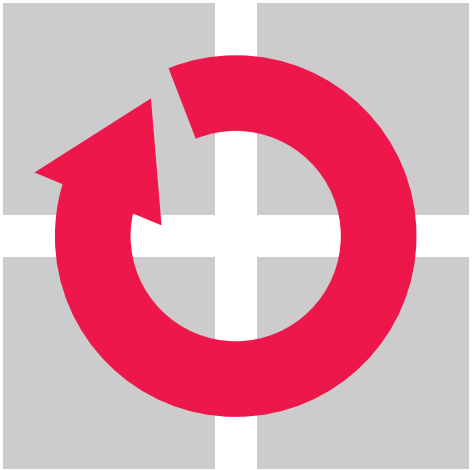
Software-basierter Speicherschutz durch spezialisierte Java-VMs auf Mikrocontrollersystemen

Christian Wawersich

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme
Universität Erlangen-Nürnberg

wawersich@informatik.uni-erlangen.de
http://www4.informatik.uni-erlangen.de/~wawi/



Friedrich-Alexander-Universität
Erlangen-Nürnberg

TECHNISCHE FAKULTÄT

Überblick

- Motivation
- Softwarebasierter Speicherschutz mit Java
- Migration von Betriebssystem-Konzepten
- Spezialisierte Java-VMs
- Zusammenfassung

Überblick

- **Motivation**

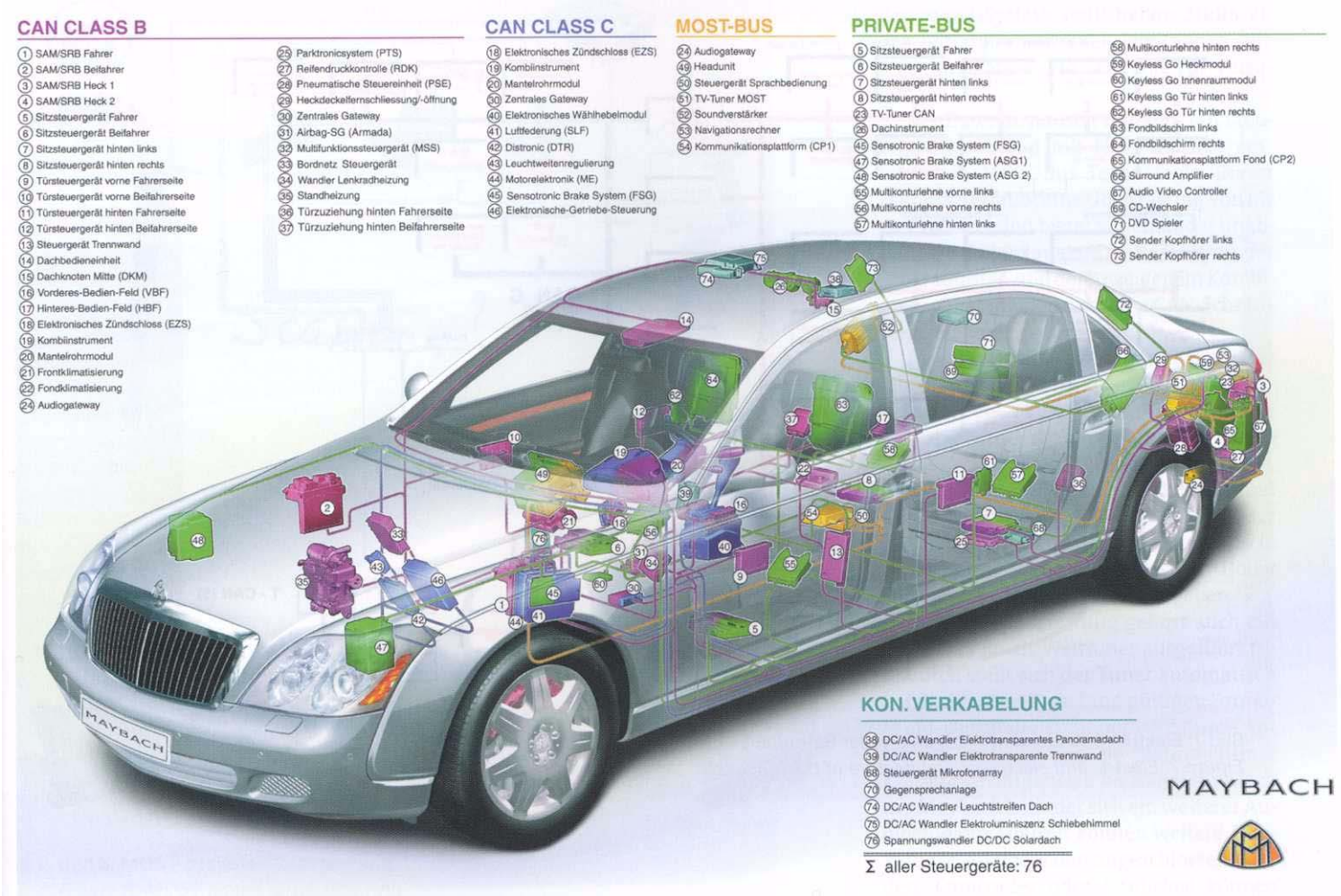
  ➔ Warum Speicherschutz und Java auf Mikrocontrollern?

- Migration von Betriebssystem-Konzepten
- Softwarebasierter Speicherschutz mit Java
- Spezialisierte Java-VMs
- Zusammenfassung

Anwendungsdomäne

- Hardware
  - 8 bis 32 Bit Systeme
  - 4 KiBi Byte bis einige MeBi Byte Speicher
  - MMU: nicht vorhanden oder heterogene Konzepte
- Betriebssysteme
  - über 50% Speziallösungen
  - OSEK/VDX im Automobilbereich
- Programmiermodell
  - meistens Assembler und C
  - Speziallösungen

Mikrocontroller im Automobil



Probleme

- Vielzahl der Mikrocontroller
  - Steuergerät und Software aus einer Hand
  - “Kapselung” durch Hardware
- Heterogene Mikrocontroller (Hardware und Software)
- Unterschiedliche Hersteller

➔ Integration und Variantenvielfalt führt zu Problemen

Folgen

RÜCKRUF VON 1,3 MILLIONEN AUTOS

Blamage für Benz (Spiegel 31.03.2005)

Mercedes ruiniert seinen Namen (TAZ 02.04.2005)

Größte Rückrufaktion für mangelhafte Autos in der Geschichte des Unternehmens. Fehler bei Software ...

- Hohe Kosten
- Verlust an Reputation
- Gefahr für den Menschen

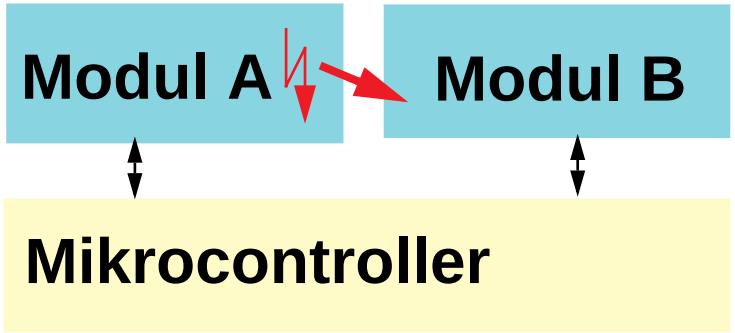
Lösungsansatz: Reduktion

- Verzicht auf Funktionalität
- Weniger, aber leistungsfähigere Mikrocontroller
  - Funktionen zusammenfassen
  - Software unterschiedlicher Hersteller

➔ Verlagerung der Integrationsprobleme

Integrationsproblem

- Verlust der Kapselung
  - Fehlerausbreitung
  - unklare Verantwortlichkeit



- Steigende Komplexität einzelner Geräte
- Heterogene Software

➔ Neue Probleme, aber eigentlich bekannte Probleme

Lösung: Migration von Konzepten

- Kapselung durch Speicherschutz
  - hardwarebasierter Speicherschutz
  - softwarebasierter Speicherschutz
- Hochsprachen und OOP
  - Ada, Java, C#, C++ ...
- ➔ Java
  - OO-Hochsprache
  - softwarebasierter Speicherschutz

Überblick

- Motivation
- **Softwarebasierter Speicherschutz mit Java**
- Migration von Betriebssystem-Konzepten
- Spezialisierte Java-VMs
- Zusammenfassung

Java für eingebettete Systeme

- Java 2 Micro Edition (J2ME)
  - angepasste Standard-Bibliotheken
  - angepasste JVM
  - Erfolgsgeschichte im Infotainmentbereich
    32 Bit CPUs und >32 MeBi Byte Speicher
- JavaCard
  - hoher Sicherheitsbedarf
  - stark angepasste VM
  - geeignet für kleinste eingebettete Systeme
    8-16 Bit CPU und > 4 KiBi Byte Speicher

Bietet Java eine ausreichende Kapselung?

- Globale Klassenvariablen
- Hardwarezugriff über unsicheres JNI
- Wandern von Objektreferenzen und Threads
- Fehlende Ressourcenverwaltung
  - gemeinsamer Speicherverwaltung
  - gemeinsame Rechenzeit

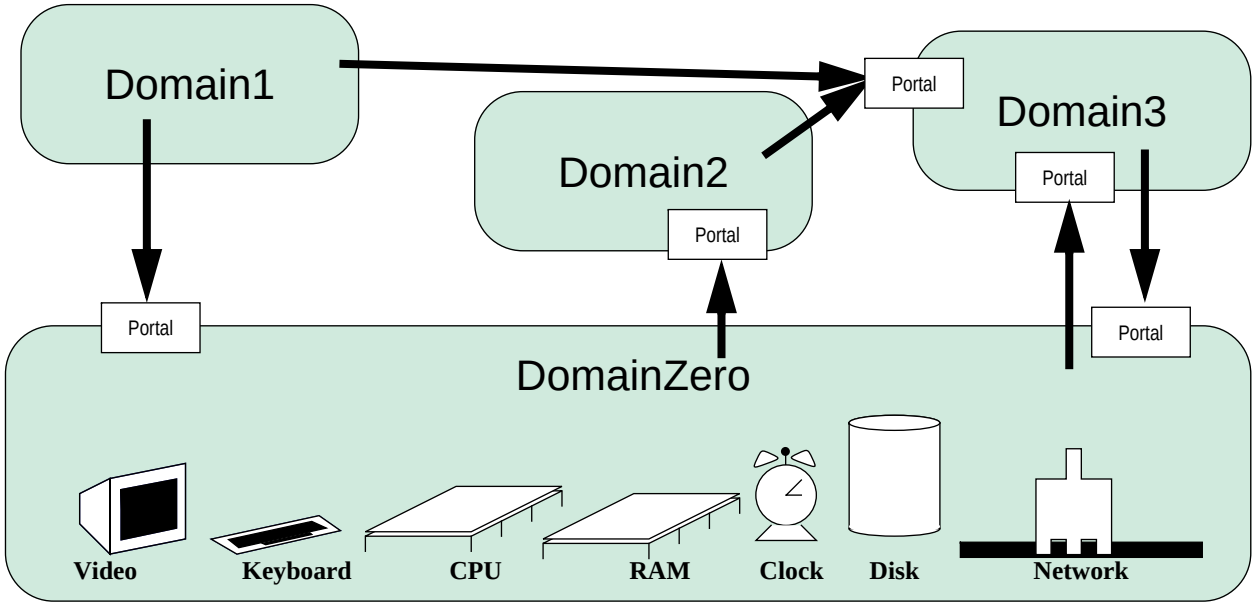
→ Java-Speicherschutzkonzept alleine greift zu kurz

Überblick

- Motivation
- Softwarebasierter Speicherschutz mit Java
- **Migration von Betriebssystem-Konzepten**
- Spezialisierte Java-VMs
- Zusammenfassung

JX: Systemarchitektur

- Kooperierende & isolierte *Domains* (100% Java)
  - Ressourcenverwaltung pro Domain möglich
- DomainZero repräsentiert Minimal-Kern (C/Asm)
- Kommunikation über *Portale*
  - vollständige Entkopplung (Referenzen, Threads, Speicher)



JX: Stand

- Architekturen
  - x86-kompatible PC
- Bytecode-Übersetzer
  - IA32, H8, ANSI C
- Hardware
  - IDE-Platten
  - Netzwerkkarten (NE2000, 3COM, EE100)
  - CAN-BUS

JX für Mikrocontroller geeignet?

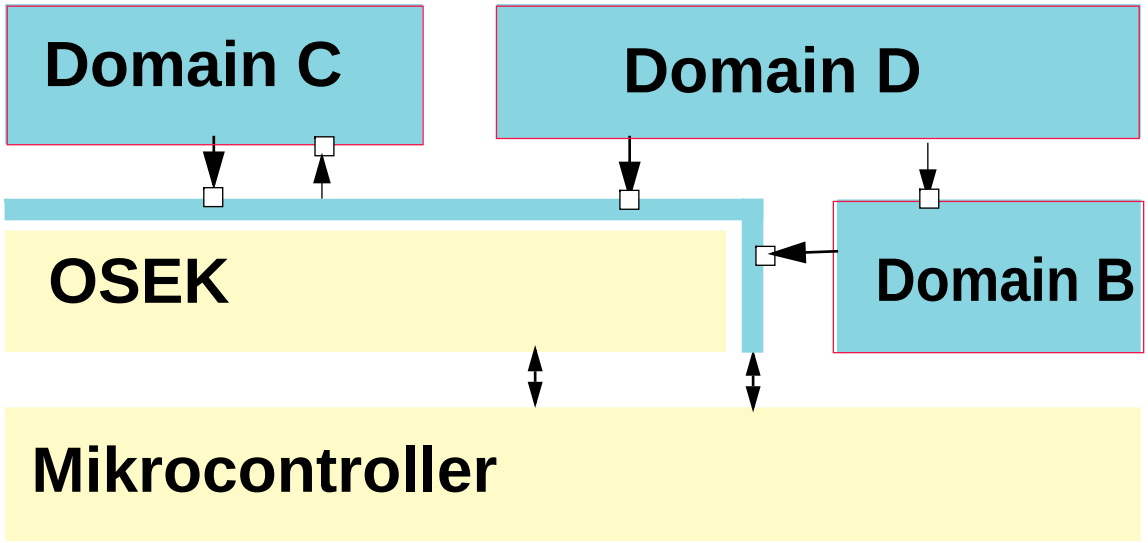
- Lego MindStorm prototypische Portierung
  32 KiBi Speicher, 16 Bit CPU
- 70-100 KiBi Byte DomainZero
- Grosse Verwaltungsstrukturen
  - Dynamisches Laden von Klassen
  - automatische Speicherbereinigung
  - Typinformationen über alle Klassen

➔ Migration von JX-Speicherschutz-Konzepten

  - Idee: Kombination mit OSEK-ähnlichen Systemarchitekturen

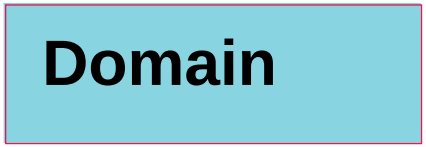
JX+OSEK: Architekturüberblick

- Bestehendes OSEK-System nutzen
- Architekturkonzept von JX übertragen



JX+OSEK: Programmiermodell

- Domain kapselt Anwendung
  - Speicherschutz
  - Ressourcenschutz
  - jede Domain ist eine eigenständige JVM
- Kommunikation über Portale
  - Hardwarezugriff
  - OSEK-Dienste
  - Inter-Domain-Kommunikation



Vergleich: Middleware-Ansatz im AutoSAR-Projekt !

Überblick

- Motivation
- Softwarebasierter Speicherschutz mit Java
- Migration von Betriebsystem-Konzepten
- **Spezialisierte Java-VMs**
- Zusammenfassung

Globale Sicht zum Übersetzungszeitpunkt

- Statische Konfiguration der gesamten Software
- Übersetzung vor der Laufzeit
  - kein dynamisches Laden von Klassen
  - Verzicht auf Interpreter oder JIT
  - Verzicht auf Verifikation zur Laufzeit
- Erzeugen der OSEK-Konfiguration
- Feedback
  - was kostet welche Operation?

Spezialisierung der JVM

- Spezialisierung auf Bytecode-Ebene
  - 8, 16, 32-Bit
  - Fliesskomma-Operationen
  - mehrdimensionale Arrays ...
- Spezialisierung von Portal-Aufrufen
- Spezialisierung der Speicherverwaltung
- Exceptionhandler, Synchronisation, ...

Spezialisierung von Portal-Aufrufen

- Berücksichtigung von OSEK-Konzepten
  - kein Thread-Konzept
  - Interrupts, Tasks und Events

➔ asynchrone Nachrichten?

- Primitive Portal-Aufrufe
  - primitive Argumente und Portalreferenzen

➔ Keine Kopie der transitiven Hülle eines Objekts

Spezialisierung von Portal-Aufrufen (2)

- Erzeugen von spezialisiertem Maschinencode
  - Memory-Objekt in JX
  - Übersetzer kennt die Implementierung der Klasse

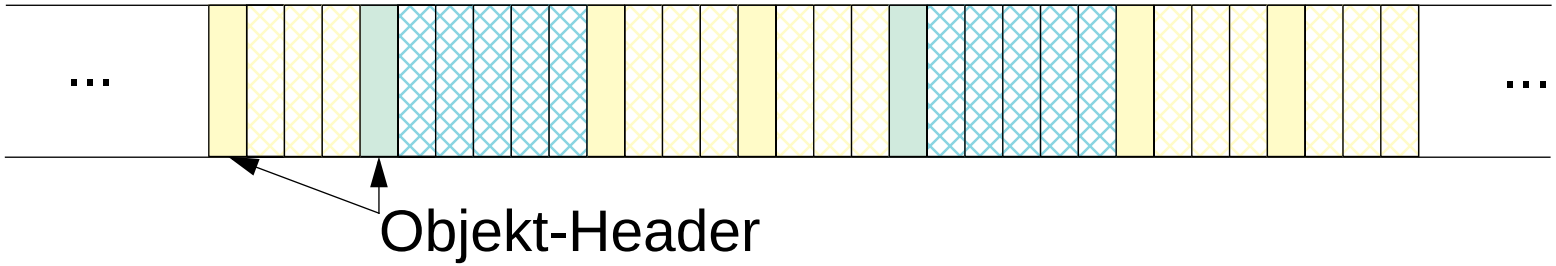
➔ keine Thread-/Task-Umschaltung

Spezialisiertes Speicherverwaltung

- Nur statische Objekte
  - Objekterzeugung im Klassen-Konstruktor
  - keine Verwaltung zur Laufzeit
  - minimaler Platzbedarf
- Nur wenige dynamische Typen von Objekten
  - spezialisiertes Speicherlayout
  - kontrollierter Platzbedarf
- Viele dynamische Objekte
  - auswählbare Speicherbereinigung
  - Standardverhalten

Spezialisiertes Speicherlayout

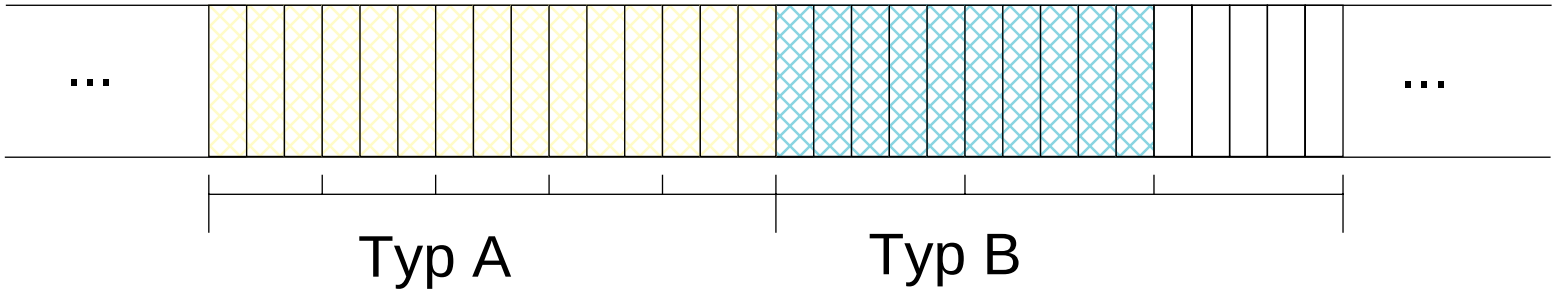
- Typischer Speicheraufbau



- Objekt-Header (4-12 Bytes)
  - Zeiger auf Typ-Informationen
  - Zeiger auf virtuelle Methodentabelle
  - Zeiger oder Flags für die Synchronisation
  - Flags für die Speicherverwaltung
  - Hash-Code

Spezialisiertes Speicherlayout (2)

- Spezialisiertes Speichelayout



- Keine Objekt-Header (0 Byte)
  - Typ-Information ist Position im Speicher
  - Hash-Code ist Objekt-Addresse
  - Synchronisation über Hashtabelle

Zusammenfassung

- Integrationsprobleme erfordern neue Konzepte
- Lösung: Migration von Konzepten
  - Speicherschutz
  - Hochsprachen
- Ansatz: Migration von JX-Konzepten
- Spezialisierung der Laufzeitumgebung
  - Spezialisierte Portal-Aufrufen
  - Spezialisierte Speicherverwaltung