Real Time Systems (RETSY)

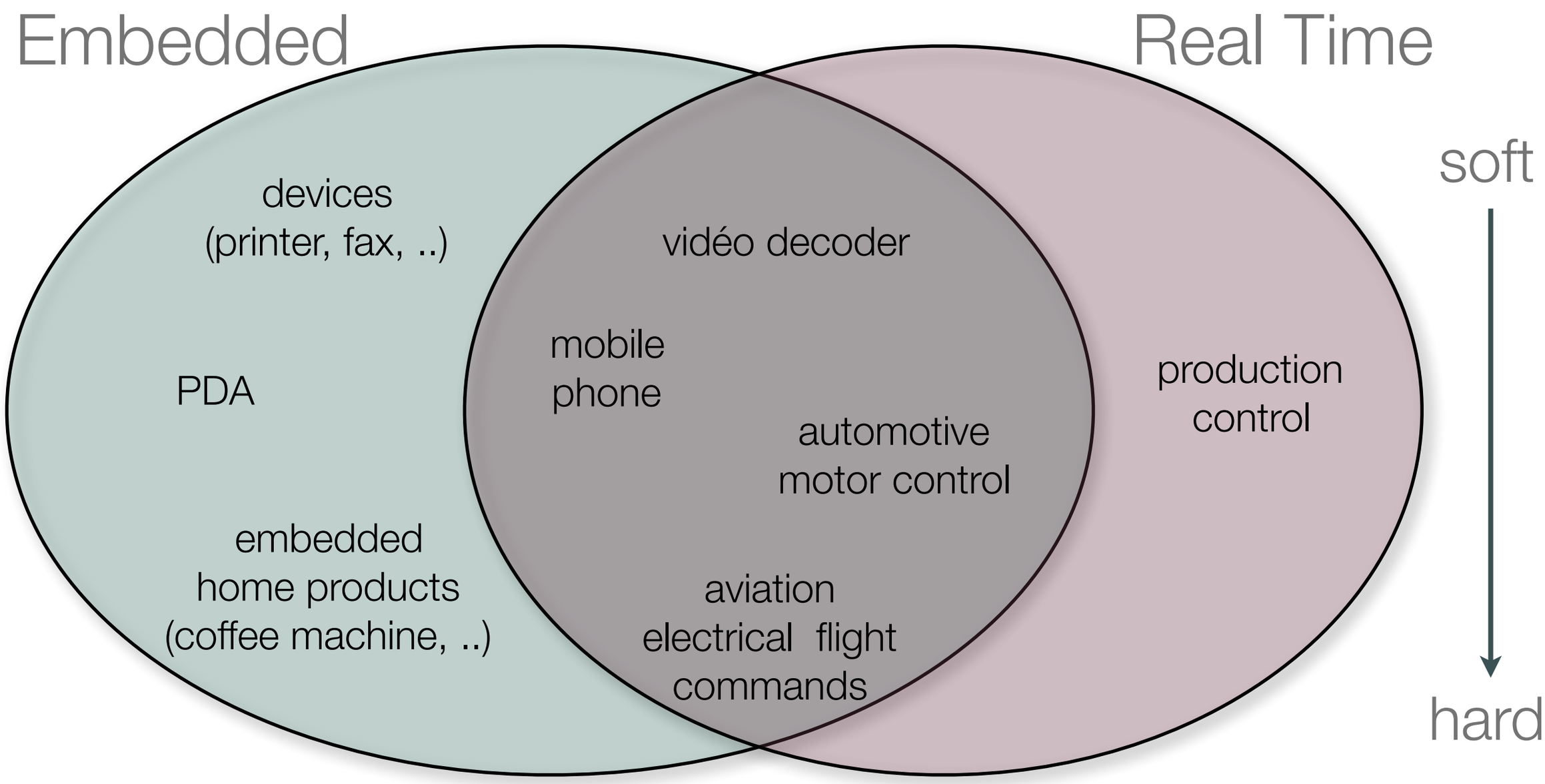
Jean-Luc Béchennec - Jean-Luc.Bechennec@irccyn.ec-nantes.fr
Sébastien Faucou - Sebastien.Faucou@univ-nantes.fr

Generalities

Real-time systems

- A real-time system is a computing system that must produce a response within a specified (and known) time.
  - Implies a deadline.
  - Response after the deadline is a failure.
- Usually the input is an event or a data in the physical world.
  - Output is related to that event or data.
  - Output has to be produced quick enough to be compatible with the controlled process.
- Correctness of the system depends on :
  - The functional correctness (as in any system)
  - The timeliness : time critical systems

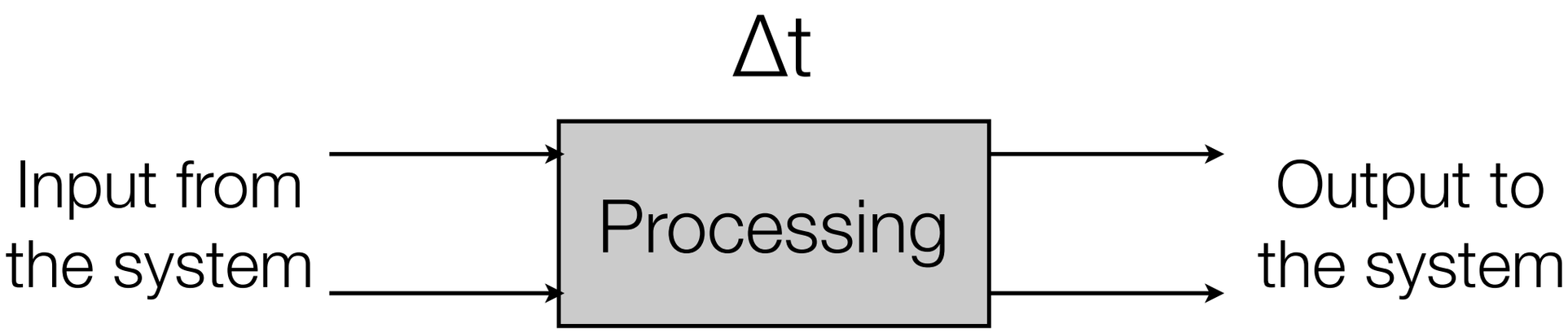
Embedded systems *vs* Real Time systems



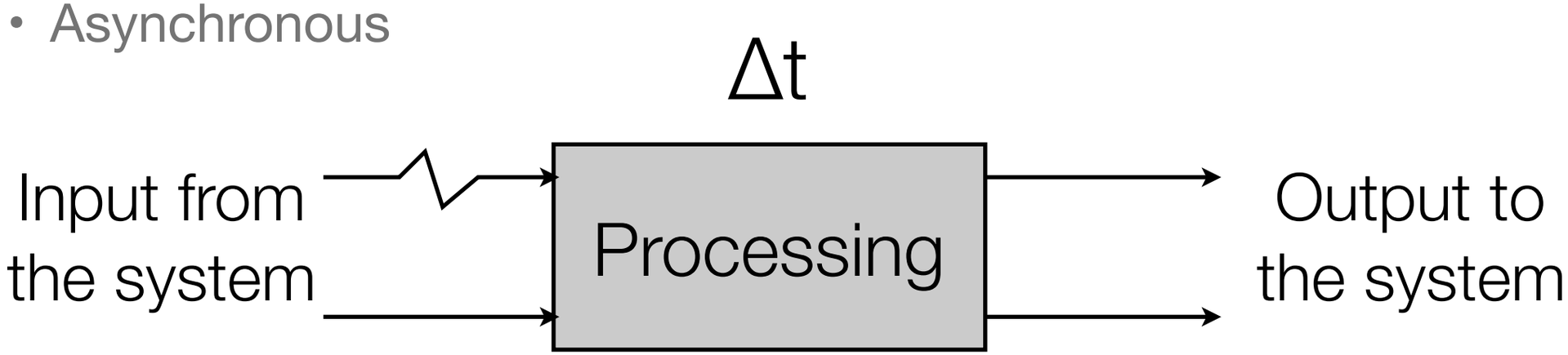
source:
I. Puaut

Real-time systems

- Simple system :
  - One task only



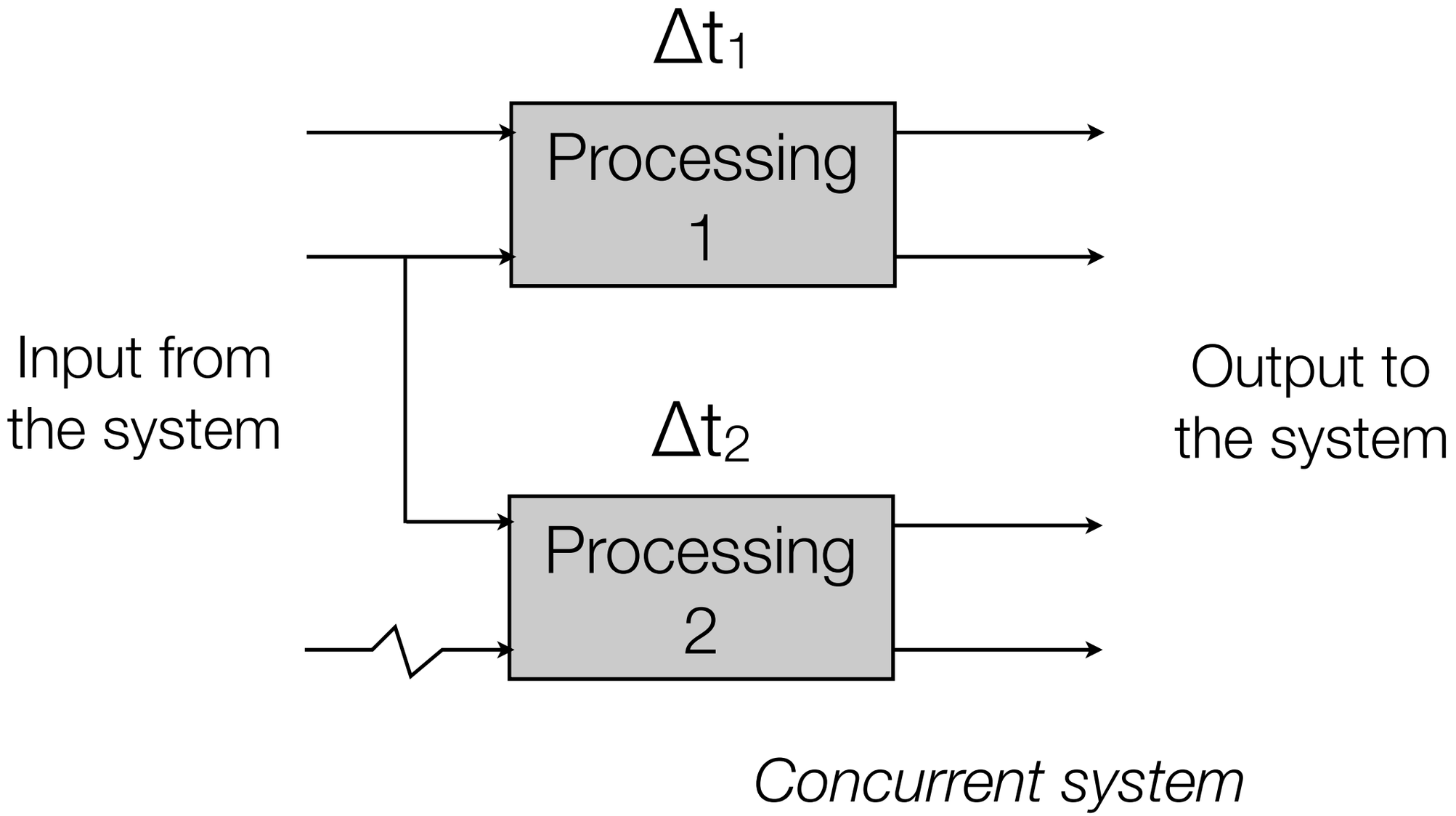
- Asynchronous



a bit of concurrency

Real-time systems

- More complex :
  - Multitask



Concurrent system

What means concurrent ?

- A computer is a sequential machine
  - A program is sequential (most of the time)
  - A program is a sequence of instructions
  - A computer executes the instructions one after the other
- Processing many tasks on the same computer means tasks are *concurrent* for their execution

How to manage concurrency ?

- Bare metal approach
  - Program runs directly on the hardware
  - Asynchronous events are managed by interrupt handlers
  - Concurrency between interrupt handlers and normal execution
  - Limited to small and less complex systems.

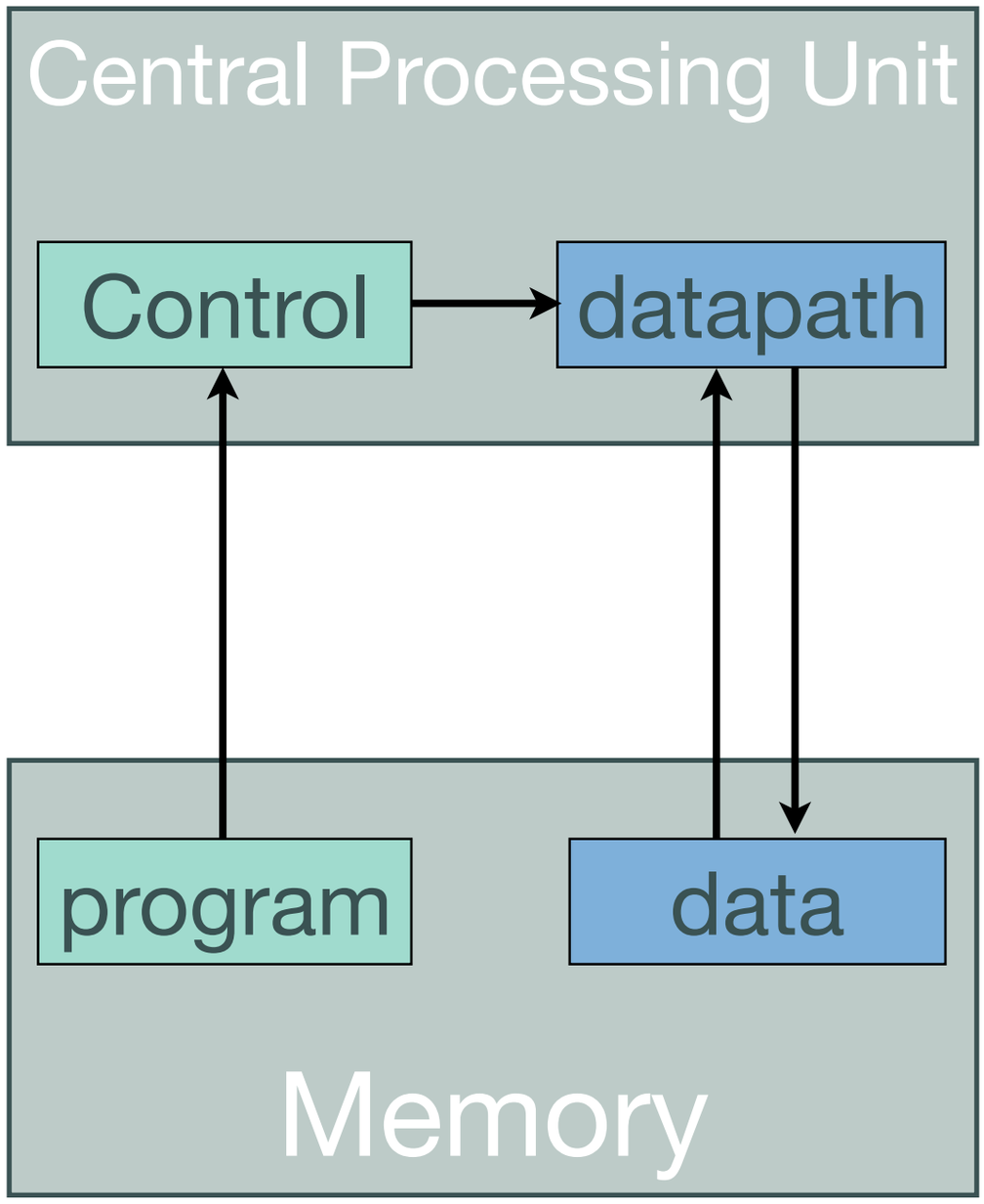
How to manage concurrency ?

- Sequencer approach
  - The sequencer is a sequential program running in loop
  - The sequencer measures the time and calls functions (tasks) according to their activation period
  - Periodic system, predictable, deterministic
  - Not very flexible, all tasks are mixed, non preemptable.

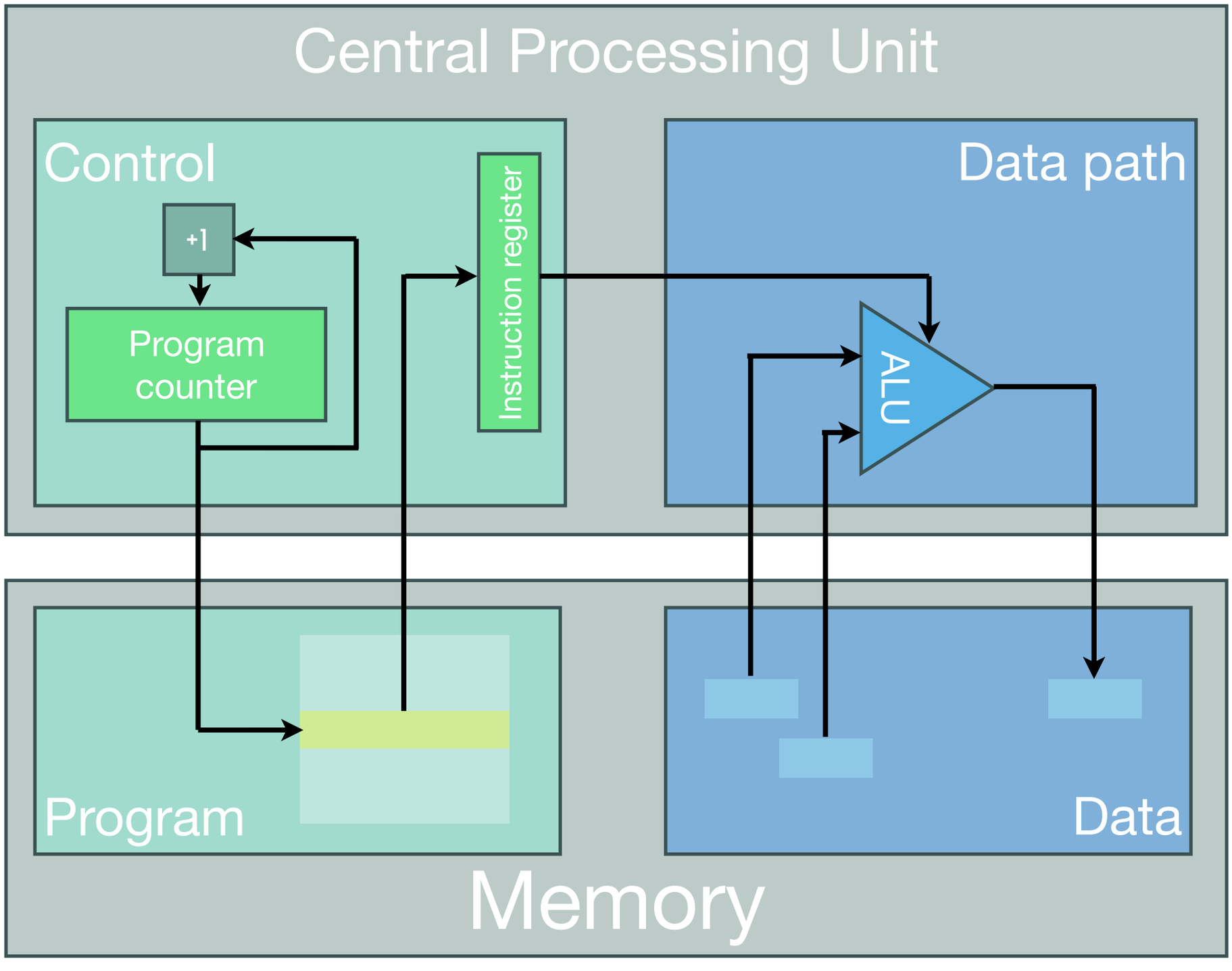
How to manage concurrency ?

- Real-time operating system (RTOS) preemptive
  - Uses real-time *scheduling*
  - Periodic system, predictable, deterministic
  - Preemptive system :
    - A running task mais be stopped to allow a higher priority task to run and resumed later.
- Tasks of the application are separated from the RTOS
- Separation of concerns, reusability, abstraction layer.

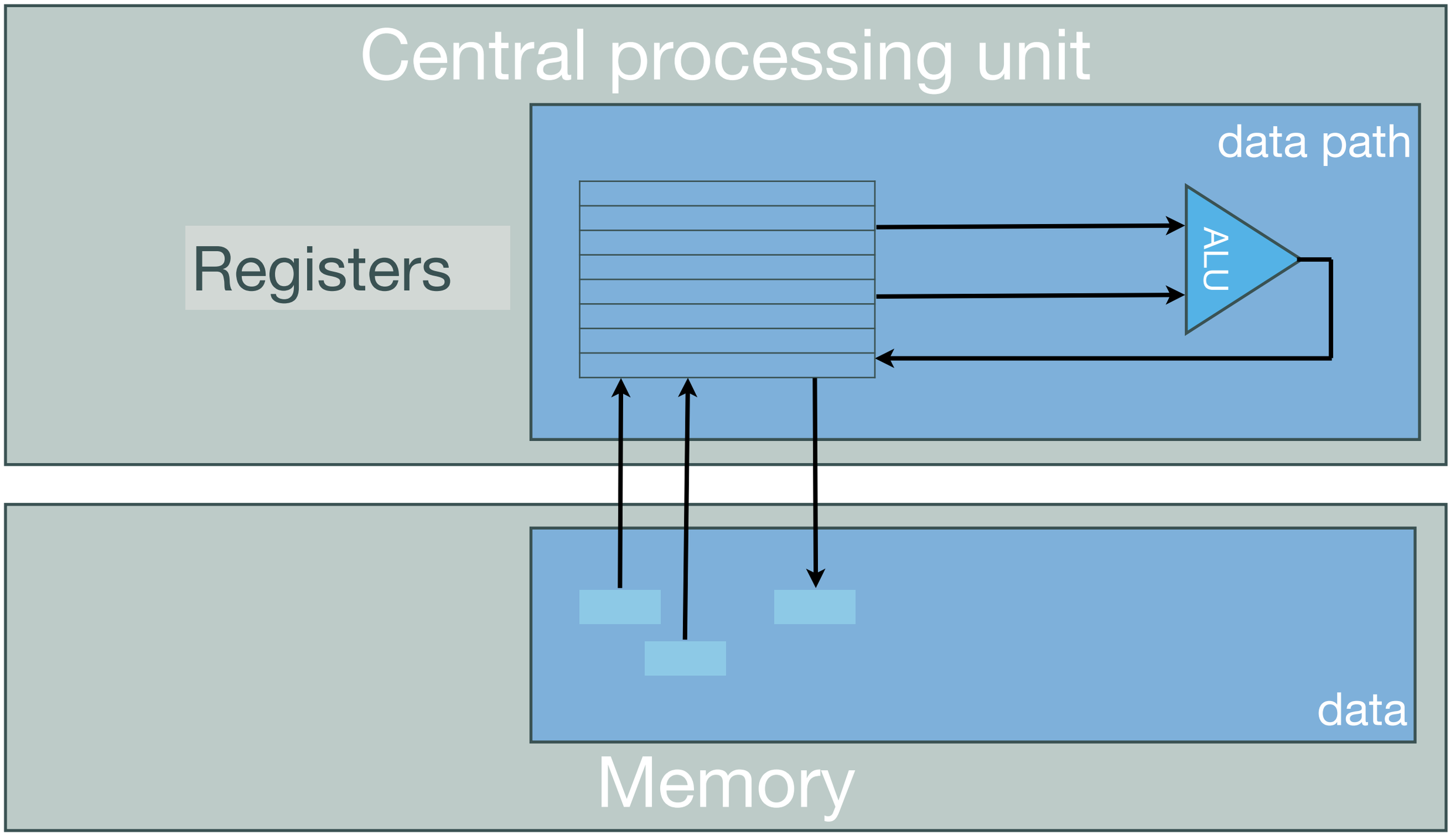
Very simplified drawing



A little bit less simple



Focus on registers

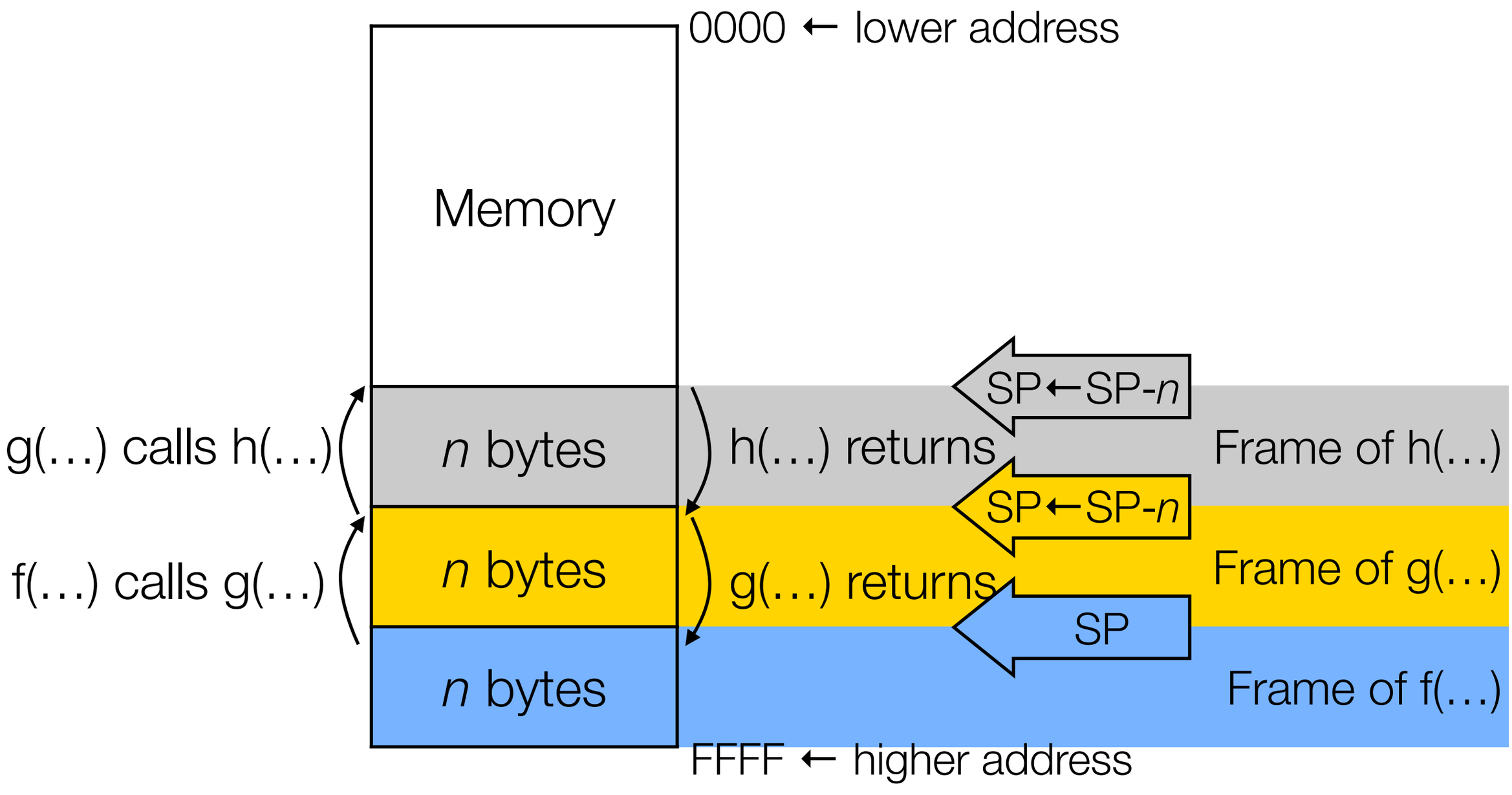


So what means preemptible multitasking ?

- A program executes in a context : the content of a part of the registers of the CPU
- Registers are used to :
  - store values to compute : general purpose registers
  - store the address of the instruction to execute : the program counter (PC)
  - store the address of the stack : the stack pointer (SP)
    - The stack is a zone in memory used to store local variables of a function
    - When a function is called the stack pointer is decremented to make room for the local variables
    - When a function returns the stack pointer is restored to its previous value

So what means preemptible multitasking ?

- Stack and stack frames



So what means preemptible multitasking ?

- Each task is a separate program. It has:
  - A context and a memory zone to store it
  - A memory zone to store its stack and its variables
- « stopping » a task means:
  - storing the context (registers) in its memory zone
- « starting » a task means:
  - loading the context (registers) from its memory zone. SP being a register the stack is changed accordingly
- « preemptable » means:
  - **at any time, the RTOS may, in response to an interrupt (so asynchronously), stop the running task and start another task. The running task is preempted.**

RTOS vs General purpose OS (GPOS)

- a GPOS scheduler is designed for high throughput :
  - time is divided in slice and the priority of a task is used to allocate a part of the slice proportional to the priority.
  - no bounded latency, no deadline garantee.
- systems are bigger: more RAM, more MHz, disk storage

- a RTOS scheduler is designed with predictability in mind :
  - predictable latencies.
  - predictable execution.
  - precise dispatch of tasks along a timeline.
- systems are small from a few KBytes to less than a MBytes

OSEK/VDX and AUTOSAR compliant RTOS

Jean-Luc Béchennec - Jean-Luc.Bechennec@irccyn.ec-nantes.fr
Sébastien Faucou - Sebastien.Faucou@univ-nantes.fr

Context 1/3

- Embedded electronic in vehicles with hard and soft real-time constraints
  - PowerTrain, Chassis, Body, Telematics
- High economical constraints
  - Small computers (16 bits, few RAM)
- Distributed systems
  - Based on standards like CAN, LIN and now FlexRay
- High dependability expected
  - ABS, ESP, AirBag, ...

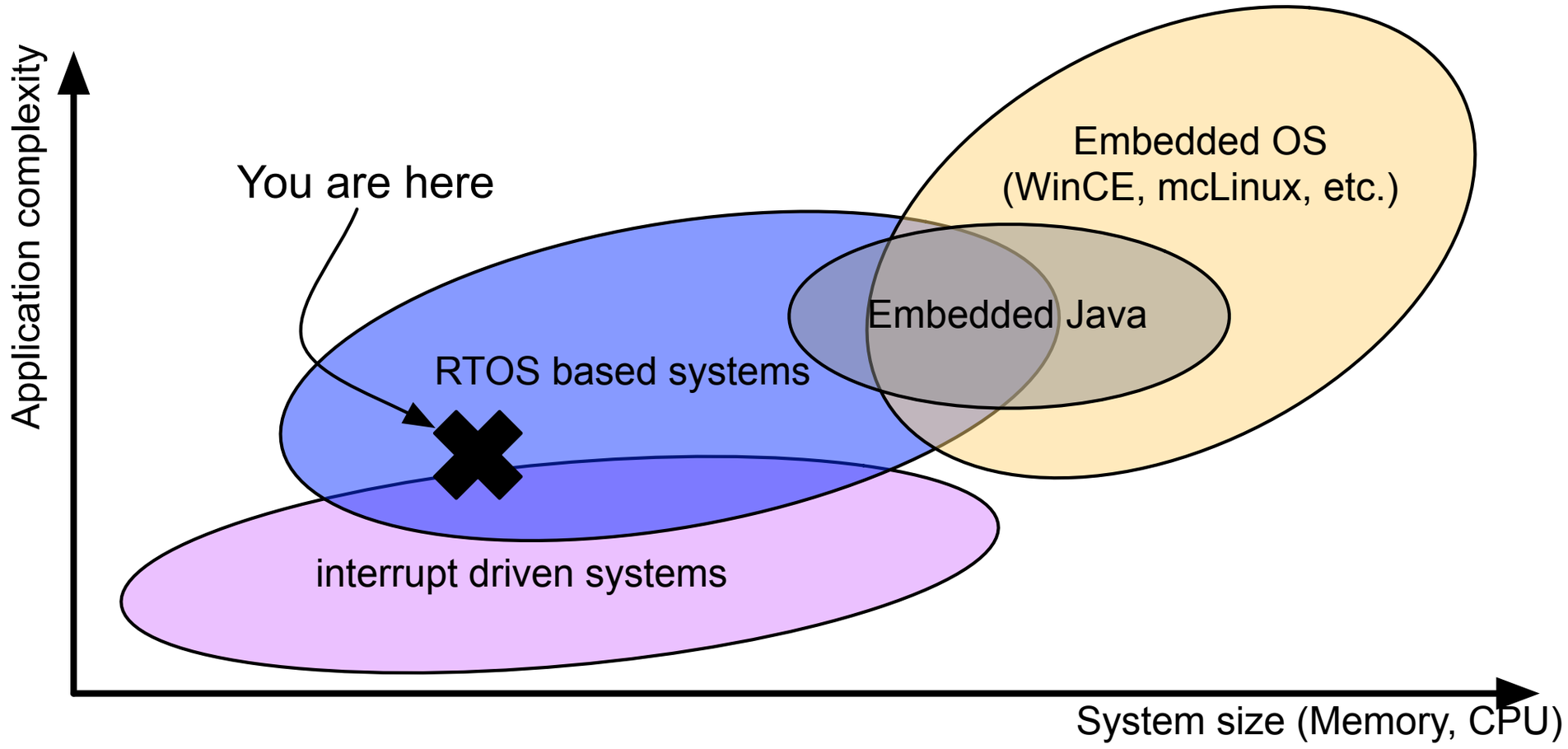
Context 2/3

- OSEK/VDX : "Öffene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug / Vehicle Distributed Executive” (Open Systems and their Interfaces for the Electronics in Motor Vehicles)
  - Industrial and academic (from automotive industry and research) consortium
    - steering committee: Opel, BMW, DaimlerChrisler, PSA, Renault, Volkswagen, Robert Bosch, Siemens, University of Karlsruhe
- System specification (architecture, interfaces et behavior) for automotive electronic embedded systems (http://www.osek-vdx.org)
- Foundation of AUTOSAR system
- ISO 17356 standard

Context 3/3

- Motivations
  - High expenses coming from the development and/or port of operating system
  - No interoperability of systems built by different providers
- Chosen approach : specification of the architecture and specification of the software building blocks.
  - Interfaces that are independent from the hardware
  - Well defined behavior to ease the portability
  - A dedicated approach to take into account domain specificities (embedded, real-time, ... and cost)
- Expected advantages
  - Basic software reusability, Application portability.

Where is OSEK compared to other OS



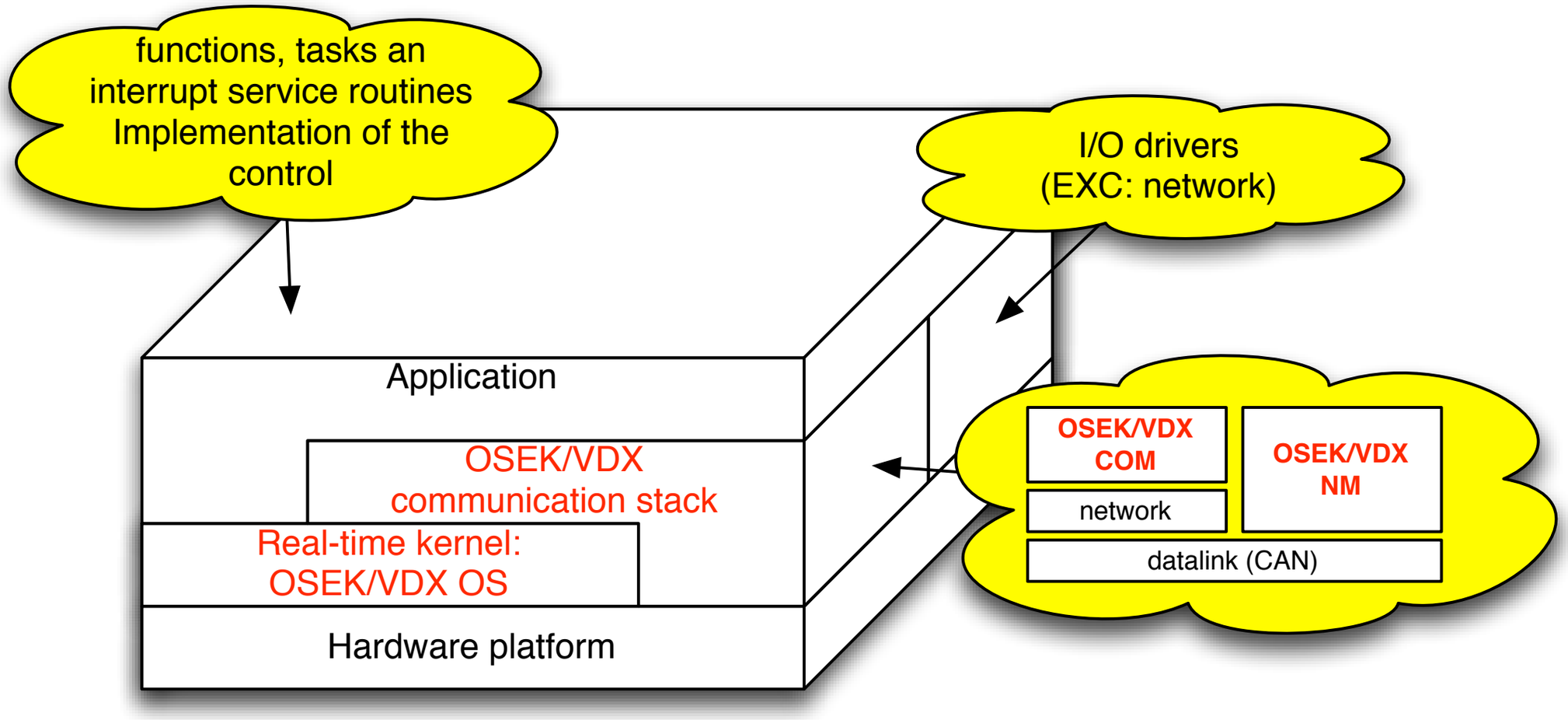
OSEK/VDX specifications

- OSEK/VDX OS : “event-triggered” Real-time kernel
- OSEK/VDX COM : Application level communication protocol
- OSEK/VDX NM : Network management
- OSEK/VDX OIL : Offline application description and configuration language
- OSEK/VDX ORTI : Debugging interface
- OSEK/VDX ttOS et FTCOM : “time-triggered” architecture and components for the most critical systems

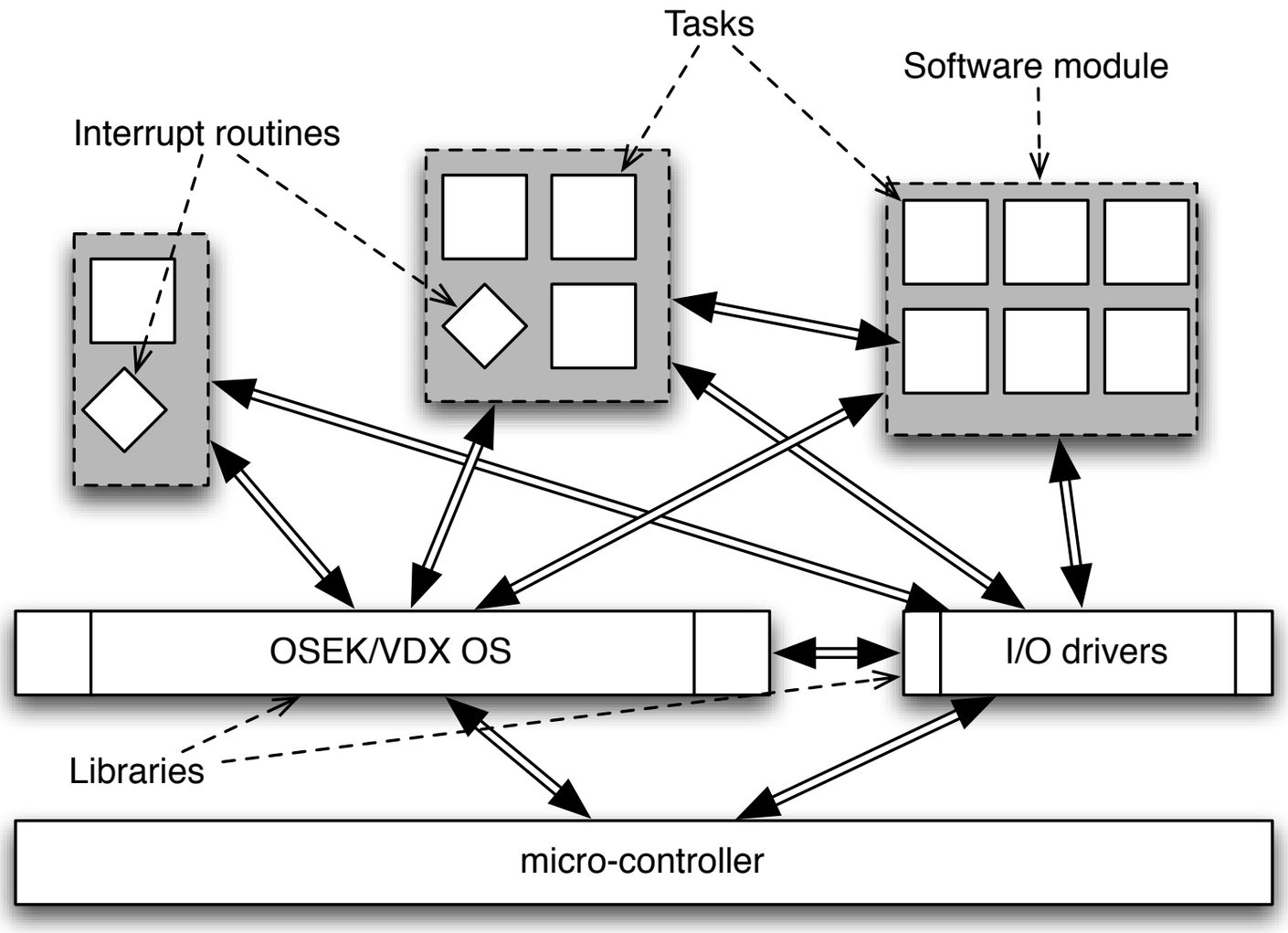
Who is using Trampoline

- EE/CS department of universities in Nantes, Rennes, Angers, Toulouse (??)
  - Teaching (labs in embedded and real-time systems)
  - Research
  - In both cases, an open source project is a good choice (freedom to study and modify)
- Car makers (Renault, PSA, VOLVO), suppliers (See4sys, AUTOLIV), software editors (Trialog, Dassault Systems), Research institutes (IRT SystemX)
  - Internal R&D projects, internal training
  - Some others are interested by the freedom of use as in free speech
- + Some anonymous users from Germany, India, UK, China, etc. (from the web site logs)

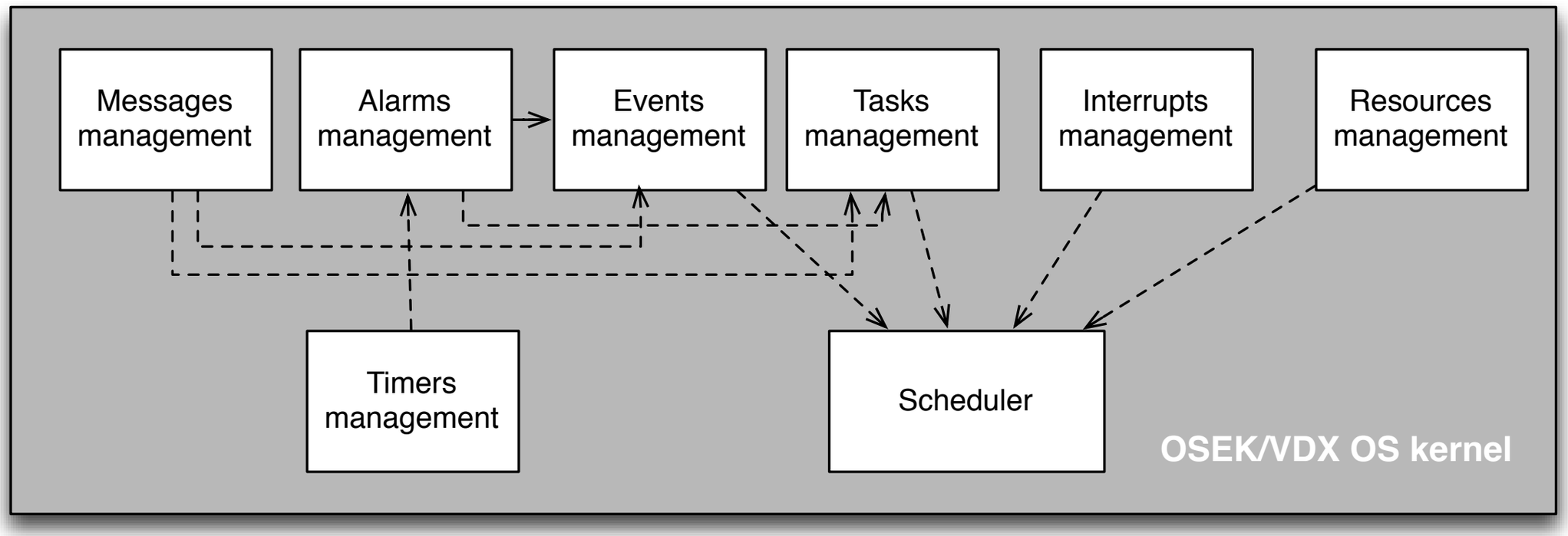
ECU software architecture



From the OSEK/VDX OS point of view



Zoom on OSEK/VDX OS



Main features

- Designed for automotive domain:
  - based on few (but enough) concepts
  - static configuration (offline) : The application architecture is completely known
    - Greatly simplify the design and the writing of the kernel
  - allow to embed only the functions of the OS that are really used
  - allow to store the program and the configuration in ROM
  - unified address space, unique execution mode
    - simplify the design and the programming of the kernel.
    - Focus on performance instead of robustness
  - Predictable behavior. Fit requirements of real-time applications.

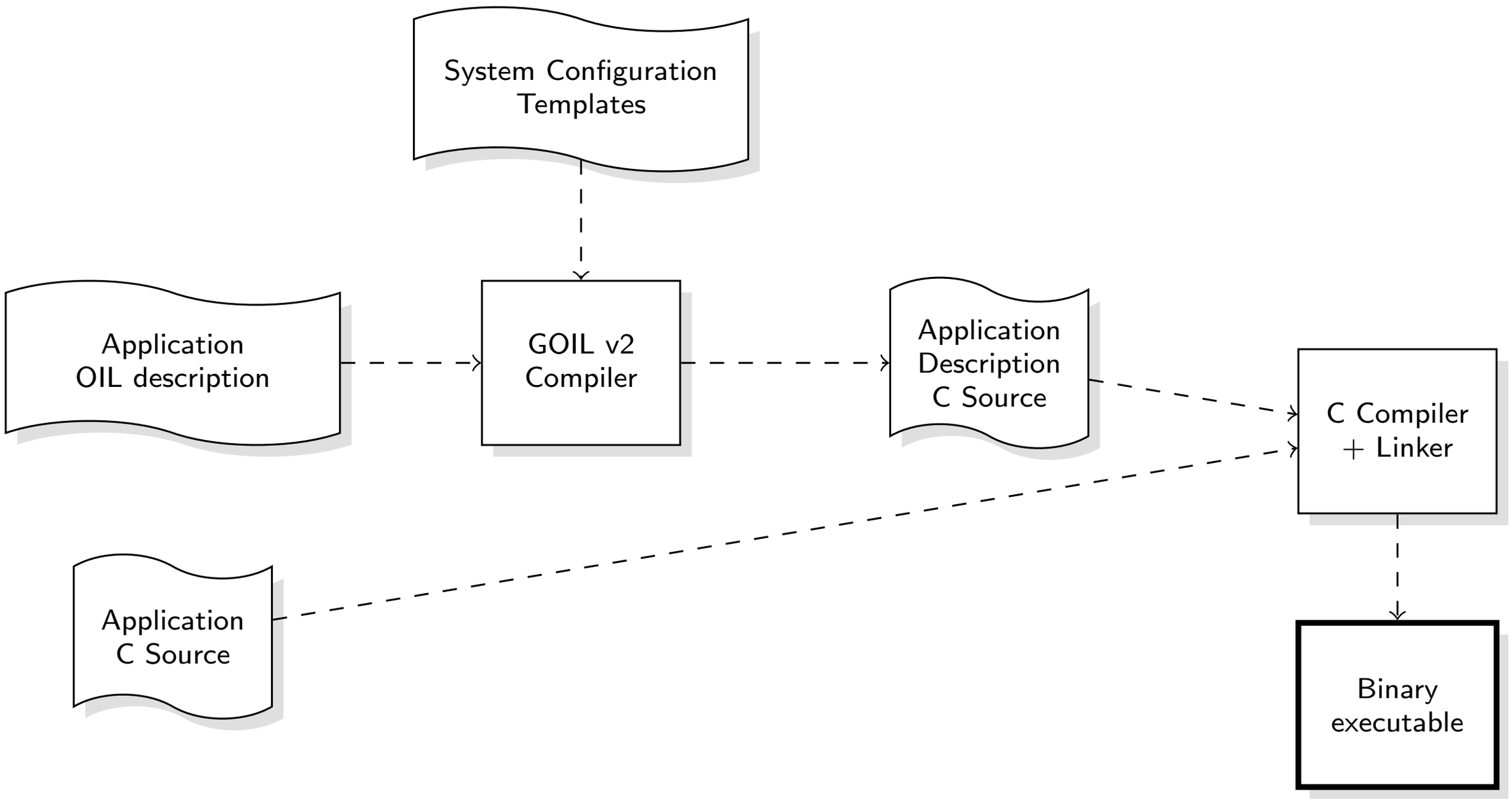
Development process of an OSEK OS + OIL Application

- Objects of an OSEK application are all defined when the application is designed
  - Objects are static. i.e: there are no creation/deletion of tasks, resources, ... dynamically during the execution of the application.
- Data structures are used to store the properties of the objects and are defined statically when the application is built.
- These structures are generally complex, hard to maintain and depends on the OSEK vendor.
- A language has been defined (and standardized) to define the attributes of the objects in a simple way:
  - OIL: OSEK Implementation Language

Development process of an OSEK OS + OIL Application

- The OIL syntax is a simple one: based on objects (tasks, resources, …) with a value for each attribute.
  Some attributes have sub-attributes.
- Starting from the description of the application (text file), data structures are automatically generated:
  - fast;
  - less error prone;
  - Independant of the OSEK vendor (the data structures are not included in the standard);
  - easy to update.

Development process of an OSEK OS + OIL Application



Development process of an OSEK OS + OIL Application

- An implementation definition part (IMPLEMENTATION):
  - This part allows to define default values for objects. For instance:
    - Task stack size defaults to 512 bytes;
    - Interrupt Service Routine stack size defaults to 256 bytes;
    - Task priority defaults to 1 ...
- This allow to define min-max for parameters to optimize data structures. For instance:
  - Task priority is within 1..10. This way the OIL compile put priority of tasks in one byte only.

Development process of an OSEK OS + OIL Application

- A description of the application: (CPU)
  - This part contains the objects of our application (tasks, ISR category 2, alarms, counters, …) that we will see soon.
  - An application mode, APPMODE, is defined in CPU (required). Application modes are used to define variants of the application (ex: diffent behaviors among different vehicles).

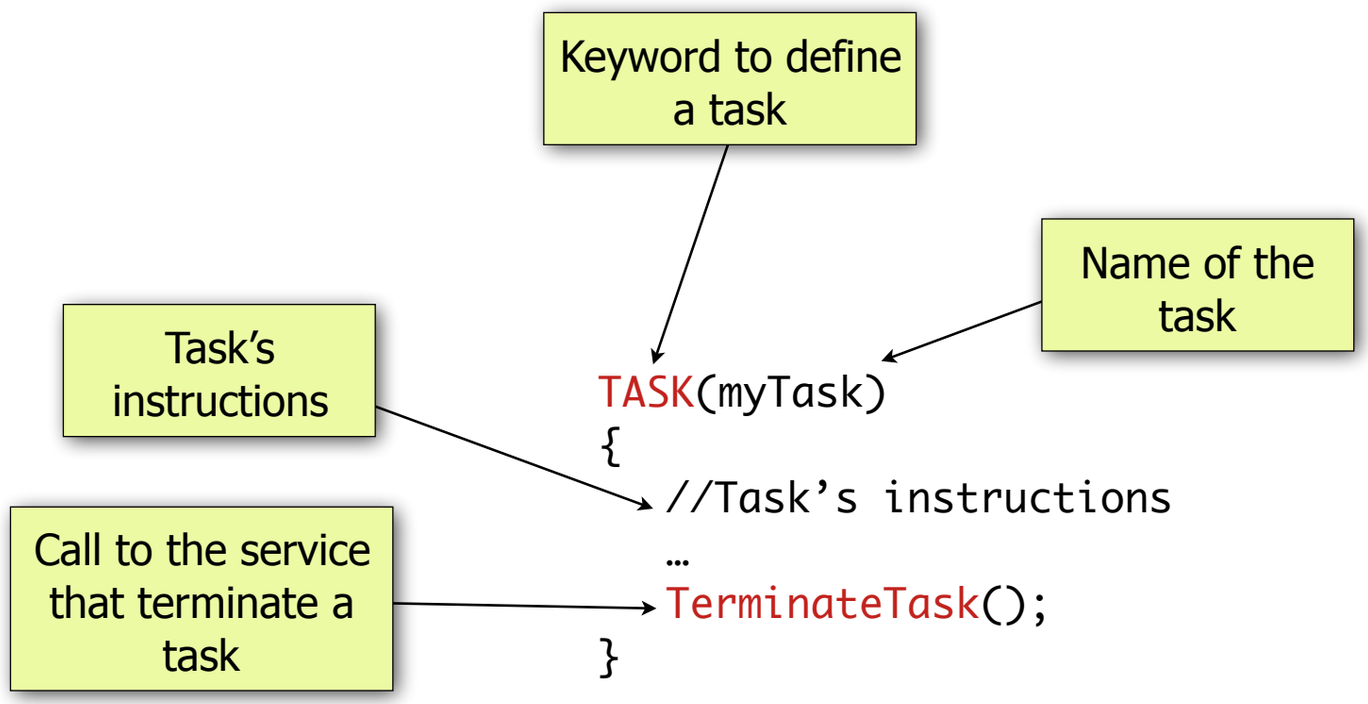
Services of OSEK

- Task services
- Synchronization services (events)
- Mutual exclusion services (resources)
- One-shot and periodical services (counters and alarms)
- Interrupt management services
- Communication services
- System services and error management

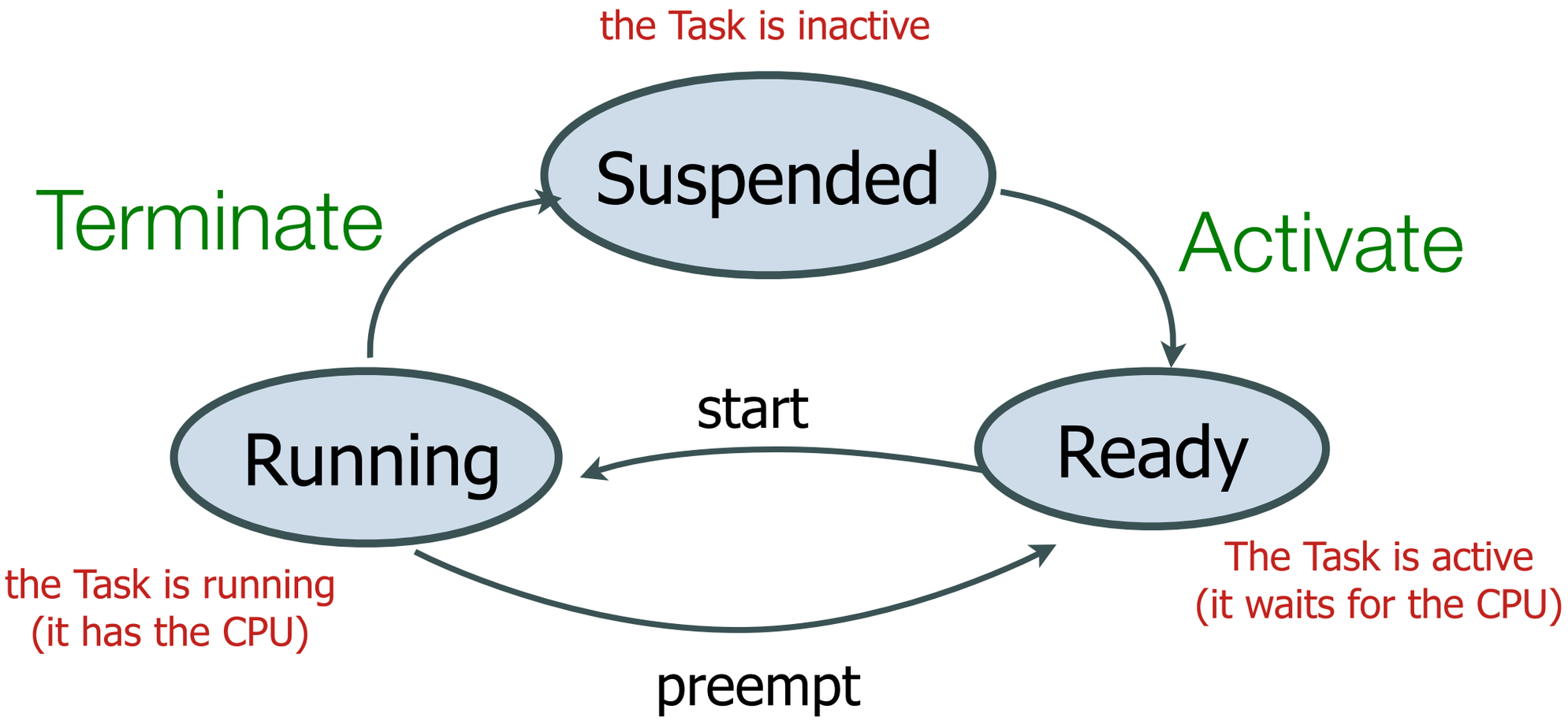
Objects are static
No creation/deletion of objects!!

Tasks in OSEK

- Tasks are « active » elements of the application
- 2 categories of tasks exist in OSEK/VDX:
  - Basic tasks
  - Extended tasks (that will be presented in next chapter)
- A basic task is a sequential C code that must terminate (no infinite loop)



Basic task states



OSEK scheduling policy

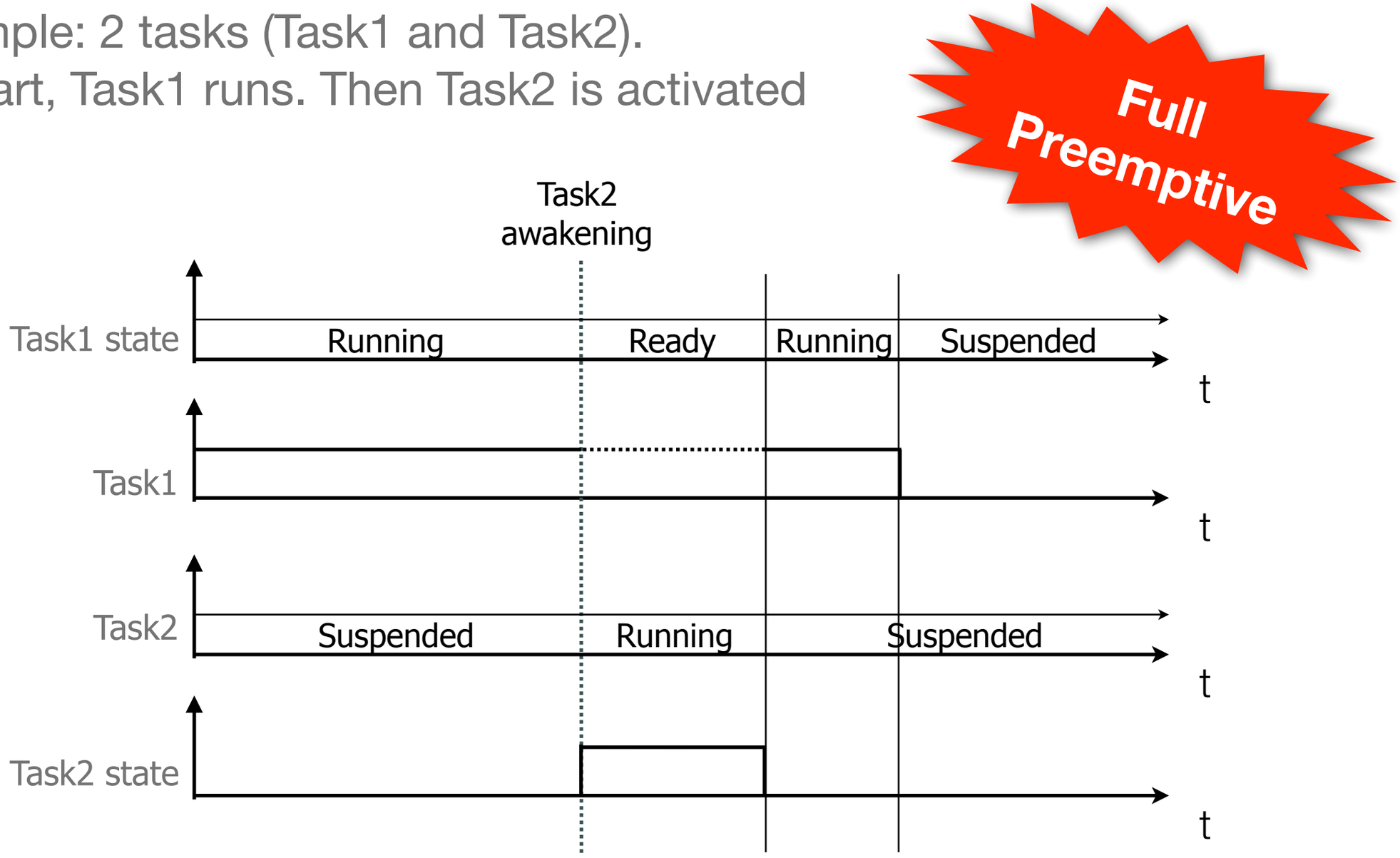
- Scheduling is done in-line
  - Scheduling is done dynamically during the execution of the application.
- Tasks have a fixed priority
  - The priority of a task is given at design stage;
  - The priority does not change (almost, taking and releasing resources may change the priority);
  - No round-robin. If more than one task have the same priority, tasks are run one after the other. ie a task may not preempt a task having the same priority
- Tasks may be preemptable or not (almost)
  - This property is defined at design stage.

Scheduling modes

- “Full preemptive”: All tasks are preemptable
  - It is the most reactive model because any task may be preempted. The highest priority Task is sure to get the CPU as soon as it is activated.
- “Full non preemptive”: All tasks are non-preemptable.
  - It is the most predictive model because a task which get the CPU will never be preempted. Scheduling is a straightforward and the OS memory footprint may be smaller.
- “Mixed”: Each task may be configured as preemptable or non-preemptable.
  - It is the most flexible model.
  - For instance, a very short task (in execution time) may be configured as non-preemptable because the context switch is longer than its execution.

Scheduling modes

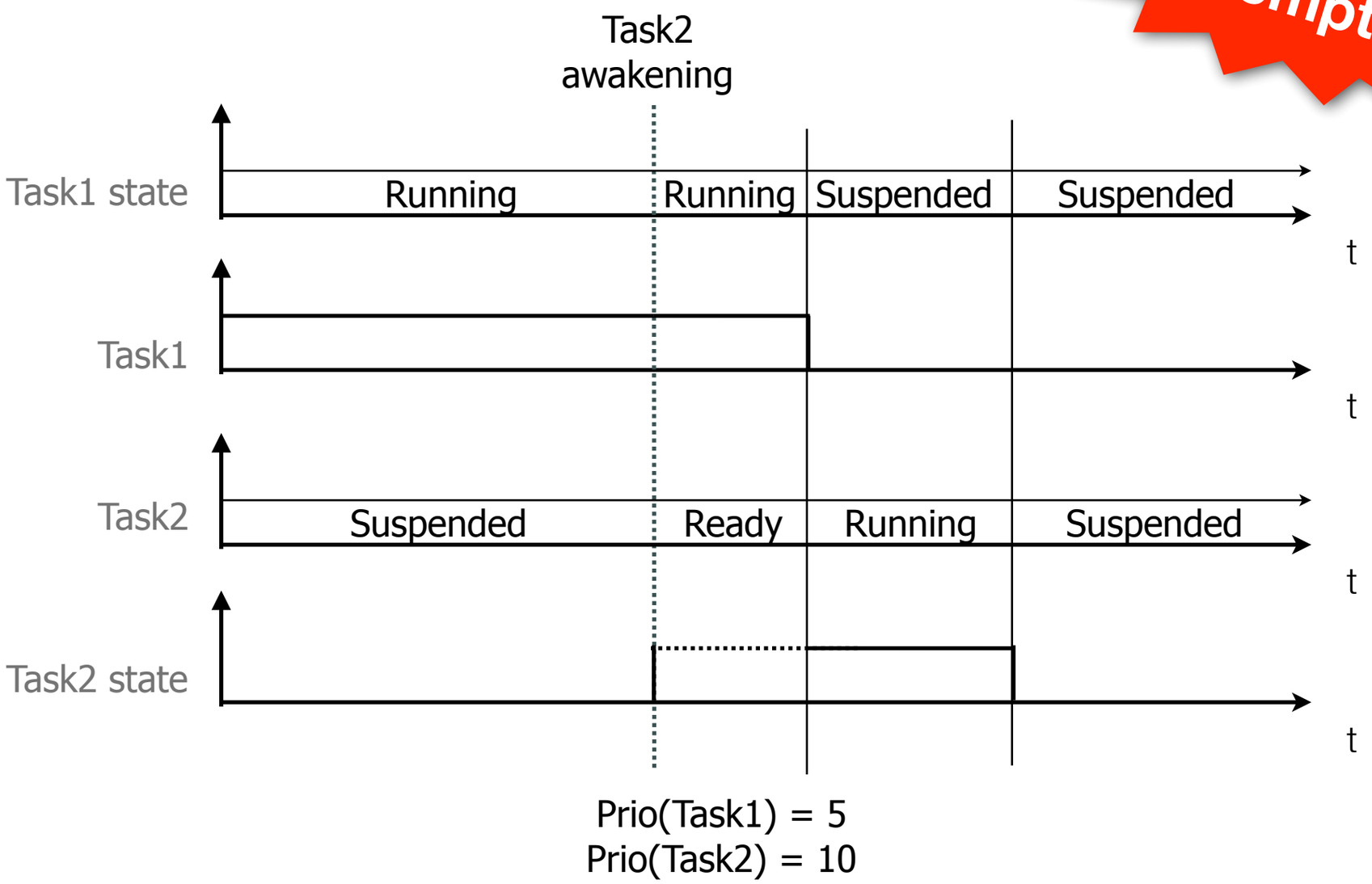
- Example: 2 tasks (Task1 and Task2).
  At start, Task1 runs. Then Task2 is activated



Scheduling modes

- Example: 2 tasks (Task1 and Task2).
  At start, Task1 runs. Then Task2 is activated

Full Non-preemptive



Prio(Task1) = 5
Prio(Task2) = 10

Tasks' services

- **TerminateTask** service
  - `StatusType TerminateTask(void);`
  - StatusType is an error code:
    - E_OK: no error
    - E_OS_RESOURCE: the task hold a resource
    - E_OS_CALLEVEL: the service is called from an interrupt
- The service stops the calling task. The task goes from running state to suspended state.
- A task may not stop another task!
  - forgetting to call TerminateTask may crash the application (and maybe the OS)!

```
TASK(myTask)
{
   //Task's instructions
   …
   TerminateTask();
}
```

Tasks' services

- **ActivateTask** service:
  - `StatusType ActivateTask(TaskType <TaskId>);`
  - The argument is the id of the task to activate.
  - StatusType is an error code:
    - E_OK: no error
    - E_OS_ID: invalid TaskId (no task with such an id)
    - E_OS_LIMIT: too many activations of the task
- This service puts the task <TaskId> in ready state
  - If the activated task has a higher priority, the calling task is put in the ready state. The new one goes in the running state.
  - A scheduling may be done (preemptable task or not, called from an interrupt).

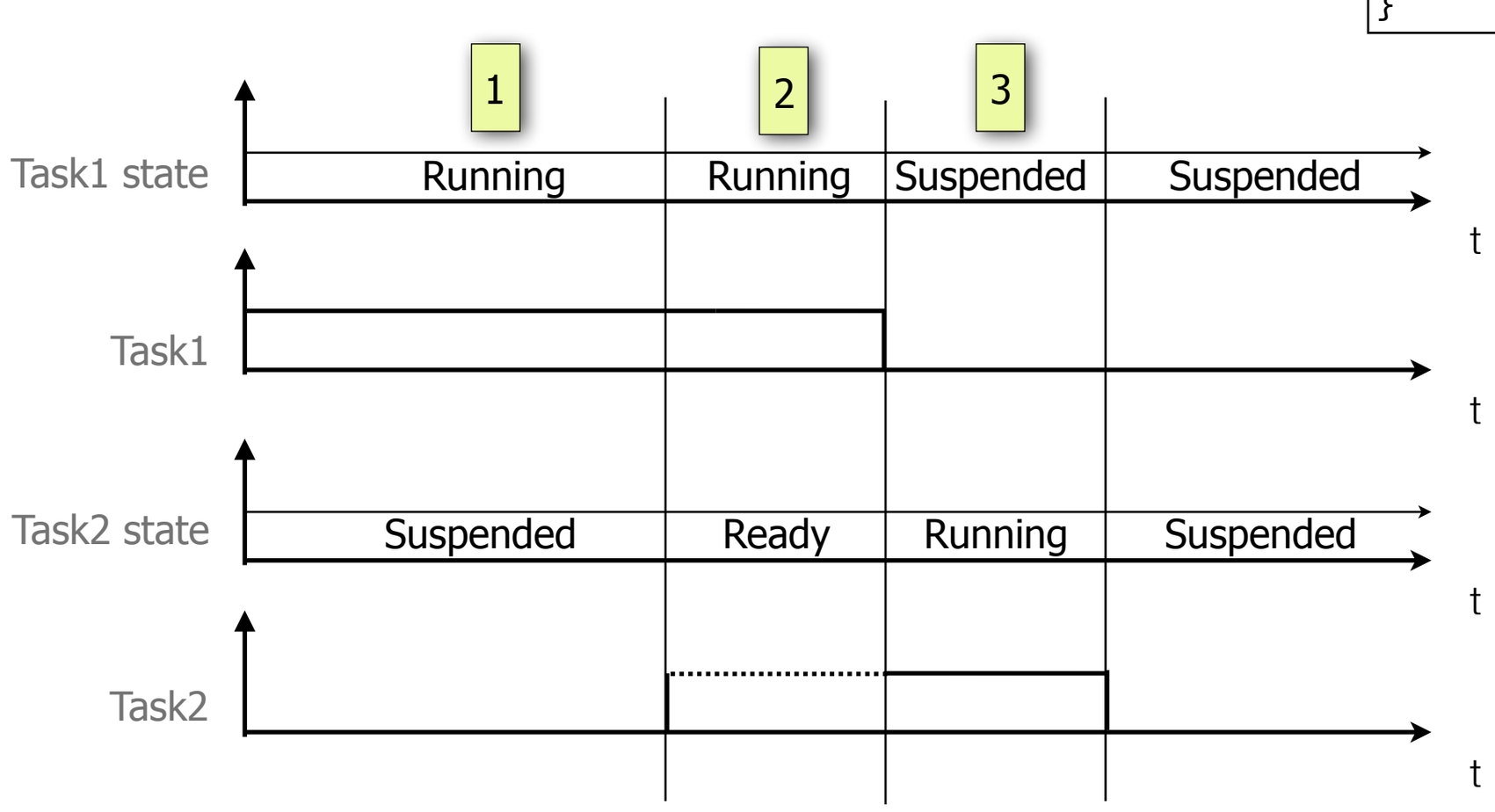
Tasks' services

- Example with 2 tasks: Task1 is active at start of the application (AUTOSTART parameter)

Prio(Task1)≥Prio(Task2)

```
TASK(Task2)
{
  3 …
    TerminateTask();
}
```

```
TASK(Task1)
{          1
    …
    ActivateTask(Task2)
  2 …
    TerminateTask();
}
```



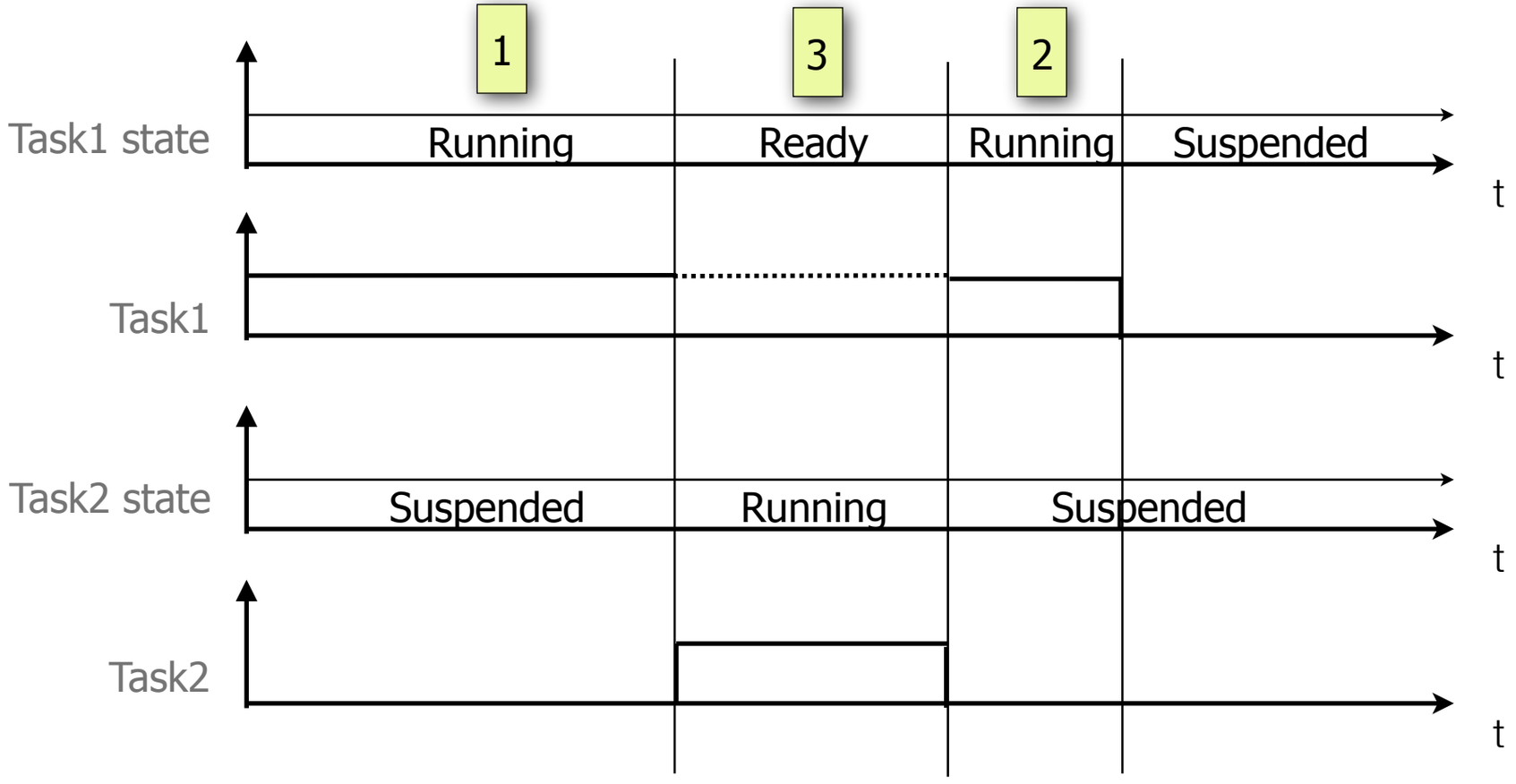
Tasks' services

- Example with 2 tasks: Task1 is active at start of the application (AUTOSTART parameter)

Prio(Task1)<Prio(Task2)

```
TASK(Task2)
{
    …  3
    TerminateTask();
}
```

```
TASK(Task1)
{
    …  1
    ActivateTask(Task2)
  2 …
    TerminateTask();
}
```

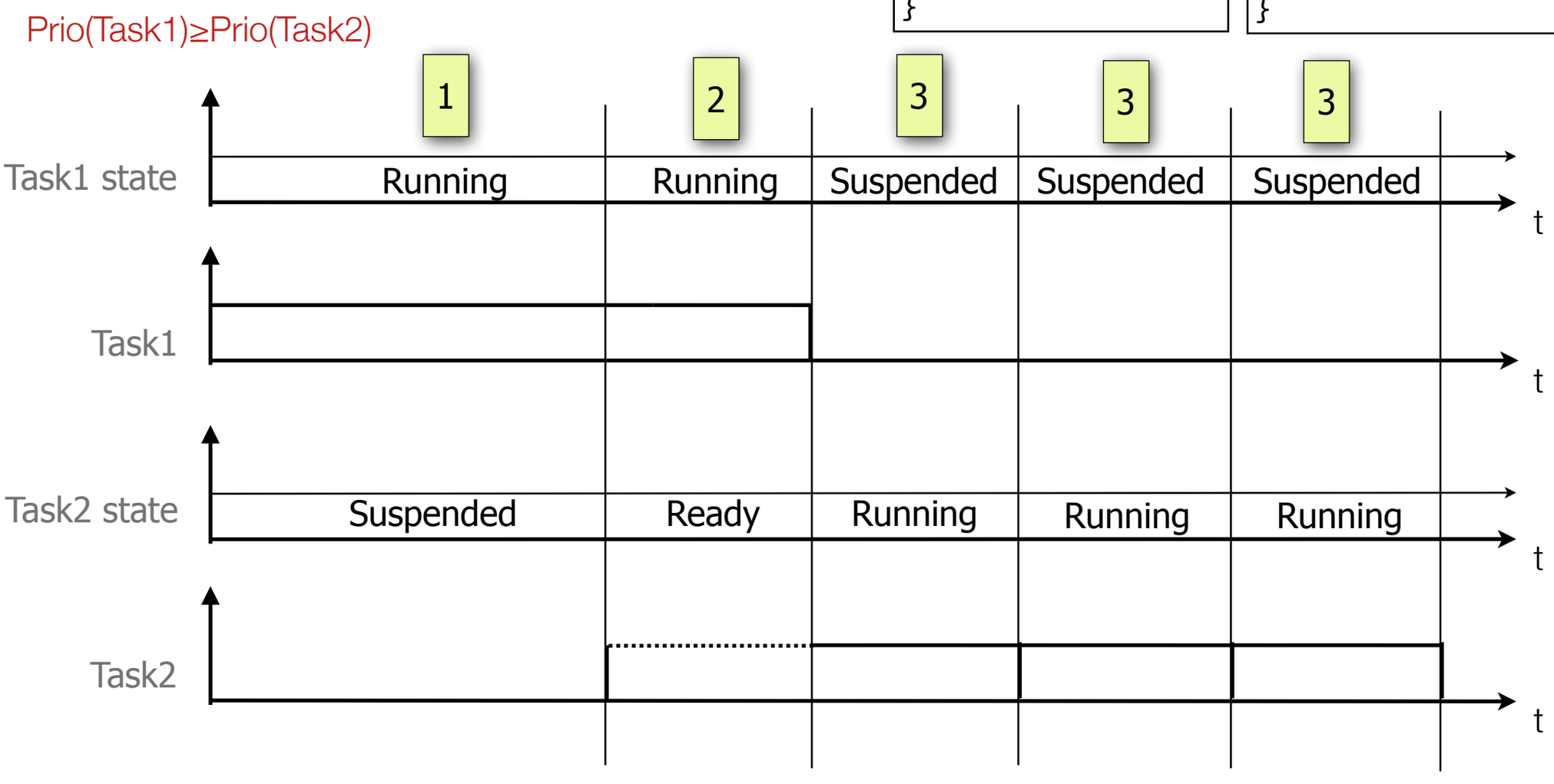


Tasks' services

- When multiple activations occur, OSEK allows to memorize them up to a value defined at design time.

```
TASK(Task2)
{
    …
    TerminateTask();
}
```

```
TASK(Task1)
{
    …
    ActivateTask(Task2)
    ActivateTask(Task2)
    ActivateTask(Task2)
    …
    TerminateTask();
}
```



Tasks' services

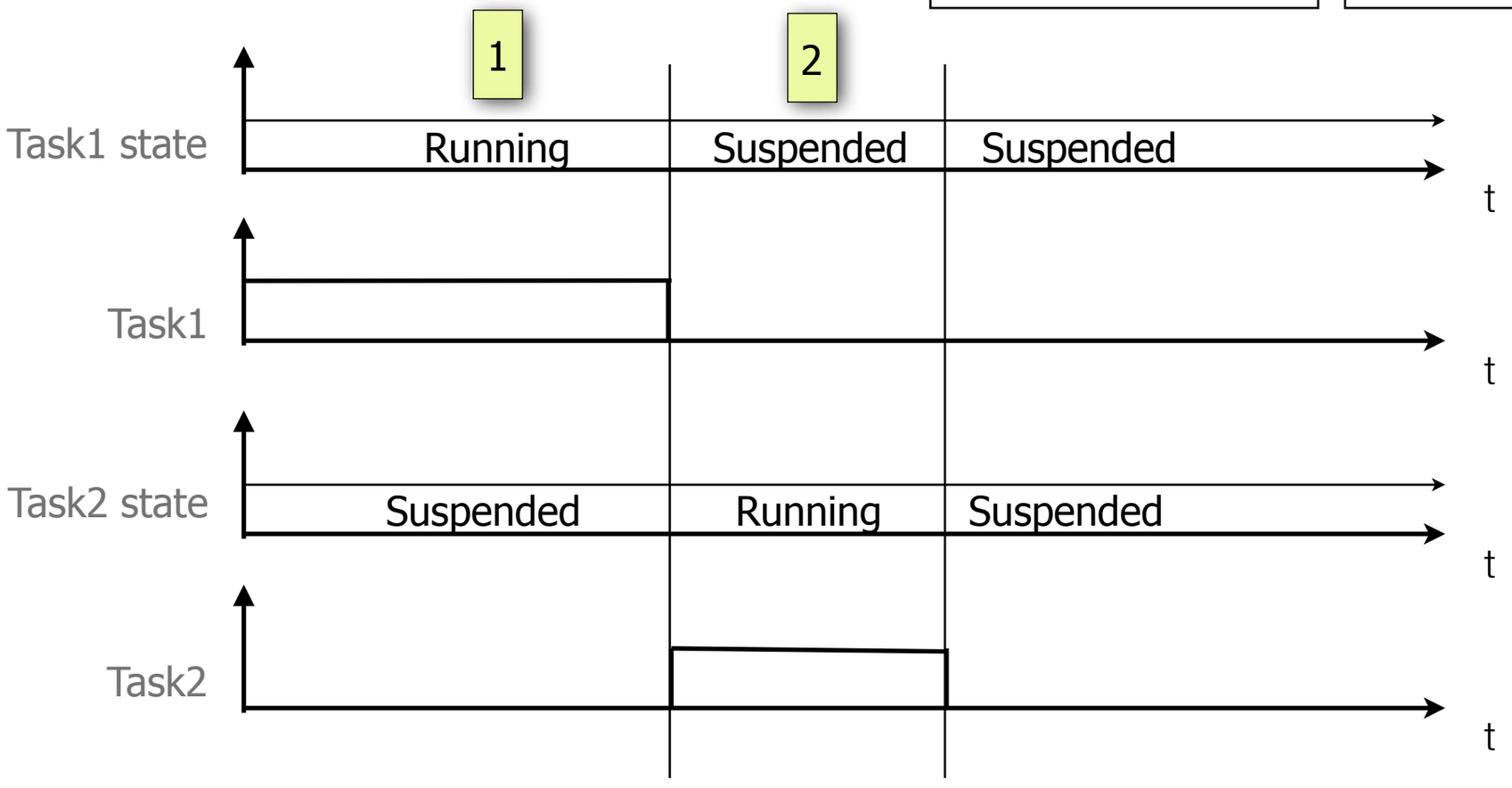
- **ChainTask** service:
  - StatusType ChainTask(TaskType <TaskId>);
  - The argument is the id of the task to activate;
  - StatusType is an error code:
    - E_OK: No error
    - E_OS_ID: invalid TaskId (no task with such an id)
    - E_OS_LIMIT: too many activations of the task
- This service puts task <TaskId> in ready state, and the calling task in the suspended state.
  - This service replaces TerminateTask for the calling task.

Tasks' services

- Example with 2 tasks: Task1 is active at start of the application (AUTOSTART parameter)

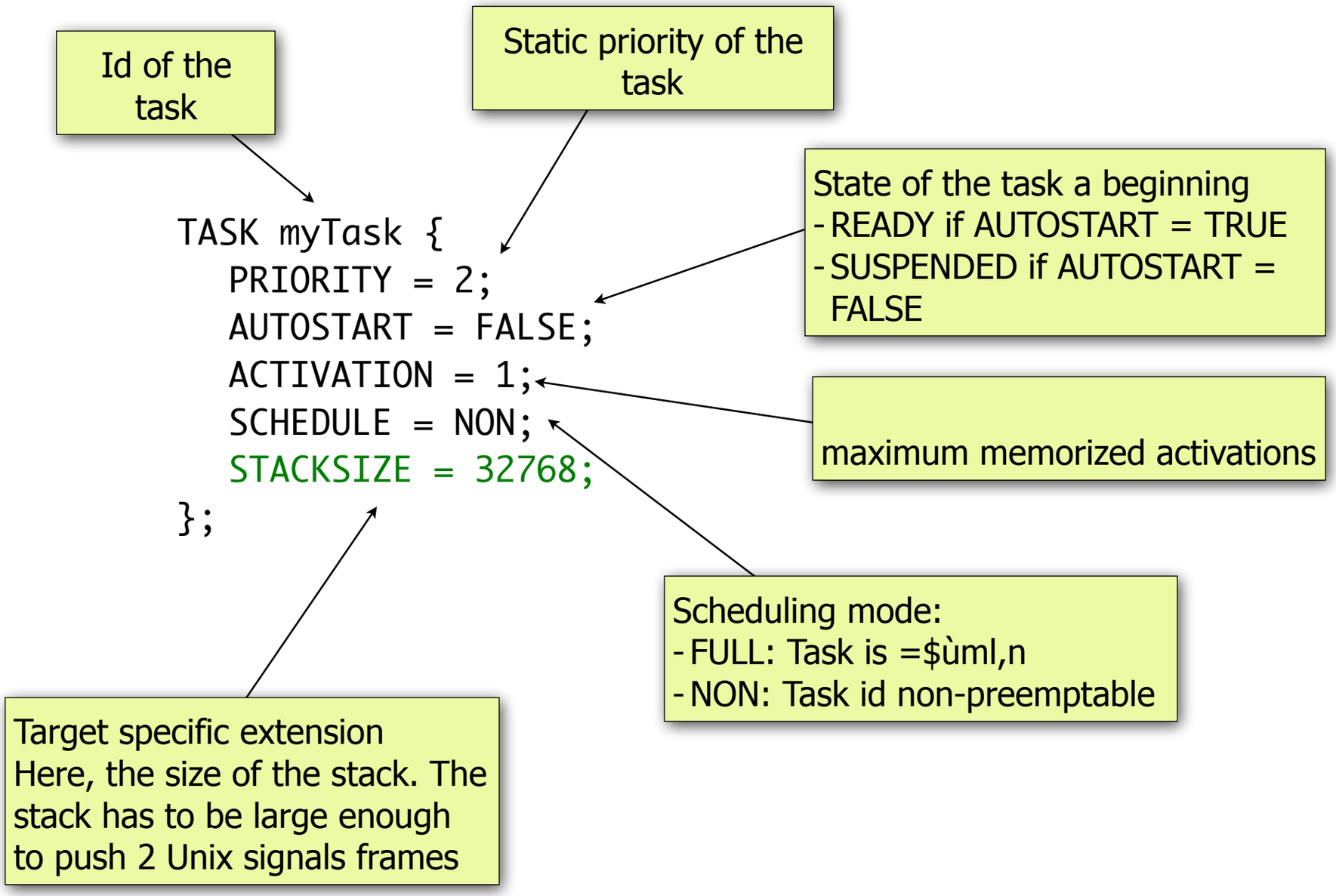
```
TASK(Task2)
{
  …
  TerminateTask();
}
```

```
TASK(Task1)
{
  …
  ChainTask(Task2);
}
```



Tasks' services

- OIL description of a task



Tasks' services

- OIL description of a task

```
TASK myTask {
   PRIORITY = 2;
   AUTOSTART = TRUE {
        APPMODE = std;
   };
   ACTIVATION = 1;
   SCHEDULE = NON;
   STACKSIZE = 32768;
};
```

If the task is put in READY state at start, a sub-attribute corresponding to the application mode has to be defined

Hook Routines

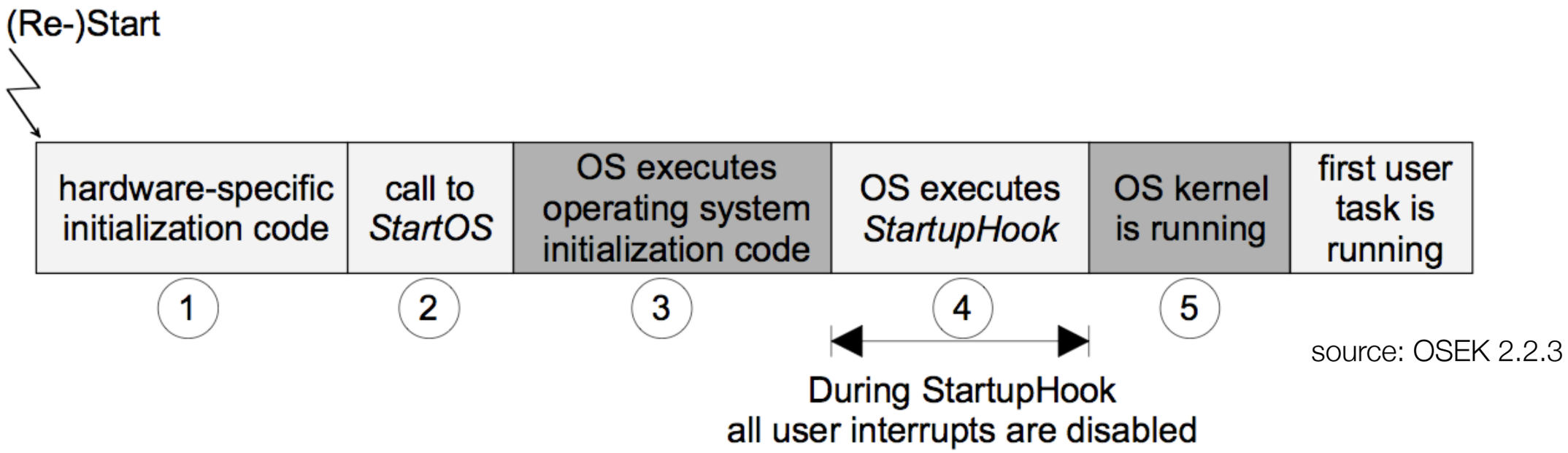
- Features
  - OSEK proposes dedicated routines (or functions) to allow the user to « hook » an action at important stages in system calls.
  - “hook routines” are/have:
    - called by the operating system;
    - a priority greater than all tasks;
    - not interrupted by ISR2 (presented after);
    - a standardized interface;
    - able to call a subset of the operating system services.

Hook Routines

- Usage
  - System startup
    - allow initializations before the schedule start but after the operating system is started.
  - System Shutdown
    - allow to do something when the system is shutdown (usually a very bad thing occured !).
- Tracing of system behavior
  - allow to get the task scheduling;
  - we will use it in labs.
- Error management

Hook Routines

- StartupHook
  - This routine is called after the startup of the OS but before the startup of the scheduler



source: OSEK 2.2.3

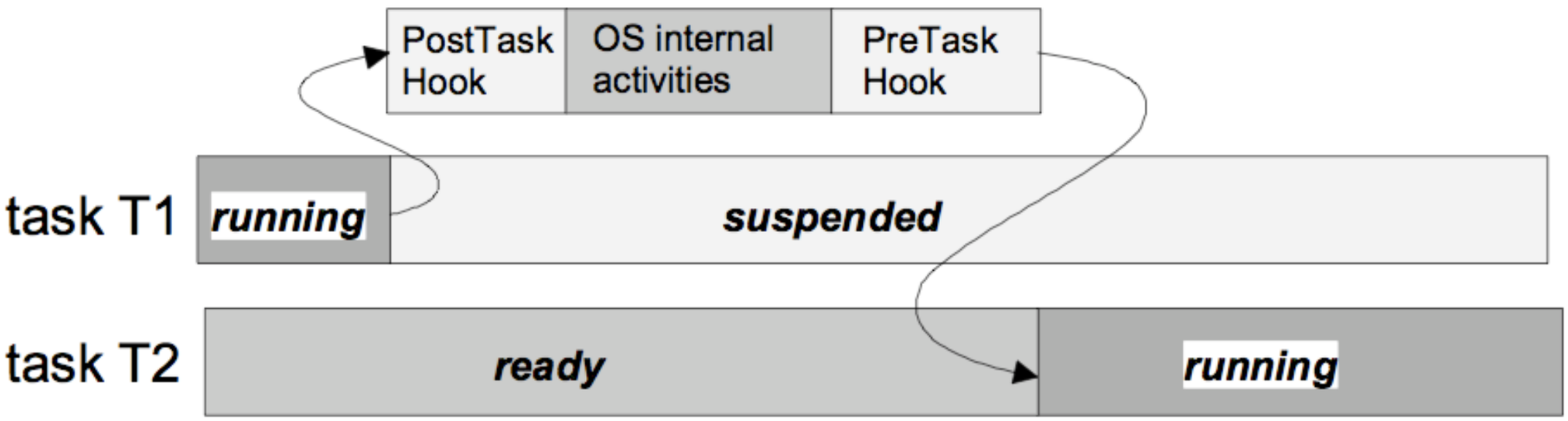
- ShutdownHook
  - This routine is called when ShutdownOS() is called and should be used for fatal error handling.

Hook Routines

- ErrorHook
  - This routine est called when a system call does not return E_OK, that is if an error occurs during a system call.
  - Exception: It is not called if the error occured in a system call called by the ErrorHook (to prevent recursive calls).

Hook Routines

- PreTaskHook and PostTaskHook
  - PreTaskHook is called just before a task goes from READY state to RUNNING state;
  - PostTaskHook is called just before a task goes from RUNNING state to READY or SUSPENDED state;
- It is the only way to detect a task preemption.



source: OSEK 2.2.3

Hook Routines

- OIL declaration
  - The hooks which are used must be declared in the OS object in the implementation part of the OIL file

```
OS config {
     STATUS = EXTENDED;
     ERRORHOOK = TRUE;
     PRETASKHOOK = TRUE;
} ;
```

STATUS:
It may be EXTENDED (additional checking and errors), or STANDARD.

- In the C source:

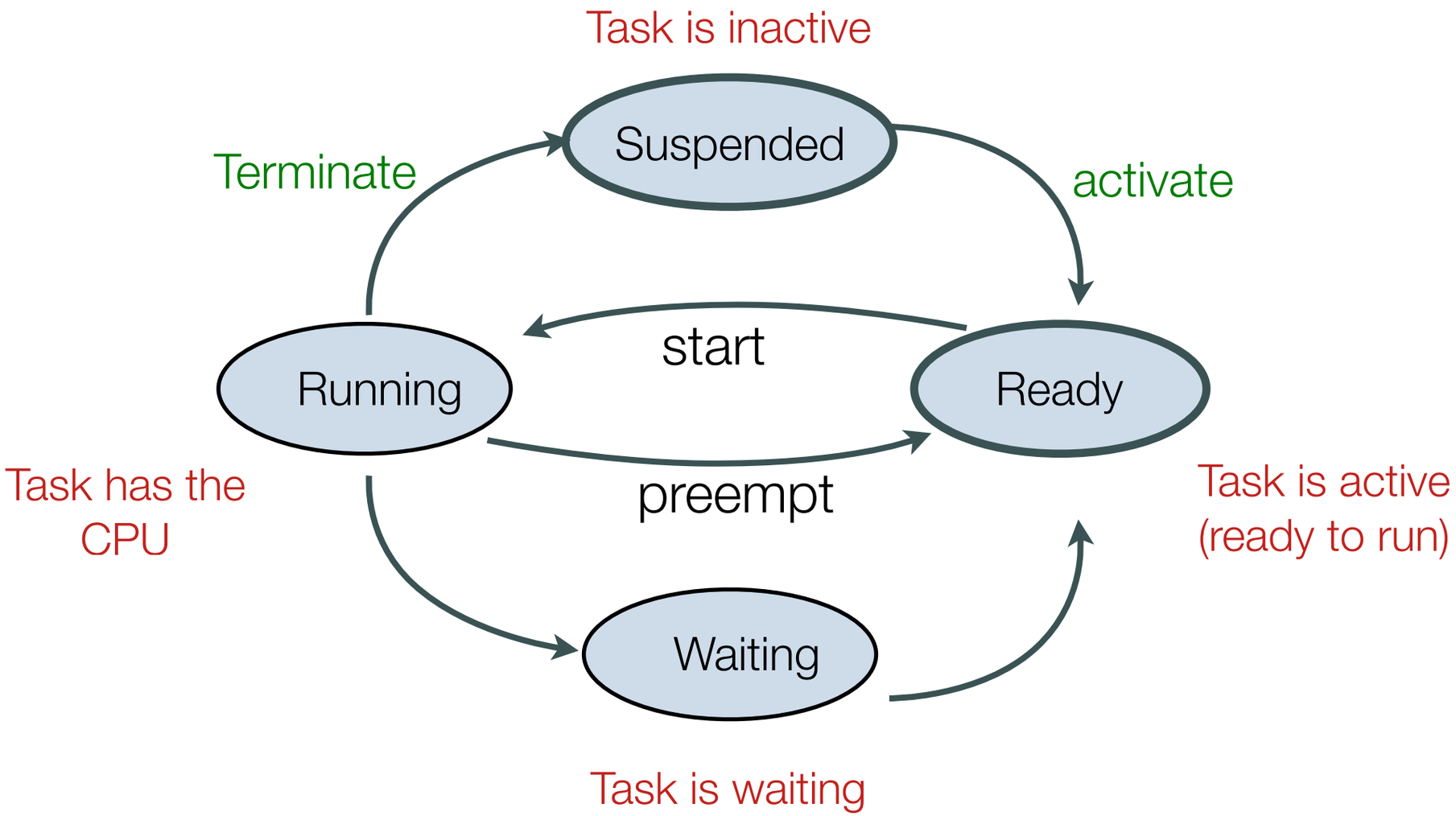
```
void ErrorHook(StatusType error)
{
}

void PreTaskHook(void)
{
    TaskType id;
    GetTaskID(&id);
    printf("Before %d\n",id);
}
```

Tasks' synchronization

- Synchronization of tasks: A task should be able to wait an external event (a verified condition).
- To implement this feature, OSEK uses events.
- Task's model is modified to add a new state: waiting.
  - The task that are able to wait for an event are called Extended tasks
  - The drawback is a more complex scheduler (a little bit slower and a little bit bigger in code size)

States of an extended task



The concept of event

- An event is like a flag that is raised to signal something just happened.
- An event is private: It is a property of an Extended Task. Only the owning task may wait for the event.
- It is a N producers / 1 consumer model
  - Any task (extended or basic) or Interrupt Service Routines Category 2 (will be explained later) may invoke the service which set an event.
  - One task (an only one) may get the event (ie invoke the service which wait for an event.
- The maximum number of event per task relies on the implementation (16 in Trampoline)

Event mask

- An Extended Task may wait for many events simultaneously
  - The first to come wakes up the task.
- To implement this feature, an event corresponds to a binary mask: 0x01, 0x02, 0x04, ...
- An event vector is associated to 1 or more bytes. Each event is represented by one bit in this vector
- So each task owns:
  - a vector of the events set
  - a vector of the event it waits for

Event mask

- Operation:
  - Event X signaling : ev_set |= mask_X;
  - Is event X arrived ? : ev_set & mask_X;
  - Wait for event X : ev_wait |= mask_X;
  - Clear event X : ev_set &= ~mask_X;
- In practice, these operations are done in a simpler way by using the following services.

Events' services

- **SetEvent**
  - StatusType SetEvent(TaskType <TaskID>, EventMaskType <Mask>);
  - Events of task <TaskID> are set according to the <Mask> passed as 2nd argument.
  - StatusType is an error code:
    - E_OK: no error;
    - E_OS_ID: invalid TaskId;
    - E_OS_ACCESS: TaskID is not an extended task (not able to manage events);
    - E_OS_STATE: Events cannot be set because the target task is in the SUSPENDED state.
- This service is not blocking and may be called from a task or an ISR2

Events’ services

- **ClearEvent**
  - StatusType ClearEvent(EventMaskType <Mask>);
  - The events selected by <Mask> are cleared.
  - May be called by the owning task (only);
  - StatusType is an error code:
    - E_OK: no error;
    - E_OS_ACCESS: The calling task is not an extended one (so it does not manage events);
    - E_OS_CALLEVEL: The caller is not a task.
- non-blocking service.

Events' services

- **GetEvent**
  - StatusType GetEvent(TaskType <TaskId>, EventMaskRefType event);
  - The event mask of the task <TaskId> is copied to the variable event (A pointer to an EventMaskType is passed [illegible]);
  - StatusType is an error code:
    - E_OK: no error;
    - E_OS_ID: invalid TaskID;
    - E_OS_ACCESS: TaskID is nor an extended task;
    - E_OS_STATE: Events may not be copied because the target task is in the SUSPENDED state.
  - Non-blocking service, might be called from a task or an ISR2.

Events' services

- **WaitEvent**
  - StatusType WaitEvent(EventMaskType <EventID>);
  - Put the calling task in the WAITING state until one of the events is set.
  - May be called by the event owning (extended) task only;
  - StatusType is an error code:
    - E_OK: no error;
    - E_OS_ACCESS: The calling task is not an extended one;
    - E_OS_RESOURCE: The task has not released all the resources (will be explained later);
    - E_OS_CALLEVEL: The caller is not a task
  - Blocking service.

Events in OIL

- OIL description of an EVENT

```
EVENT ev1 {
  MASK = AUTO;
};

EVENT ev2 {
  MASK = 0x4;
};
```

Definition of the mask. It is:
- AUTO, the actual value is computed by the OIL compiler.
- A litteral value which is the binary mask.

List of the event the task uses. The task is the owner of these events

```
TASK myTask {
   PRIORITY = 2;
   AUTOSTART = FALSE;
   ACTIVATION = 1;
   SCHEDULE = NON;
   STACKSIZE = 32768;
   EVENT = ev1;
   EVENT = ev2;
};
```

If an event is used in more than one task, only the name is shared: **An event is private**.

- myTask is automatically an **Extended task** because it uses at least one event.

Example

Set ev1 which is owned by myTask

```
TASK(Task1)
{
   …
   SetEvent(myTask, ev1);
   …
   TerminateTask();
}
```

Wait for 2 events simultaneously
The task will be waked up when at least one of the 2 events will be set

```
TASK(myTask)
{
   EventMaskType event_got;
   …
   WaitEvent(ev1 | ev2);
   GetEvent(myTask, &event_got);
   ClearEvent(event_got);
   if (event_got & ev1) {
      //manage ev1
   }
   if (event_got & ev2) {
      //manage ev2
   }
   …
   TerminateTask();
}
```

Useful to know what event has been set

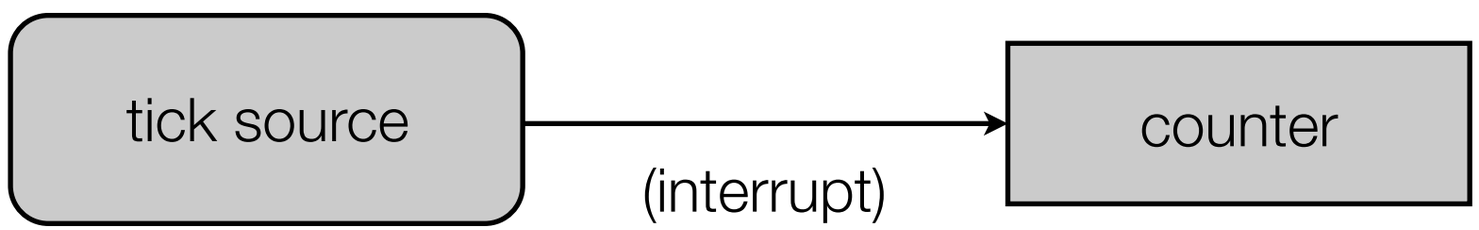
```
TASK(Task3)
{
   …
   SetEvent(myTask, ev2);
   …
   TerminateTask();
}
```

Counters and Alarms

- Goal: perform an “action” after a number of “ticks” from an hardware device:
  - Typical case: periodic activation of a task with a hardware timer.
- The “action” may be:
  - signalization of an event;
  - activation of a task;
  - function call (a callback since it is a user function). The function is executed on the context of the running task.
- The hardware device may be:
  - a timer;
  - any periodic interrupt source (for instance an interrupt triggered by the low position of a piston of a motor. The frequency is not a constant in this case.

The counters

- The counter is an abstraction of the hardware “tick” source (timer, interrupt source, ...)
  - The “tick” source is heavily dependent of the target platform;
  - The counter is a standard component;
  - Moreover, the counter has a divider.



The counters

- A counter defines 3 values:
  - The maximum value of the counter (MaxAllowedValue);
  - A division factor (TicksPerBase): for instance with a TicksPerBase equal to 5, 5 ticks are needed to have the counter increased by 1;
  - The minimum number of cycles before the alarm is triggered (explained after);
- The counter restart to 0 after reaching MaxAllowedValue.

OIL description of a counter

```
COUNTER generalCounter {
   TICKSPERBASE = 10;
   MAXALLOWEDVALUE = 65535;
   MINCYCLE = 128;
};
```

number of "ticks" (from the interrupt source) needed to have the counter increased by one.

Maximum value of the counter. This value is used by the OIL compiler to generate the size of the variable used to store the value of the counter.

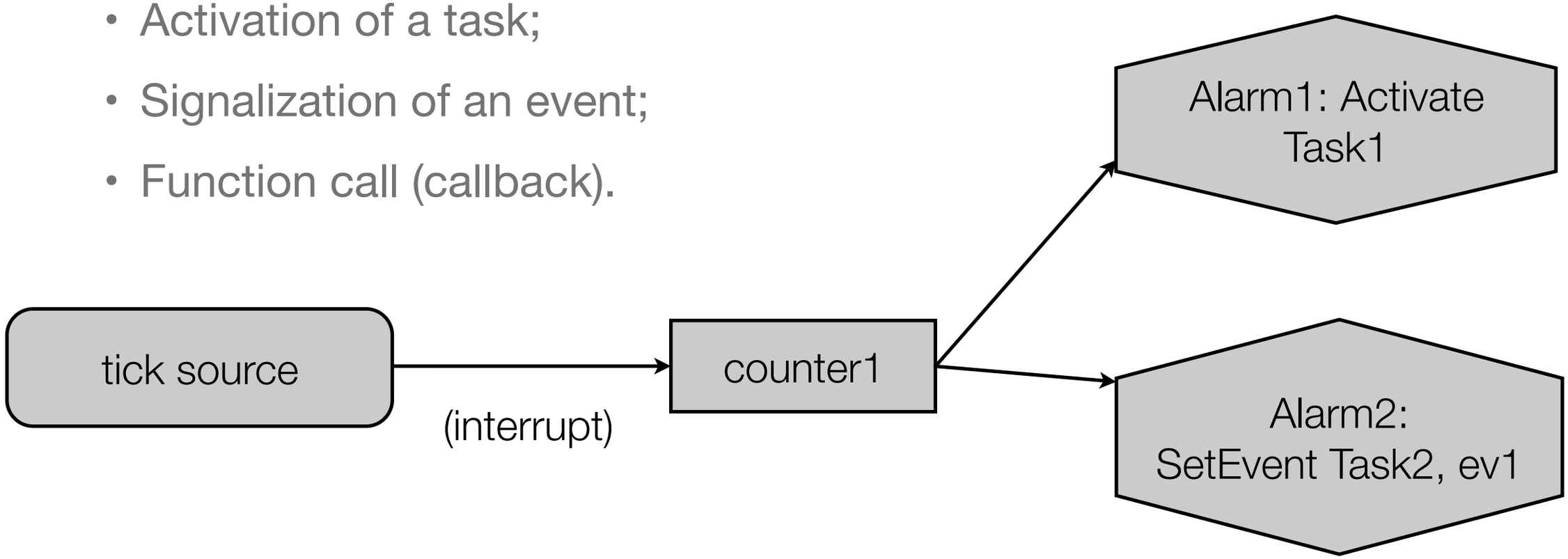
minimum interval between 2 triggering of the alarm.

The counters

- At least one counter is available: SystemCounter
- No system call to modify the counters.
  - Their behavior are masked by the alarms.
- A hardware interrupt must be associated to a counter
  - This part is not explained in the standard and depends on the target platform and the OSEK/VDX vendor.
- Features of the Trampoline UNIX port
  - For Trampoline running on UNIX, a separate tool acts as a programmable interrupt source.
  - SystemCounter has a MaxAllowedValue equal to 32767, a TicksPerBase and a MinCycle equal to 1. There is one tick every 10ms.

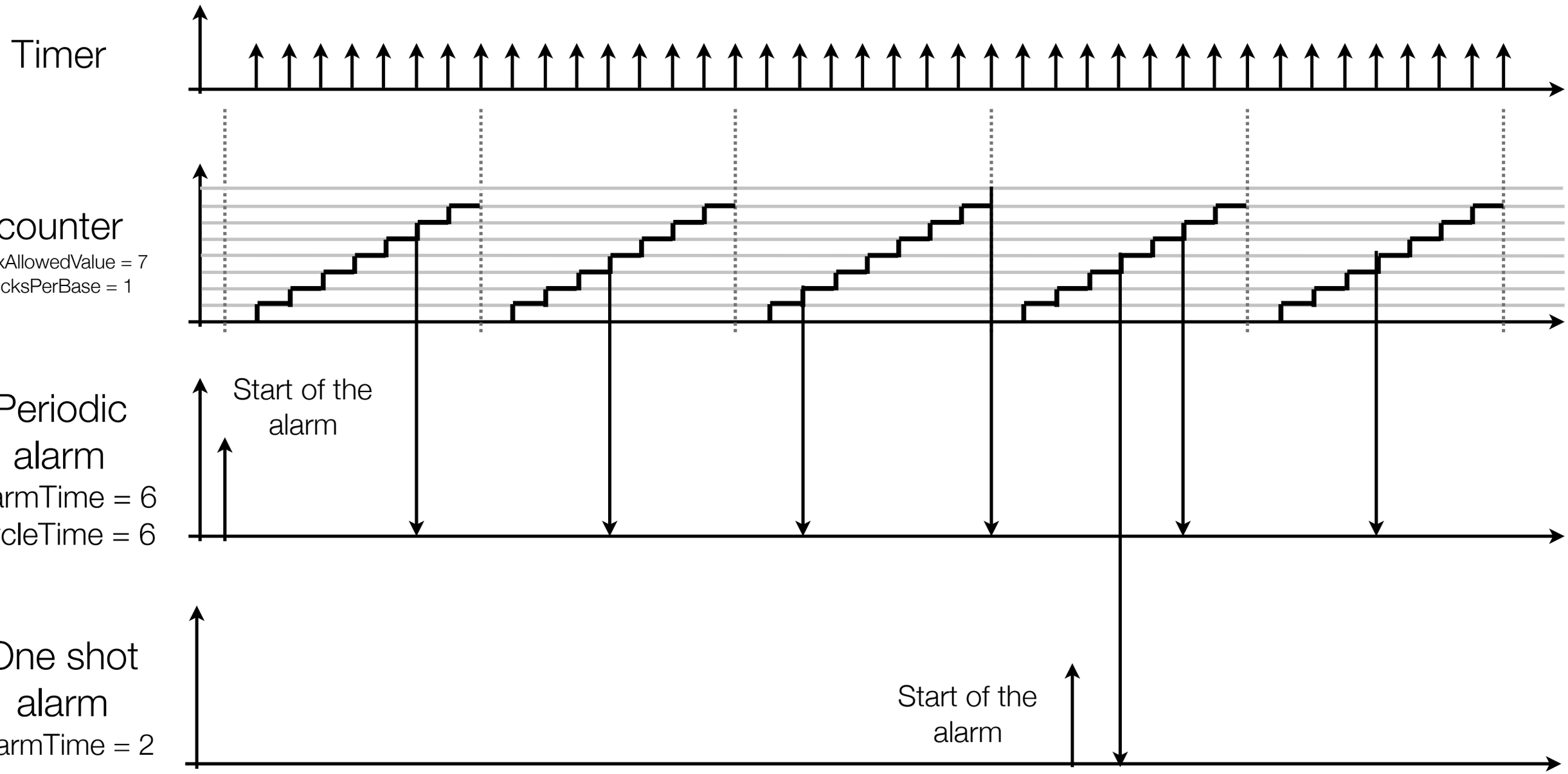
The Alarms

- An alarm is connected to a counter an performs an action.
  - An alarm is associated to **1** counter
  - A counter may be used for several alarms
- When the counter reach a value of the alarm (CycleTime, AlarmTime), the alarm expires and an action is performed:
  - Activation of a task;
  - Signalization of an event;
  - Function call (callback).



Counters/Alarms

- Example



Counters/Alarms

- Counters do not have system calls.
  - They are set up statically and behave that way while the system is up and running.
  - The hardware tick source may be stopped.
- Alarms may be started and stopped dynamically.

Alarms' services

- **SetAbsAlarm**
  - `StatusType SetAbsAlarm (`
    `AlarmType <AlarmID>,`
    `TickType <start>,`
    `TickType <cycle>)`
    - AlarmID is the id of the alarm to start.
    - start is the absolute date at which the alarm expire
    - cycle is the relative date (counted from the start date) at which the alarm expire again. If 0, it is a one shot alarm.
  - StatusType is an error code:
    - E_OK: no error;
    - E_OS_STATE: The alarm is already started;
    - E_OS_ID: The AlarmID is invalid.
    - E_OS_VALUE: start is < 0 or > MaxAllowedValue *and/or* cycle is < MinCycle or > MaxAllowedValue.

Alarms' services

- **SetRelAlarm**
  - `StatusType SetRelAlarm ( AlarmType <AlarmID>, TickType <increment>, TickType <cycle>)`
    - AlarmID is the id of the alarm to start.
    - increment is the relative date at which the alarm expire
    - cycle is the relative date (counted from the start date) at which the alarm expire again. If 0, it is a one shot alarm.
  - StatusType is an error code:
    - E_OK: no error;
    - E_OS_STATE: The alarm is already started;
    - E_OS_ID: The AlarmID is invalid.
    - E_OS_VALUE: increment is < MinCycle or > MaxAllowedValue *and/or* cycle is < MinCycle or > MaxAllowedValue.

Alarms' services

- **CancelAlarm**
  - Stop an alarm.
  - `StatusType CancelAlarm (AlarmType <AlarmID>)`
    - AlarmID is the id of the alarm to stop.
- StatusType is an error code:
  - E_OK: no error;
  - E_OS_NOFUNC: The alarm is not started;
  - E_OS_ID: The AlarmID is invalid.

Alarms' services

- **GetAlarm**
  - Get the remaining ticks before the alarm expires.
  - `StatusType GetAlarm (`
    `AlarmType <AlarmID>,`
    `TickRefType <tick>)`
    - AlarmID is the id of the alarm to get.
    - tick is a pointer to a TickType where GetAlarm store the remaining ticks before the alarm expire.
- StatusType is an error code:
  - E_OK: no error;
  - E_OS_NOFUNC: The alarm is not started;
  - E_OS_ID: The AlarmID is invalid.

Alarms' services (almost)

- **GetAlarmBase**
  - Get the parameters of the underlying counter.
  - `StatusType GetAlarmBase (`
    `AlarmType <AlarmID>,`
    `AlarmBaseRefType <info>)`
    - AlarmID is the id of the alarm.
    - info is a pointer to an AlarmBaseType where GetAlarmBase store the parameters of the underlying counter.
  - StatusType is an error code:
    - E_OK: no error;
    - E_OS_ID: The AlarmID is invalid.

OIL description of an alarm

```
ALARM alarm_1 {
    COUNTER = generalCounter;
    ACTION = ACTIVATETASK {
        TASK = task_1;
    };
    AUTOSTART = TRUE {
        ALARMTIME = 10;
        CYCLETIME = 5000;
        APPMODE = std;
    };
};
```

action to be performed by the alarm. Here, activation of task task_1.

The alarm is triggered à 10

It is a periodic alarm which is triggered every 5000 counter tick

Interrupts

- 2 kinds of interrupts (Interrupt Service Routine or ISR) are defined in OSEK, according to the richness needed for the ISR.
- Anyway, the execution time of an ISR must be short because it delays the execution of tasks.
- Level 1 interrupts
  - are very fast;
  - stick to the hardware capabilities of the micro-controller;
  - are not allowed to do a system call;
  - usually difficult to port to another micro-controller;
- Level 2 interrupts
  - are not as fast as level 1 interrupts
  - are allowed to do some system calls (activate a task, get a resource, ...)

ISR1

- Are not allowed to do system calls;
- In fact, ISR1 are ignored by the operating system and defined as classical interrupts:
  - Init interrupt registers of the hardware peripheral;
  - Init the related interrupt mask
  - Do not touch the other interrupt masks (which are managed by the operating system).

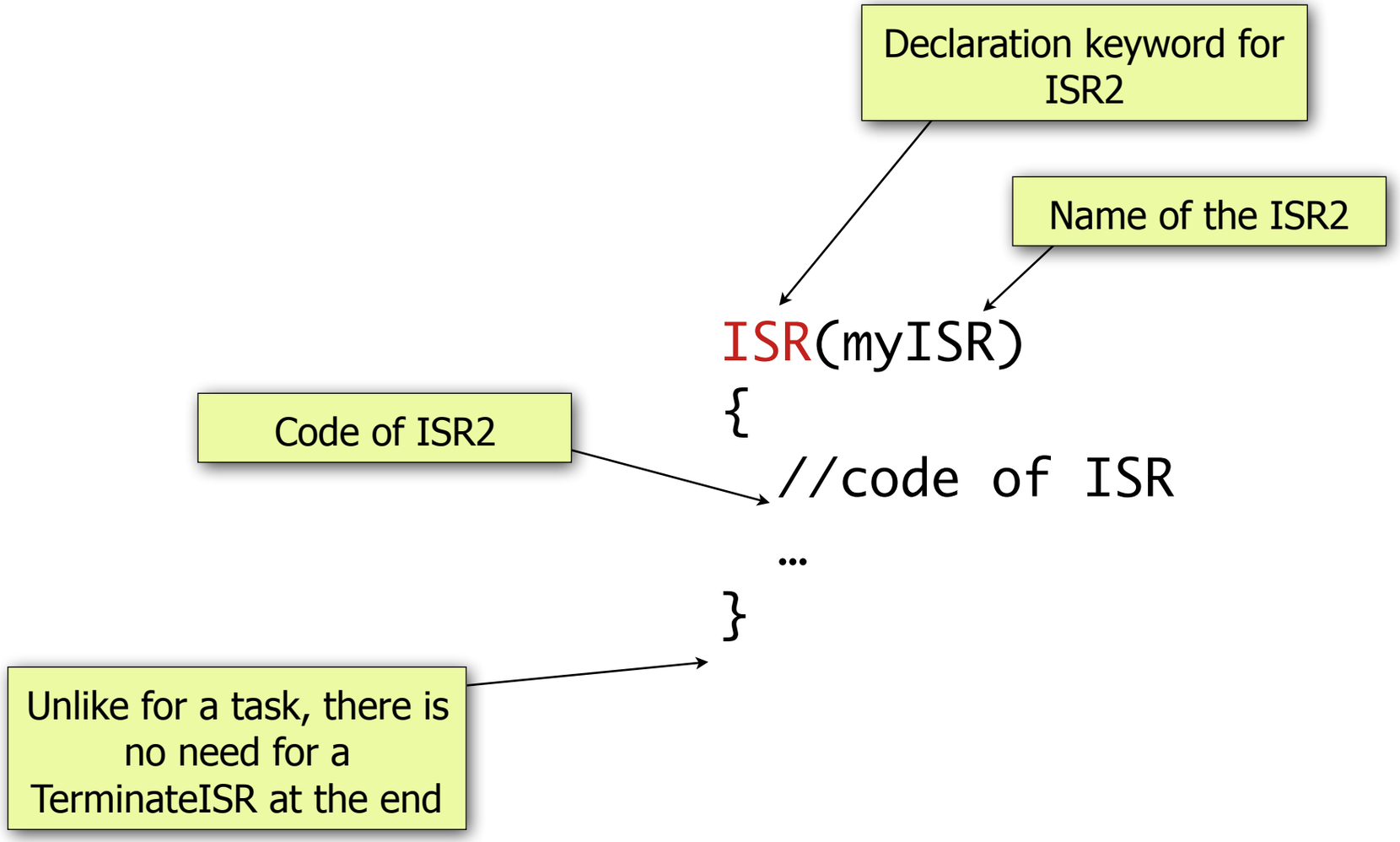
ISR2

- May (must?) do system calls (activate a task, get a resource, …)
- Roughly the same behavior as a task
  - they have a priority (greater than the higher priority of tasks). ISR2 priority is a logical one and may not be related to the hardware priority level.
  - they have a context (registers, stack, …)
- In addition an ISR2
  - is associated to a hardware interrupt (triggered by an event)

ISR2

- To use an ISR2, it is necessary to
  - declare it in the OIL file with the interrupt source identifier (depends on the target platform) to indicate where the interrupt handler is installed;
  - initialize the related interrupt registers of the peripheral which will trigger the interrupt.

ISR2


Declaration keyword for ISR2
Name of the ISR2
ISR(myISR)
{
//code of ISR
…
}
Code of ISR2
Unlike for a task, there is no need for a TerminateISR at the end

ISR2

- OIL Description of an ISR2

```
ISR myISR {
  CATEGORY  = 2;
  PRIORITY  = 1;
  STACKSIZE = 32768;
  SOURCE    = SIGUSR1;
};
```

interrupt category (ISR2)

Static priority. The OIL compiler insure the actual priority of ISRs start above the higher priority of tasks

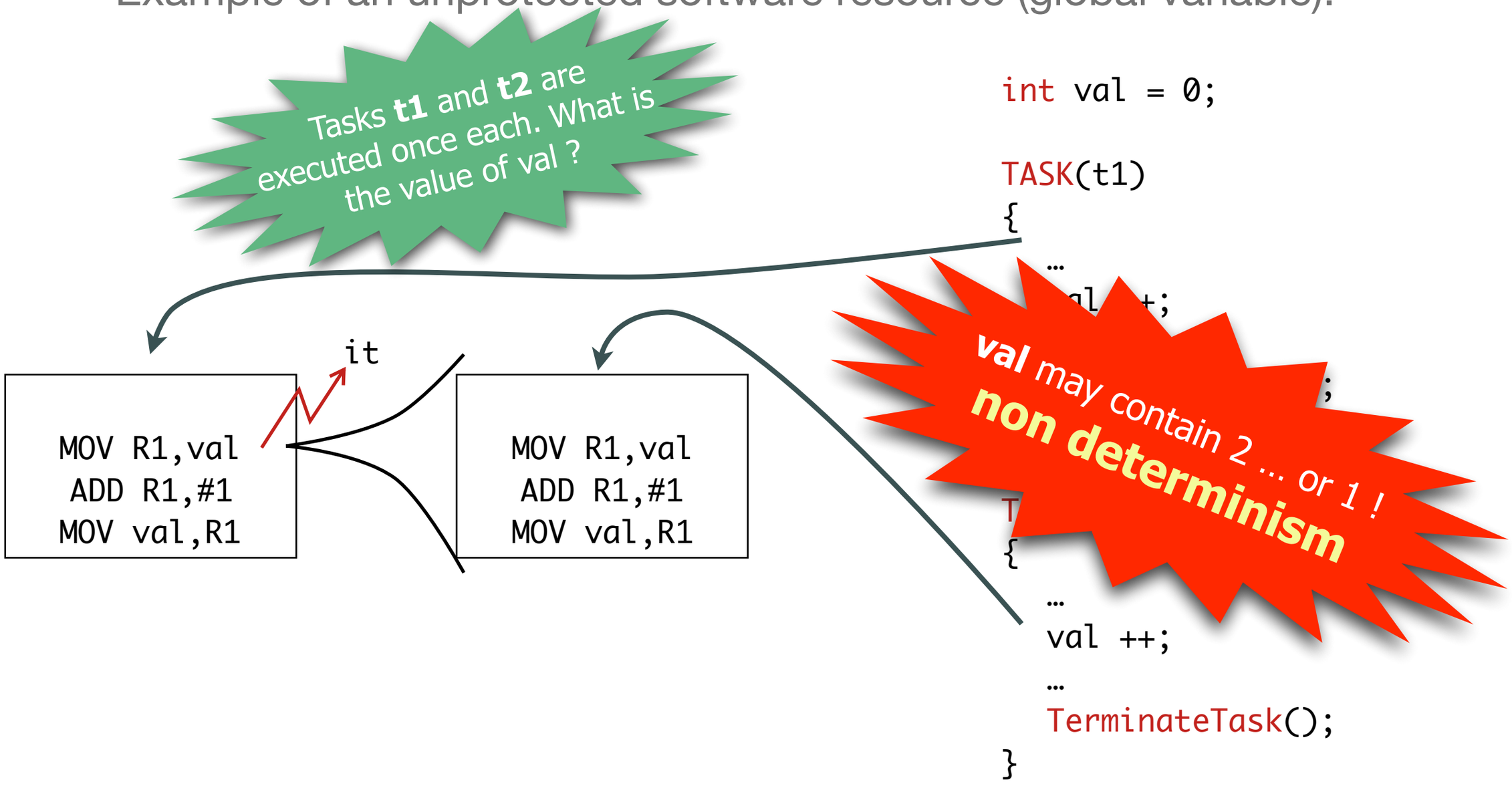
Target specific extension:
- Size of the stack
- Id of the Unix signal (roughly the same thing as a interrupt vector id on a micro-controller)

Accessing shared resources

- Hardware and software resources may be shared between tasks (an optionally between tasks and ISR2 in OSEK)
  - Resource sharing implies a task which access a resource should not be preempted by a task which will access to the same resource.
  - This leads to allow to modify the scheduling policy to give the CPU to a low priority task which access the resource while a high priority task which access the same resource is ready to run.
    - In some cases, priority inversion may occur;
    - Deadlocks may occur when the design is bad.
- In OSEK, the Priority Ceiling Protocol is used to solve this problem.

Example

- Example of an unprotected software resource (global variable):



Semaphore

- Proposed by Edger Dijkstra
- It allows to protect access to shared resources
- This mechanism, available in some OS (not OSEK) offers 3 functions:
  - Init() : initialize the semaphore;
  - P() to test the semaphore (Probieren);
  - V() pour increment the semaphore (Verhogen).

Semaphore

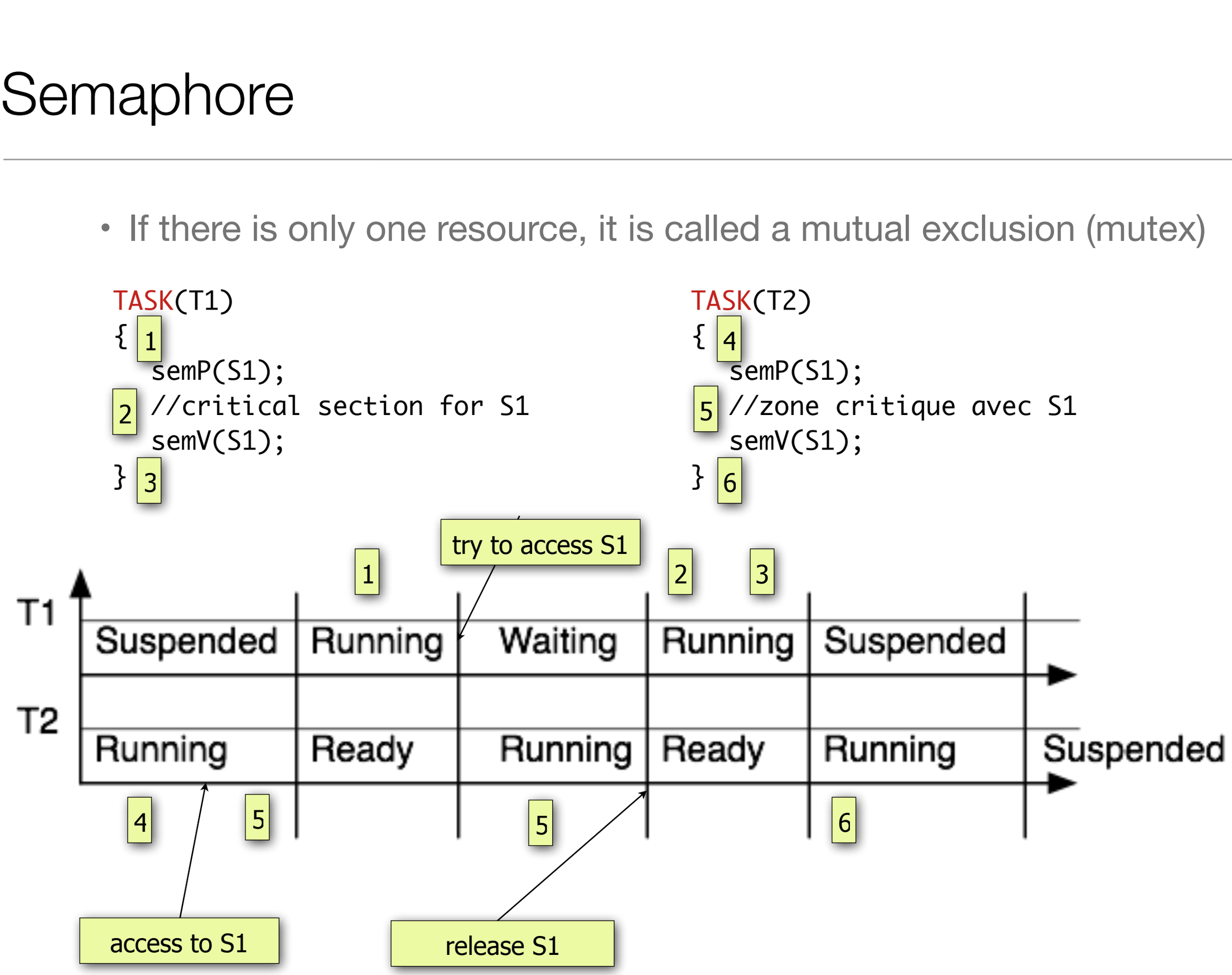
- A counter is associated to the semaphore.
  - A call to P() is used to ask for resource access:
    - If the counter is > 0, it is decremented and the resource may be taken.
    - If the counter is = 0, the task which called P() is put in the waiting state until the counter became > 0. At that time the task will be awaken and the counter will be decremented again.
  - A call to V() is used release a resource:
    - The counter is incremented and a task which is waiting for the resource may be put in ready state.

Semaphore

- If there is only one resource, it is called a mutual exclusion (mutex)

```
TASK(T1)
{ 1
   semP(S1);
 2 //critical section for S1
   semV(S1);
} 3
```

```
TASK(T2)
{ 4
   semP(S1);
 5 //zone critique avec S1
   semV(S1);
} 6
```



Semaphore

- The counter associated to the semaphore may have a non-binary value
- example: access to a buffer:
  - Here are 2 function to read and write the buffer. These functions may be called by concurrent tasks.
- The S1 semaphore which has the initial value of its counter equal to 5 allows up to 5 writes. After that, a task which try to write is put in the waiting state until a task does

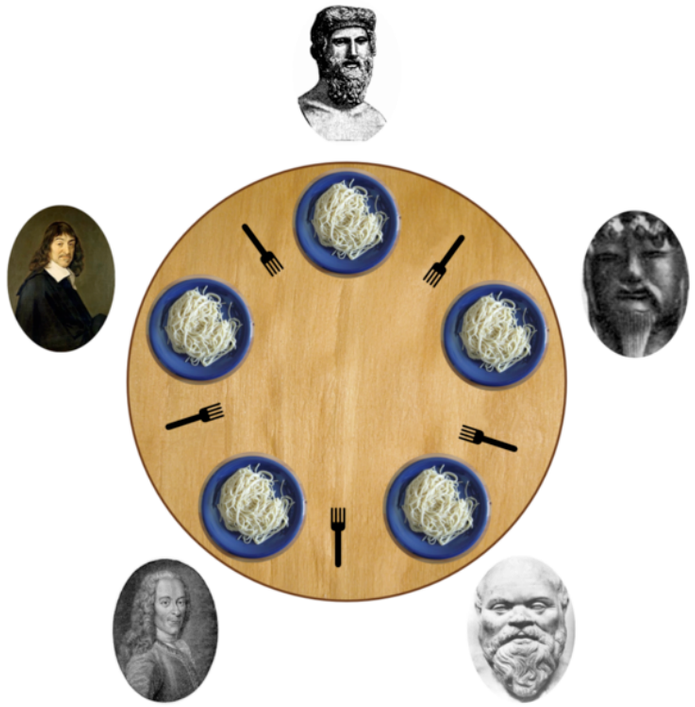
```
void init() {
    semInit(S1, 5);
}
```

```
void WriteBuffer(int data) {
   semP(S1);
   //buffer write
}

void ReadBuffer(int *data) {
   semV(S1);
   //buffer read
}
```

Classical example: the philosophers

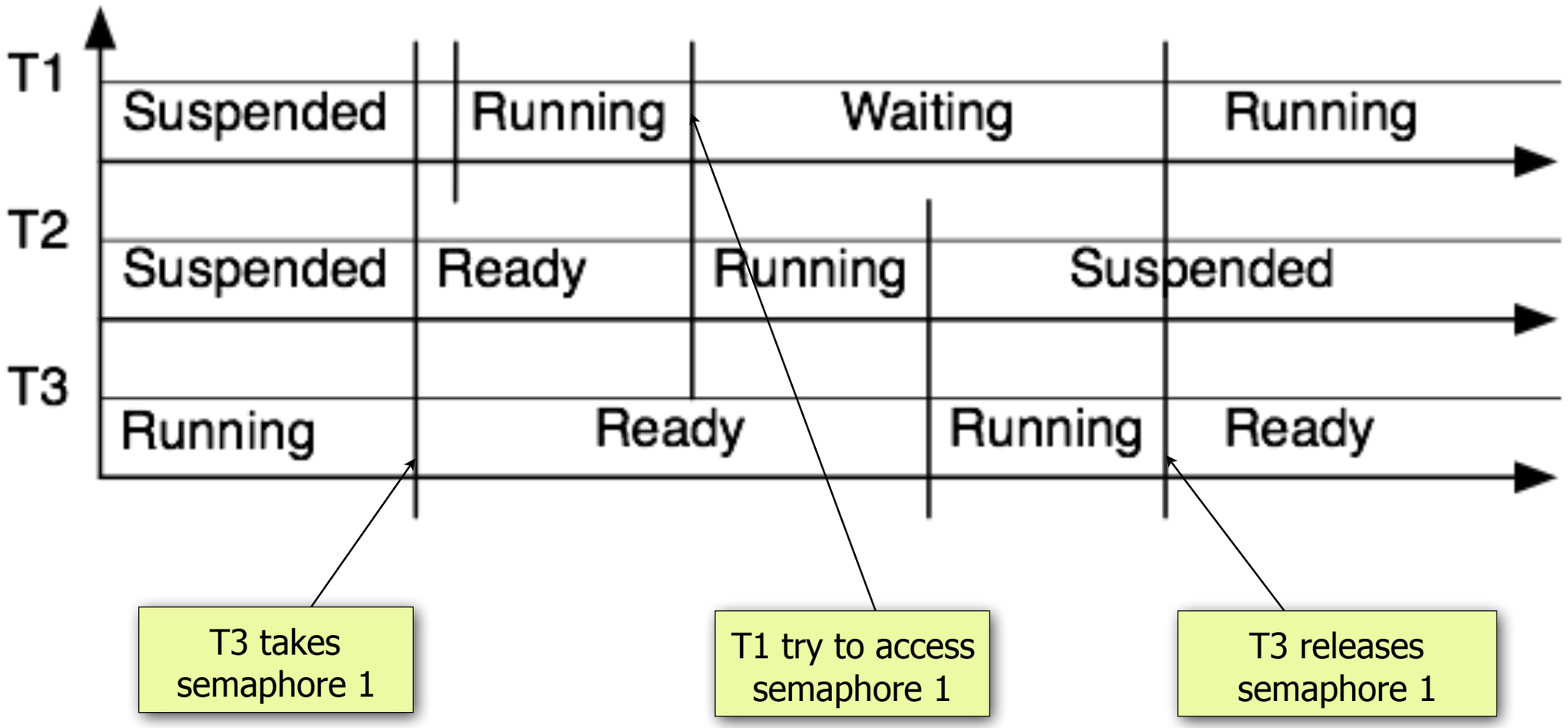
- Given n philosophers (ici n=5), each philosopher may:
  - think (during an unspecified time)
  - eat (during a finite time).
  - wait to eat (during a finite time, otherwise he starves).
- Constrain:
  - To eat, a philosopher needs the fork on its right and on its left.
- Problem:
  - find a solution allowing philosophers to wait a finite time before eating, without deadlock.



source: Wikipédia - licence GNU FDL

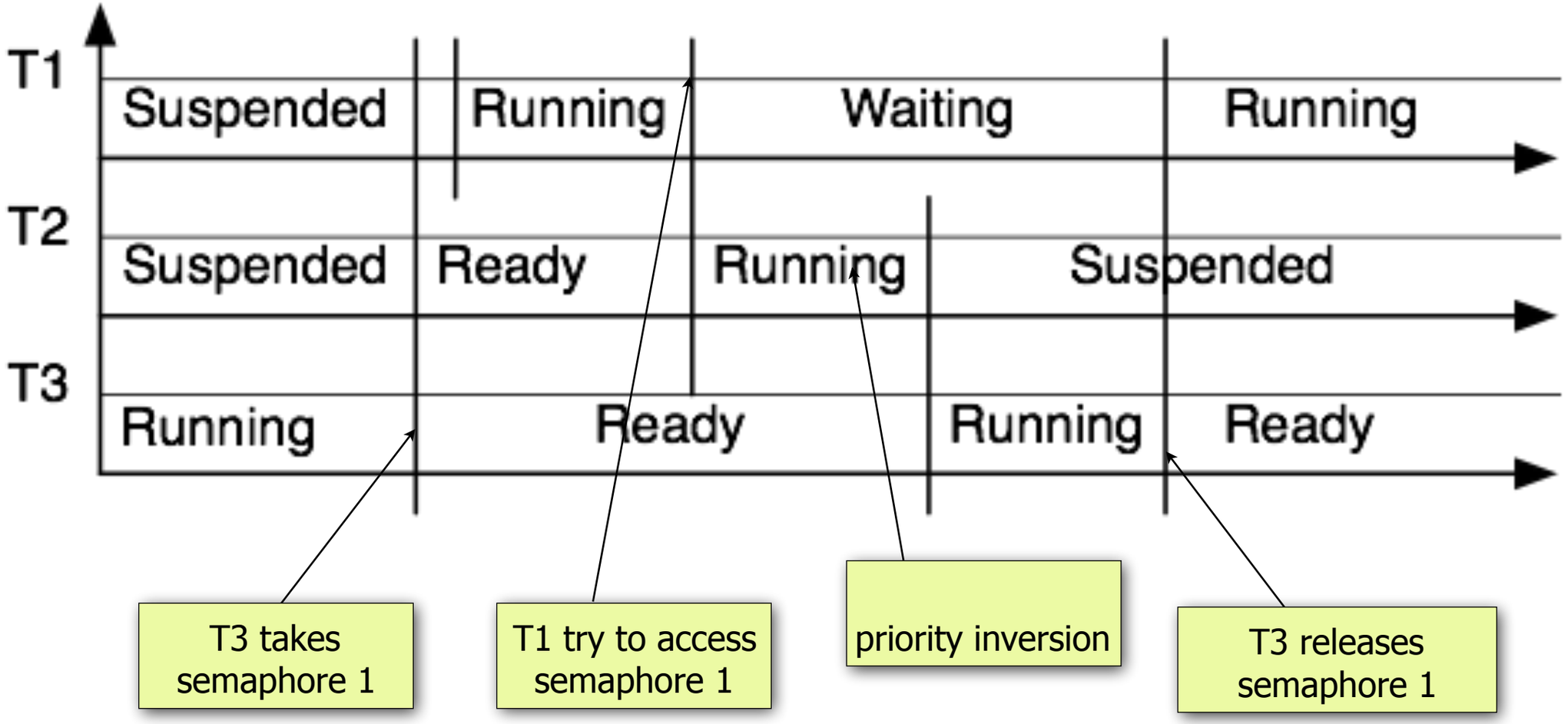
Problem: priority inversion

- Classical synchronization mechanism (semaphore, mutex) may have the priority inversion problem:
  - A task with a lower priority may delay a higher priority task.
  - The following example sows 3 preemptable tasks, T1 has the higher priority:



Problem: priority inversion

- Task 1 (higher priority) try to access semaphore 1 which is used by task 3.
- There is a priority inversion because task1 waits for task 3 which has a lower priority than task 2.
- So task 1 appears to run at lower priority than task 2.

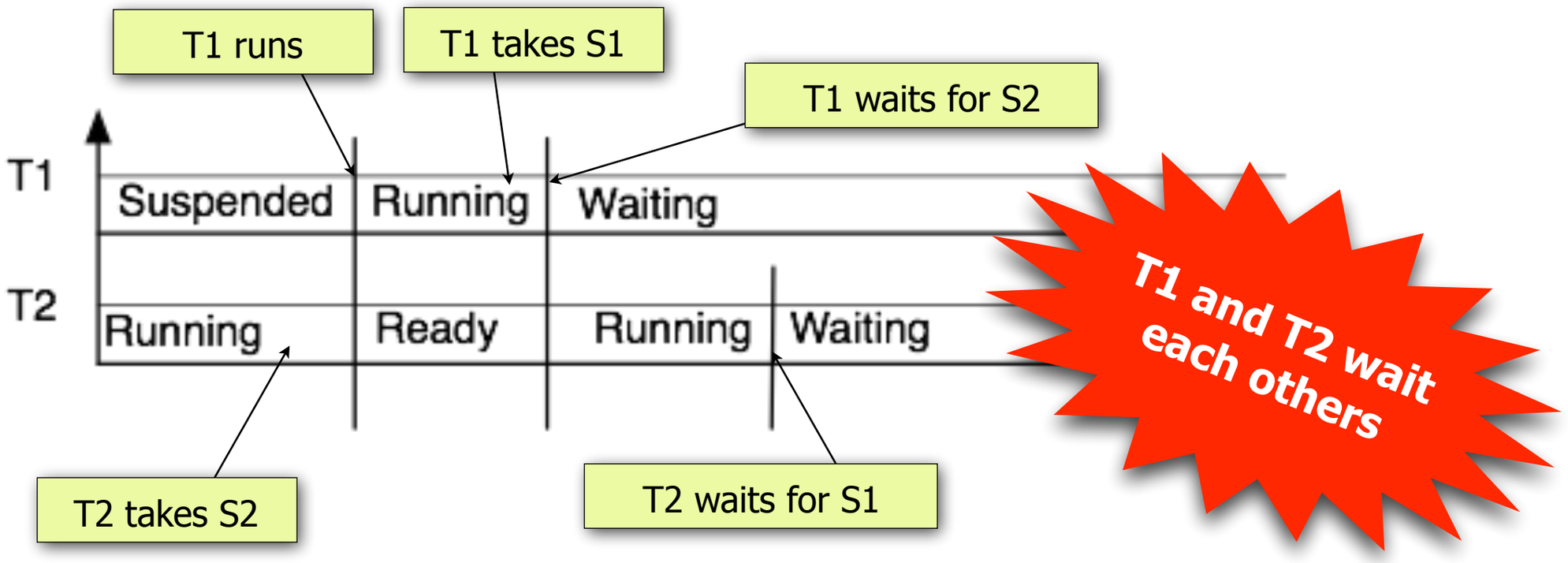


Problem: deadlock

- The biggest problem is the deadlock. It is a design problem

```
TASK(T1)
{
   semP(S1);
   semP(S2);
   //Critical section with S1 and S2
   semV(S2);
   semV(S1);
}
```

```
TASK(T2)
{
   semP(S2);
   semP(S1);
   //Critical section with S1 and S2
   semV(S1);
   semV(S2);
}
```



OSEK resources

- OSEK resources are used to do mutual exclusion between several tasks (or ISR2) to protect the access to a shared hardware or software entity.
- Example of hardware entity:
  - LCD display;
  - Communication network (CAN, ethernet, …).
- Example of software entity:
  - a global variable;
  - the scheduler (in this case, the task may not be preempted).
- OSEK/VDX offers a RESOURCE mechanism with 2 associated system calls (to Get and Release a Resource.

A word about shared global variables

- As we saw the CPU has registers and variables are temporarily copied in the registers (the compiler generates such a code).
- So the memory is not always up to date
- A shared global variable is shared by using the memory. So its value in memory should always be up to date
- In the C language, the `volatile` qualifier tells the compiler to always load and store a global variable from memory instead of copying it in a register each time it is used.

```
volatile int myVar;
```

OSEK resources

- **GetResource**
  - `StatusType GetResource ( ResourceType <ResID> ) ;`
    - Get the resource ResID;
    - StatusType is an error code:
      - E_OK: no error;
      - E_OS_ID: the resource id is invalid;
      - E_OS_ACCESS: trying to get a resource that is already in use (it is a design error).
- A task that « own » the resource may not be preempted by another task that will try to get the resource.

  ⇒ What about the fixed priority scheduling?

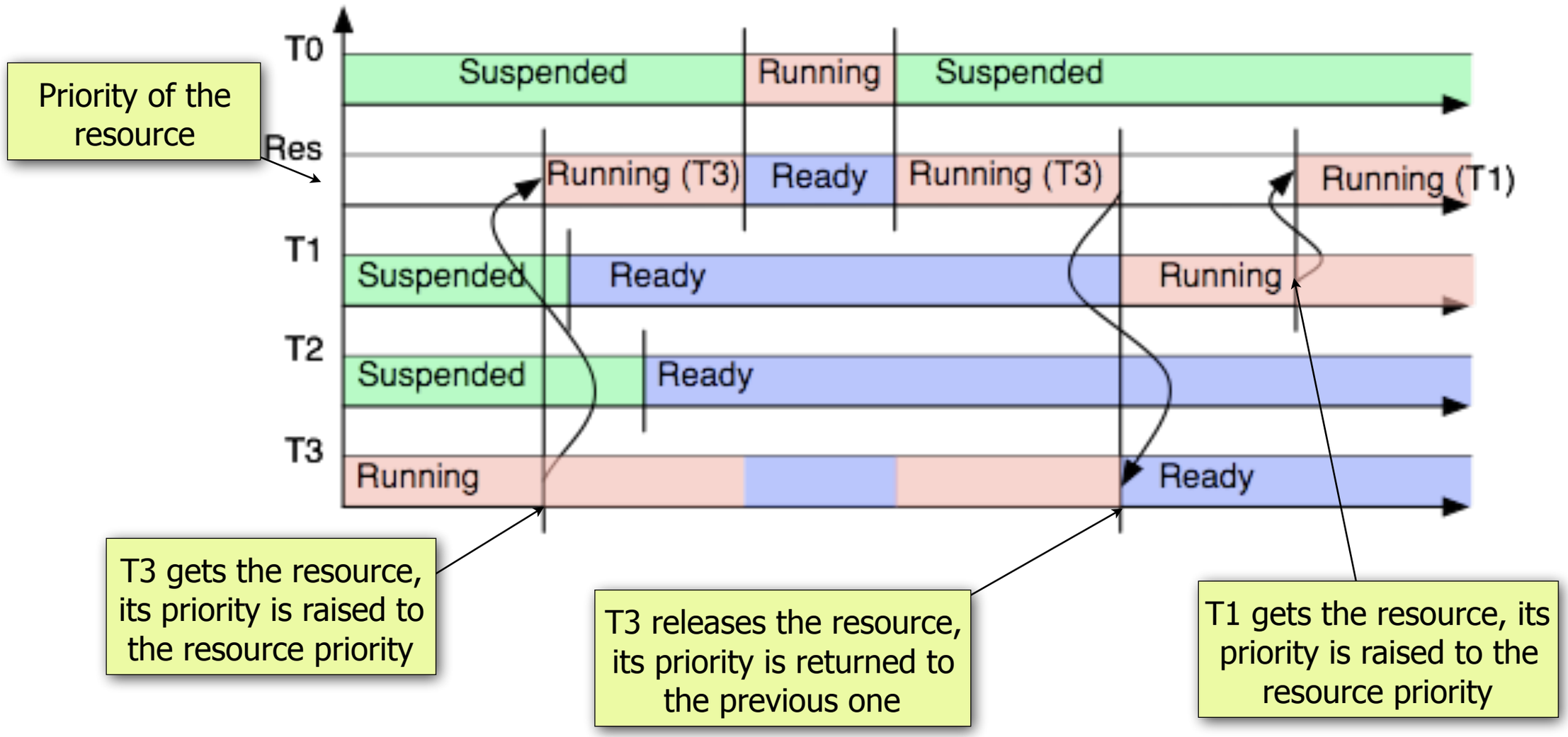
OSEK resources

- **ReleaseResource**
  - `StatusType ReleaseResource ( ResourceType <ResID> ) ;`
    - Release the resource ResID;
  - StatusType is an error code:
    - E_OK: no error;
    - E_OS_ID: the resource id is invalid;
    - E_OS_ACCESS: trying to release a resource that is not in use (it is a design error).

OSEK resources

- To take resources into account in scheduling, a slightly modified PCP (Priority Ceiling Protocol) is used.
- Each resource has a priority such as:
  - The priority is ≥ to max of priorities of tasks which may get the resource;
  - When a task get a resource, its priority is raised to the priority of the resource
  - When a task release the resource, its priority is lowered to the previous one.

OSEK resources



OSEK resources

- T0 has a higher priority than the resource. Its behavior is not modified.
- T2 has a priority set between the priority of T1 and the priority of T3. T2 is delayed while T3 uses the resource.
- T1 is delayed when T3 uses the resource but is never delayed by T2
  - **No priority inversion**
- **No deadlock possible.**

OSEK resources

- Some remarks:
  - An ISR2 may take a resource;
  - Res_scheduler is a resource that disable scheduling when in use. A task which gets Res_scheduler becomes non-preemptable until it releases it;
  - There is no need to get a resource if a task is configured as non-preemptable in the OIL file;
  - A task should get a resource for a time as short as possible. ie only to access a shared entity because higher priority tasks may be delayed.

OSEK resources

- Exceptions:
  - if a shared variable is an atomic one (ie the CPU reads or write it with only one assembly instruction, **AND**
  - the variable is written (and not read) by only 1 task, there is no need to get a resource
  - if a resource is not needed, an ISR2 may be replaced by an ISR1 with better performance.

Tasks group

- This feature allow to mix non-preemptable tasks and preemptable tasks.
  - In the same group all the tasks are seen as non-preemptable by the other tasks of the group.
  - A task having a higher priority of all the tasks of the group (and not part of the group) may preempt any task of the group.
- This feature uses an internal resource for each group:
  - The internal resource is got automatically when the task start to run;
  - The internal resource is released automatically when the task terminates;
  - An internal resource may not be reference with GetResource() or ReleaseResource().

RES_SCHEDULER

- A default internal resource exists:
  - RES_SCHEDULER internal resource has a priority equal to the max priority of the tasks.
  - Any task declared as non-preemptable is in fact in a task group with the internal resource RES_SCHEDULER.

OSEK Resources

- OIL Description of a resource

```
RESOURCE resA {
   RESOURCEPROPERTY = STANDARD;
};
```

the RESOURCEPROPERTY parameter may be STANDARD or INTERNAL. For the latter the resource is got automatically when the task run and released automatically when it calls TerminateTask();

```
TASK myTask {
   PRIORITY = 2;
   AUTOSTART = FALSE;
   ACTIVATION = 1;
   SCHEDULE = NON;
   RESOURCE = ResA;
   STACKSIZE = 32768;
};
```

The priority of the resource is computed according to the priority of all the tasks and ISR2 that use it. So the **resource must be declared.** Otherwise, unpredictable behavior may occur.

Communication in OSEK/VDX OS

- 2 ways to move from one task to an other in the same ECU (Electronic control unit)
  - Shared global variable
    - A resource is needed to synchronize accesses and insure mutual exclusion
    - Events are needed to notify the receiving task a new data have been put in the global variable (control flow and data flow synchronization)
    - Not the best way to implement communication

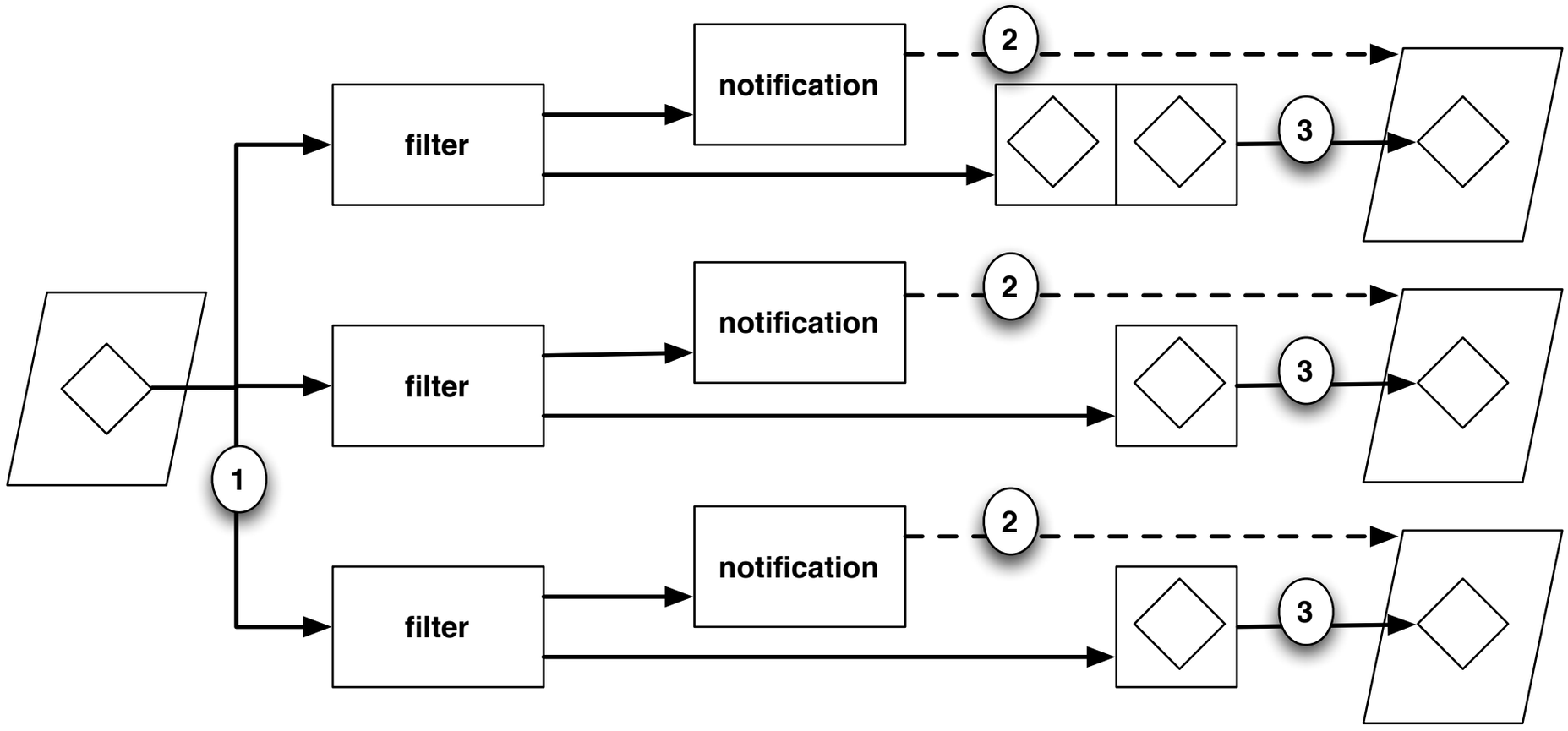
Communication in OSEK/VDX OS

- Message passing
  - No resource needed
  - Mechanisms to synchronize the control flow and the data flow are provided
  - Several configurable schemes are provided to satisfy communication needs in real-time systems
- A better way to communicate but it takes more RAM!

Communication concepts

- Message objects
  - sending
  - or receiving
- Filters
  - associated to a receiving message object
  - allows to determine whether the message must be transmitted or not.
- Notification
  - associated to a receiving message object
  - allows to notify the receiver a new message is arrived

General arrangement



1 : SendMessage (by the sender)
2 : ActivateTask ou SetEvent (by the comminication service)
3 : ReceiveMessage (by the receiver)

Message sending (1)

- A task sends a value (stored in one of its local variables) in a message
  - `StatusType SendMessage( MessageIdentifier <MsgId>, ApplicationDataRef <DataRef> )`
    - <MsgId> : identifier of the message as defined in the OIL file.
    - <DataRef> : memory address (pointer) of the variable to send
- StatusType is an error code:
  - E_OK : no error
  - E_OS_ID : <MsgId> does not exist or is the id of a receiving message

Message sending (2)

```
MESSAGE msgDataSend {
  MESSAGEPROPERTY = SEND_STATIC_INTERNAL {
    CDATATYPE = “uint32”;
  };
};
```

MESSAGEPROPERTY attribute gives the kind of message. For an **internal sending message**, it is always SEND_STATIC_INTERNAL.

CDATATYPE attribute is a string, it corresponds to the C type of the variable.

```
TASK sender1 {
  ...
  MESSAGE = msgDataSend;
  ...
};
```

```
ISR sender2 {
  ...
  MESSAGE = msgDataSend;
  ...
};
```

```
TASK sender1 {
  uint32 truc;
  truc = compute_truc(...);
  SendMessage(msgDataSend, &truc);
  ...
}
```

```
ISR sender2 {
  uint32 muche;
  muche = read_muche(...);
  SendMessage(msgDataSend, &muche);
  ...
}
```

Message receiving (1)

- A task receive a message in one of its local variables. In the meantime, the message is stored in a receiving message object. There are two kinds of receiving message objects:
  - UNQUEUED : Only the last value is stored. Each new value override the previous one. This kind of message is used to transmit the value of a state of the system (the current value is of interest). This is also called a *state message* or a *blackboard transmission*.
  - QUEUED : *n* last values are stored in a queue. If a message arrives while the queue is full, this message is lost. This kind of message is used when each value has to be taken into account. This is also called *event message*, or a *letterbox transmission*.

Message receiving (2)

- A task receives a message in one of its local variables
  - `StatusType ReceiveMessage ( MessageIdentifier  <MsgId>, ApplicationDataRef <DataRef> )`
    - <MsgId> : identifier of the message as defined in the OIL file
    - <DataRef> : Memory address of the variable to receive
  - StatusType is an error code
    - E_OK : no error
    - E_COM_NOMSG : the message is a QUEUED one and the queue is empty. Nothing is stored in <DataRef>
    - E_COM_LIMIT : the message is a QUEUED one and a least one message has been lost
    - E_OS_ID : <MsgId> does not exist or is a sending message.
  - The receiving task is never blocked when it calls ReceiveMessage

Message receiving, queued messages (3)

```
MESSAGE msgDataRec1 {
  MESSAGEPROPERTY = RECEIVE_QUEUED_INTERNAL {
    SENDING_MESSAGE = msgDataSend;
    FILTER = NEWISDIFFERENT;
    QUEUESIZE = 4;
  };
  NOTIFICATION = SETEVENT {
    TASK = Receiver1;
    EVENT = msgIn;
  };
  CDATATYPE = “uint32”;
};
```

FILTER attribute is one of the predefined filters (explained hereafter).

QUEUESIZE attribute is the size of the queue (number of messages)

NOTIFICATION may be ACTIVATETASK, SETEVENT or NONE. The notification is performed when a new message arrives

Message receiving, unqueued message (4)

```
MESSAGE msgDataRec2 {
  MESSAGEPROPERTY = RECEIVE_UNQUEUED_INTERNAL {
    SENDINGMESSAGE = msgDataSend;
    FILTER = NEWISDIFFERENT;
    INITIALVALUE = 0x00;
  };
  NOTIFICATION = ACTIVATETASK {
    TASK = Receiver2;
  };
};
```

INITIALVALUE defines the initial value stored in the message. It is a UINT64. It cannot be used for any datatype.

Message receiving (5)

```
TASK Receiver1 {
  AUTOSTART = TRUE{
    APPMODE = DefaultAppMode;
  };
  MESSAGE = msgDataRec1;
  EVENT = msgIn;
  ...
};

EVENT msgIn {
  MASK = AUTO;
};
```

```
TASK(Receiver1)
{
  uint32 data;
  ...
  do_some_stuff(...);
  while(1) {
    WaitEvent(msgIn);
    ClearEvent(msgIn);
    ReceiveMessage(msgDataRec1, &data);
    process_data(data);
  }
  TerminateTask();
}
```

Message receiving (6)

```
TASK Receiver2 {
  AUTOSTART = FALSE;
  MESSAGE = msgDataRec2;
};
```

```
TASK(Receiver2)
{
  uint32 data;
  ...
  ReceiveMessage(msgDataRec2, &data);
  process_data(data);
  TerminateTask();
}
```

Message filtering (1)

- It is not always useful to receive all the messages…
  - when 2 values in a row are the same
  - when the value is greater than a threshold. The message is used by a different task. For instance a control system where an alternate law is used according to the value
  - when the sender has a period *x* times smaller than the period of the receiver. In this case 1 message every *x* is received
- For these cases (an for others) OSEK/COM allows to set a filter for each receiving message.
- The message is delivered if the filter result is TRUE

Message filtering (2)

Filter	Sub-attribute	Result
ALWAYS		VRAI
NEVER		FALSE
MASKEDNEWEQUALSX	MASK, X	new & MASK == X
MASKEDNEWDIFFERSX	MASK, X	new & MASK != X
NEWISEQUAL		new == old
NEWISDIFFERENT		new != old
MASKEDNEWEQUALSMASKEDOLD	MASK	new & MASK == old & MASK
MASKEDNEWDIFFERSMASKEDOLD	MASK	new & MASK != old & MASK
NEWISWITHIN	MIN,MAX	(MIN <= new) && (new <= MAX)
NEWISOUTSIDE	MIN, MAX	(new < MIN) || (new > MAX)
NEWISGREATER		new > old
NEWISLESSOREQUAL		new <= old
NEWISLESS		new < old
NEWISGREATEROREQUAL		new >= old
ONEEVERYN	PERIOD, OFFSET	msg_number % PERIOD == OFFSET

Message filtering (3)

```
MESSAGE msgDataRec1 {
  MESSAGEPROPERTY = RECEIVE_QUEUED_INTERNAL {
    SENDING_MESSAGE = msgDataSend;
    FILTER = ONEEVERYN {
      PERIOD = 3;
      OFFSET = 1;
    };
  ...
};
```

The receiver receives **1 message every N from start**. So it will receive messages 1, 4, 7, 10, 13, ...

```
MESSAGE msgDataRec2 {
  MESSAGEPROPERTY = RECEIVE_UNQUEUED_INTERNAL {
    SENDING_MESSAGE = msgDataSend;
    FILTER = MASKEDNEWDIFFERSMASKEDOLD {
      MASK = 0xFF000000;
    };
  ...
};
```

The receiver receives the message **if the 3 lower bytes of the current message and of the previous one are different.**

Summary

- Communication services allows tasks to exchange messages
- A message may be sent by an ISR2 too
- A sending message object may have many producers
- A receiving message object has one consumer only
- Many receiving messages may be associated to the same sending message object
- Filtering allows select messages which are received
- Notification mechanism is used to inform the consumer a new message is available by activating a task or setting an event.

Summary of OSEK/VDX OS

Needed support	OSEK/VDX OS
asynchronous real-time applications	fixed-priority preemptive scheduling, « real-time » resource sharing protocol, management of time with counters and alarms, interrupts: **OK**
many hardware platform with many profiles (performance, memory size)	static system, conformance classes: **OK**
“safety-critical” applications (ABS, ESP, active steering, …)	?

Is OSEK/VDX OS compatible with the requirements of critical applications ?

Critical application ?

- A critical application must have a predictable behavior, even when an error occurs
  - Design or programming error: for instance an unexpected infinite loop, a pointer to a memory zone which is not owned by the task, a stack overflow, …
  - Hardware failure: a peripheral that has an interrupt rate higher than the expected one, memory corruption, …
- These failures lead generally to a runtime error
  - A task or ISR is activated too often, runs longer than expected, …
- These failures may lead to a failure of the ECU, failure of one of the functions (or more) and lastly failure of the vehicle itself
- Safety must be taken into account at every stage of the design process of all the components. The OS must be robust and should detect and tolerate faults. Is it anticipated in OSEK/VDX OS ?

OSEK/VDX OS and critical applications

- The OSEK/VDX OS standard does not specify:
  - that a memory protection mechanism must be implemented. Without such a mechanism, a task may access anywhere in memory and corrupt it. The kernel may crash
  - that a temporal protection mechanism must be implemented. Without such a mechanism, a task may run for a longer time than expected or may be activated with a higher frequency than expected. Low priority tasks may not run and the real-time behavior is wrong.

- Conclusion : a kernel implemented with the OSEK/VDX OS standard does not fit critical applications requirements because its behavior is unpredictable when an error occurs!

Summary of OSEK/VDX OS

Needed support	OSEK/VDX OS
asynchronous real-time applications	fixed-priority preemptive scheduling, « real-time » resource sharing protocol, management of time with counters and alarms, interrupts: **OK**
many hardware platform with many profiles (performance, memory size)	static system, conformance classes: **OK**
“safety-critical” applications (ABS, ESP, active steering, …)	No requirement in the standard: **KO**

OSEK/VDX OS fits the needs of automotive applications except the critical ones

AUTOSAR

- AUTOSAR = AUTomotive Open System Architecture
- Consortium gathering most of companies of the automotive industry: Car manufacturers, components manufacturers, silicon founder, software editor.
- Set of specification spanning from design to implementation of embedded electronic systems
  - latest public version: 4.2
- More informations on the site: http://www.autosar.org

AUTOSAR OS

- AUTOSAR OS is the specification of an OS kernel for AUTOSAR architecture. It is based on OSEK/VDX OS :

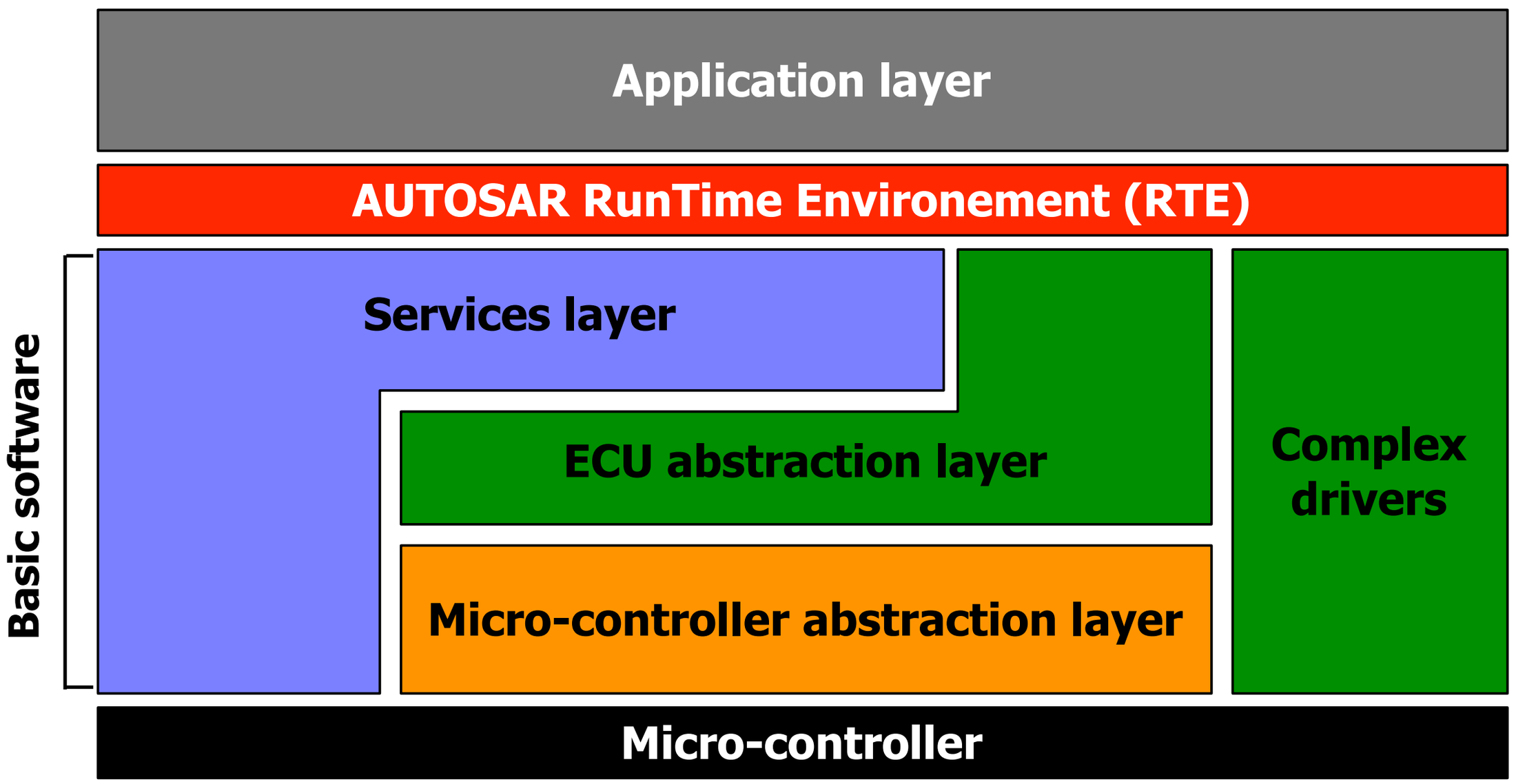
  - It adds new concepts: OS-Application, Schedule Table, Global Time, Software Counter (presented hereafter)

  - But the main improvement is the addition of configurable mechanisms for error detection in the kernel (called “protection facilities”)
    This rectify the OSEK/VDX OS inability to support critical systems

Goals

- Definition of a flexible and modular architecture which fits the different hardware platforms and the different applications' requirements
- Ease portability and software component reuse (application and system components)
- Insure the interoperability of the software components from different sources
- Leverage the embedded software component market

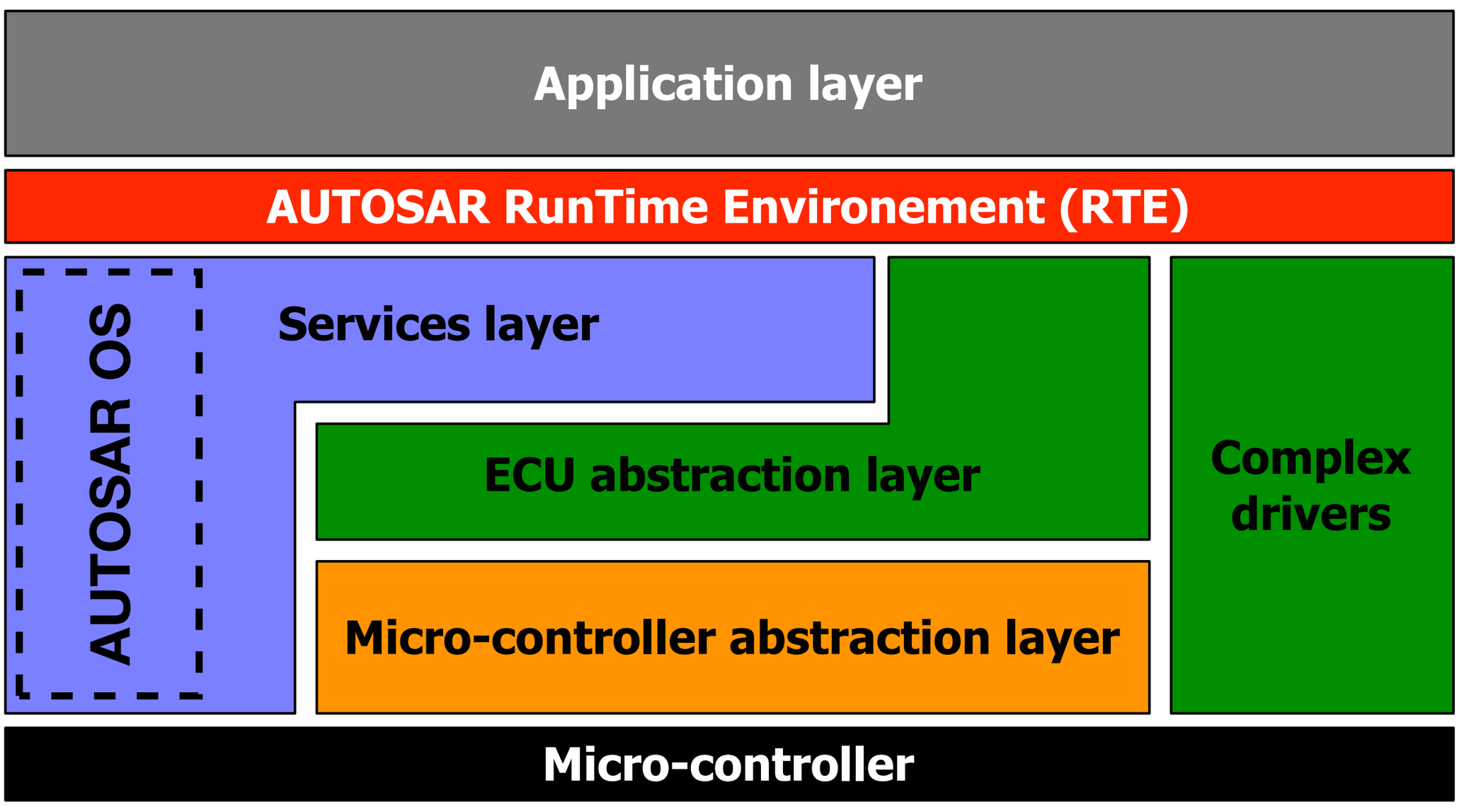
Reference architecture (1)



Reference architecture (2)

- Application layer: software components which implement ECU functions
- RTE: middleware which implements and access to the hardware resources for the application layer
- Services layer: AUTOSAR OS kernel, communication protocols, NVRAM access services, diagnostic services
- ECU abstraction layer: high level access to on-board and off-board devices
- Micro-controller abstraction layer: low level access to on-board devices.
- Complex drivers: free slot in the architecture to implement special device drivers.

AUTOSAR OS in the architecture



AUTOSAR OS

- AUTOSAR OS is the specification of an OS kernel for AUTOSAR architecture. It is based on OSEK/VDX OS :

  - It adds new concepts: OS-Application, Schedule Table, Global Time, Software Counter (presented hereafter)

  - But the main improvement is the addition of configurable mechanisms for error detection in the kernel (called “protection facilities”)
    This rectify the OSEK/VDX OS inability to support critical systems

AUTOSAR OS

- AUTOSAR OS is based on OSEK/VDX OS and adds (and sometimes remove or modify) some features :
  - What have been added
    - Software counters
    - Schedule Tables
    - Global time synchronization
    - Stack monitoring
    - OS Applications
    - Protection mechanisms

Software counters

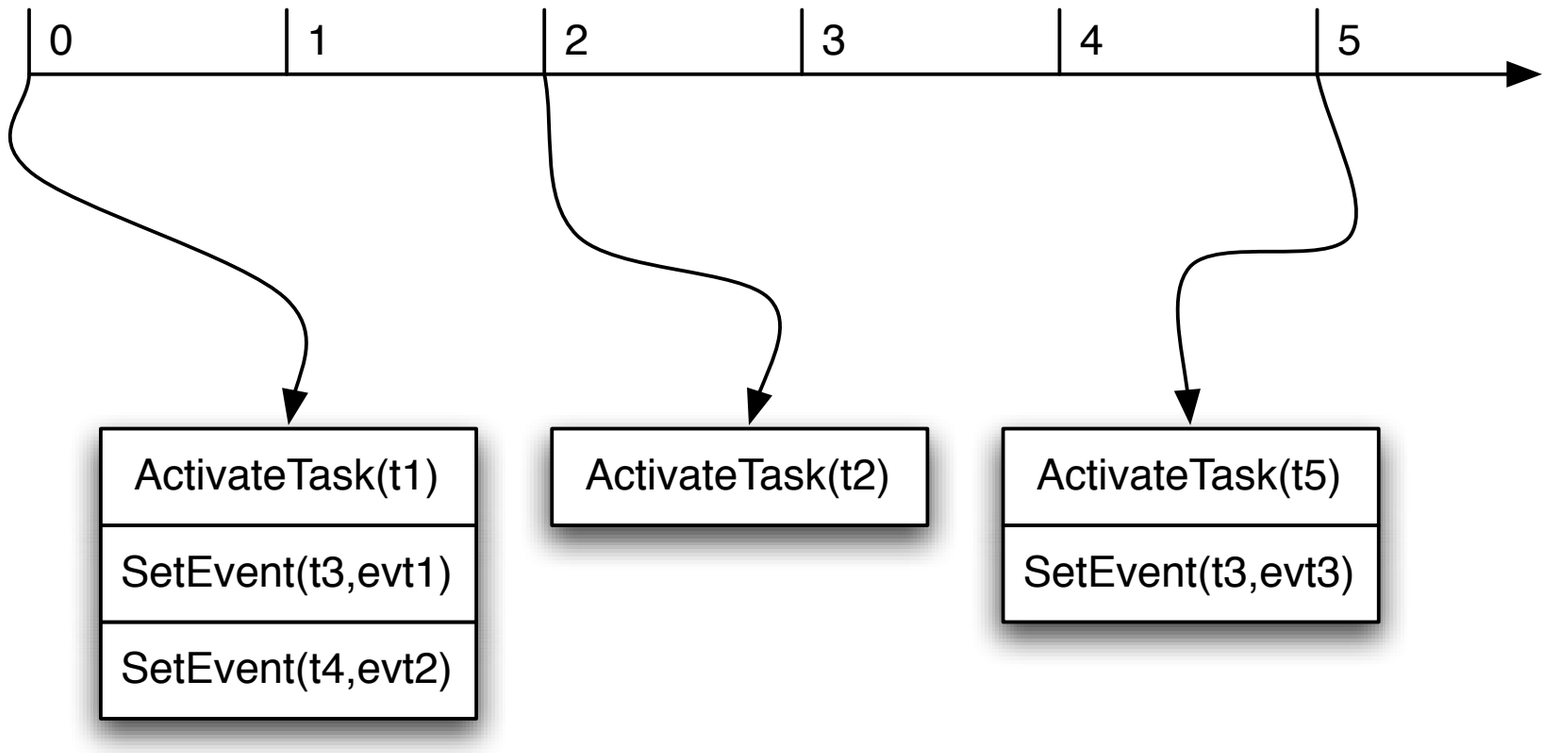
- Like OSEK hardware counters + API to manage them.
  - `StatusType IncrementCounter(CounterType cpt)`
    - Increment the counter by 1. The TicksPerBase is taken into account.
  - `StatusType GetCounterValue(`
    `CounterType  cpt,`
    `TickRefType  value`
    `)`
    - returns in value the value of the counter
  - `StatusType GetElapsedCounterValue(`
    `CounterType  cpt,`
    `TickType     previousValue,`
    `TickRefType  value`
    `)`
    - returns in value the difference between the current value and previousValue

Schedule Tables (1)

- In OSEK/VDX the implementation of periodic operations (task activation for instance) requires at least one alarm and one task:
  - it is an overused design pattern
  - not so easy to implement and error prone
- Schedules tables offer this kind of mechanism for :
  - task activation
  - event setting
- A schedule table may be periodic.

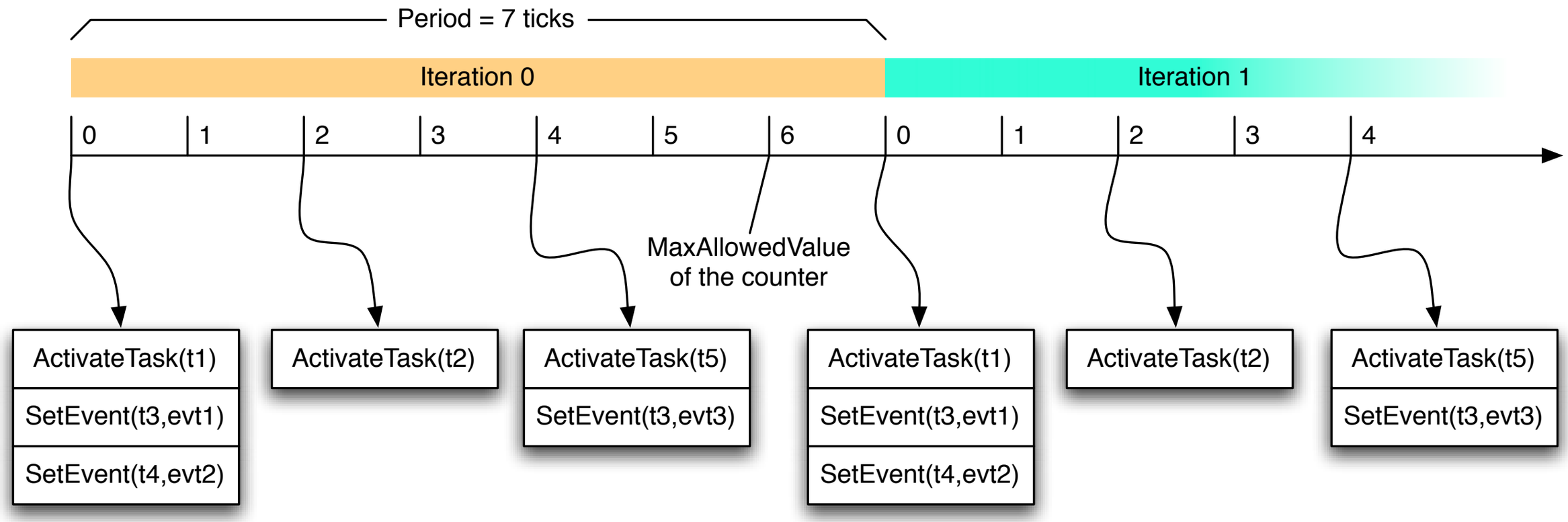
Schedule Tables (2)

- A schedule table is a set of dated Expiry Points
  - A schedule table is driven by a counter
    - So schedule table dates are between 0 and the MaxAllowedValue of the counter.



Schedule Tables (3)

- For a periodic schedule table, the period sets the position of offset 0 for the next iteration



Schedule Tables (4)

- Services to start, stop and chain schedule tables
  - `StartScheduleTableAbs(ScheduleTableType st, TickType date);`
    - Start a schedule table `st` at an absolute `date`, the expiry point at date 0 will be executed when the counter will reach the date.
  - `StartScheduleTableRel(ScheduleTableType st, TickType delay);`
    - Start a schedule table `st` at a relative `delay`, the expiry point at date 0 will be executed after `delay` ticks starting from now.
  - `StopScheduleTable(ScheduleTableType st);`
    - Stop the processing of schedule table `st`.
  - `NextScheduleTable(ScheduleTableType cst, ScheduleTableType`

OIL description

- New object SCHEDULETABLE

```
SCHEDULETABLE st1 {
    COUNTER = counter100ms ;
    PERIODIC = TRUE ;
    AUTOSTART = FALSE ;
    LENGTH = 10 ;
    ACTION = ACTIVATETASK {
        OFFSET = 0;
        TASK = t1 ;
    };
    ACTION = ACTIVATETASK {
        OFFSET = 3;
        TASK = t1 ;
    };
    ACTION = ACTIVATETASK {
        OFFSET = 8;
        TASK = t1 ;
    };
};
```