a large extent at compile time, preventing expensive string comparisons at runtime and reducing a lookup to a single or double array lookup. Only the runtime data structures of the name service, that are actually used by the application are added to the generated KESO system.

7.2 IdleRoundRobin Heap Implementation

In the second part of this thesis, a heap implementation was presented that provides automatic memory management adapted for embedded systems. During the design of the IdleRoundRobin heap, real-time capabilities were kept in mind and the heap was optimized to provide a low latency in the reaction to external events. The task model of the underlying OSEK operating system was incorporated in the design of the garbage collector, and a low priority scheduling of the garbage collector was chosen because the effort needed to scan the root set is minimal at the idle time of the system.

To achieve the low interrupt latency, a fully preemptible garbage collector was implemented. The developed algorithms do only require the disabling of interrupts for short critical sections of constant complexity. The worst-case interrupt latency is therefore low and predictable.

Resources were used to synchronize the scanning of Task stacks with the applications, which is not of constant complexity. This allows high priority Tasks to be scheduled during the scanning of a stack. Priority inversion is prevented by the OSEK priority ceiling protocol that is used for OSEK Resources.

Many of the critical sections, that are currently protected by the disabling of interrupts, can be implemented using special instructions such as the compare-and-swap instruction, on architectures where such or similar instructions are available.

The dynamic allocation of new objects is currently allowed for interrupt service routines and alarm callback functions. In common practice, however, memory is not dynamically allocated from such functions. Therefore, as a reasonable constraint, the dynamic allocation of memory could be prohibited for functions on the interrupt level. This would allow to synchronize most of the critical sections, that are currently protected by disabling the interrupts, by disabling the OSEK scheduler. This would even further improve the reaction times to external events. Preliminary benchmarks have shown, however, that the IdleRoundRobin garbage collector already has a very low impact on the interrupt latency.

The garbage collector of the IdleRoundRobin heap is not yet suitable for the use by applications that need to meet hard real-time constraints. It can, however, be used for non real-time parts of the system, while still allowing hard real-time Tasks in other domains.

The main problem of the garbage collector, that renders it unsuitable to real-time requirements, is the yet unsolved fragmentation problem of the heap. One solution to this problem is compacting the heap during garbage collection. This does, however, imply the atomic copying of discovered objects, which imposes system pause times that are probably not acceptable. As an alternative solution, objects exceed the size of a slot could be represented by a linked list of slots that does not need to be sequential in memory, similar to the approach taken in JamaicaVM [Sie04]. This would, however, drastically increase the access times to object fields that exceed the first slot of the object.

A smaller issue of the garbage collector is the currently conservative estimated worst-case size of the working stack. With hints by the application developer, however, close estimates for the worst case size of the working stack are possible.

Write barriers were used to synchronize the mutation of the object graph by the user applications with the garbage collector during a scan phase. The overhead measured for the write barriers is expected to be tolerable to most applications. Realistic applications have not yet been developed for KESO, and benchmarks with such applications still have to be taken to determine the actual impact of write barriers on applications.

7.3 KESO Future Development

To provide a full-featured JVM, Java exceptions and Java monitors still need to be implemented. Furthermore, drivers for the serial port and the CAN interface of the Tricore controller are currently in development.

A port of a stripped down KESO to the AVR architecture is also in development. Because of the extreme resource constraints on AVR controllers, this KESO variant is not based on an OSEK operating system and runs on the bare hardware.

A larger milestone for the future developments is the distribution of a KESO system across several microcontrollers, while retaining a uniform view on the entire system. The domains of the total system are located on different controllers, and inter-controller communication is done through the portal mechanism. One could, for instance, think of a network of low performance AVR nodes, that are connected through a more powerful controller, such as the Tricore, and relocate complex operation to the more powerful controller.

Bibliography

[Bak92] Henry G. Baker. The Treadmill: Real-Time Garbage Collection without Motion Sickness. *ACM Sigplan Notices*, 27(3):66–70, March 1992.

[BBG+00] Greg Bollella, Benjamin Brosgol, James Gosling, Peter Dibble, Steve Furr, and Mark Turnbull. *The Real-Time Specification for Java*. Addison Wesley Longman, 1st edition, January 2000.

[con02] AJACS consortium. AJACS: Applying Java to Automotive Control Systems Concluding Paper V2.0. Technical report, AJACS consortium, 2002.

[Cza00] Grzegorz Czajkowski. Application isolation in the Java virtual machine. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications*, pages 354–366, New York, NY, USA, 2000. ACM Press.

[DLM+76] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, Carel S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: an exercise in cooperation. In *Language Hierarchies and Interfaces, International Summer School*, pages 43–56, London, UK, 1976. Springer-Verlag.

[Dom04] Jörg Domaschka. Entwurf und Implementierung einer statischen Java-Laufzeitumgebung für den LEGO Mindstorms RCX. Study thesis, University of Erlangen-Nuremberg, Germany, October 2004.

[Gab05] Stefan Gabriel. Evaluierung und Implementierung einer echtzeitfähigen Speicherbereinigung für das Betriebssystem JX. Study thesis, University of Erlangen-Nuremberg, Germany, November 2005.

[Gag02] Etienne Gagnon. *A portable research framework for the execution of Java bytecode*. PhD thesis, School of Computer Science, McGill University, Montreal, 2002.

[GFWK02] Michael Golm, Meik Felser, Christian Wawersich, and Jürgen Kleinöder. The JX operating system. In *General Track 2002 USENIX Annual Technical Conference*, pages 45–58. USENIX Association, June 2002.

[GH01] Etienne M. Gagnon and Laurie J. Hendren. SableVM: A research framework for the efficient execution of Java bytecode. pages 27–40. Java Virtual Machine Research and Technology Symposium (JVM '01), April 2001.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The Java Series*. Addison-Wesley Professional, 3rd edition, July 2005.

[GLS75] Jr. Guy L. Steele. Multiprocessing compactifying garbage collection. *Commun. ACM*, 18(9):495–508, 1975.

[Har06] Till Harbaum. NanoVM - Java for the AVR. `http://www.harbaum.org/till/nanovm`, July 2006.

[JL96] Richard Jones and Rafael Lins. *Garbage collection: algorithms for automatic dynamic memory management*. John Wiley & Sons, Inc., New York, NY, USA, 1996.

[jsr06] JSR 121 application isolation API specification. `http://jcp.org/aboutJava/communityprocess/final/jsr121/index.html`, 2006.

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison Wesley Longman, Inc., second edition, 1999.

[Mic97] Sun Microsystems. Java Remote Method Invokation Specification. 1997.

[OSE04] OSEK/VDX Group. *OSEK Implementation Language Specification 2.5*. OSEK/VDX Group, 2004.

[OSE05] OSEK/VDX Group. *Operating System Specification 2.2.3*. OSEK/VDX Group, February 2005.

[SCC$^+$06] Doug Simon, Cristina Cifuentes, Dave Cleal, John Daniels, and Derek White. Java^{TM} on the bare metal of wireless sensor devices: the Squawk Java virtual machine. In *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments*, pages 78–88, New York, NY, USA, 2006. ACM Press.

[Sie04] Fridtjof Siebert. The impact of realtime garbage collection on realtime Java programming. *isorc*, 00:33–40, 2004.

[Sol00] Jose H. Solorzano. TinyVM - Java VM for Lego Mindstorms RCX. `http://tinyvm.sourceforge.net/`, 2000.

[Sun04] Sun Microsystems. JSR-000139 Connected Limited Device Configuration 1.1. `http://www.jcp.org/aboutJava/communityprocess/final/jsr139/`, 2004.

[Wil92] Paul R. Wilson. Uniprocessor garbage collection techniques. In *Proc. Int. Workshop on Memory Management*, number 637, Saint-Malo (France), 1992. Springer-Verlag.

[Zor90] Benjamin Zorn. Barrier methods for garbage collection. Technical Report CU-CS-494-90, University of Colorado, Boulder, Nov 1990.

List of Figures