Embedded Software in Java with KESO

HotSys-Praktikum

Michael Stilkerich

Friedrich-Alexander-Universität
Erlangen-Nürnberg

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

http://www4.informatik.uni-erlangen.de/~mike
mike@cs.fau.de

Outline

- Motivation
- KESO Overview
- Spatial Isolation
- Programming Device Drivers
- Homework: Highstriker in KESO

Java instead of C(++)? Why?

- many programming errors are detected...
  - `NULL` pointer dereferences
  - array out-of-bounds accesses

```
void foo(char *str) {
   char buffer[100];
   strcpy(buffer,str);
}
```

- ...or avoided
  - dangling pointers (garbage collection instead of malloc/free)

```
free(foo);
int *p=malloc(sizeof int);
foo->bar = 5;
```

- control flow integrity
  - no buffer overflows
  - no explicit indirect jumps

```
void foo(char *str) {
   void (*fn)() = ...;
   fn();
}
```

- type safety paves the way for
  - state migration
  - spatial isolation of applications (without hardware support)
  - ...

But isn't Java is too big/slow for tiny MCUs?

- dynamic class loading
  - fully-featured Java runtimes (cf., J2ME configurations)
  - provides a standard execution platform for portable code
- interpreter, just-in-time compiler
  - footprint
  - execution time, "warm-up" time
- dynamic linking
  - footprint
- reflection
  - footprint
  - analyzability

Java in Embedded Systems

- subset class library and the Java Virtual Machine
- custom class file formats
- portability-focused approaches widely spread (WORA)
- Example: Java 2 Micro Edition (J2ME)
  - Configurations and Profiles
  - Configurations: Target device classes (~ available resources)
    - Connected Device Configuration (CDC):
      32-bit CPU, 2MB RAM, 2.5MB ROM
    - Connected Limited Device Configuration (CLDC):
      16/32-bit CPU >= 16MHz, 192 kB RAM, 160 kB ROM
  - Profiles: Application domain (needed functionality, APIs)
    - MIDP (Mobile Information Device Profile)
    - Personal Profile

Statically-Configured Embedded Systems

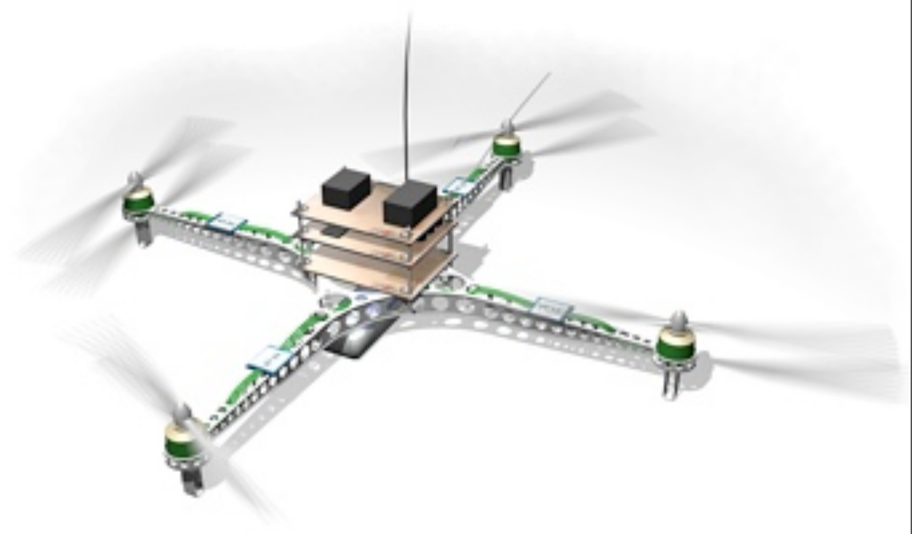
- comprehensive a-priori knowledge
  - code
  - operating system objects (threads, locks, events, ...)
- significant subset of the embedded systems market
  - real-time systems
  - safety critical systems
  - appliances
- often mass products
  - cost pressure
  - choose the smallest possible hardware platform
  - operating systems tailored to the application
- portable profiles contain unneeded functionality

KESO: Idea

- JVM tailoring (instead of fixed configurations)
  - automatic selection of Java features by static application analysis
  - static applications, no dynamic class loading
  - no Java reflection
- ahead-of-time compilation to C
  - VM bundled with application
- scheduling/synchronization provided by underlying OS
  - currently AUTOSAR/OSEK OS
  - accustomed programming model remains
  - no dynamic creation of threads!

Performance: KESO vs. C

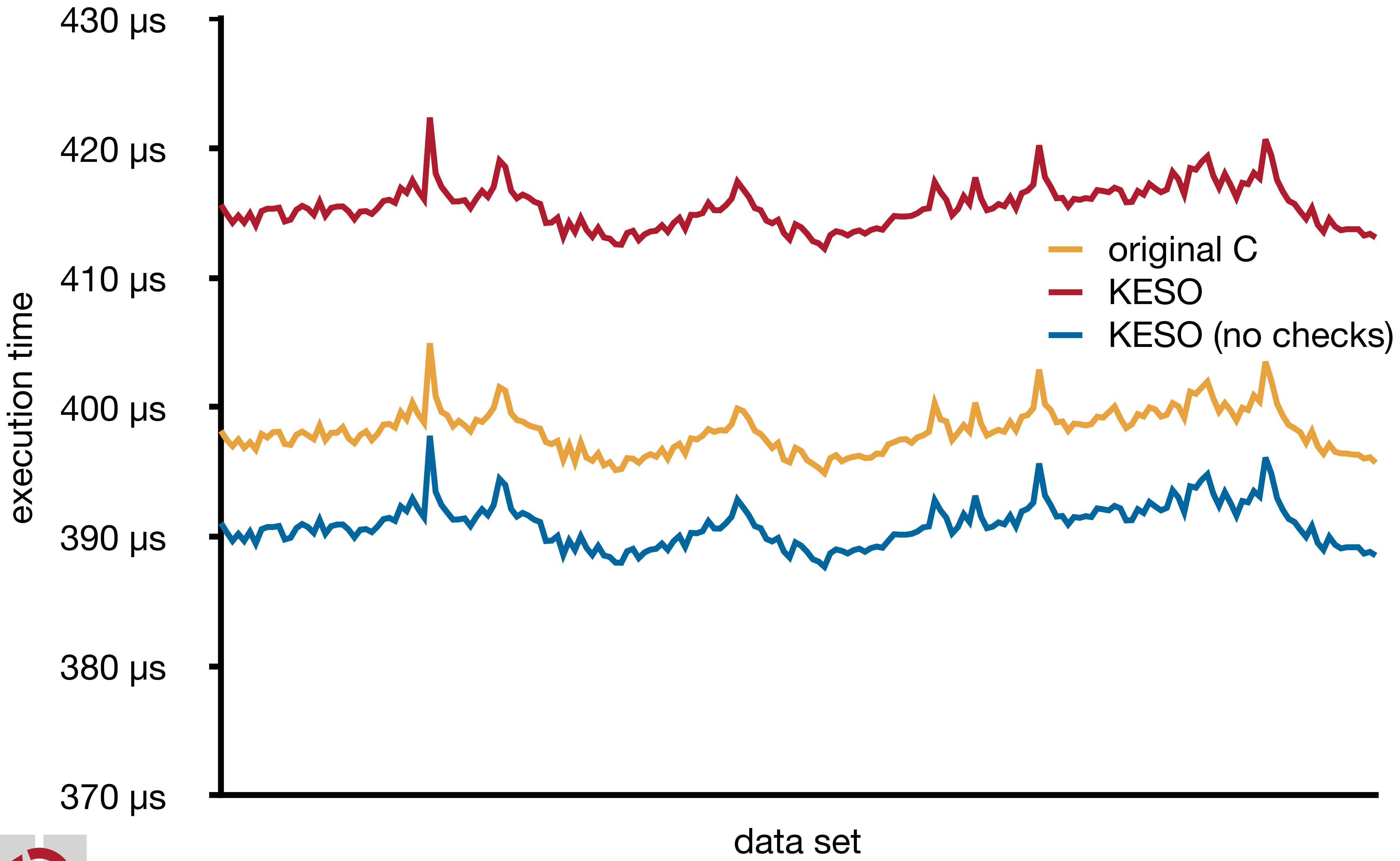
- Flight Attitude Control Algorithm of the I4Copter
  - http://www4.cs.fau.de/Research/I4Copter
  - C Code generated from a Simulink Model
  - Input: sensor and steering data
  - Output: engine thrust levels
  - Tricore TC1796b@150 MHz (1 MiB MRAM)



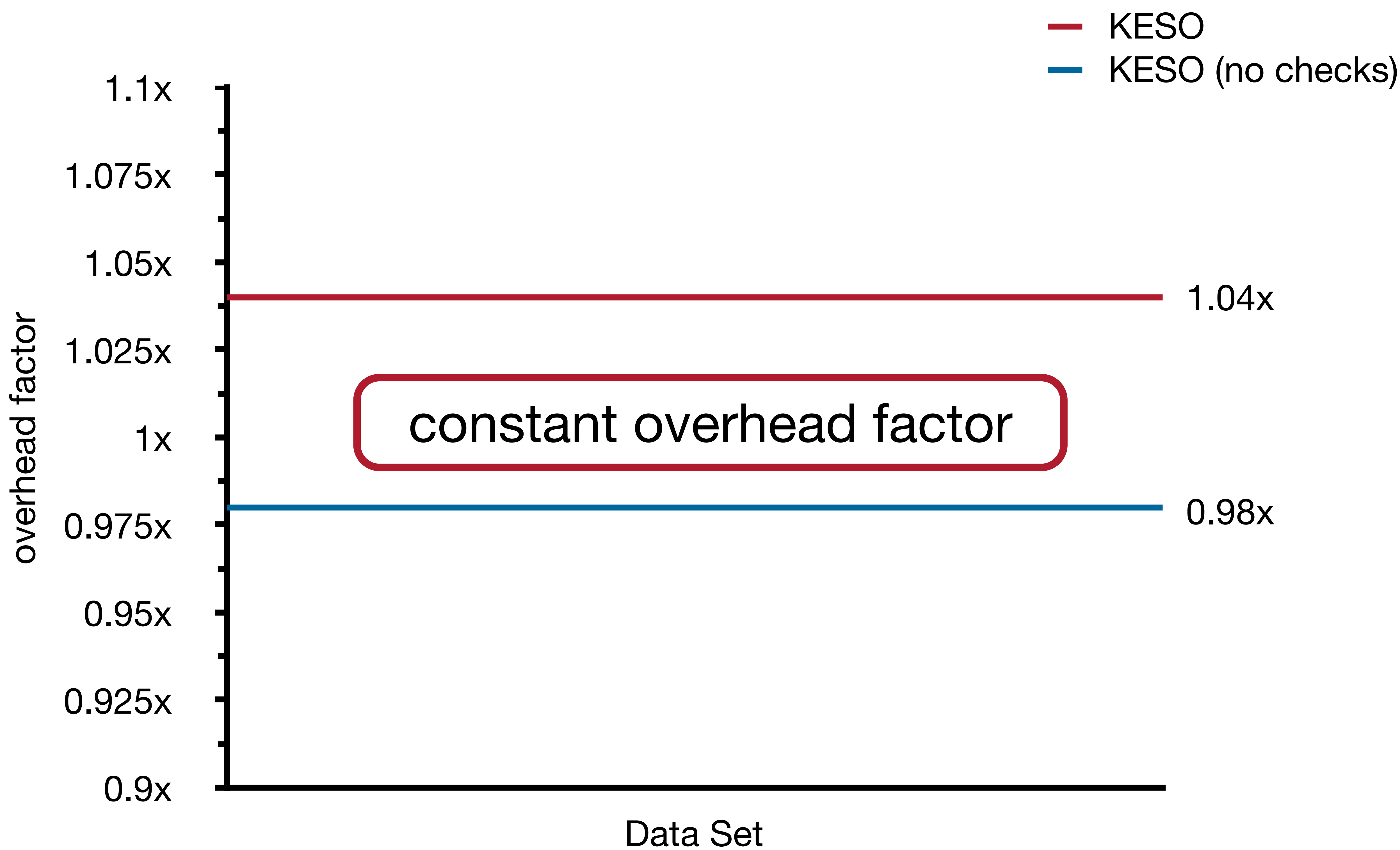
- Java port close to the C version

- Recorded trace of inflight sensor and steering data
  - Verified that C and Java version output the same actuator values
  - Replayed 200 data samples to measure execution time

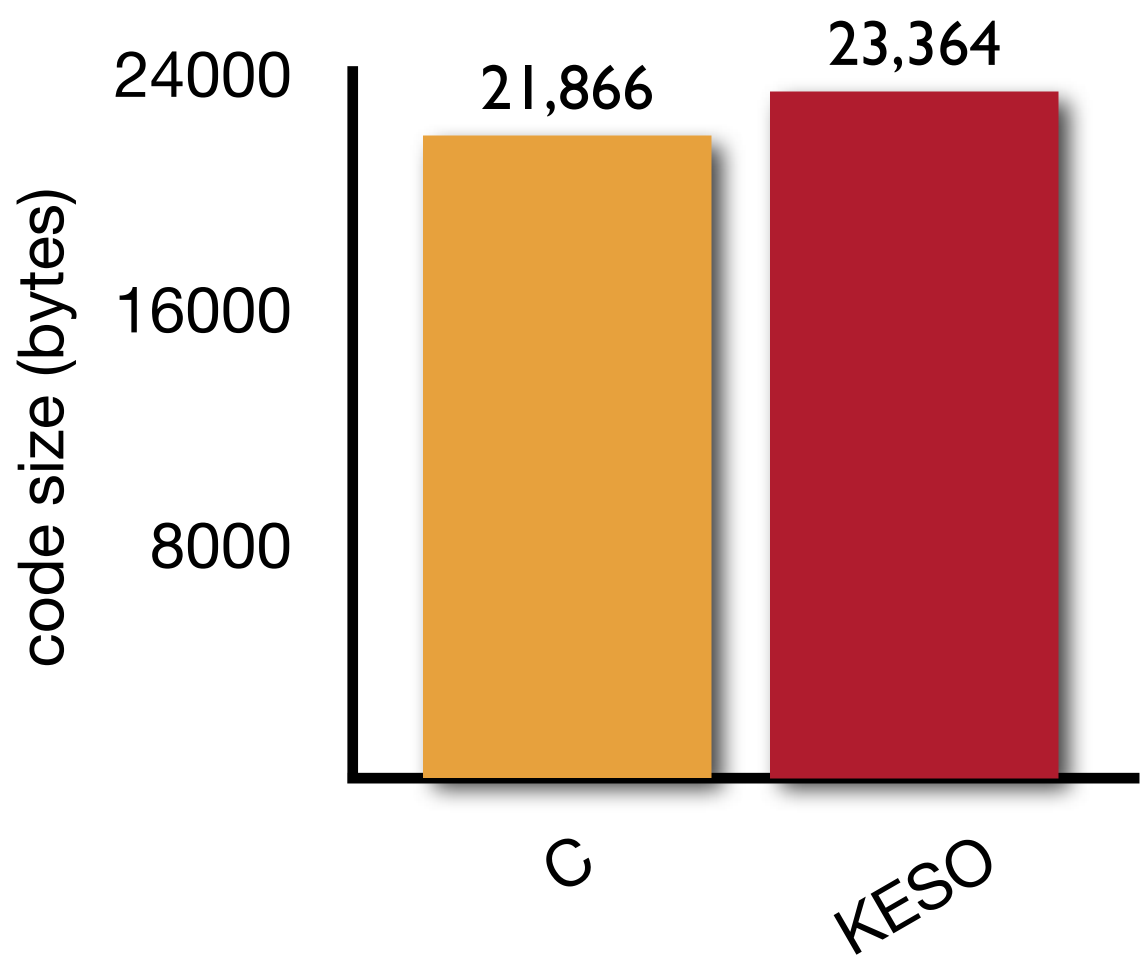
Attitude Controller - Execution Times

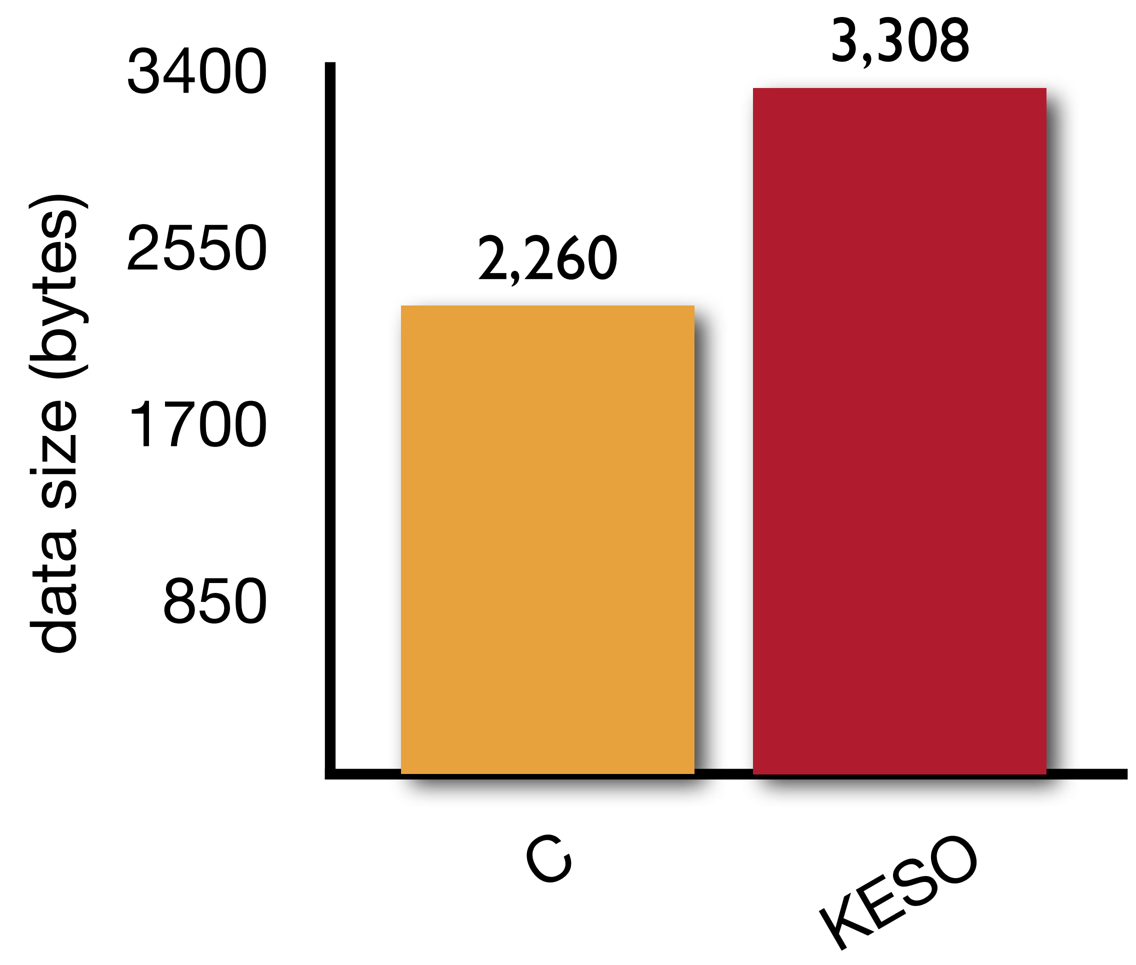


Attitude Controller - Execution Times



Attitude Controller - Footprint

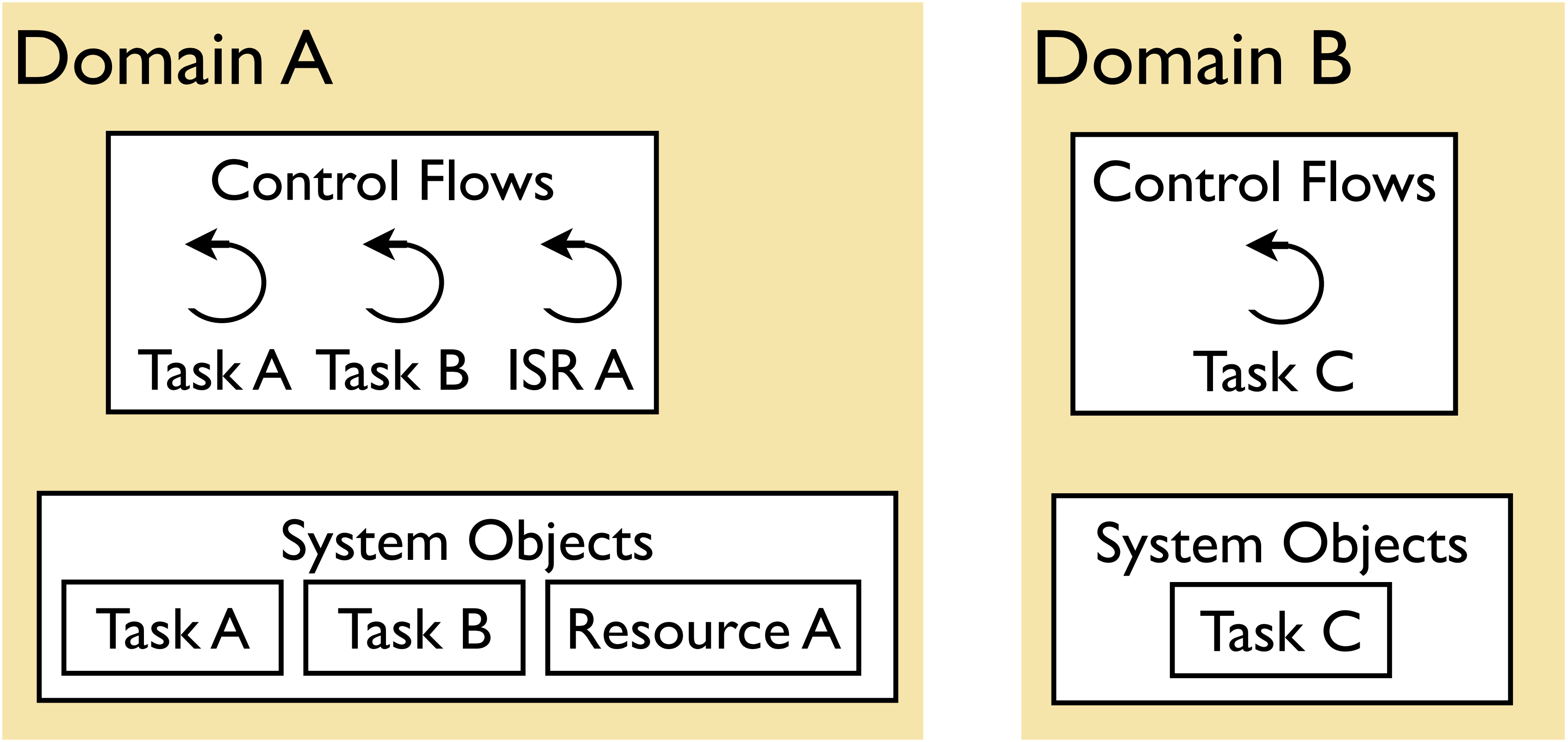




- 930b (5.4%) KESO runtime
- 586b (3.5%) checks
- 1.5k (6%) overhead to C

- 1k heap
- 20b system objects

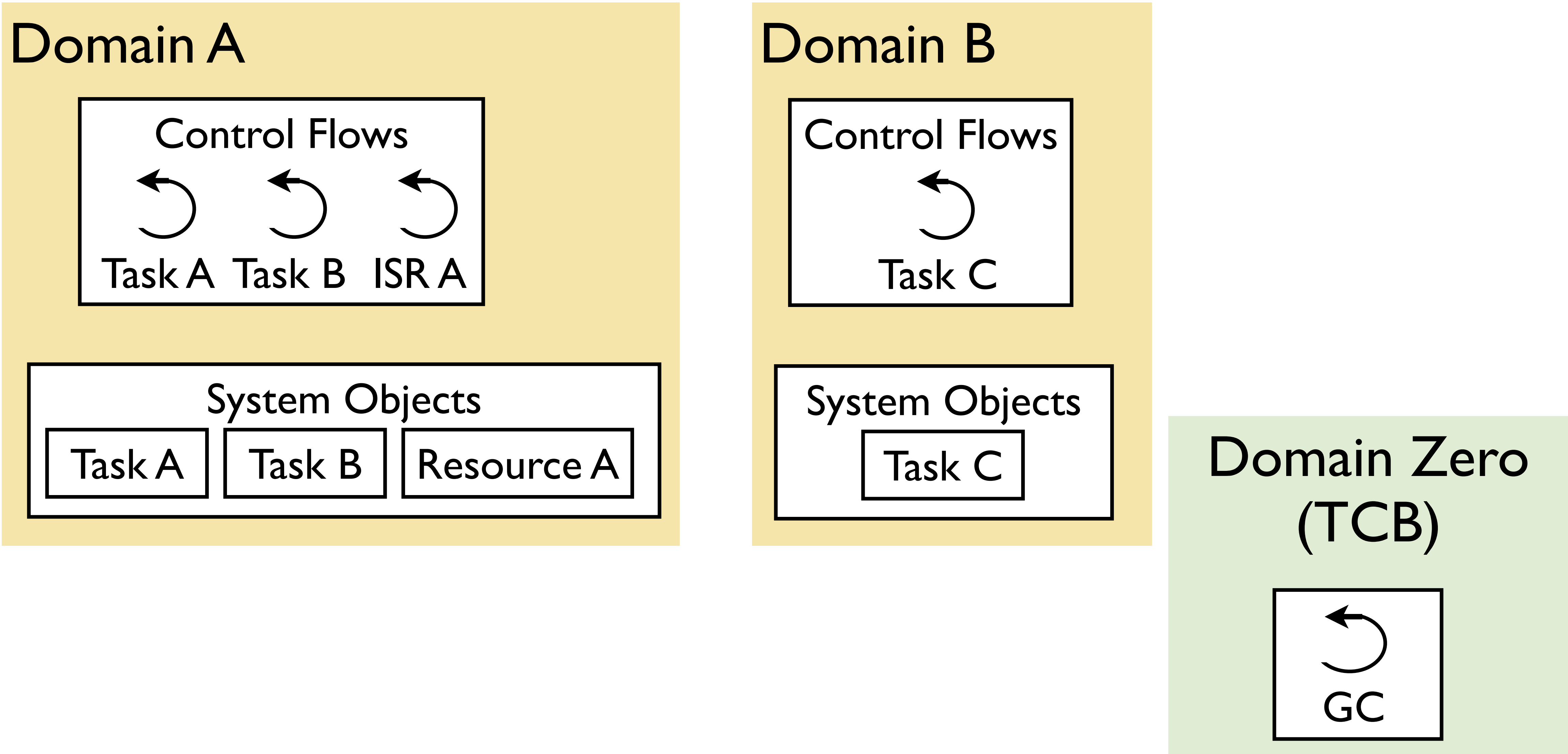
KESO: Architecture



Domains: realms of {memory,service}protection

- containers for control flows and system objects
- appear as a separate JVMs to the application

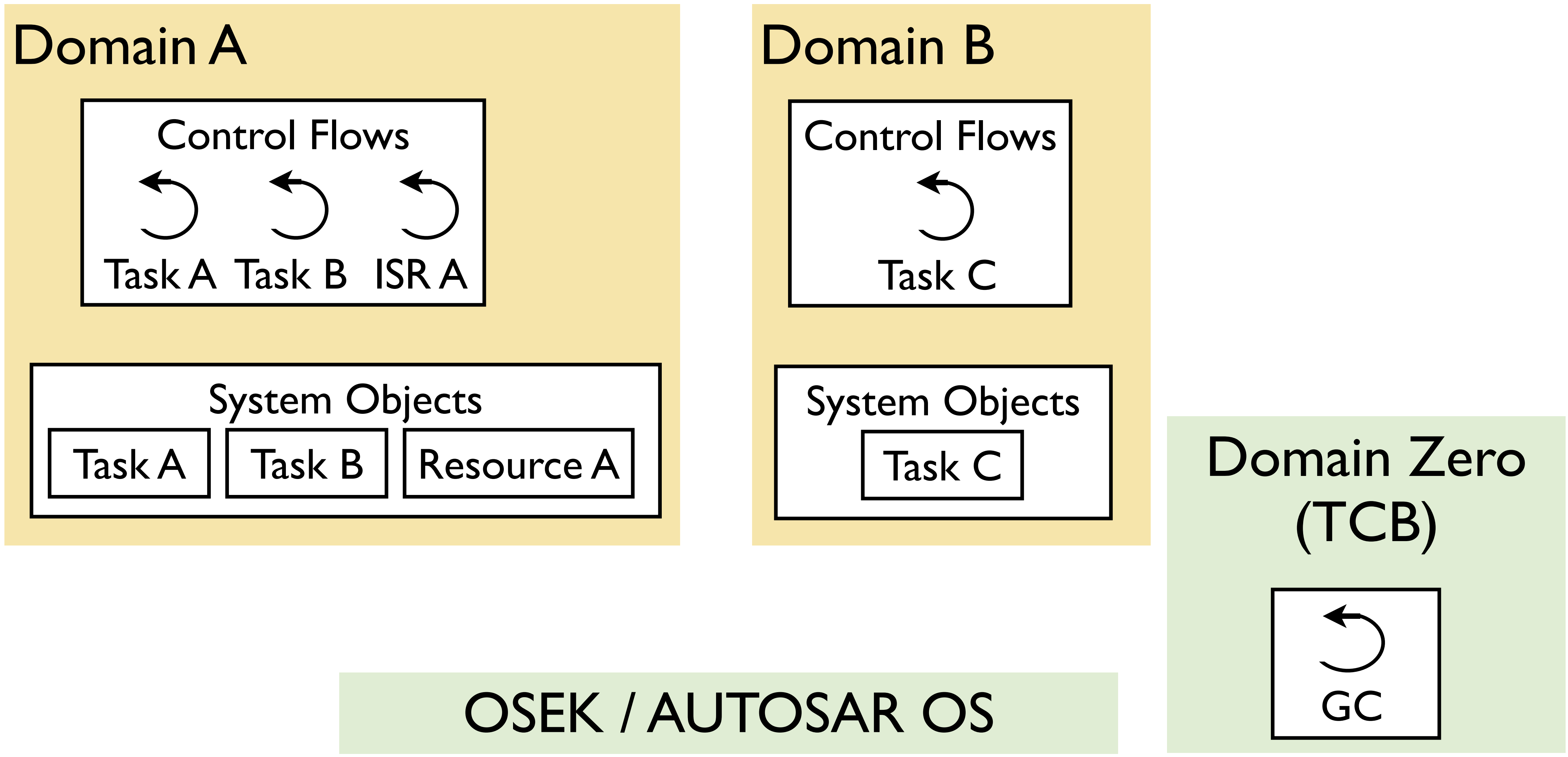
KESO: Architecture



Domain Zero

- trusted control flows of KESO's runtime environment
- currently only the garbage collector

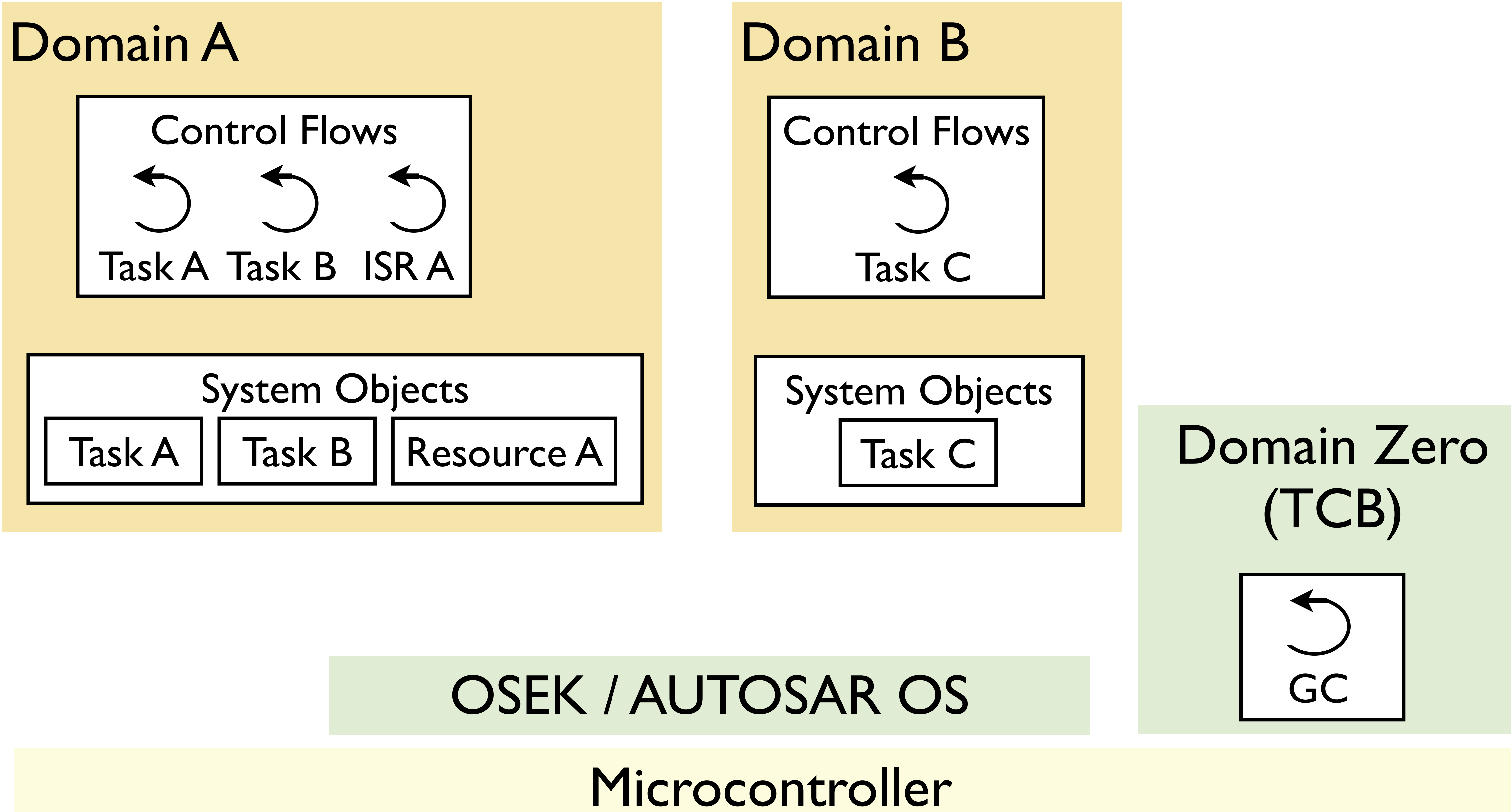
KESO: Architecture



OSEK / AUTOSAR OS

- provides threading/scheduling facilities
- temporal isolation, hardware-based spatial isolation

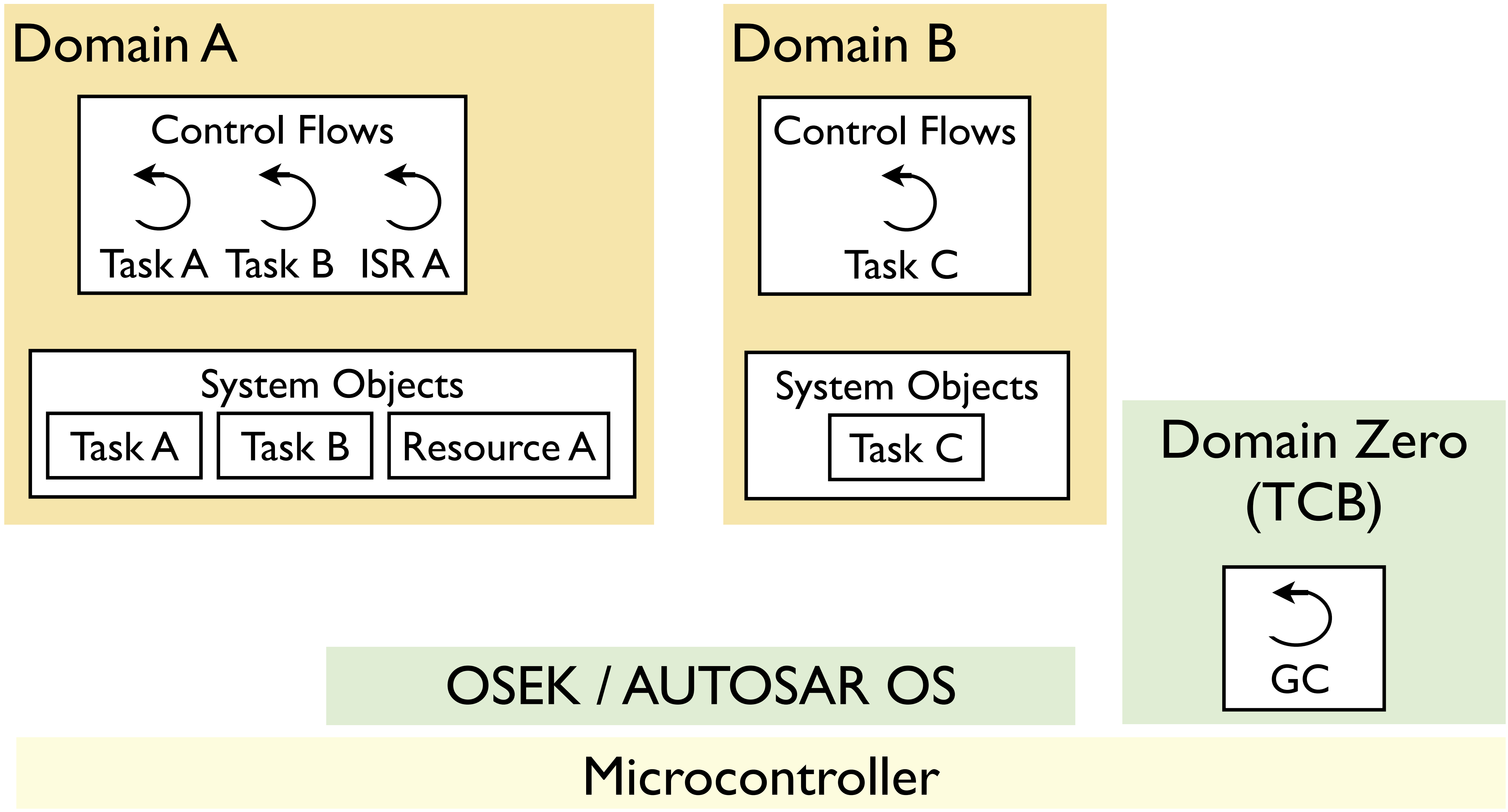
KESO: Architecture



Typical Targets

- low-end: 8-bit AVR (ATmega8535, 8K ROM, 512b RAM)
- higher-end: 32-bit Tricore (TC1796, 2M ROM, 256K RAM)

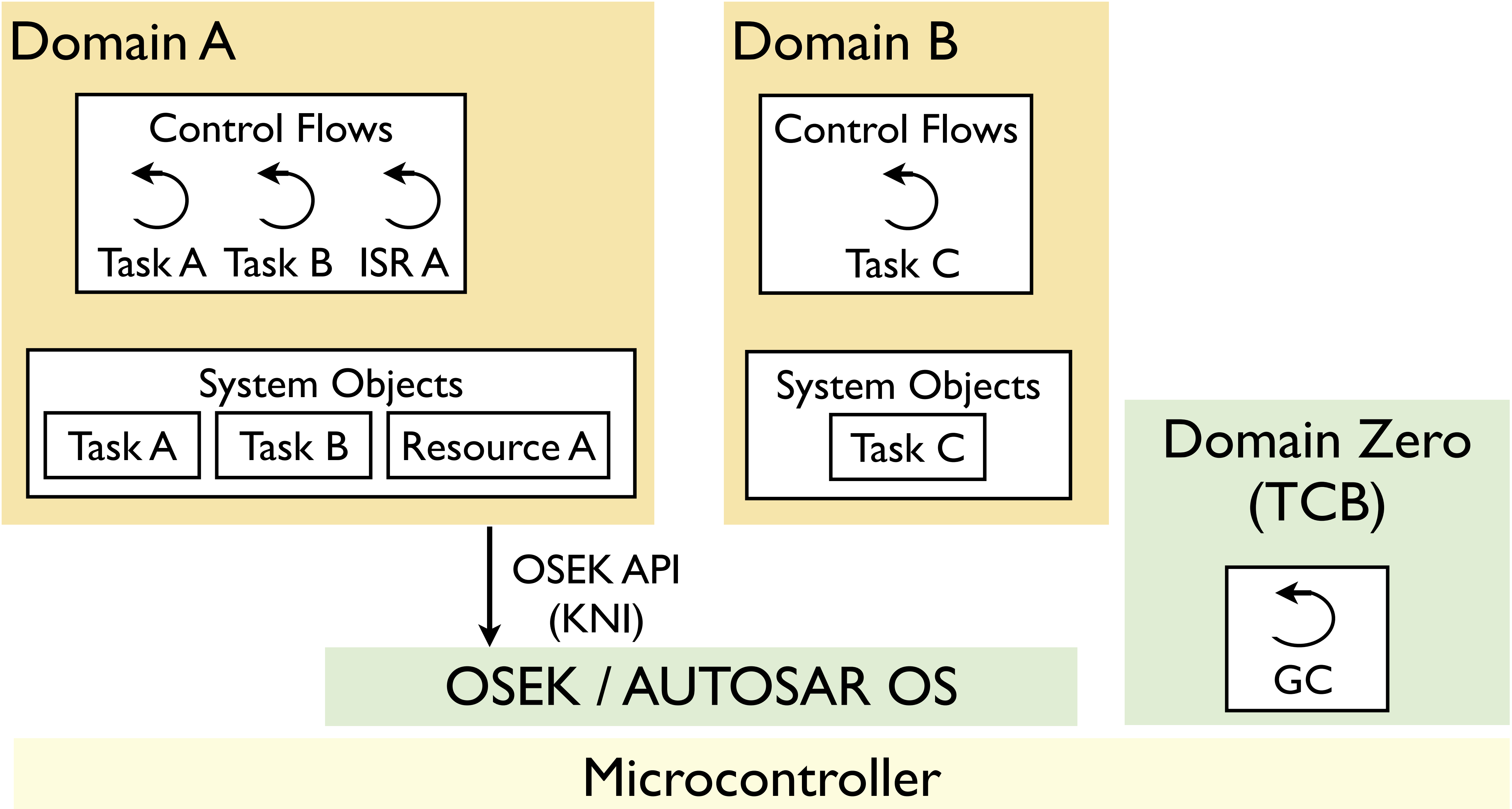
KESO: Architecture



KESO Native Interface (KNI)

- aspect-oriented mechanism for unsafe interactions
- full access to the internal state of Java-to-C compiler

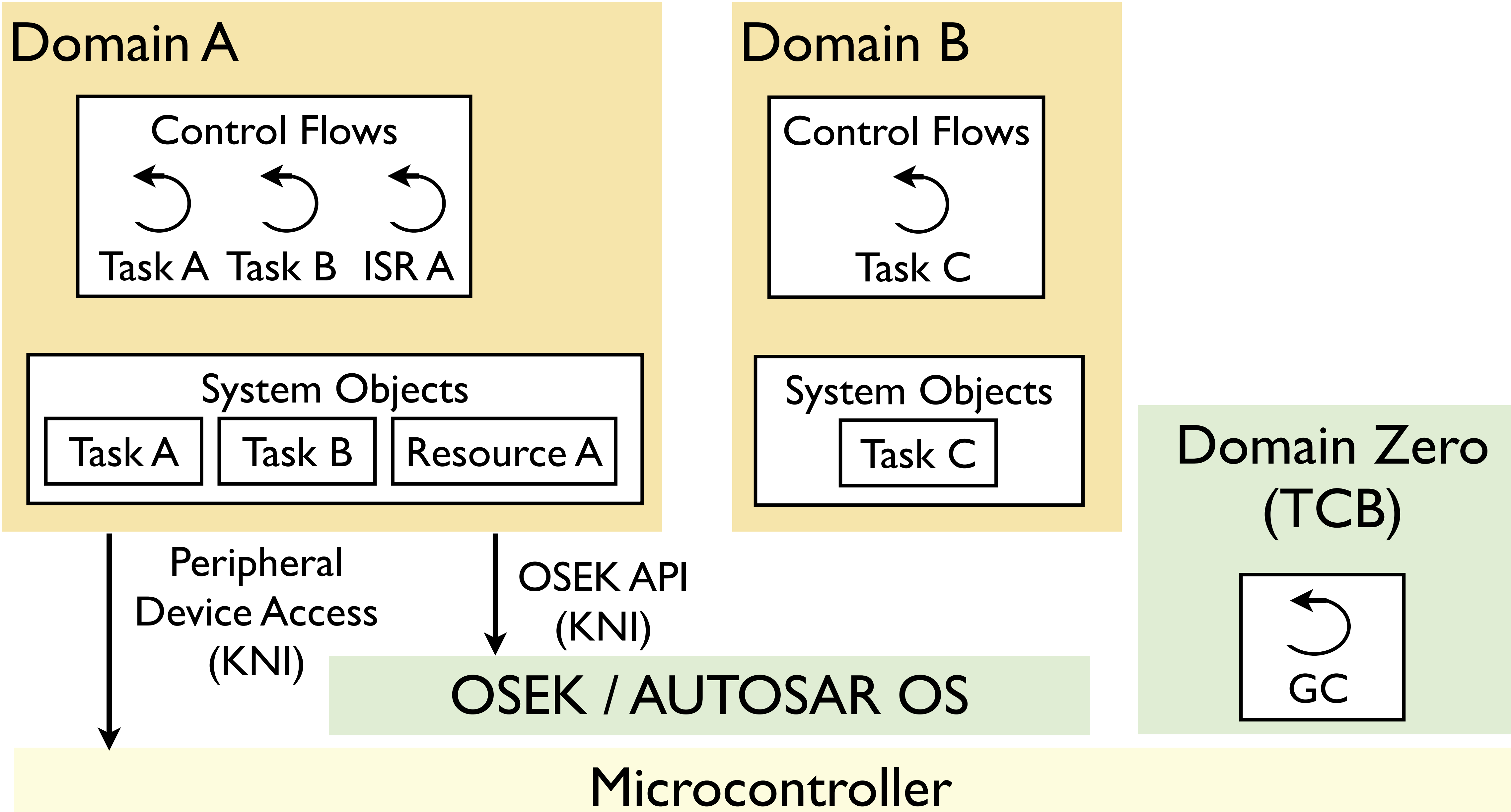
KESO: Architecture



OSEK Java API

- access to OSEK / AUTOSAR system services
- language-based service protection

KESO: Architecture



Peripheral Device Access

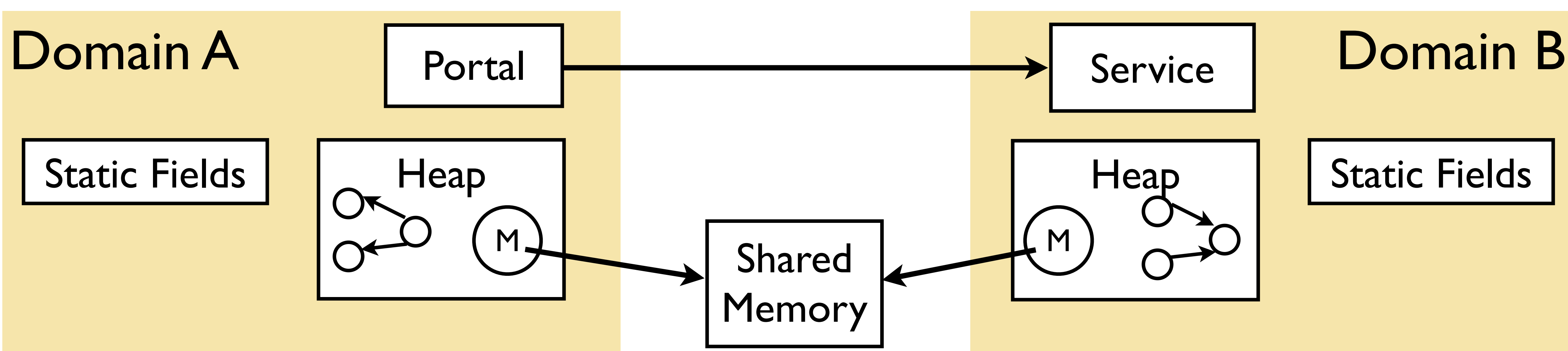
- RawMemory (similar to RTSJ)
- Memory-mapped objects (similar to C structs)

KESO: Unsupported Java Features

- dynamic class loading
- dynamic thread creation (no `new Thread()`)
- `synchronized`
  - instead explicit use of OSEK/VDX Resources (priority ceiling)
- exception handling
  - exceptions considered fatal errors (program stops)
- reflection
- standard class library
  - core functionality of CLDC 1.1 provided

Spatial Isolation

- inhibit shared data among different domains
- own set of static fields in each domain
- logical heap separation (no cross-domain references)
  - current implementation: heaps physically separated
- Inter-domain Communication
  - Shared Memory (≈ RawMemory with reference counting)
  - Portals (RMI-like mechanism)

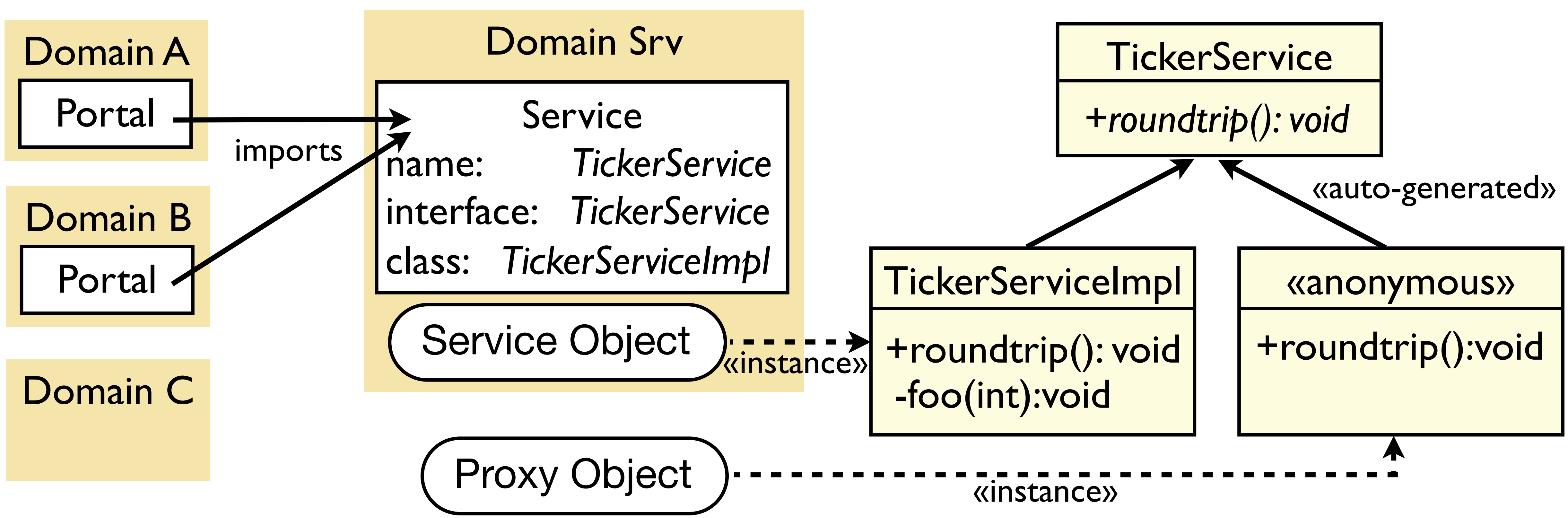


Memory Management

- physical heap separation
  - static partitioning of the available memory
  - each domain is allocated a fixed-size heap
- choose from different heap management strategies
  - no garbage collection (*RestrictedDomainScope*)
  - stop-the-world GC (*CoffeeBreak*)
  - incremental GC (*IdleRoundRobin*)
- domains may use different heap strategies
  - currently the two GCs cannot co-exist in one system
- GC is performed individually for each domain

Inter-Domain Communication with Portals

- service domain: exports interface as named service
  - service object is allocated in the service domain
  - proxy object (portal) is statically allocated for the client domains
- client domains: statically import this service
  - client domains acquire proxy object reference via name service
  - other domains cannot access the service at runtime



Portals: Parameter Passing

- strictly call-by-value
  - retain logical heap separation
- reference parameters
  - deep copy to the service domain's heap
  - GC needed in the service domain
- marker interface `NonCopyable` to prevent copying
  - reference replaced by `null`
  - used for system objects

Portals: Implementation and Overhead

```
public void foo() {
  TickerService srv = (TickerService)
        PortalService.lookup("TickerService");

  srv.roundtrip();
}
```

Name Service

- compiled to an array lookup
- returns
  - service object in service domain
  - portal object in client domains
  - `null` otherwise

Portals: Implementation and Overhead

```
public void foo() {
  TickerService srv = (TickerService)
        PortalService.lookup("TickerService");

  srv.roundtrip();
}
```

Portal Call

- regular virtual method call

Portals: Implementation and Overhead

```
public void foo() {
  TickerService srv = (Tick
        PortalService.looku

  srv.roundtrip();
}
```

Switch Protection Context

- backup current domain on stack
- change current execution context
- migrate task to service domain
- restore original domain on return

```
public void roundtrip_portal(object_t *proxy) {
   domain_t prev_domain = CURRENT_DOMAIN;
   CURRENT_TASK->effective_domain = DomainSrv_ID;
   CURRENT_DOMAIN = DomainSrv_ID;
   PUSH_STACK_PARTITION(DomainSrv_ID);
   roundtrip_impl(&tickerservice_srvobj);
   POP_STACK_PARTITION();
   CURRENT_DOMAIN = prev_domain;
   CURRENT_TASK->effective_domain = prev_domain;
}
```

Portals: Implementation and Overhead

```
public void foo() {
  TickerService srv = (TickerService)
        PortalService

  srv.roundtrip();
}
```

Partition Stack

- enables GC to skip irrelevant partitions
- only if service method potentially blocks

```
public void roundtrip_portal(object_t *proxy) {
   domain_t prev_domain = CURRENT_DOMAIN;
   CURRENT_TASK->effective_domain = DomainSrv_ID;
   CURRENT_DOMAIN = DomainSrv_ID;
   PUSH_STACK_PARTITION(DomainSrv_ID);
   roundtrip_impl(&tickerservice_srvobj);
   POP_STACK_PARTITION();
   CURRENT_DOMAIN = prev_domain;
   CURRENT_TASK->effective_domain = prev_domain;
}
```

Portals: Implementation and Overhead

```
public void foo() {
  TickerService sr
        PortalServ

  srv.roundtrip();
}
```

Invoke Service Method

- statically bound call
- service object is passed as `this` reference
- primitive parameters are passed through
- references are deep copied

```
public void roundtrip_portal(object_t *proxy) {
   domain_t prev_domain = CURRENT_DOMAIN;
   CURRENT_TASK->effective_domain = DomainSrv_ID;
   CURRENT_DOMAIN = DomainSrv_ID;
   PUSH_STACK_PARTITION(DomainSrv_ID);
   roundtrip_impl(&tickerservice_srvobj);
   POP_STACK_PARTITION();
   CURRENT_DOMAIN = prev_domain;
   CURRENT_TASK->effective_domain = prev_domain;
}
```

Portals: Implementation and Overhead

```
public void foo() {
  TickerService srv = (TickerService)
        PortalService.lookup("TickerService");

  srv.roundtrip();
}
```

Cost

- primitive parameters: same order of magnitude
- reference parameters: next order(s) of magnitude

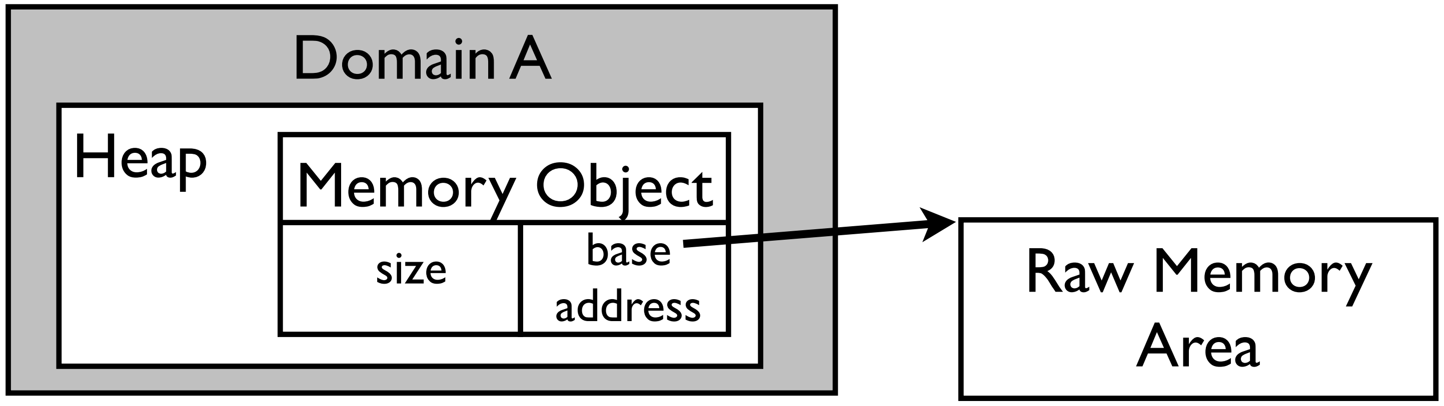
```
public vo
   domain
   CURRENT_TASK->effective_domain = DomainSrv_ID;
   CURRENT_DOMAIN = DomainSrv_ID;
   PUSH_STACK_PARTITION(DomainSrv_ID);
   roundtrip_impl(&tickerservice_srvobj);
   POP_STACK_PARTITION();
   CURRENT_DOMAIN = prev_domain;
   CURRENT_TASK->effective_domain = prev_domain;
}
```

Example: GPIO Port on 8-Bit AVR

- 8 pins, each selectively usable as input or output
- three 8-bit configuration registers per Port, e.g. PORTA
  - Data Direction Register (DDRA): I/O mode (0=input, 1=output)
  - Data Register (PORTA)
    - input PIN: configure pull-up resistor (1 = pull-up resistor activated)
    - output PIN: set output level (0 = low, 1 = high)
  - Input Pins (PINA, read-only): read the level of a pin
- registers are mapped to the data address space
  - PINA @0x19, DDRA @0x1a, PORTA @0x1b
- How do we access these registers from Java code?
  - RawMemory
  - Memory-mapped Objects

RawMemory

- Problem
  - access memory outside the managed areas (“raw memory”)
  - without impairing the soundness of the type system
- RawMemory (Real-Time Specification for Java (RTSJ))
  - proxy object enables access to a raw memory region
  - raw and managed memory areas must not overlap
  - read and write <u>primitive</u> data
  - KESO API: `keso.core.Memory`, `keso.core.MemoryService`



KESO's RawMemory API (excerpt)

```
package keso.core;

// all methods of this class are implemented by KNI
public final class Memory {

    private Memory() { /* instantiation by the RTE only */ }
    public int getSize() { return 0; }

    // getters
    public byte   get8(int offset) { return 0; }
    public short get16(int offset) { return 0; }
    public int   get32(int offset) { return 0; }

    // setters
    public void  set8(int offset, byte  value) { }
    public void set16(int offset, short value) { }
    public void set32(int offset, int   value) { }

    // common bit operations
    public void and32(int offset, int mask) { }
    public void xor16(int offset, int mask) { }
}
```

Example: 8-bit AVR IO-Port with RawMemory

```
import keso.core.*;
public final class PortA {
    private static final int BASE=0x19,PIN=0,DDR=1,PORT=2;
    private static final Memory regs =
       MemoryService.allocStaticDeviceMemory(BASE, 3);

    public static void setMode(int pin, boolean isOutput) {
        if(isOutput) {
            regs.or8(DDR, (1<<pin));
        } else {
            regs.and8(DDR, ~(1<<pin));
            writePin(pin, true);    // activate pull-up resistor
        }
    }
    public static void writePin(int pin, boolean level) {
        if(level) regs.or8(PORT,  (1<<pin));
        else regs.and8(PORT, ~(1<<pin));
    }
    public static boolean readPin(int pin) {
        return (regs.get8(PIN) & (1<<pin)) != 0;
    }
}
```

Memory-Mapped Objects

- RawMemory is flexible but inconvenient in simple uses
  - programmer needs to deal with explicit offsets
  - offsets need to be bounds-checked (same as arrays)

- Memory-Mapped Objects
  - memory layout defined by a class (similar to a C `struct`)
  - memory-mapped objects may contain regular and mapped fields
  - regular fields become part of the proxy object
  - accesses to mapped fields are redirected to the raw region

- Advantages
  - access fields by name rather than offsets
  - bounds check only once when creating the mapping

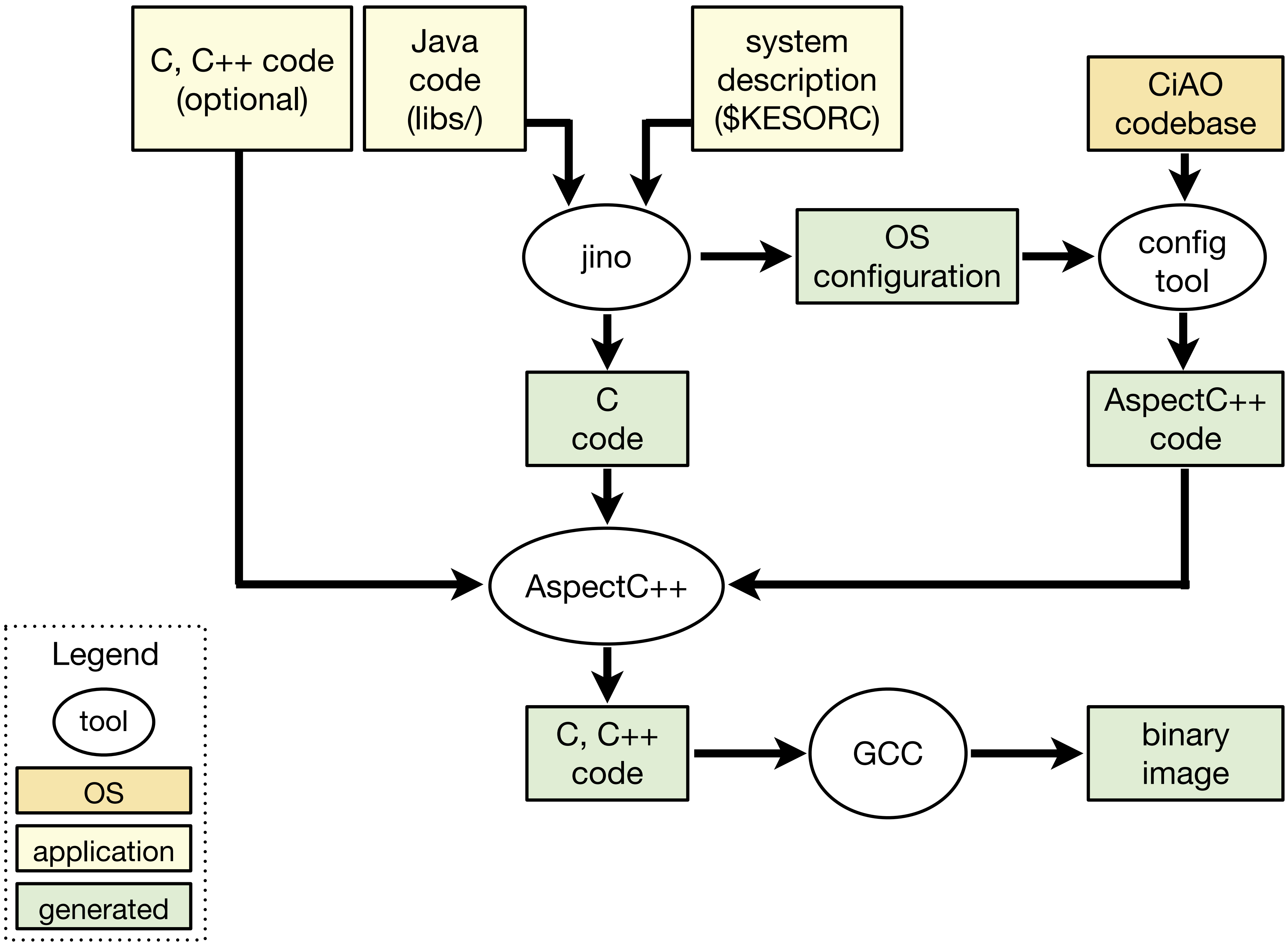
Example with Memory-Mapped Objects

```
package driver.avr;
import keso.core.*;
public final class AVRPort implements MemoryMappedObject {
   private MT_U8 PIN;   // offset 0
   private MT_U8 DDR;   // offset 1
   private MT_U8 PORT;  // offset 2

   public void setMode(int pin, boolean isOutput) {
        if(isOutput) {
            DDR.setBit(pin);
        } else {
            DDR.clearBit(pin);
            writePin(pin, true);    // activate pull-up resistor
        }
    }
    public void writePin(int pin, boolean level) {
        if(level) PORT.setBit(pin);
        else PORT.clearBit(pin);
    }
    public boolean readPin(int pin) {
        return PIN.isBitSet(pin);
    }
}
```

Example with Memory-Mapped Objects

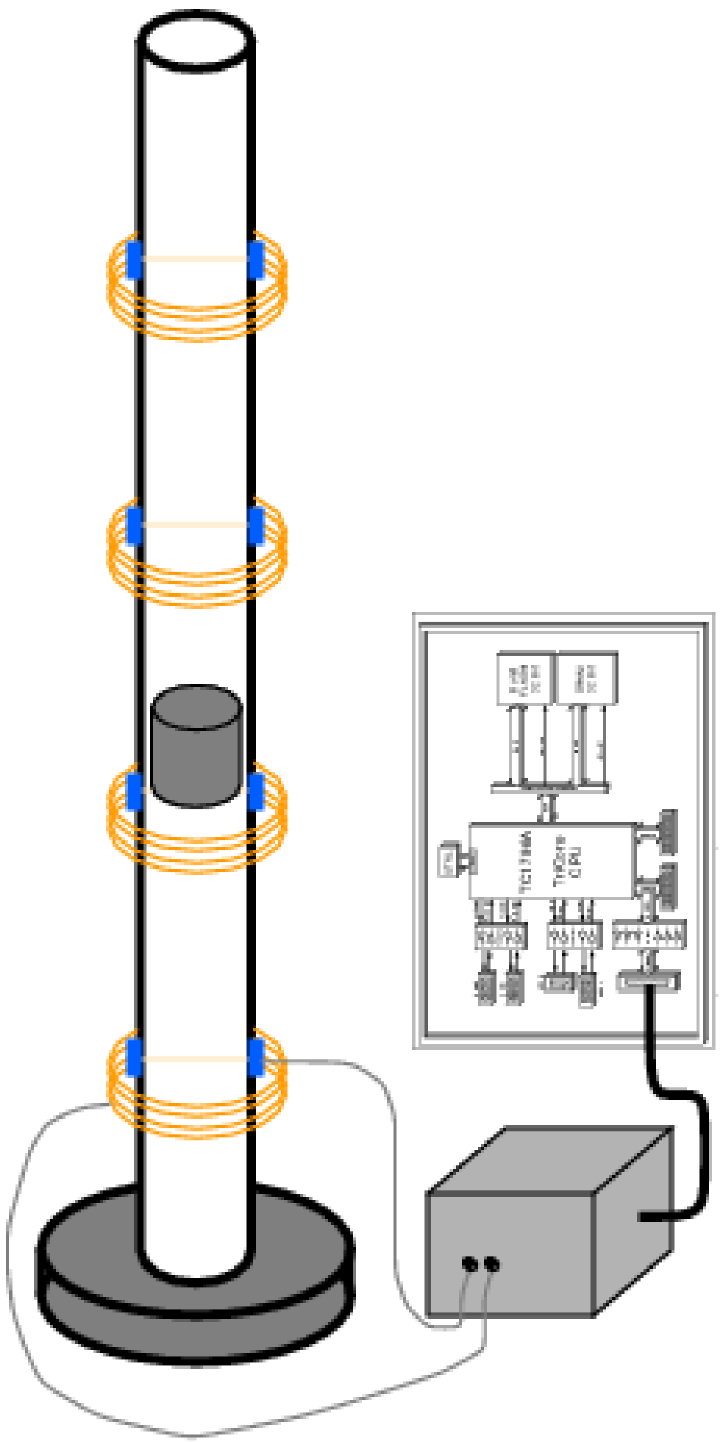
```
import keso.core.*;
import driver.avr.AVRPort;

public final class Foo {
   public void bar() {

      // create a new mapped object at base address 0x19
      AVRPort portA = (AVRPort)
       MemoryService.mapStaticDeviceMemory(0x19,“driver/avr/AVRPort”);

      /* Alternative: the mapping could also be created on the
                      base of a RawMemory object
      Memory regs= MemoryService.allocStaticDeviceMemory(0x19,3);
      AVRPort portA = (AVRPort)
       MemoryService.mapMemorytoStaticObject(regs,“driver/avr/AVRPort”);
      */

      // configure PIN 3 as output
      portA.setMode(3, true);
      // set output level of PIN 3 low
      portA.writePin(3, false);
   }
}
```

Toolchain with CiAO-Backend



Highstriker Exercise

- plexiglas tube containing an iron projectile
- TC1796b microcontroller
- 8 electric coils to move the projectile
  - GPIO Pins P0.5 - P0.12
- 1 photosensor next to each coil
  - to track the projectile's position
  - GPIO Pins P7.0 - P7.7
  - or'd at Pin P1.0 (ext. IRQ DMA_SYSSRC2)
- exercise:
  - perform a compiled-in motion sequence
  - C/CiAO variant of the application provided



Basic Commands to Support

- 1: UP(x)
  - rise the projectile's position by x coils
- 2: DOWN(x)
  - lower the projectile's position by x coils
  - brake the projectile at intermediate coils
- 3: RELEASE(x)
  - disable all coils for x ms
  - releases the projectile
- 4: HOLD(x)
  - hold the current position for x ms
  - caution: activating the coils for a longer time causes overheating
- 5: END(x): marks end of sequence (x discarded)
  - all coils are turned off, tasks terminated and ISRs disabled