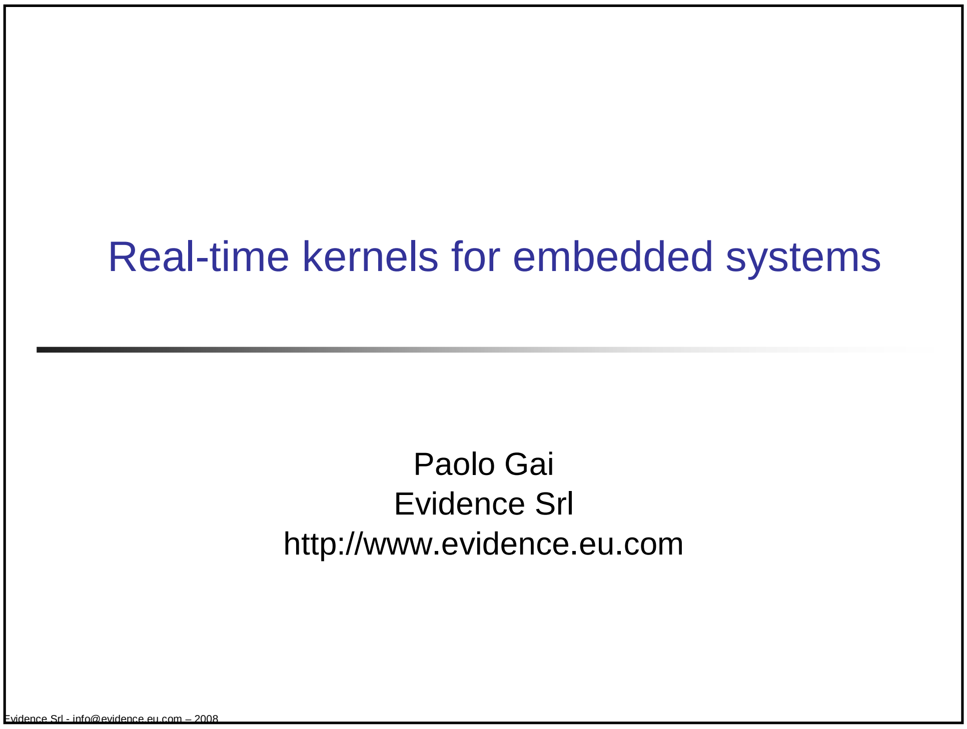
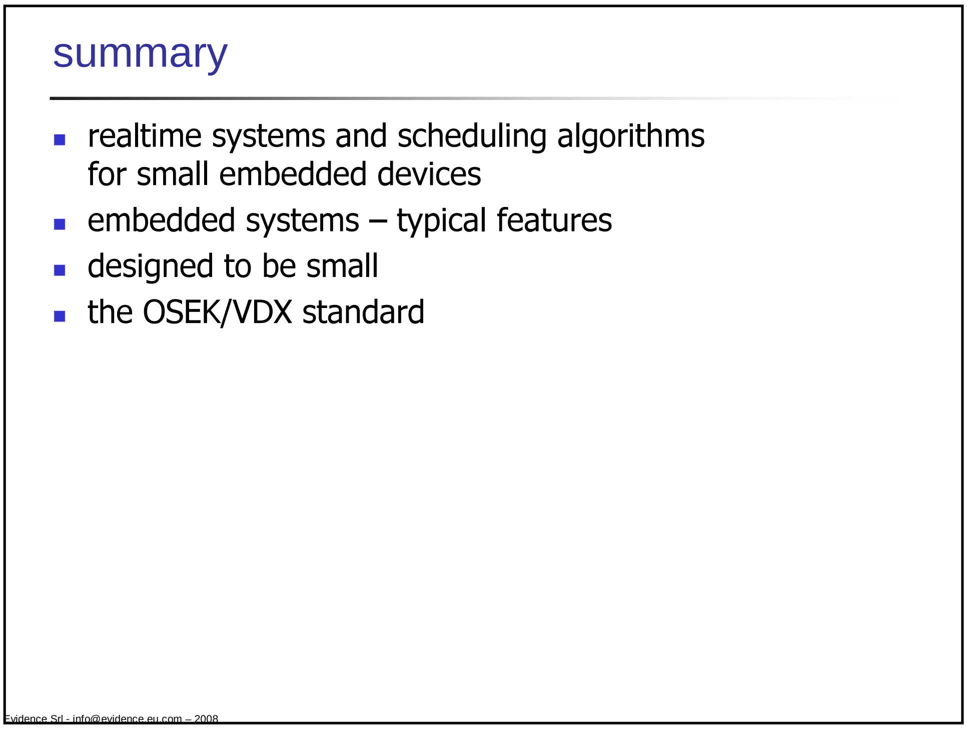


Real-time kernels for embedded systems

Paolo Gai
Evidence Srl
http://www.evidence.eu.com



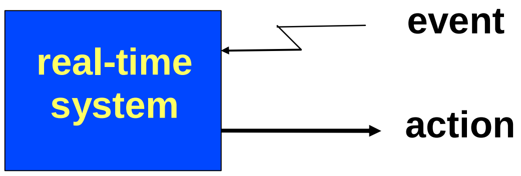
summary

- realtime systems and scheduling algorithms for small embedded devices
- embedded systems – typical features
- designed to be small
- the OSEK/VDX standard

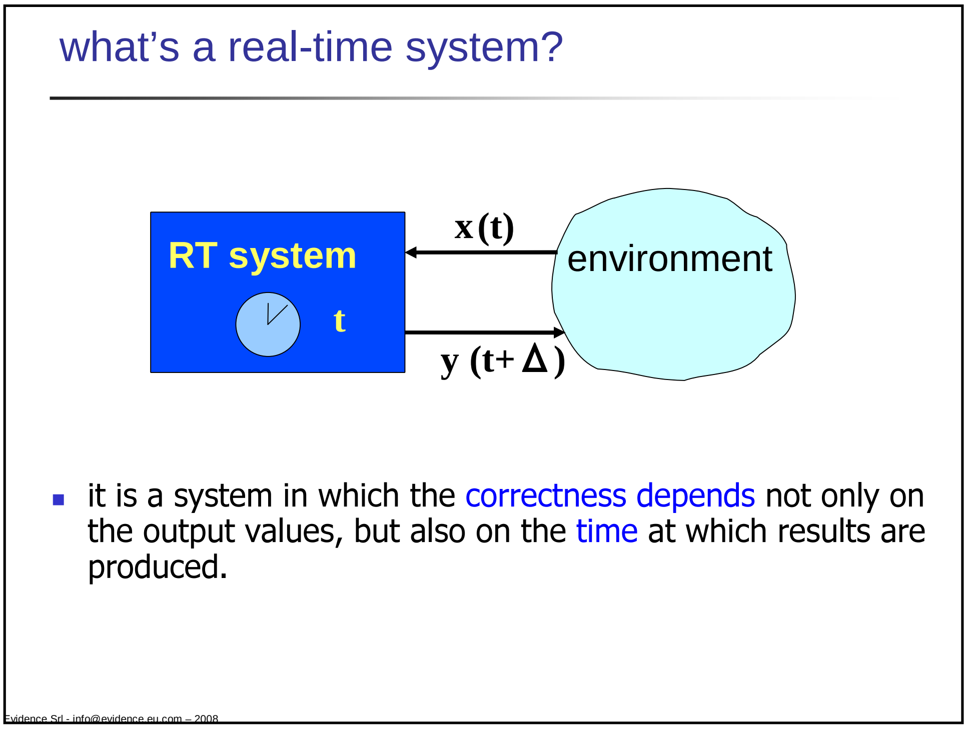
part I

realtime systems and scheduling algorithms
for small embedded devices

real-time systems

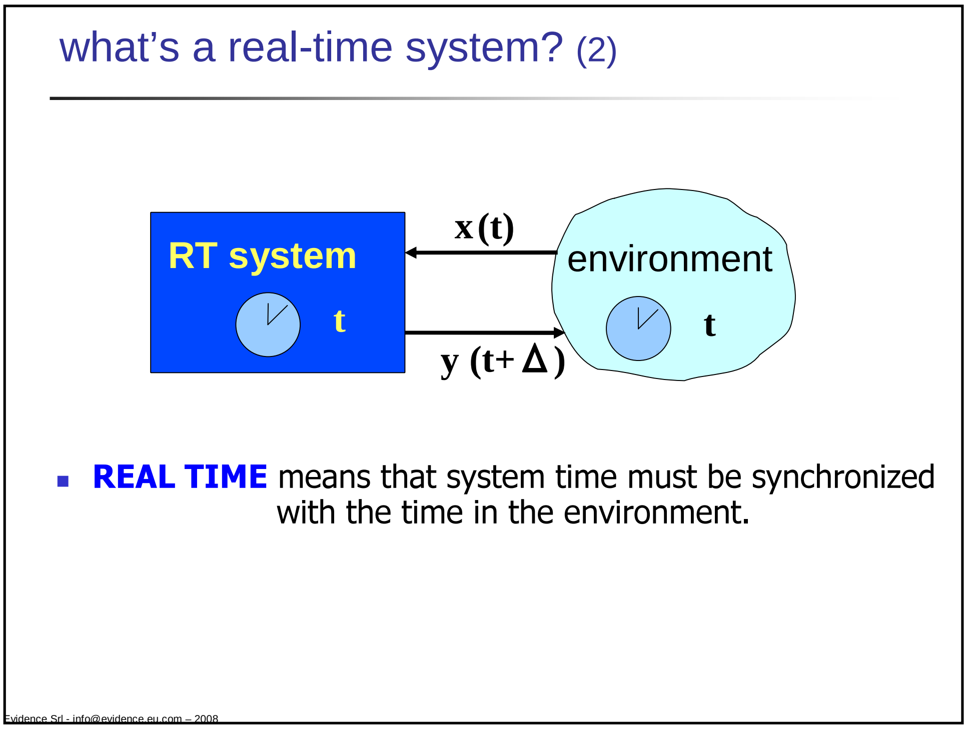


- a computing system able to respond to events within precise timing constraints is called a **Real-Time System**



what's a real-time system?

- it is a system in which the correctness depends not only on the output values, but also on the time at which results are produced.

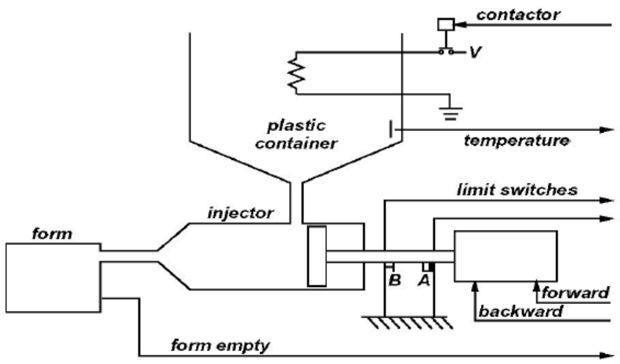


what's a real-time system? (2)

- **REAL TIME** means that system time must be synchronized with the time in the environment.

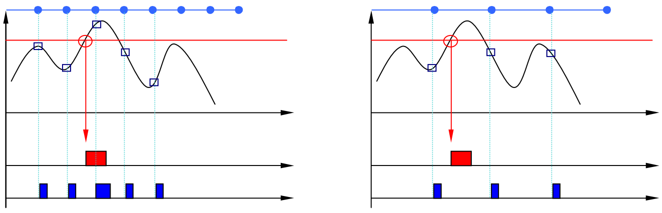
Sample RT applications

- Control of an injection molding system

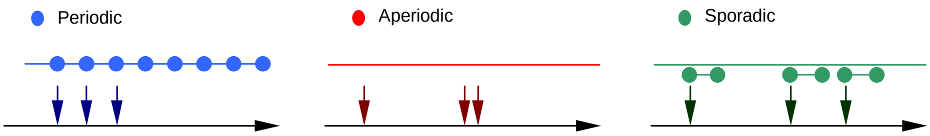


event vs time triggered

- Example, activity to be executed when the temperature exceeds the *warn* level:
- *event triggered*
  - action triggered only when temperature > *warn*
- *time triggered*
  - controls temperature every *int* time units; recovery is triggered when temperature > *warn*



activation models



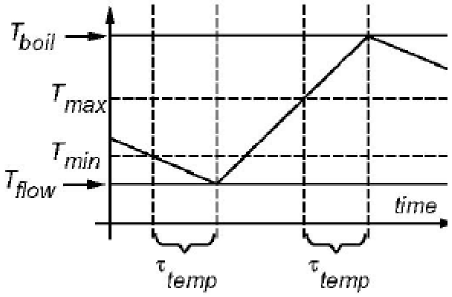
Modeling Real-time systems

- We need to identify (in the specification and design phase)
  - Events and Actions (System responses).
- Some temporal constraints are explicitly expressed as a results of system analysis
  - “The alarm must be delivered within 2s from the time instant a dangerous situation is detected”
- More often, timing constraints are hidden behind sentences that are apparently not related to time …

Modeling Real-time systems

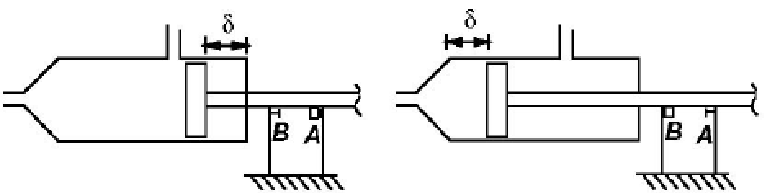
Example1: plastic molding

- The controller of the temperature must be activated within τ_{temp} seconds from the time instant when temperature thresholds are surpassed, such that $T_{boil} < T < T_{flow}$

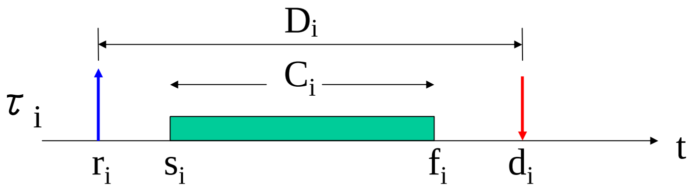


Modeling Real-time systems

- … the injector must be shut down no more than τ_{inj} seconds after receiving the end-run signals A or B such that $v_{inj}\tau_{inj} < \delta$



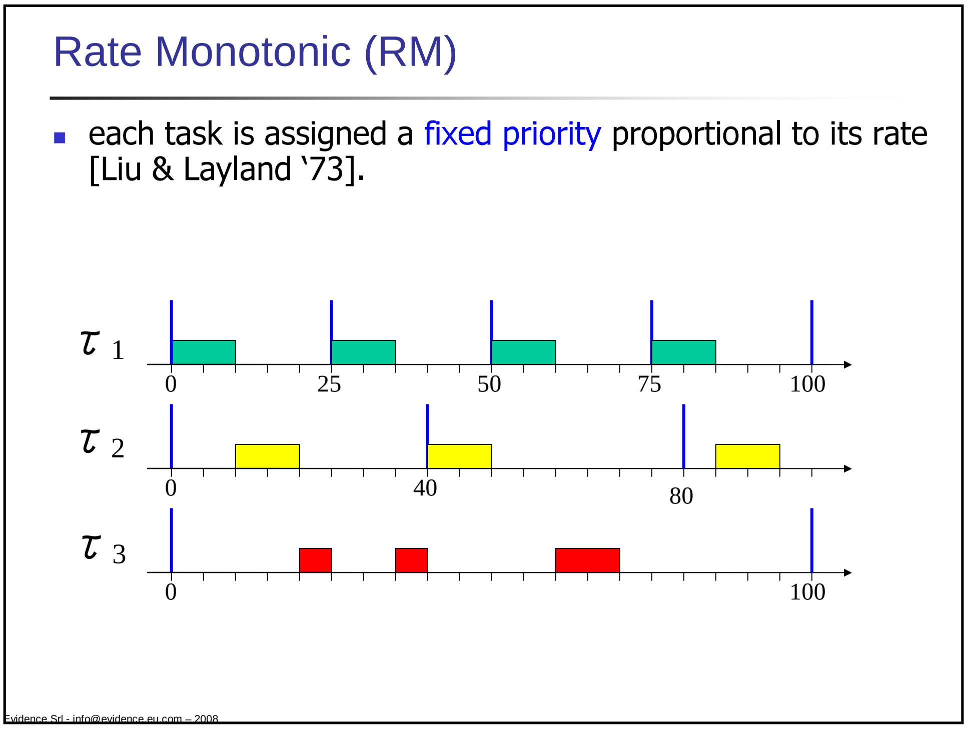
real-time tasks



- **r_i** request time (arrival time **a_i**)
- **s_i** start time
- **C_i** worst-case execution time (wcet)
- **d_i** absolute deadline
- **D_i** relative deadline
- **f_i** finishing time

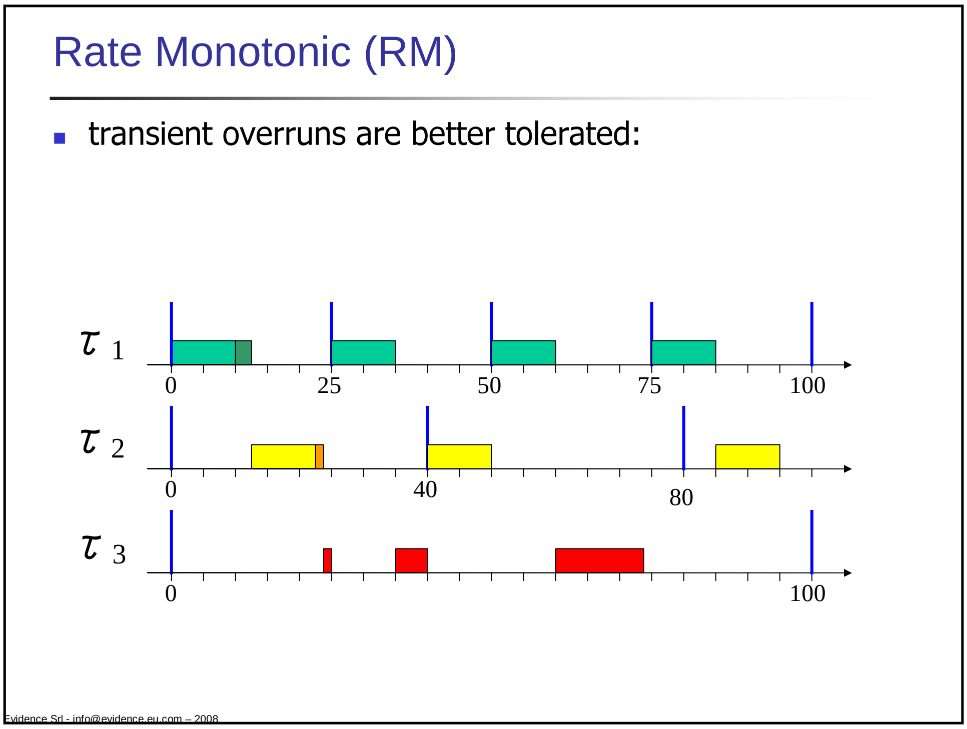
priority scheduling

- tasks are independent
- execution time is constant
- tasks are executed on a priority-based kernel
- each task is assigned a priority based on its timing constraints
- we verify the feasibility of the schedule using analytical techniques



Rate Monotonic (RM)

- each task is assigned a fixed priority proportional to its rate [Liu & Layland '73].



Rate Monotonic (RM)

- transient overruns are better tolerated:

how can we verify feasibility?

- each task uses the processor for a fraction of time:

$$U_i = \frac{C_i}{T_i}$$

- hence the total processor utilization is:

$$U_p = \sum_{i=1}^{n} \frac{C_i}{T_i}$$

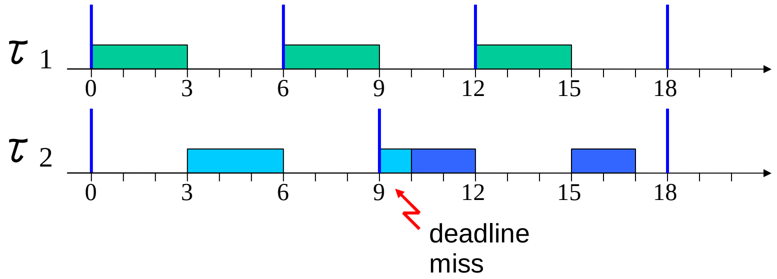
- U_p is a misure of the processor load

a necessary condition

- if Up > 1 the processor is overloaded hence the task set cannot be schedulable.
- however, there are cases in which Up < 1 but the task is not schedulable by RM

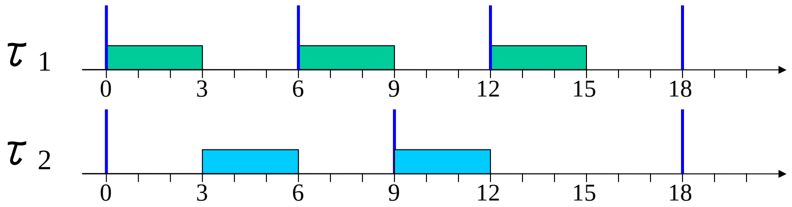
an unfeasible RM schedule

$$U_p = \frac{3}{6} + \frac{4}{9} = 0.944$$



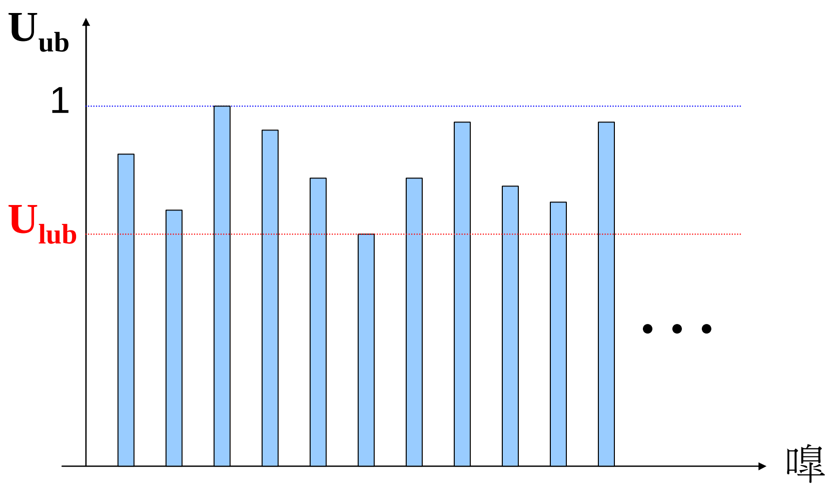
utilization upper bound

$$U_p = \frac{3}{6} + \frac{3}{9} = 0.833$$



NOTE: if C_1 or C_2 is increased, task 2 will miss its deadline!

the least upper bound



U_{lub} for RM

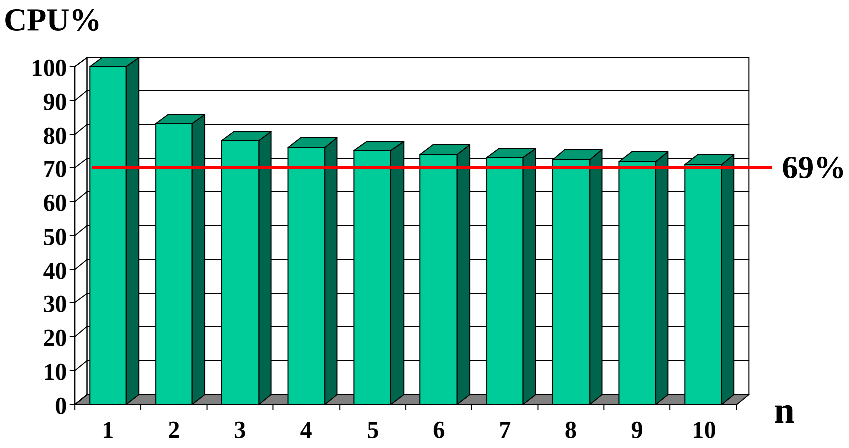
- in 1973, Liu and Layland proved that for a set of n periodic tasks:

$$U_{\text{lub}}^{RM} = n(2^{1/n} - 1)$$

for n → ∞ $U_{lub} \to \mathit{ln}\ 2$

- this is a sufficient condition
- if the periods are harmonic, Ulub = 1

RM Schedulability



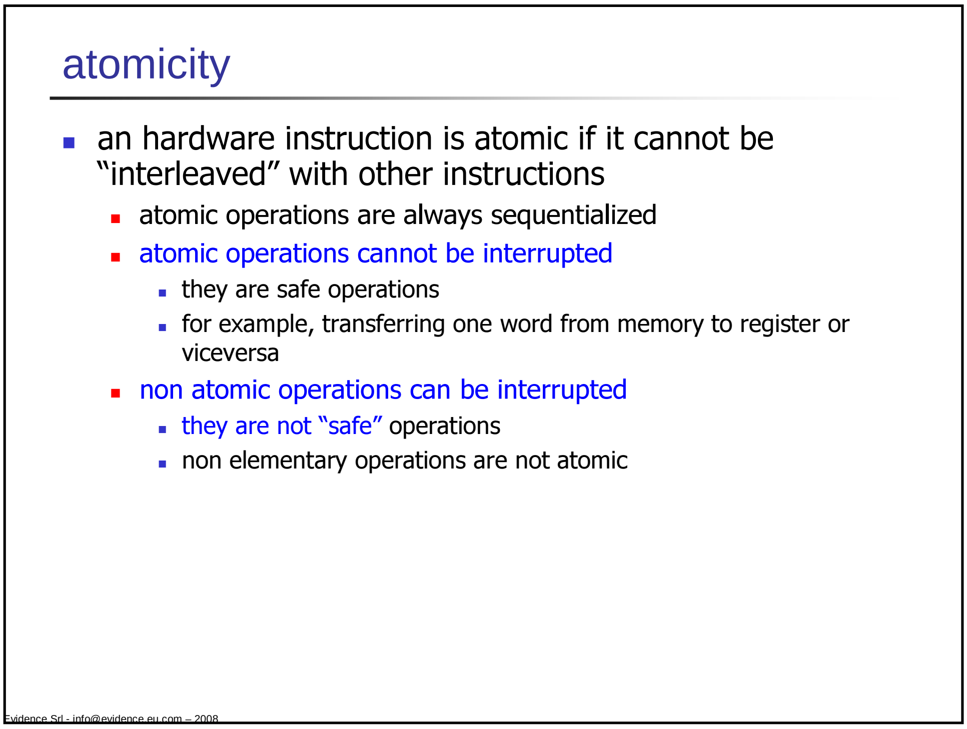
RM guarantee test

- we compute the processor utilization as:

$$U_p = \sum_{i=1}^{n} \frac{C_i}{T_i}$$

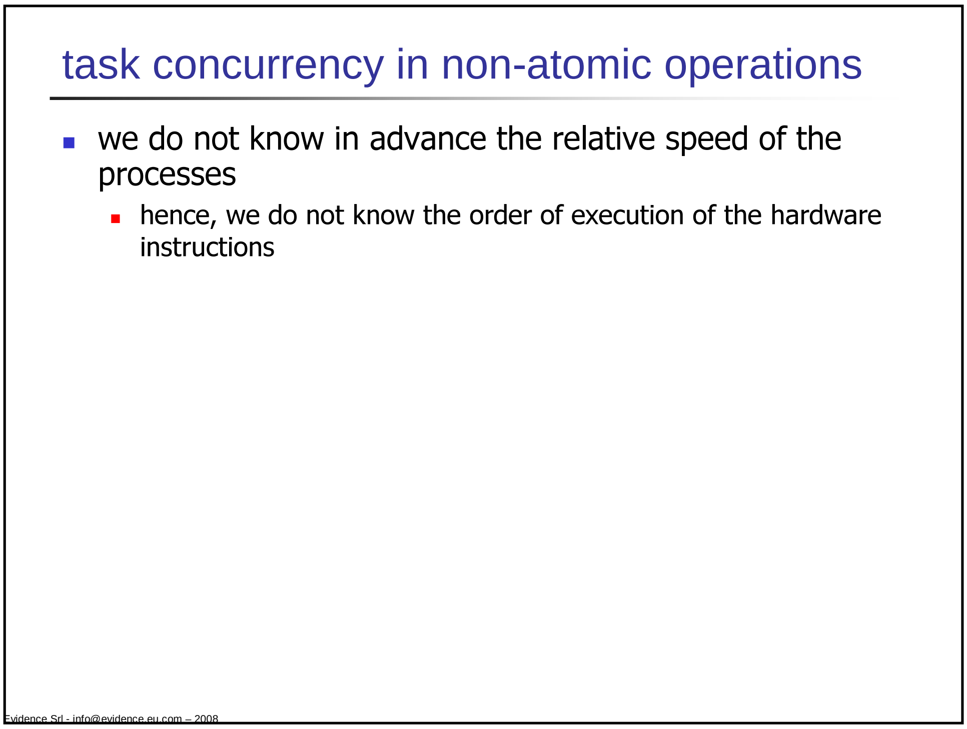
- guarantee test (only sufficient):

$$U_p < n(2^{1/n} - 1)$$



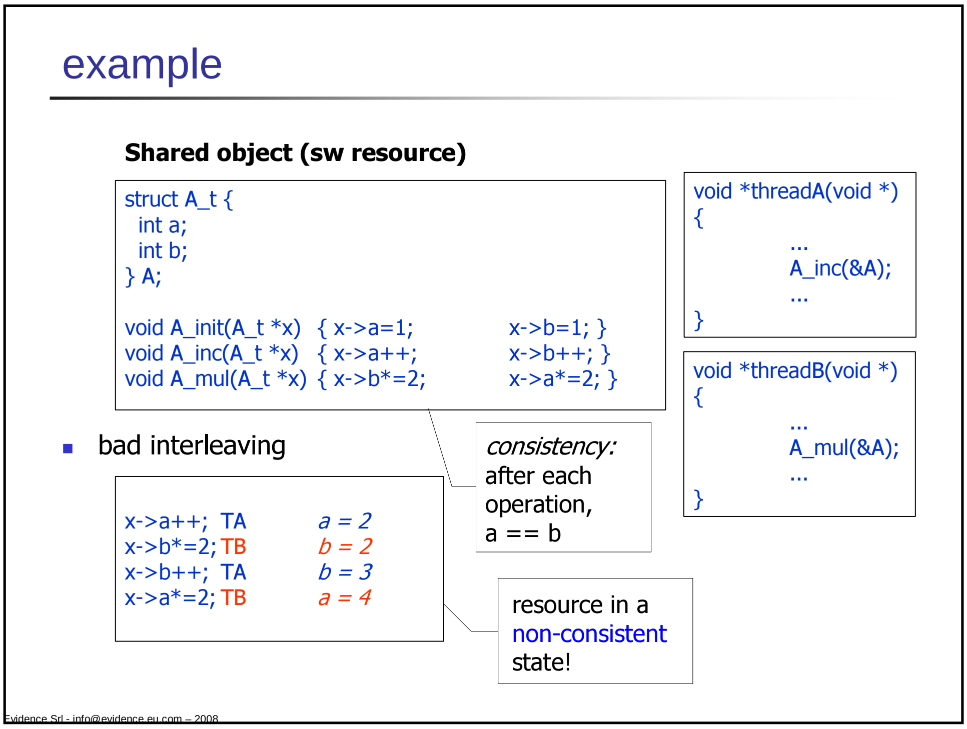
atomicity

- an hardware instruction is atomic if it cannot be “interleaved” with other instructions
  - atomic operations are always sequentialized
  - atomic operations cannot be interrupted
    - they are safe operations
    - for example, transferring one word from memory to register or viceversa
  - non atomic operations can be interrupted
    - they are not “safe” operations
    - non elementary operations are not atomic



task concurrency in non-atomic operations

- we do not know in advance the relative speed of the processes
  - hence, we do not know the order of execution of the hardware instructions



example

Shared object (sw resource)

```
struct A_t {
  int a;
  int b;
} A;

void A_init(A_t *x)  { x->a=1;          x->b=1; }
void A_inc(A_t *x)   { x->a++;          x->b++; }
void A_mul(A_t *x)   { x->b*=2;         x->a*=2; }
```

```
void *threadA(void *)
{
        …
        A_inc(&A);
        …
}
```

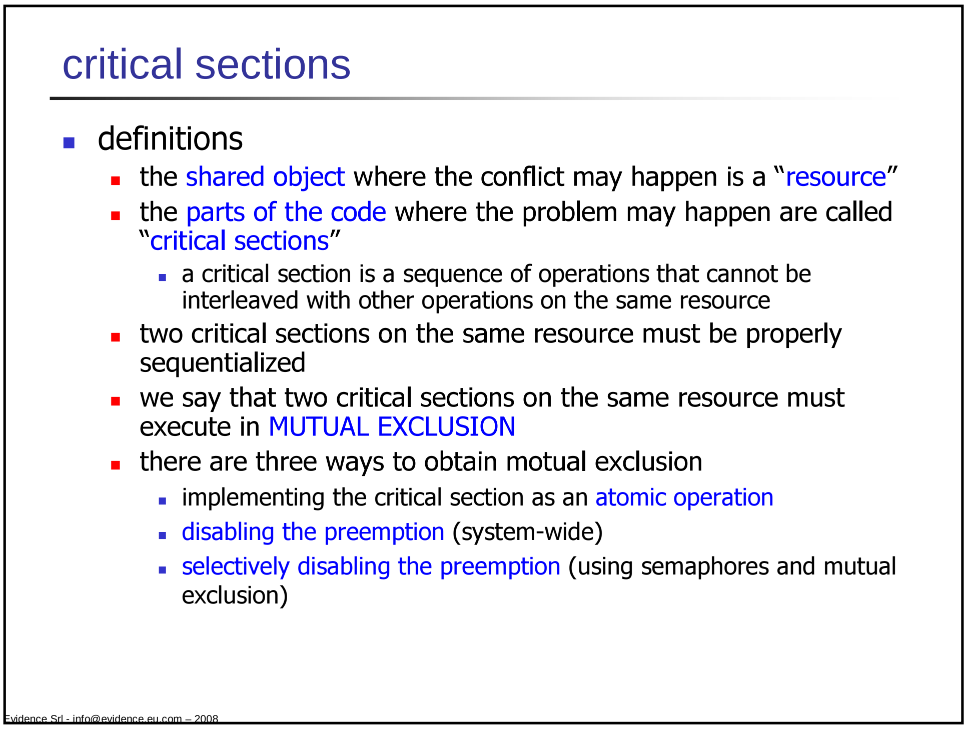
```
void *threadB(void *)
{
        …
        A_mul(&A);
        …
}
```

- bad interleaving

```
x->a++;  TA      a = 2
x->b*=2; TB      b = 2
x->b++;  TA      b = 3
x->a*=2; TB      a = 4
```

consistency: after each operation, a == b

resource in a non-consistent state!

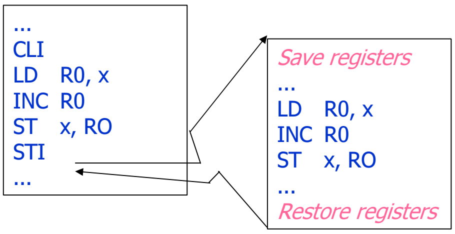


critical sections

- definitions
  - the shared object where the conflict may happen is a "resource"
  - the parts of the code where the problem may happen are called "critical sections"
    - a critical section is a sequence of operations that cannot be interleaved with other operations on the same resource
  - two critical sections on the same resource must be properly sequentialized
  - we say that two critical sections on the same resource must execute in MUTUAL EXCLUSION
  - there are three ways to obtain motual exclusion
    - implementing the critical section as an atomic operation
    - disabling the preemption (system-wide)
    - selectively disabling the preemption (using semaphores and mutual exclusion)

critical sections: atomic operations

- in single processor systems
  - disable interrupts during a critical section



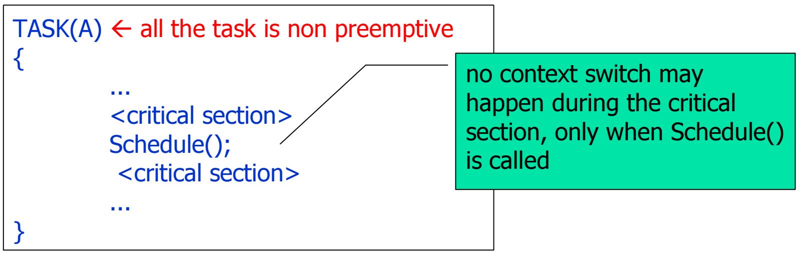
```
...
CLI
LD   R0, x
INC  R0
ST   x, RO
STI
...
```

```
Save registers
...
LD   R0, x
INC  R0
ST   x, RO
...
Restore registers
```

- problems:
  - if the critical section is long, no interrupt can arrive during the critical section
    - consider a timer interrupt that arrives every 1 msec.
    - if a critical section lasts for more than 1 msec, a timer interrupt could be lost!
  - concurrency is disabled during the critical section!
    - we must avoid conflicts on the resource, not disabling interrupts!

critical sections: disabling preemption

- single processor systems
  - it is possible to disable preemption for a limited interval of time
  - problems:
    - if a high priority critical thread needs to execute, it cannot make preemption and it is delayed
    - even if the high priority task does not access the resource!
  - in OSEK, use non-preemptive tasks together with the Schedule() primitive



```
TASK(A) ← all the task is non preemptive
{
        ...
        <critical section>
        Schedule();
         <critical section>
        ...
}
```

no context switch may happen during the critical section, only when Schedule() is called

critical sections: selectively dis. preemption

- there exist some general mechanisms to implement mutual exclusion only between the processes that uses a resource.
  - in OSEK, use the GetResource/ReleaseResource primitives

```
TASK(A)
{
            ...
            GetResource(r);
              <critical section>
            ReleaseResource(r);
            ...
}
```

```
TASK(B)
{
            ...
            GetResource(r);
              <critical section>
            ReleaseResource(r);
            ...
}
```

priority inversion

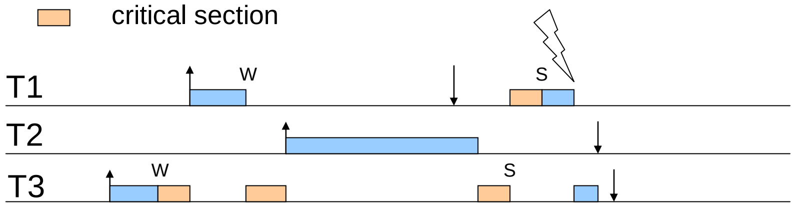
suppose to have 2 tasks that share a resource

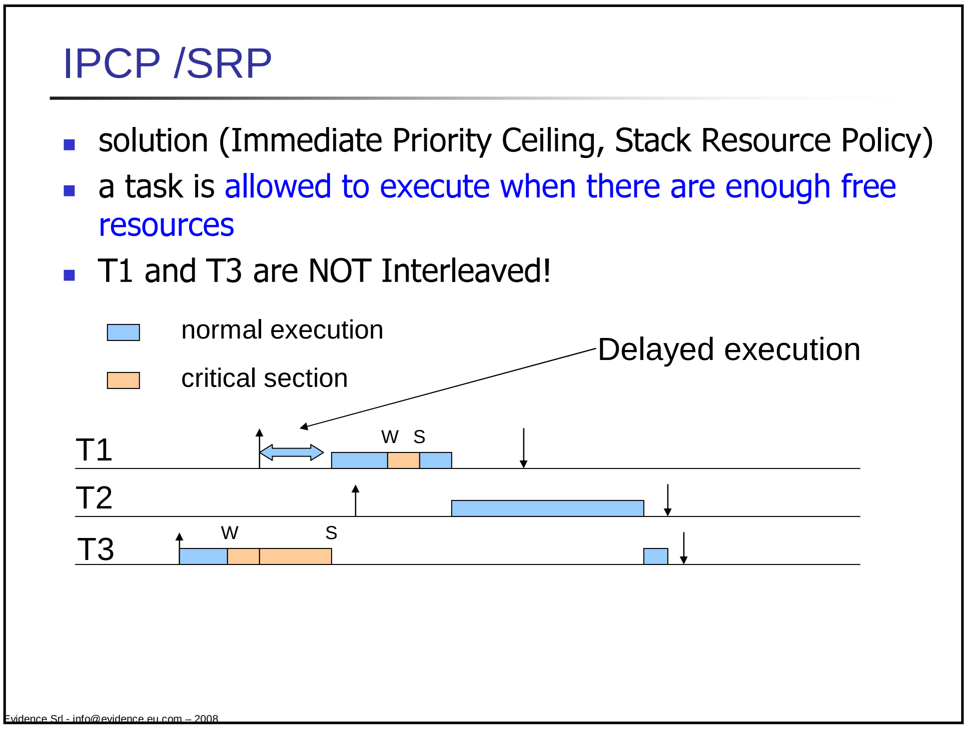
- the High Priority task can be delayed because of some low priority task

🟦 normal execution

🟧 critical section

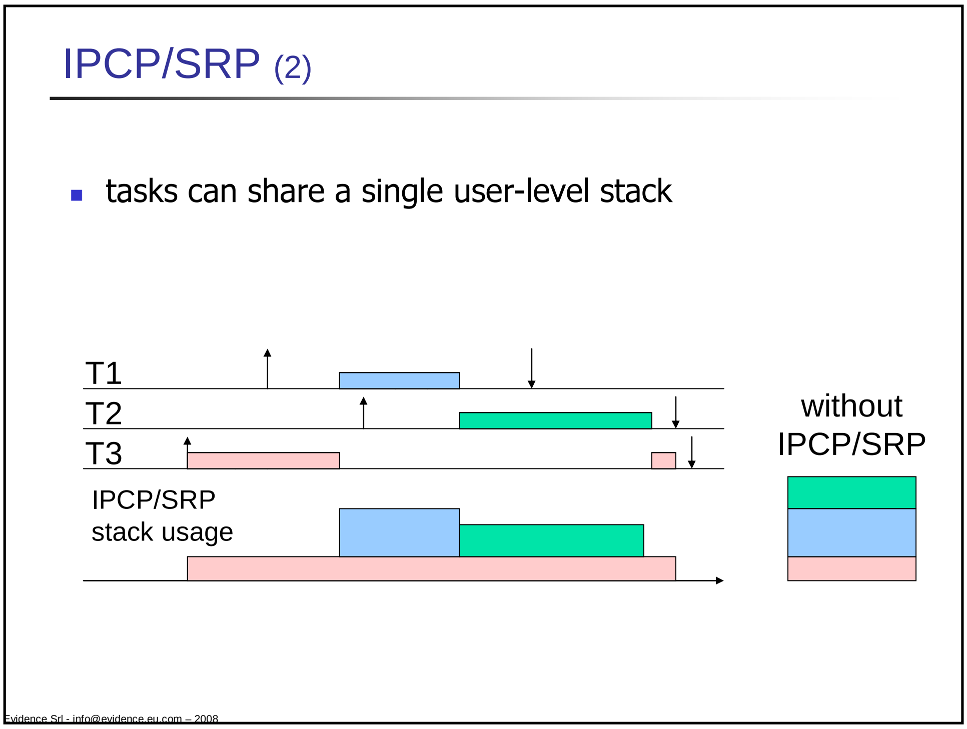
Deadline miss!!!





IPCP /SRP

- solution (Immediate Priority Ceiling, Stack Resource Policy)
- a task is allowed to execute when there are enough free resources
- T1 and T3 are NOT Interleaved!



IPCP/SRP (2)

- tasks can share a single user-level stack

implementation tips

how can two threads share the same stack space?

the tradictional thread model

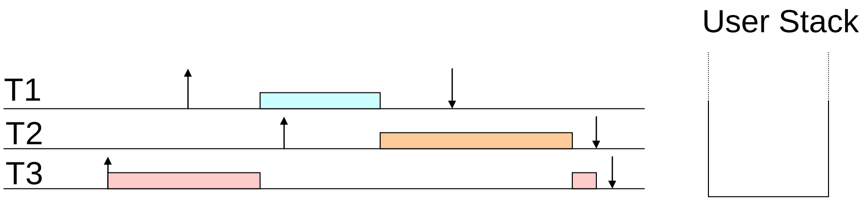
- allows a task to block
- forces a task structure

```
Task x()
{
  int local;
  initialization();
  for (;;) {
    do_instance();
    end_instance();
  }
}
```

- in general, all tasks can preempt each other
  - also, tasks can block on semaphores
- a stack is needed for each task that can be preempted
- the overall requirement for stack space is the sum of all task requirements, plus interrupt frames

kernel-supported stack sharing

- the kernel really manages only a single stack that is shared by ALL the tasks
  - also interrupts use the same stack
- kernel must ensure that tasks never block
  - it would produce interleaving between tasks, that is not supported since there is only one stack



one shot model

- to share the stack the **one shot task model** is needed
- in OSEK/VDX, these two kinds of task models are extended and basic tasks

Extended Tasks

```
Task(x)
{
  int local;
  initialization();
  for (;;) {
    do_instance();
    end_instance();
  }
}
```

Basic Tasks (one shot!)

```
int local;

Task x()
{
  do_instance();
}

System_initialization()
{
  initialization();
  ...
}
```

part I

embedded systems

-

typical features

software used in automotive systems

The software in powertrain systems

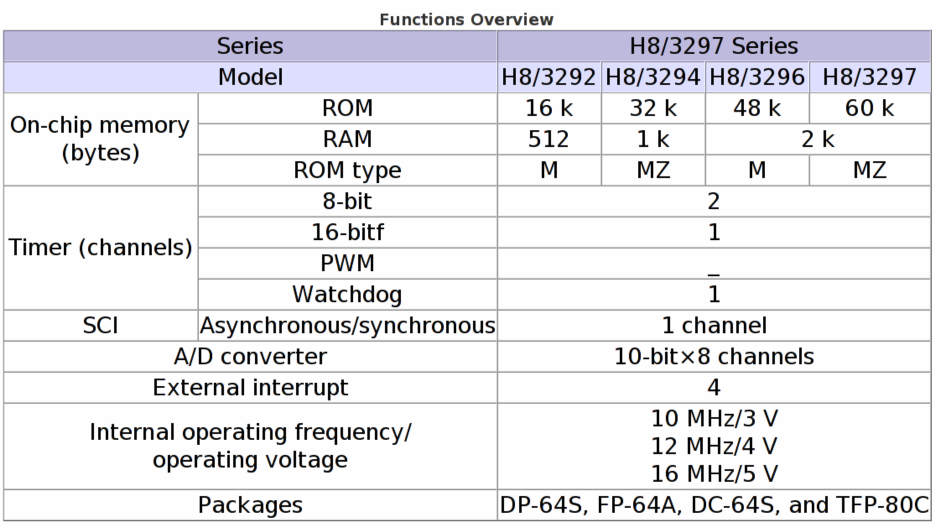
- boot and microcontroller related features
- real-time operating system
  - provides abstractions (for example: task, semaphores, …)
  - an interaction model between hardware and application
  - separates behavior from infrastructures
  - debugging simplification
- I/O Libraries
  - completes the OS with the support of the platform HW
  - 10 times bigger than a minimal OS
- application
  - implements only the behavior and not the infrastructures (libraries)
  - independent from the underlying hardware
- the operating system is a key element in the architecture of complex embedded systems

typical microcontroller features

let's try to highlight a typical scenario that applies to embedde platforms

- embedded microcontroller
  - depending on the project, that microcontroller will be @ 8, 16, or 32 bit
  - typically comes with a rich set of interfaces
    - timers (counters / CAPCOM / Watchdog / PWM)
    - A/D and D/A
    - communication interfaces (I2C, RS232, CAN, Infrared, ...)
    - ~50 interrupts (the original PC interrupt controller had only 15!!!)
- memory
  - SRAM / FLASH / ...
- other custom HW / power circuits

Hitachi H8

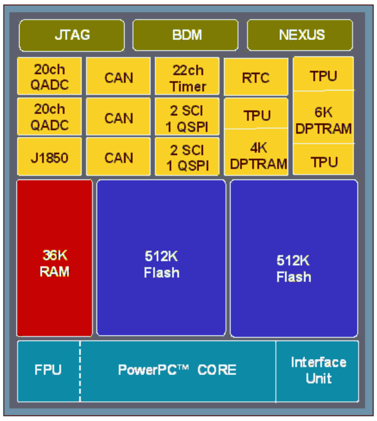


Functions Overview

Series		H8/3297 Series			
Model		H8/3292	H8/3294	H8/3296	H8/3297
On-chip memory (bytes)	ROM	16 k	32 k	48 k	60 k
	RAM	512	1 k	2 k	
	ROM type	M	MZ	M	MZ
Timer (channels)	8-bit	2			
	16-bitf	1			
	PWM	_			
	Watchdog	1			
SCI	Asynchronous/synchronous	1 channel			
A/D converter		10-bit×8 channels			
External interrupt		4			
Internal operating frequency/ operating voltage		10 MHz/3 V 12 MHz/4 V 16 MHz/5 V			
Packages		DP-64S, FP-64A, DC-64S, and TFP-80C			

Motorola MPC565

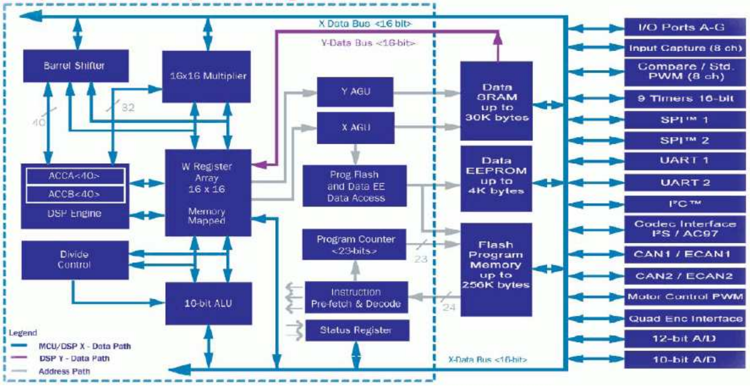
- 1M byte of internal FLASH memory (divided into two blocks of 512K bytes)
- 36K bytes Static RAM
- Three time processor units (TPU3)
- A 22-timer channel modular I/O system (MIOS14)
- Three TouCAN modules
- Two enhanced queued analog system with analog multiplexors (AMUX) for 40 total analog channels. These modules are configured so each module can access all 40 of the analog inputs to the part.
- Two queued serial multi-channel modules, each of which contains a queued serial peripheral interface (QSPI) and two serial controller interfaces (SCI/UART)
- A J1850 (DLCMD2) communications module
- A NEXUS debug port (class 3) – IEEE-ISTO 5001-1999
- JTAG and background debug mode (BDM)



Microchip dsPIC

- Single core architecture / Familiar MCU look and feel / DSP performance
- Rich peripheral options / Advanced interrupt capability / Flexible Flash memory
- Self-programming capability / Low pin count options / Optimized for C

dsPIC30F/dsPIC33F Family Block Diagram



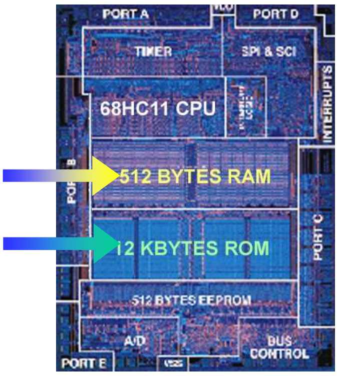
Product	Pins	Flash Memory Kbytes	RAM Kbytes	DMA # Ch	Timer 16-bit	Input Capture	Output Compare/ Standard PWM	Codec Interface	A/D- 12-bit 500 ksps	UART	SPI™	I²C™	CAN	I/O Pins (max)†	Package Code
dsPIC33FJ256GP710	100	256	30	8	9	8	8	1	2 ADC, 32 ch, 2 S/H	2	2	2	2	85	PT, PF

RAM vs ROM usage

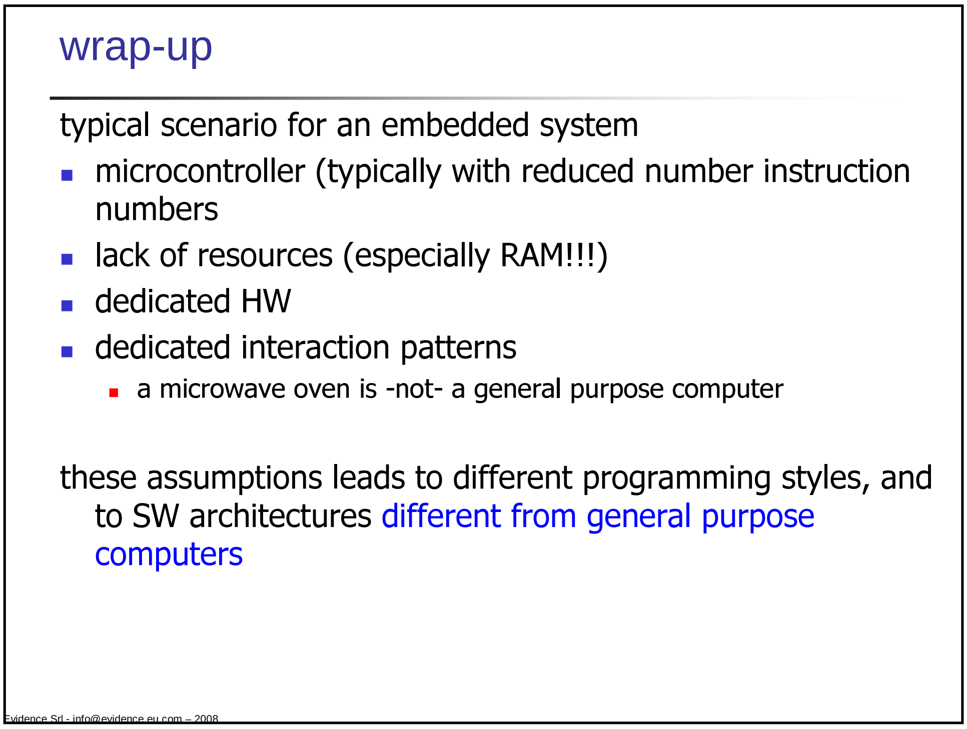
- consider a mass production market: ~ few M boards sold
- development cost impacts around 10%
- techniques for optimizing silicon space on chip
  - you can spend a few men-months to reduce the footprint of the application
- memory in a typical SoC
  - 512 Kb Flash, 16 Kb RAM

sample SoC (speech process. chip for security apps) picture

- 68HC11 micro
- 12Kb ROM
- 512 bytes RAM in approx. the same space *(24x cost!)*



Sample die of a speech-processing chip

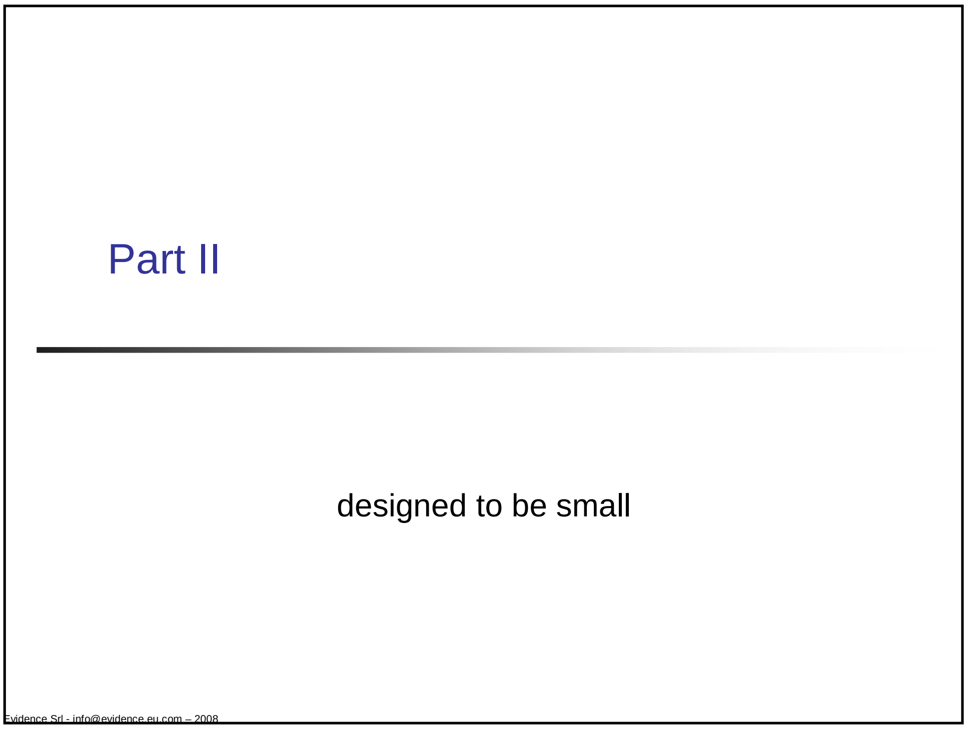


wrap-up

typical scenario for an embedded system

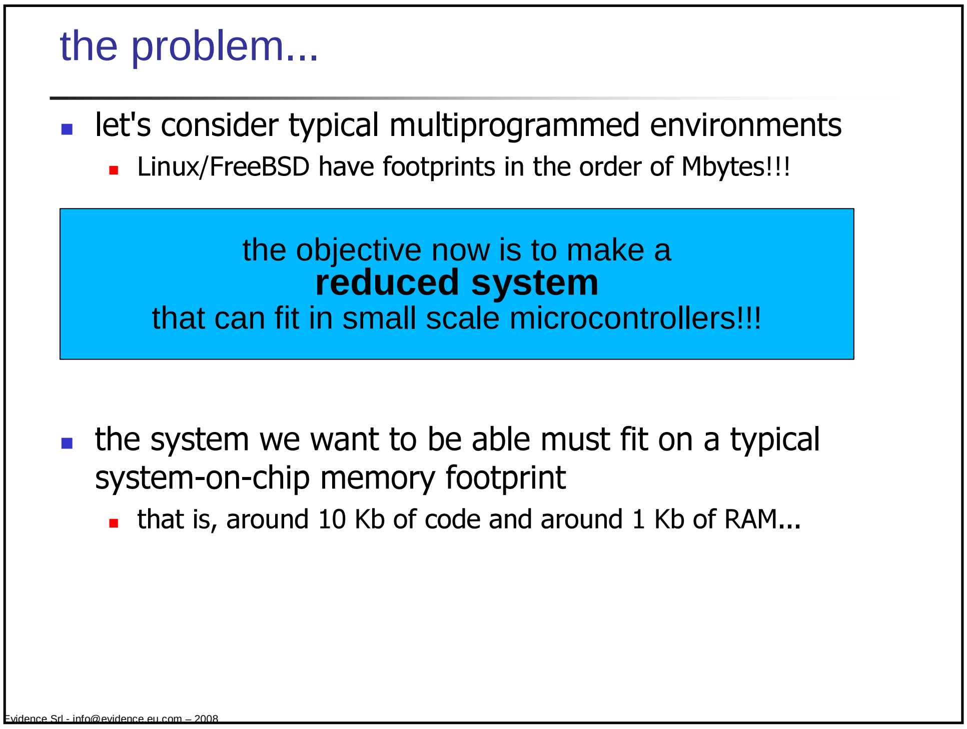
- microcontroller (typically with reduced number instruction numbers
- lack of resources (especially RAM!!!)
- dedicated HW
- dedicated interaction patterns
  - a microwave oven is -not- a general purpose computer

these assumptions leads to different programming styles, and to SW architectures different from general purpose computers



Part II

designed to be small

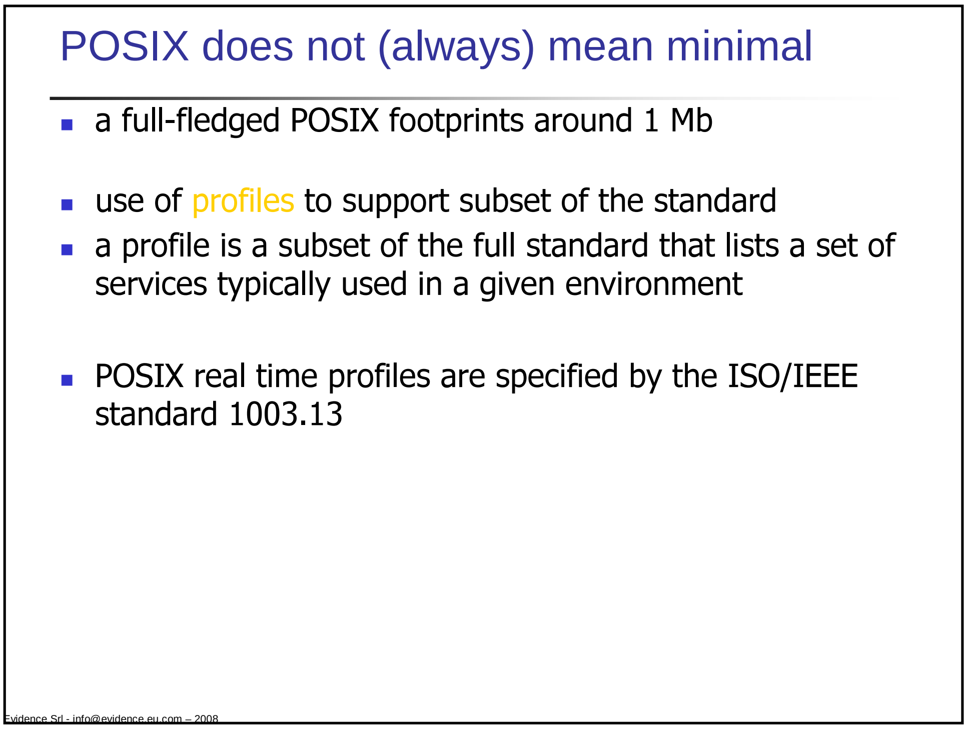


the problem...

- let's consider typical multiprogrammed environments
  - Linux/FreeBSD have footprints in the order of Mbytes!!!

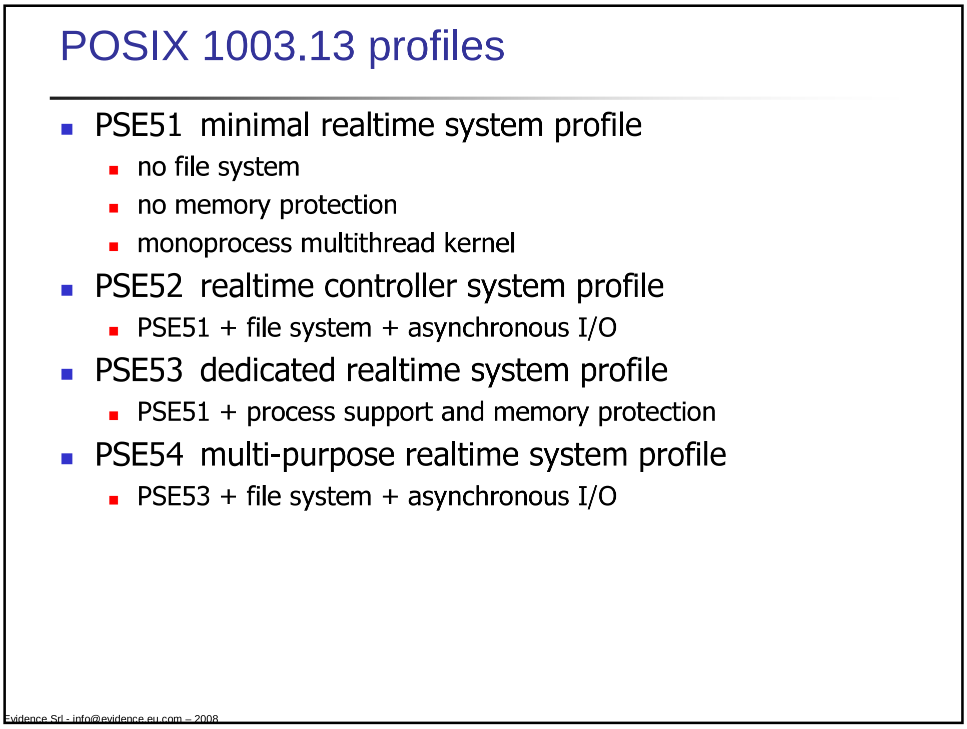
the objective now is to make a
reduced system
that can fit in small scale microcontrollers!!!

- the system we want to be able must fit on a typical system-on-chip memory footprint
  - that is, around 10 Kb of code and around 1 Kb of RAM...



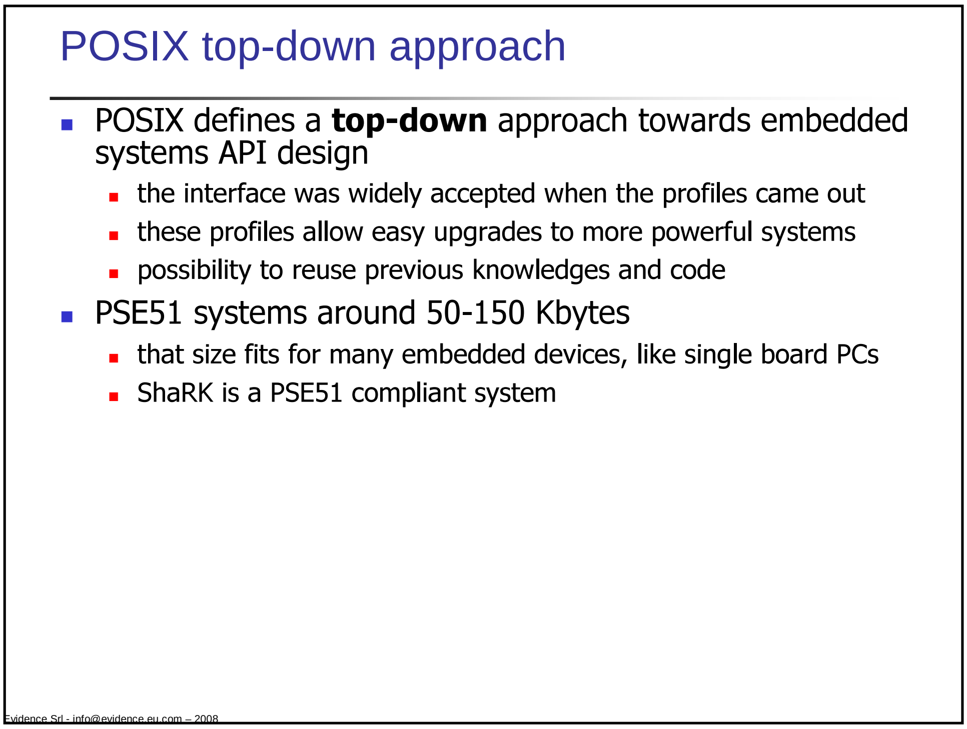
POSIX does not (always) mean minimal

- a full-fledged POSIX footprints around 1 Mb
- use of profiles to support subset of the standard
- a profile is a subset of the full standard that lists a set of services typically used in a given environment
- POSIX real time profiles are specified by the ISO/IEEE standard 1003.13



POSIX 1003.13 profiles

- PSE51 minimal realtime system profile
  - no file system
  - no memory protection
  - monoprocess multithread kernel
- PSE52 realtime controller system profile
  - PSE51 + file system + asynchronous I/O
- PSE53 dedicated realtime system profile
  - PSE51 + process support and memory protection
- PSE54 multi-purpose realtime system profile
  - PSE53 + file system + asynchronous I/O



POSIX top-down approach

- POSIX defines a **top-down** approach towards embedded systems API design
  - the interface was widely accepted when the profiles came out
  - these profiles allow easy upgrades to more powerful systems
  - possibility to reuse previous knowledges and code
- PSE51 systems around 50-150 Kbytes
  - that size fits for many embedded devices, like single board PCs
  - ShaRK is a PSE51 compliant system

SoC needs bottom-up approaches!

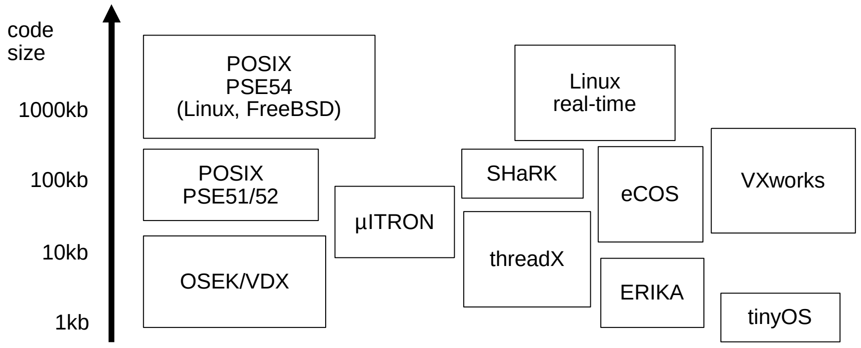
- we would like to have footprint in the order of 1-10 Kb
- the idea is to have a bottom-up approach

- starting from scratch, design
  - a minimal system
  - that provides a minimal API
  - that is able to efficiently describe embedded systems
    - with stringent temporal requirements
    - with limited resources

results:

- RTOS standards (OSEK-VDX, uITRON)
- 2 Kbytes typical footprint

typical footprints



step 1: the boot code

- starting point
  - the microcontroller
- boot code design
  - typically there will be a startup routine called at startup
  - that routine will handle
    - binary image initialization (initialized data and BSS)
    - initialization of the microcontroller services (segments/memory addresses/interrupt vectors)
  - and will finally jump to an initialization C routine
- RTOS- independent interrupt handling
  - interrupt handlers that allow an interrupt to fire and to return to the interrupted point, without any kind of rescheduling
  - OSEK calls these handlers “ISR type 1”

after step 1: a non concurrent system

- basic 1-task non-preemptive system
- good for really really small embedded devices
  - footprint around a few hundred bytes
  - e.g., PIC
- next step: add some kind of multiprogramming environment

step 2: multiprogramming environment

- right choice of the multiprogramming environment
  - cuncurrent requirements influences RAM footprint

Questions:

- what kind of multiprogramming model is really needed for automotive applications?

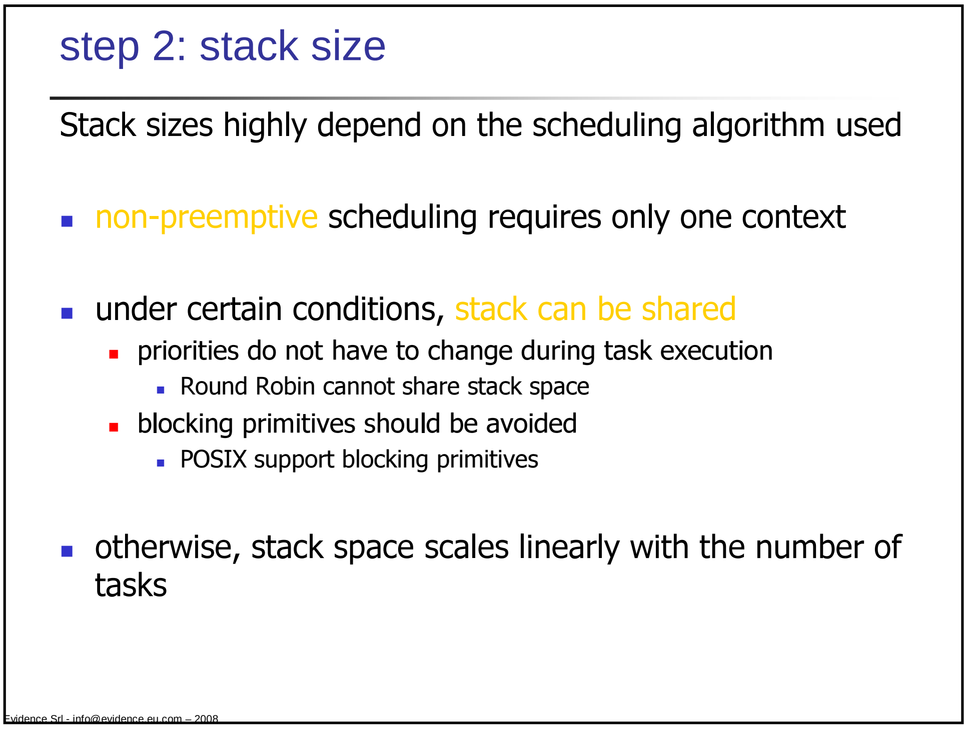
- which is the best semantic that fits the requirements?
  - preemptive or non preemptive?
  - off-line or on-line scheduling?
  - support for blocking primitives?

step 2: off-line, non real-time

- not all the systems requires full multiprogramming support
- off-line scheduled systems typically requires simpler scheduling strategies
  - example: cyclic scheduling
- non real-time systems may not require complex scheduling algorithms

TinyOS

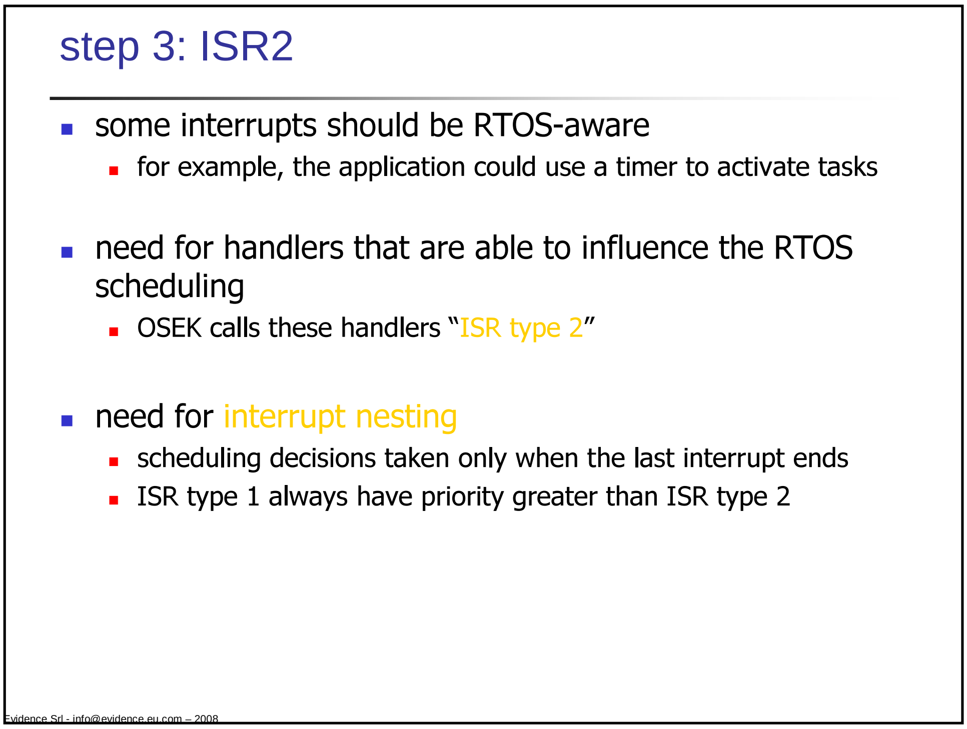
  - `http://www.tinyos.net`
  - component-based OS written in NesC
  - used for networked wireless sensors
  - provides interrupt management and FIFO scheduling in a few hundred bytes of code



step 2: stack size

Stack sizes highly depend on the scheduling algorithm used

- non-preemptive scheduling requires only one context
- under certain conditions, stack can be shared
  - priorities do not have to change during task execution
    - Round Robin cannot share stack space
  - blocking primitives should be avoided
    - POSIX support blocking primitives
- otherwise, stack space scales linearly with the number of tasks



step 3: ISR2

- some interrupts should be RTOS-aware
  - for example, the application could use a timer to activate tasks
- need for handlers that are able to influence the RTOS scheduling
  - OSEK calls these handlers "ISR type 2"
- need for interrupt nesting
  - scheduling decisions taken only when the last interrupt ends
  - ISR type 1 always have priority greater than ISR type 2

step 4: careful selection of services

- to reduce the system footprint, system services must be carefully chosen
  - no memory protection
  - no dynamic memory allocation
  - no filesystem
  - no blocking primitives
  - no software interrupts
  - no console output
- ...including only what is really needed
  - basic priority scheduling
  - mutexes for resource sharing
  - timers for periodic tasks

standardized APIs

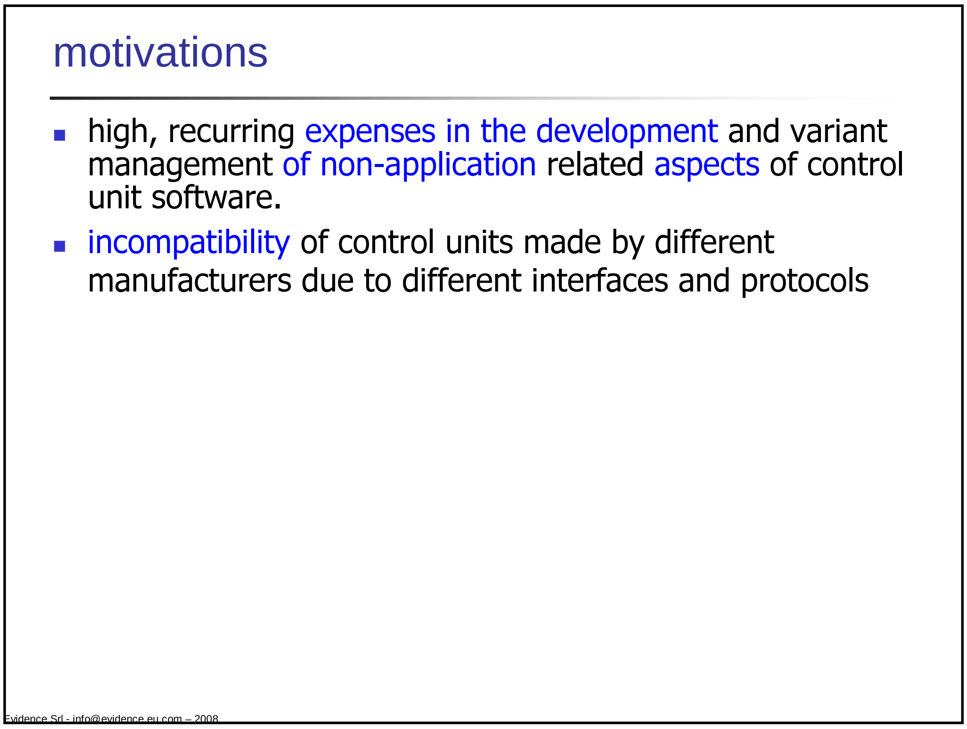
- there exists standards for minimal RTOS support
  - automotive applications, OSEK-VDX
  - japanese embedded consumers, uITRON
- and for I/O libraries
  - automotive applications, HIS working group

part IV

the OSEK/VDX standard

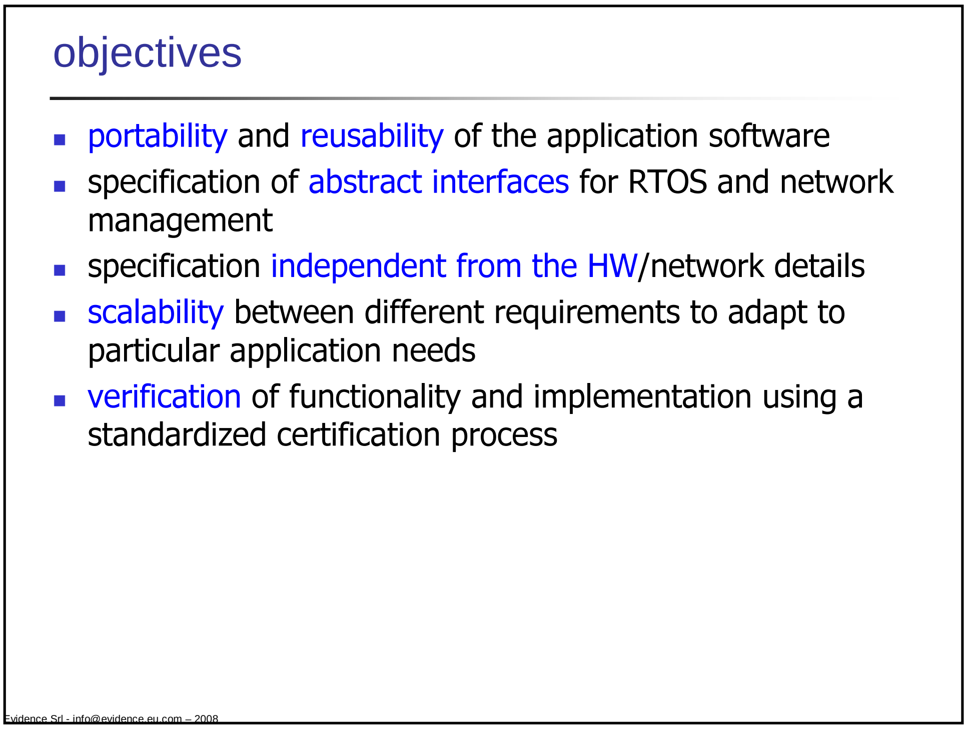
what is OSEK/VDX?

- is a standard for an open-ended architecture for distributed control units in vehicles
- the name:
  - OSEK: Offene Systeme und deren Schnittstellen für die Elektronik im Kraft-fahrzeug (Open systems and the corresponding interfaces for automotive electronics)
  - VDX: Vehicle Distributed eXecutive (another french proposal of API similar to OSEK)
  - OSEK/VDX is the interface resulted from the merge of the two projects
- `http://www.osek-vdx.org`



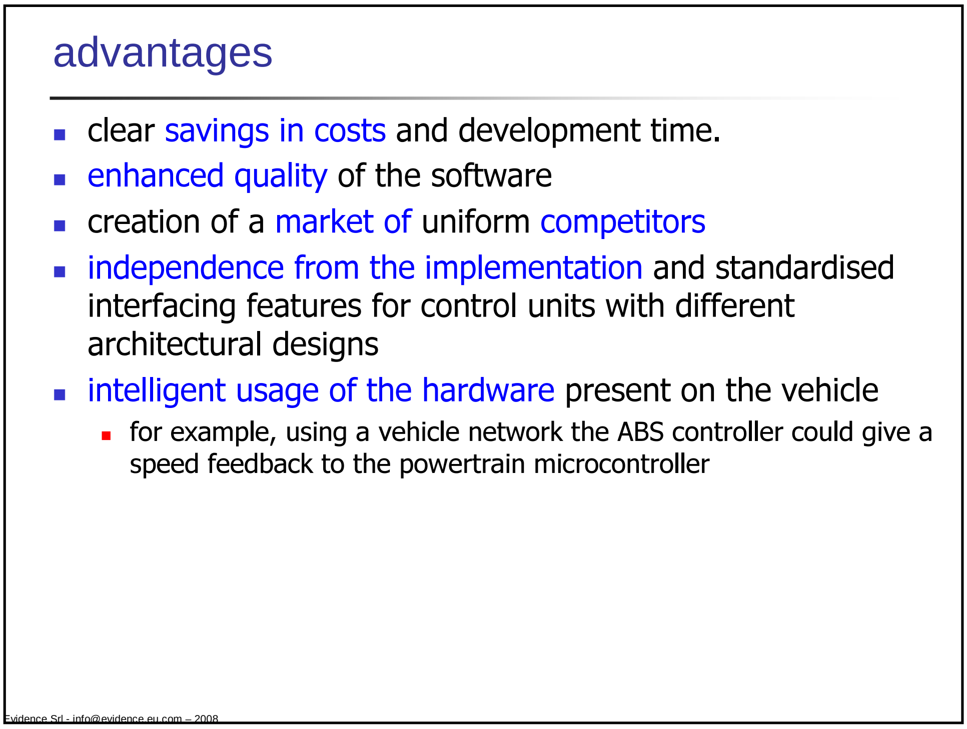
motivations

- high, recurring expenses in the development and variant management of non-application related aspects of control unit software.
- incompatibility of control units made by different manufacturers due to different interfaces and protocols



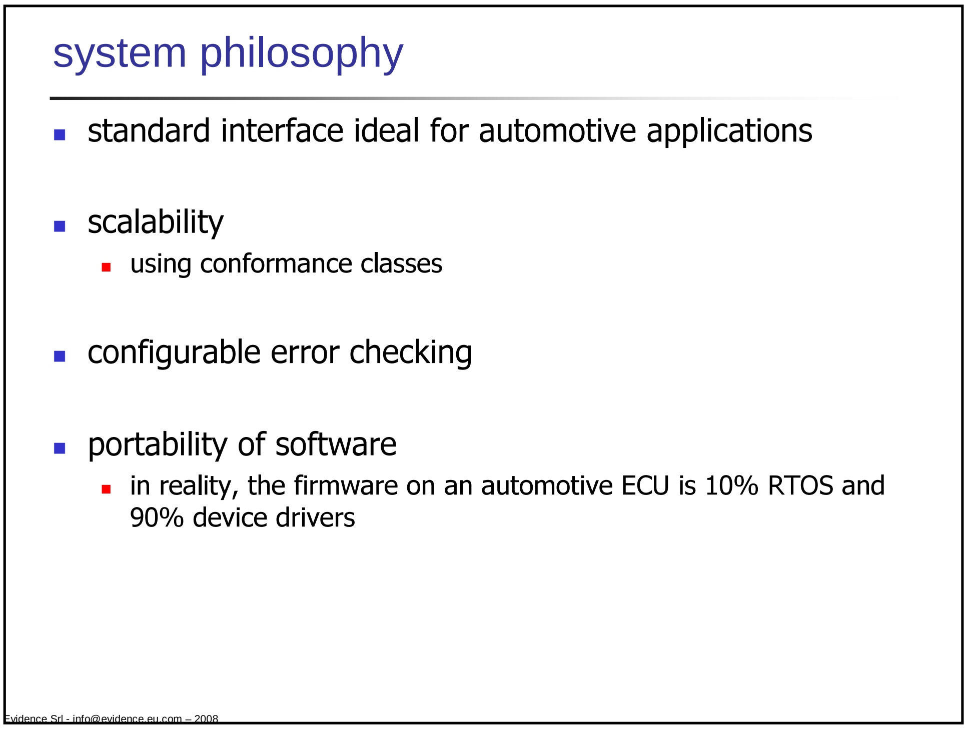
objectives

- portability and reusability of the application software
- specification of abstract interfaces for RTOS and network management
- specification independent from the HW/network details
- scalability between different requirements to adapt to particular application needs
- verification of functionality and implementation using a standardized certification process



advantages

- clear savings in costs and development time.
- enhanced quality of the software
- creation of a market of uniform competitors
- independence from the implementation and standardised interfacing features for control units with different architectural designs
- intelligent usage of the hardware present on the vehicle
  - for example, using a vehicle network the ABS controller could give a speed feedback to the powertrain microcontroller



system philosophy

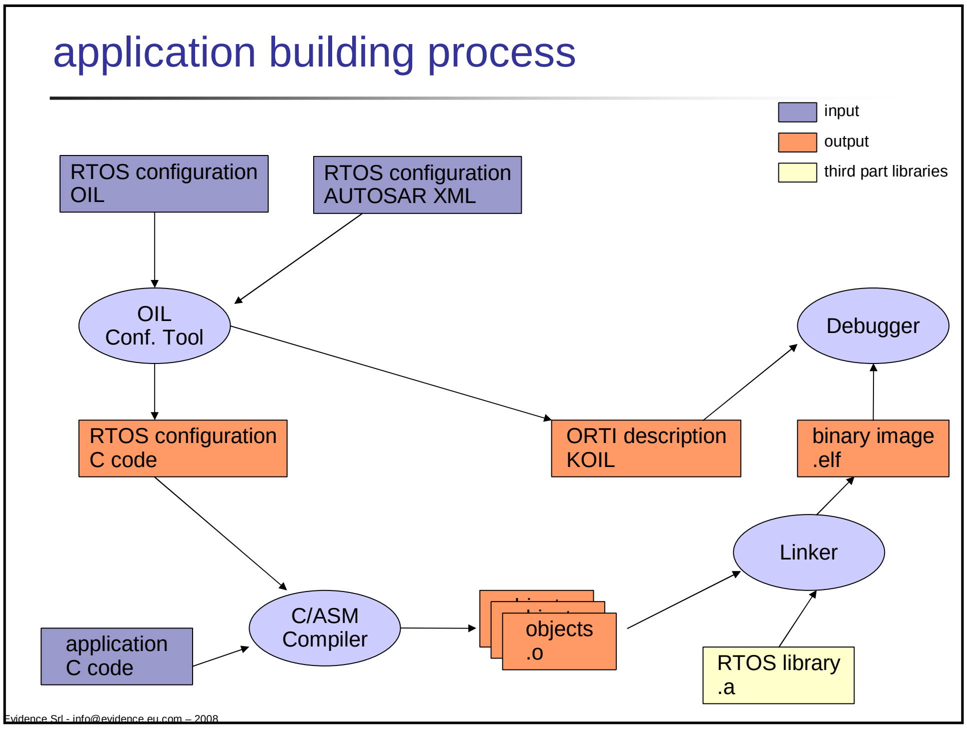
- standard interface ideal for automotive applications
- scalability
  - using conformance classes
- configurable error checking
- portability of software
  - in reality, the firmware on an automotive ECU is 10% RTOS and 90% device drivers

support for automotive requirements

- the idea is to create a system that is
  - reliable
  - with real-time predictability
- support for
  - fixed priority scheduling with immediate priority ceiling
  - non preemptive scheduling
  - preemption thresholds
  - ROM execution of code
  - stack sharing (limited support for blocking primitives)
- documented system primitives
  - behavior
  - performance of a given RTOS must be known

static is better

- everything is specified before the system runs
- static approach to system configuration
  - no dynamic allocation on memory
  - no dynamic creation of tasks
  - no flexibility in the specification of the constraints
- custom languages that helps off-line configuration of the system
  - OIL: parameters specification (tasks, resources, stacks…)
  - KOIL: kernel aware debugging



application building process

▨ input

🟧 output

□ third part libraries

RTOS configuration OIL

RTOS configuration AUTOSAR XML

OIL Conf. Tool

Debugger

RTOS configuration C code

ORTI description KOIL

binary image .elf

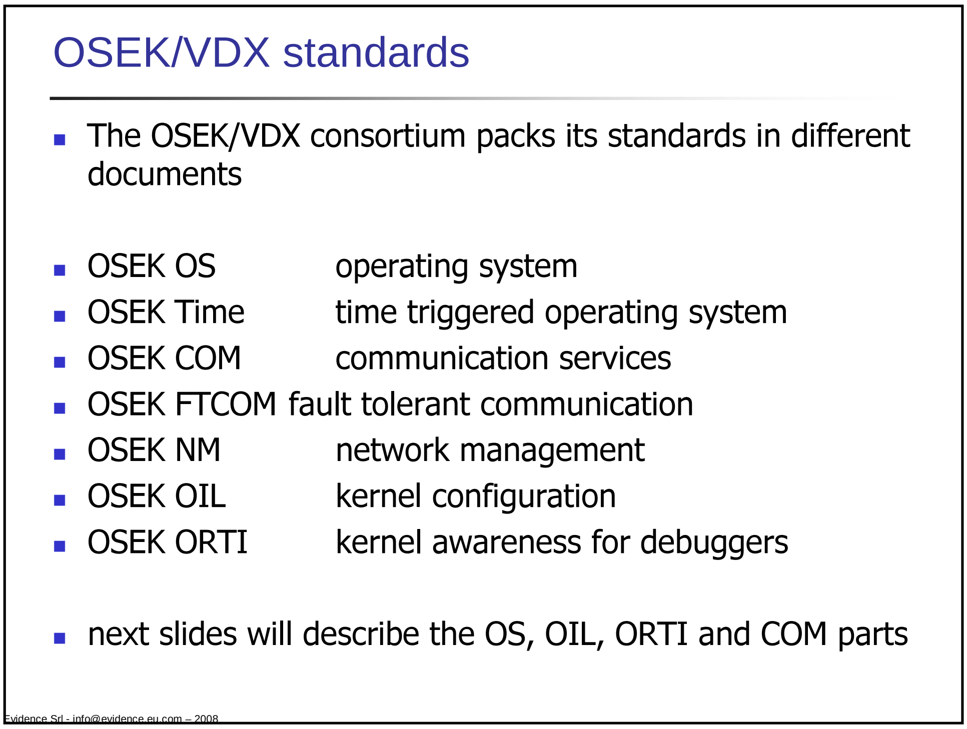
Linker

application C code

C/ASM Compiler

objects .o

RTOS library .a



OSEK/VDX standards

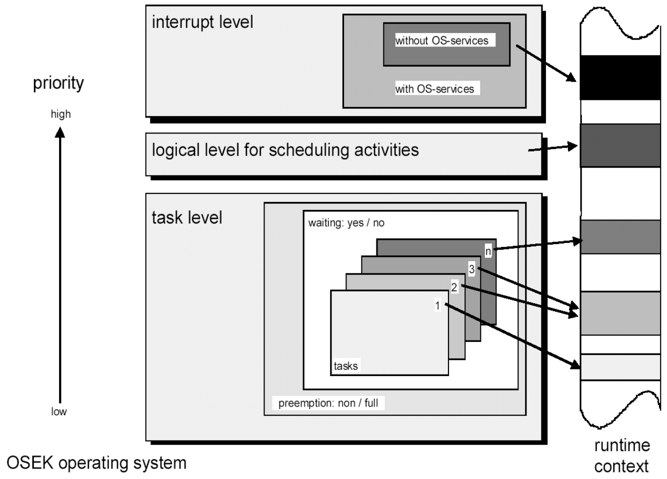
- The OSEK/VDX consortium packs its standards in different documents

OSEK OS	operating system
OSEK Time	time triggered operating system
OSEK COM	communication services
OSEK FTCOM	fault tolerant communication
OSEK NM	network management
OSEK OIL	kernel configuration
OSEK ORTI	kernel awareness for debuggers

- next slides will describe the OS, OIL, ORTI and COM parts

processing levels

- the OSEK OS specification describes the processing levels that have to be supported by an OSEK operating system

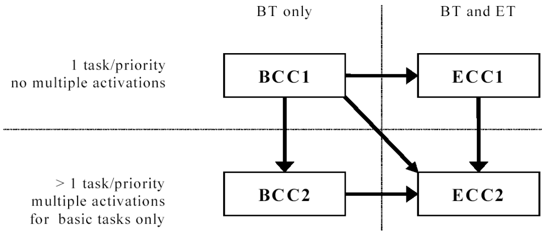


conformance classes

- OSEK OS should be scalable with the application needs
  - different applications require different services
  - the system services are mapped in Conformance Classes
- a conformance class is a subset of the OSEK OS standard
- objectives of the conformance classes
  - allow partial implementation of the standard
  - allow an upgrade path between classes
- services that discriminates the different conformance classes
  - multiple requests of task activations
  - task types
  - number of tasks per priority

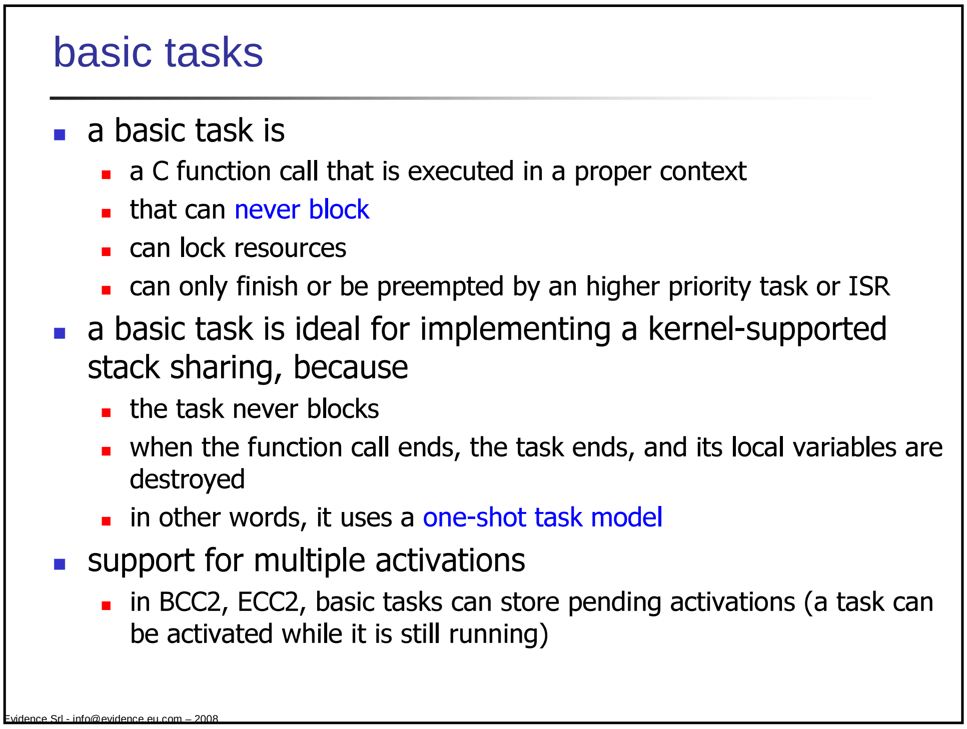
conformance classes (2)

- there are four conformance classes
  - BCC1
    basic tasks, one activation, one task per priority
  - BCC2
    BCC1 plus: > 1 activation, > 1 task per priority
  - ECC1
    BCC1 plus: extended tasks
  - ECC2
    ECC1 plus: > 1 activation (basic tasks), > 1 task per priority



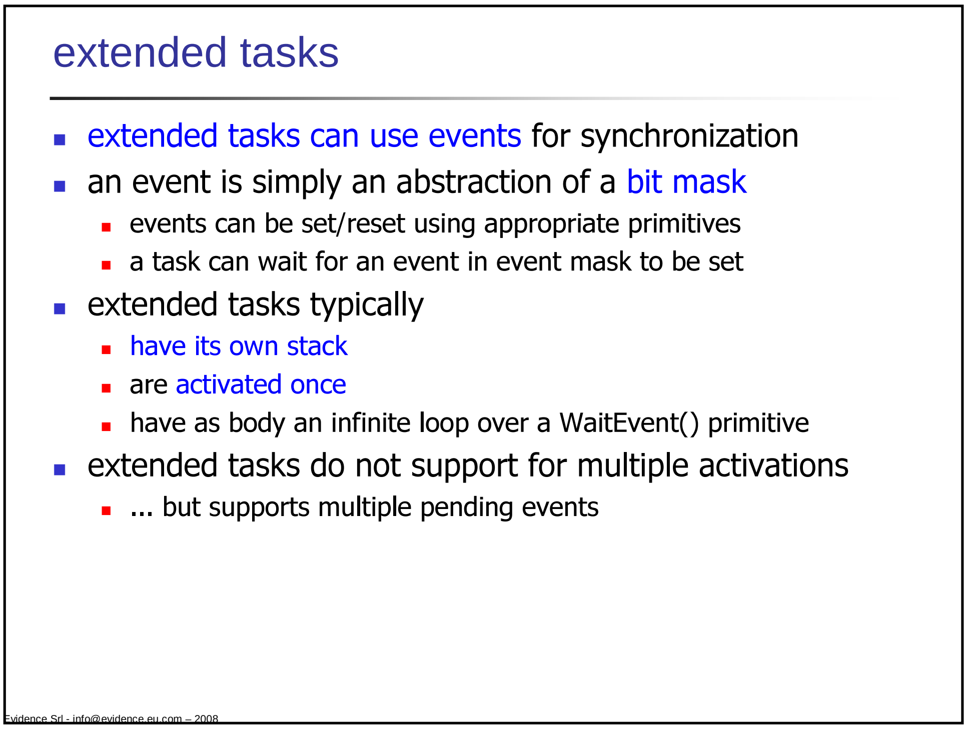
conformance classes (3)

	BCC1	BCC2	ECC1	ECC2
Multiple requesting of task activation	no	yes	BT[3]: no ET: no	BT: yes ET: no
Number of tasks which are not in the *suspended* state	8		16 (any combination of BT/ET)	
More than one task per priority	no	yes	no (both BT/ET)	yes (both BT/ET)
Number of events per task	—		8	
Number of task priorities	8		16	
Resources	RES_SCHEDULER	8 (including RES_SCHEDULER)		
Internal resources	2			
Alarm	1			
Application Mode	1			



basic tasks

- a basic task is
  - a C function call that is executed in a proper context
  - that can never block
  - can lock resources
  - can only finish or be preempted by an higher priority task or ISR
- a basic task is ideal for implementing a kernel-supported stack sharing, because
  - the task never blocks
  - when the function call ends, the task ends, and its local variables are destroyed
  - in other words, it uses a one-shot task model
- support for multiple activations
  - in BCC2, ECC2, basic tasks can store pending activations (a task can be activated while it is still running)



extended tasks

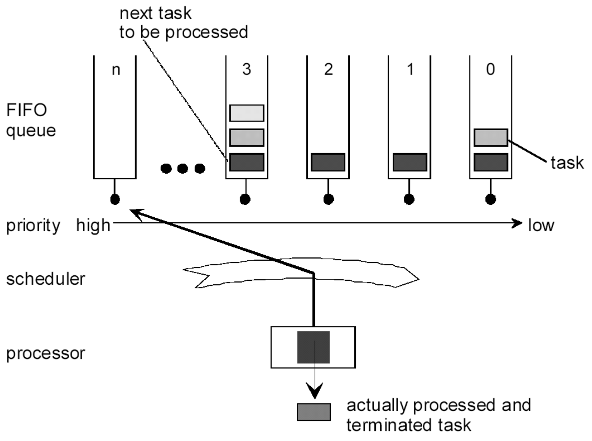
- extended tasks can use events for synchronization
- an event is simply an abstraction of a bit mask
  - events can be set/reset using appropriate primitives
  - a task can wait for an event in event mask to be set
- extended tasks typically
  - have its own stack
  - are activated once
  - have as body an infinite loop over a WaitEvent() primitive
- extended tasks do not support for multiple activations
  - ... but supports multiple pending events

scheduling algorithm

- the scheduling algorithm is fundamentally a
  - fixed priority scheduler
  - with immediate priority ceiling
  - with preemption threshold
- the approach allows the implementation of
  - preemptive scheduling
  - non preemptive scheduling
  - mixed
- with some peculiarities...

scheduling algorithm: peculiarities

- multiple activations of tasks with the same priority
  - are handled in FIFO order
  - that imposes in some sense the internal scheduling data structure



OSEK task primitives (basic and extended tasks)

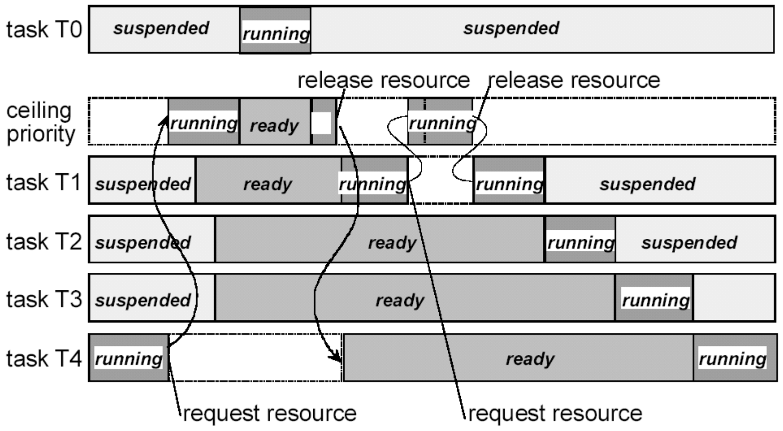
- TASK(<TaskIdentifier>) {…}
  - used to define a task body (it’s a macro!)
- DeclareTask(<TaskIdentifier>)
  - used to declare a task name (it’s a macro!)
- StatusType ActivateTask(TaskType <TaskID>)
  - activates a task
- StatusType TeminateTask(void)
  - terminates the current running task (from any function nesting!)
- StatusType ChainTask(TaskType <TaskID>)
  - atomic version of TerminateTask+ActivateTask
- StatusType Schedule(void)
  - rescheduling point for a non-preemptive task
- StatusType GetTaskID(TaskRefType <TaskID>)
  - returns the running task ID
- StatusType GetTaskState(TaskType <TaskID>, TaskStateRefType <State>)
  - returns the status of a given task

OSEK event primitives

- DeclareEvent(<EventIdentifier>)
  - declaration of an Event identifier (it’s a macro!)
- StatusType SetEvent(TaskType <TaskID>,
  EventMaskType <Mask>)
  - sets a set of event flags to an extended task
- StatusType ClearEvent(EventMaskType <Mask>)
  - clears an event mask (extended tasks only)
- StatusType GetEvent(TaskType <TaskID>,
  EventMaskRefType <Event>)
  - gets an event mask
- StatusType WaitEvent(EventMaskType <Mask>)
  - waits for an event mask (extended tasks only)
  - this is the only blocking primitive of the OSEK standard

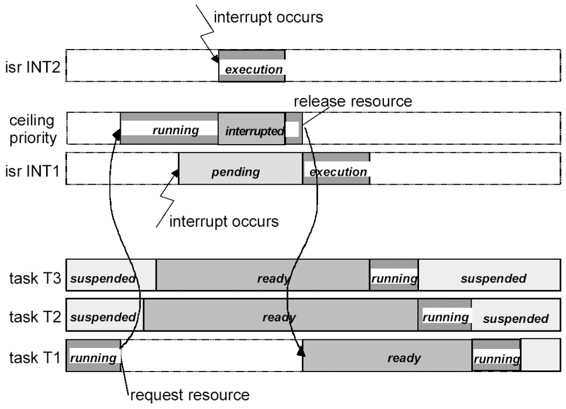
scheduling algorithm: resources

- resources
  - are typical Immediate Priority Ceiling mutexes
  - the priority of the task is raised when the task locks the resource



scheduling algorithm: resources (2)

- resources at interrupt level
  - resources can be used at interrupt level
  - for example, to protects drivers
  - the code directly have to operate on the interrupt controller

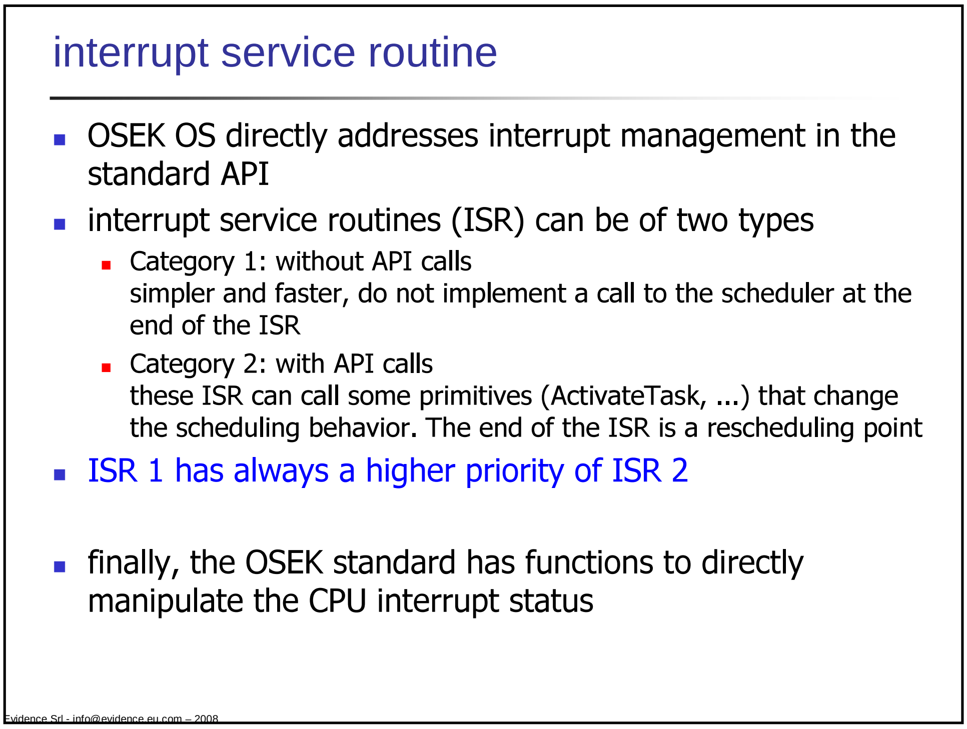


scheduling algorithm: resources (3)

- preemption threshold implementation
  - done using “internal resources” that are locked when the task starts and unlocked when the task ends
  - internal resources cannot be used by the application

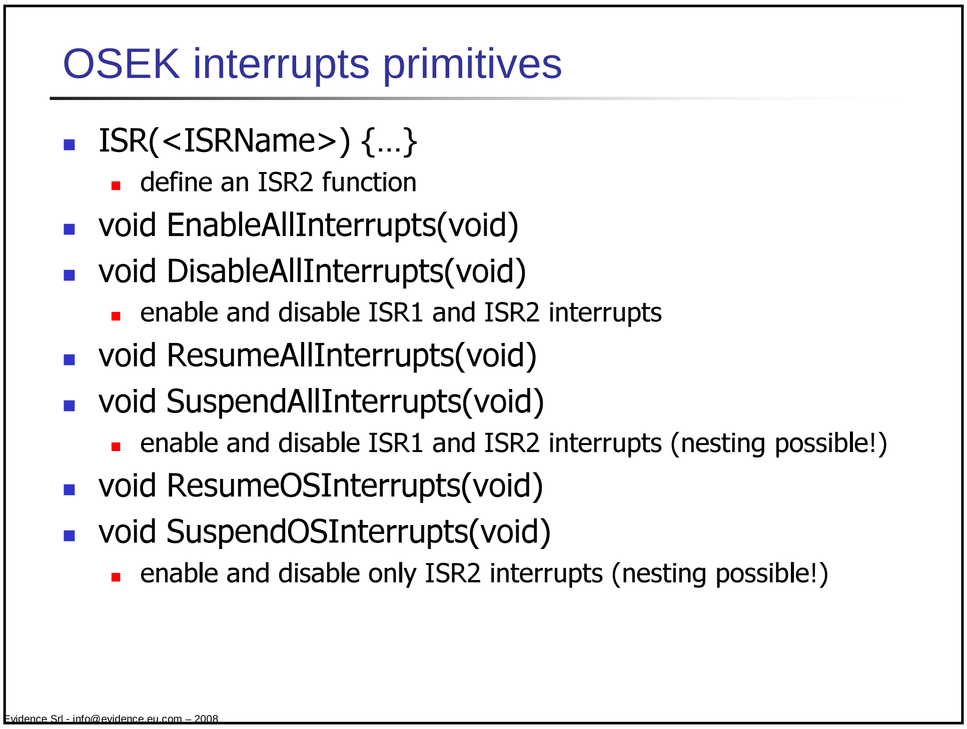
OSEK resource primitives

- DeclareResource(<ResourceIdentifier>)
  - used to define a task body (it’s a macro!)
- StatusType GetResource(ResourceType <ResID>)
  - resource lock function
- StatusType ReleaseResource(ResourceType <ResID>)
  - resource unlock function
- RES_SCHEDULER
  - resource usd by every task →the task becomes non preemptive



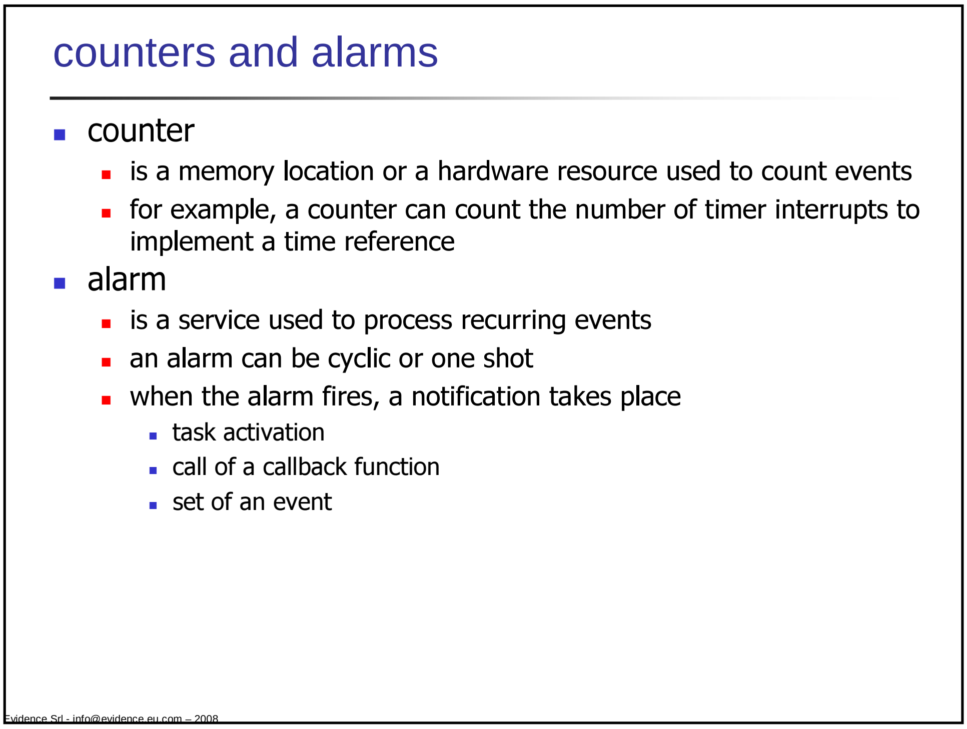
interrupt service routine

- OSEK OS directly addresses interrupt management in the standard API
- interrupt service routines (ISR) can be of two types
  - Category 1: without API calls
    simpler and faster, do not implement a call to the scheduler at the end of the ISR
  - Category 2: with API calls
    these ISR can call some primitives (ActivateTask, ...) that change the scheduling behavior. The end of the ISR is a rescheduling point
- ISR 1 has always a higher priority of ISR 2
- finally, the OSEK standard has functions to directly manipulate the CPU interrupt status



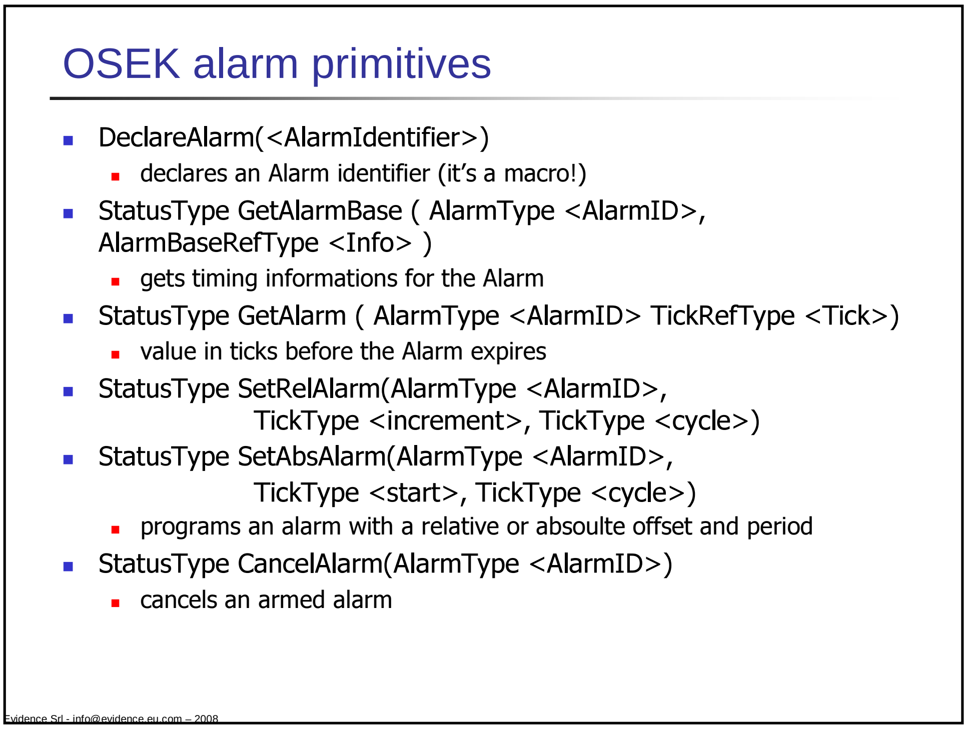
OSEK interrupts primitives

- ISR(<ISRName>) {…}
  - define an ISR2 function
- void EnableAllInterrupts(void)
- void DisableAllInterrupts(void)
  - enable and disable ISR1 and ISR2 interrupts
- void ResumeAllInterrupts(void)
- void SuspendAllInterrupts(void)
  - enable and disable ISR1 and ISR2 interrupts (nesting possible!)
- void ResumeOSInterrupts(void)
- void SuspendOSInterrupts(void)
  - enable and disable only ISR2 interrupts (nesting possible!)



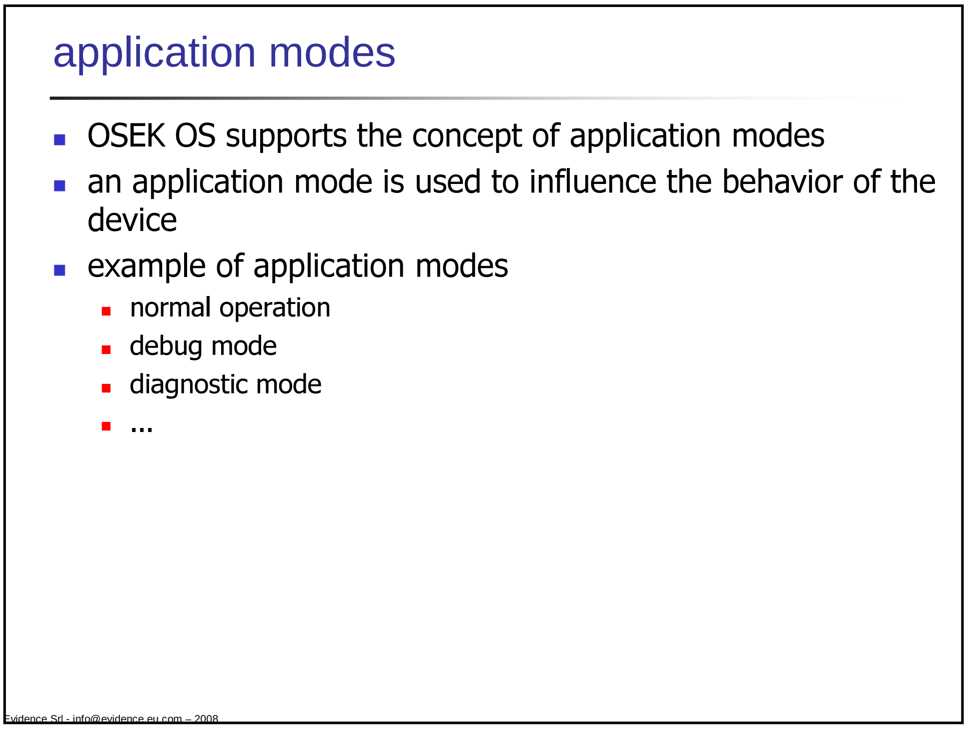
counters and alarms

- counter
  - is a memory location or a hardware resource used to count events
  - for example, a counter can count the number of timer interrupts to implement a time reference
- alarm
  - is a service used to process recurring events
  - an alarm can be cyclic or one shot
  - when the alarm fires, a notification takes place
    - task activation
    - call of a callback function
    - set of an event



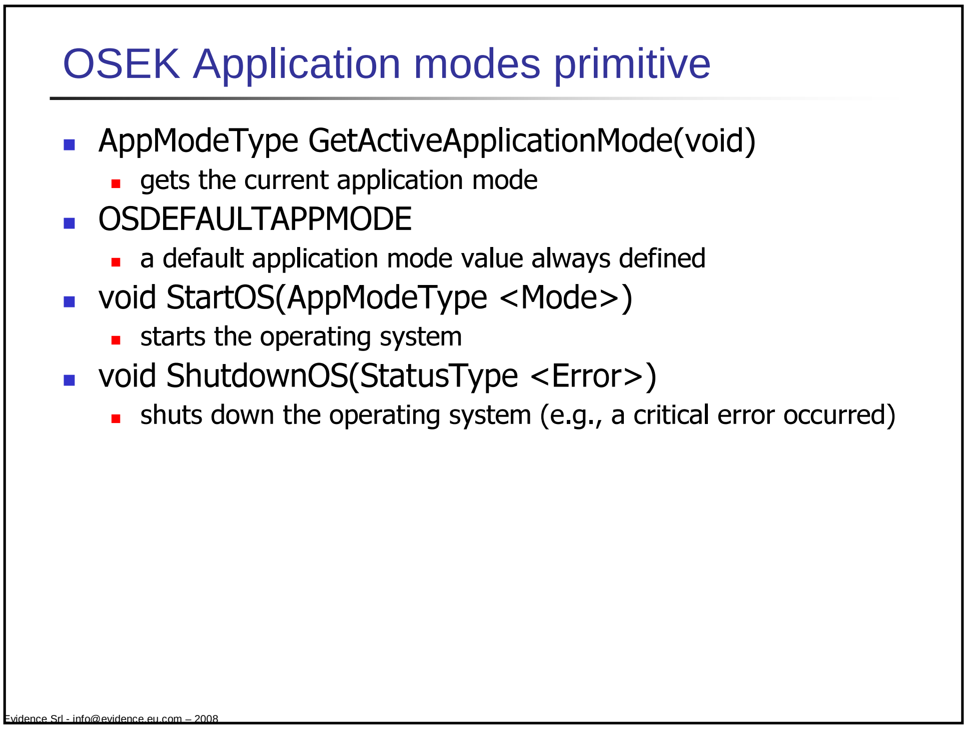
OSEK alarm primitives

- DeclareAlarm(<AlarmIdentifier>)
  - declares an Alarm identifier (it's a macro!)
- StatusType GetAlarmBase (AlarmType <AlarmID>, AlarmBaseRefType <Info>)
  - gets timing informations for the Alarm
- StatusType GetAlarm (AlarmType <AlarmID> TickRefType <Tick>)
  - value in ticks before the Alarm expires
- StatusType SetRelAlarm(AlarmType <AlarmID>,
  TickType <increment>, TickType <cycle>)
- StatusType SetAbsAlarm(AlarmType <AlarmID>,
  TickType <start>, TickType <cycle>)
  - programs an alarm with a relative or absoulte offset and period
- StatusType CancelAlarm(AlarmType <AlarmID>)
  - cancels an armed alarm



application modes

- OSEK OS supports the concept of application modes
- an application mode is used to influence the behavior of the device
- example of application modes
  - normal operation
  - debug mode
  - diagnostic mode
  - …

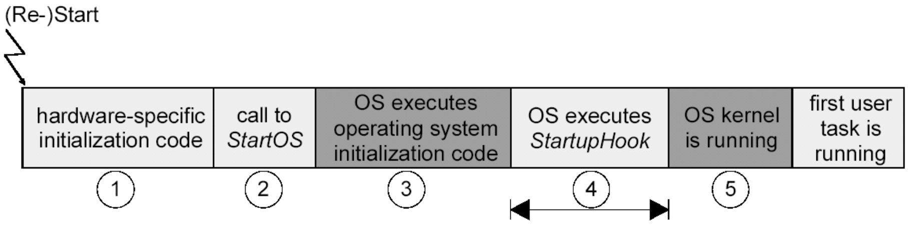


OSEK Application modes primitive

- AppModeType GetActiveApplicationMode(void)
  - gets the current application mode
- OSDEFAULTAPPMODE
  - a default application mode value always defined
- void StartOS(AppModeType <Mode>)
  - starts the operating system
- void ShutdownOS(StatusType <Error>)
  - shuts down the operating system (e.g., a critical error occurred)

hooks

- OSEK OS specifies a set of hooks that are called at specific times
  - StartupHook
    when the system starts



(Re-)Start

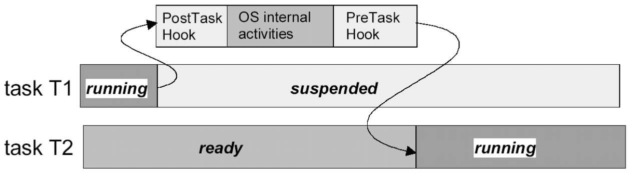
hardware-specific initialization code | call to *StartOS* | OS executes operating system initialization code | OS executes *StartupHook* | OS kernel is running | first user task is running

1 2 3 4 5

During StartupHook
all user interrupts are disabled

hooks (2)

- PreTaskHook
  before a task is scheduled
- PostTaskHook
  after a task has finished its slice

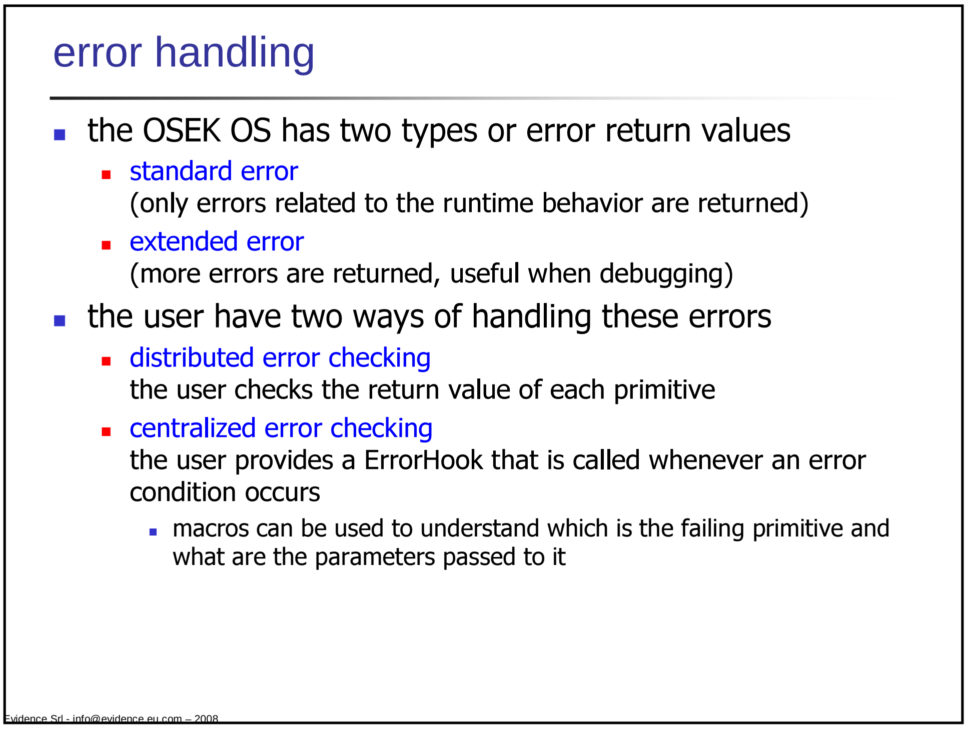


PostTask Hook | OS internal activities | PreTask Hook

task T1: ***running*** | ***suspended***

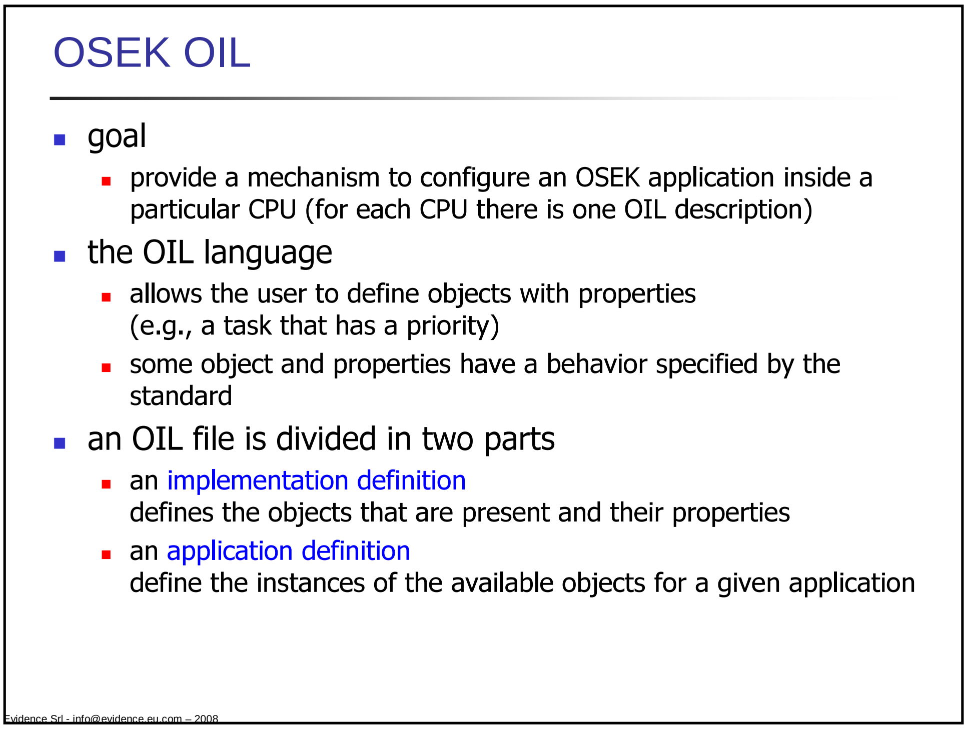
task T2: ***ready*** | ***running***

- ShutdownHook
  when the system is shutting down (usually because of an unrecoverable error)
- ErrorHook
  when a primitive returns an error



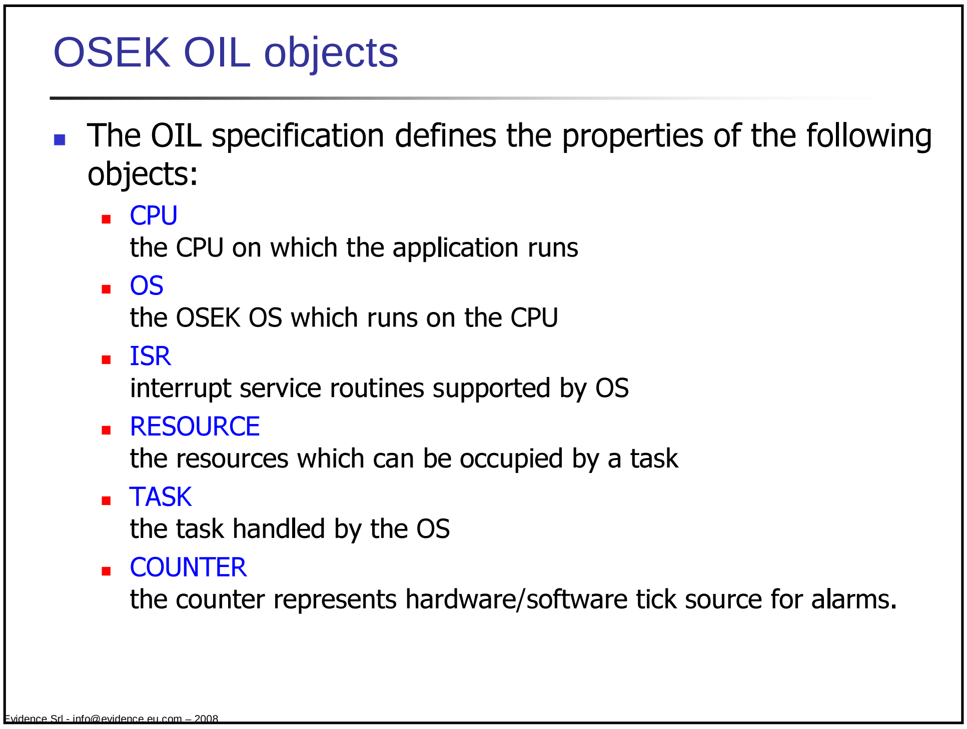
error handling

- the OSEK OS has two types or error return values
  - standard error
    (only errors related to the runtime behavior are returned)
  - extended error
    (more errors are returned, useful when debugging)
- the user have two ways of handling these errors
  - distributed error checking
    the user checks the return value of each primitive
  - centralized error checking
    the user provides a ErrorHook that is called whenever an error condition occurs
    - macros can be used to understand which is the failing primitive and what are the parameters passed to it



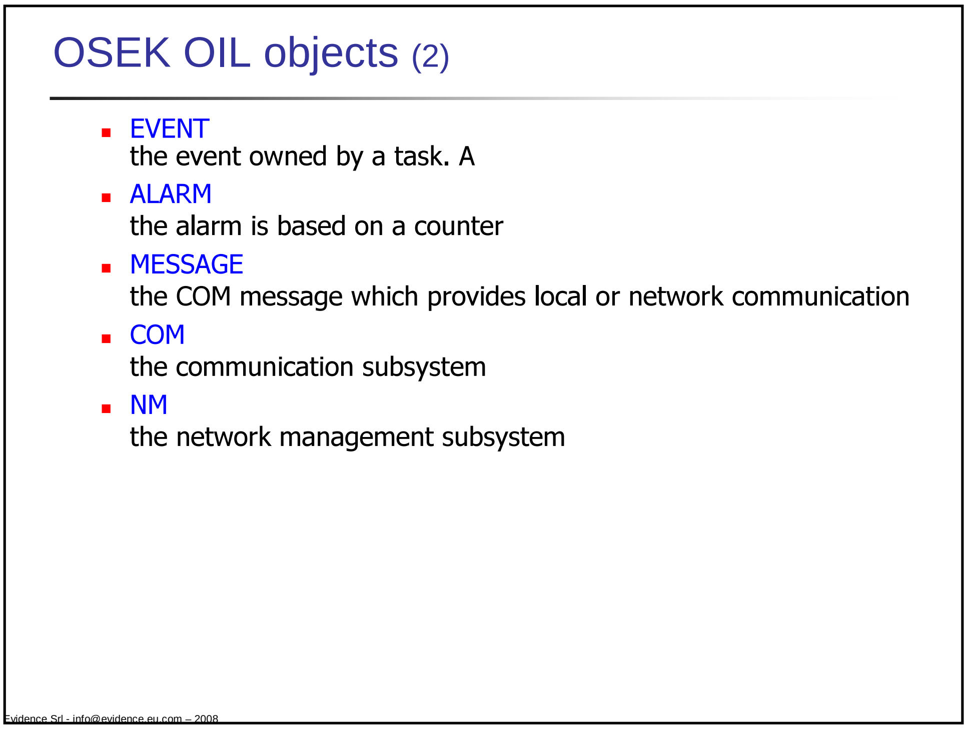
OSEK OIL

- goal
  - provide a mechanism to configure an OSEK application inside a particular CPU (for each CPU there is one OIL description)
- the OIL language
  - allows the user to define objects with properties
    (e.g., a task that has a priority)
  - some object and properties have a behavior specified by the standard
- an OIL file is divided in two parts
  - an implementation definition
    defines the objects that are present and their properties
  - an application definition
    define the instances of the available objects for a given application



OSEK OIL objects

- The OIL specification defines the properties of the following objects:
  - CPU
    the CPU on which the application runs
  - OS
    the OSEK OS which runs on the CPU
  - ISR
    interrupt service routines supported by OS
  - RESOURCE
    the resources which can be occupied by a task
  - TASK
    the task handled by the OS
  - COUNTER
    the counter represents hardware/software tick source for alarms.



OSEK OIL objects (2)

- EVENT
  the event owned by a task. A
- ALARM
  the alarm is based on a counter
- MESSAGE
  the COM message which provides local or network communication
- COM
  the communication subsystem
- NM
  the network management subsystem

OIL example: implementation definition

```
OIL_VERSION = "2.4";

IMPLEMENTATION my_osek_kernel {
[...]
    TASK {
     BOOLEAN [
         TRUE { APPMODE_TYPE APPMODE[]; },
         FALSE
        ] AUTOSTART;
        UINT32 PRIORITY;
        UINT32 ACTIVATION = 1;
        ENUM [NON, FULL] SCHEDULE;
        EVENT_TYPE EVENT[];
        RESOURCE_TYPE RESOURCE[];

        /* my_osek_kernel specific values */
        ENUM [
            SHARED,
            PRIVATE { UINT32 SIZE; }
        ] STACK;
    };
[...]
};
```

OIL example: application definition

```
CPU my_application {
    TASK Task1 {
        PRIORITY = 0x01;
        ACTIVATION = 1;
        SCHEDULE = FULL;
        AUTOSTART = TRUE;
        STACK = SHARED;
    };
};
```