OSEK/VDX and AUTOSAR compliant RTOS

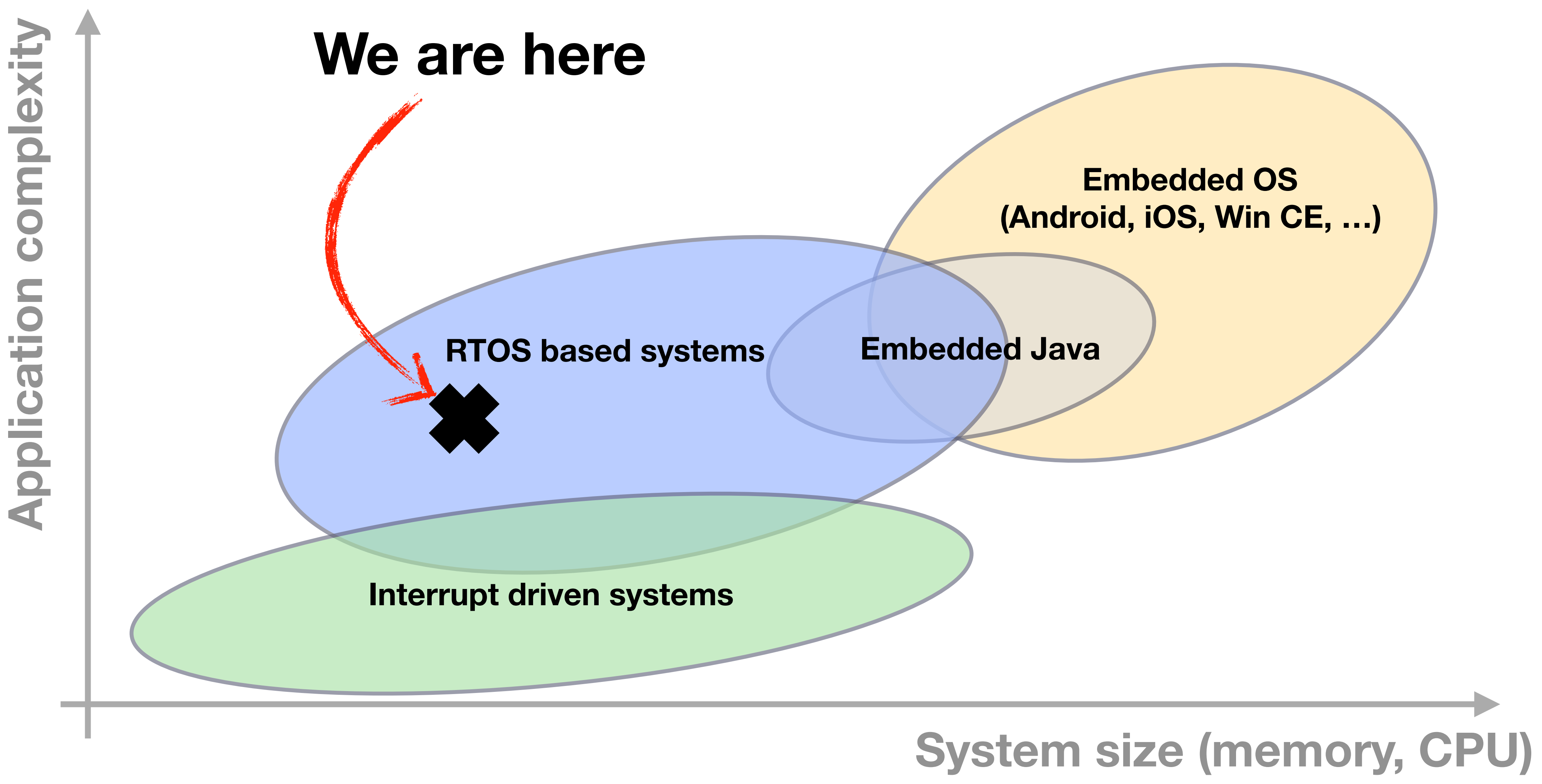
Context 1/2

- 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 2/2

- 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

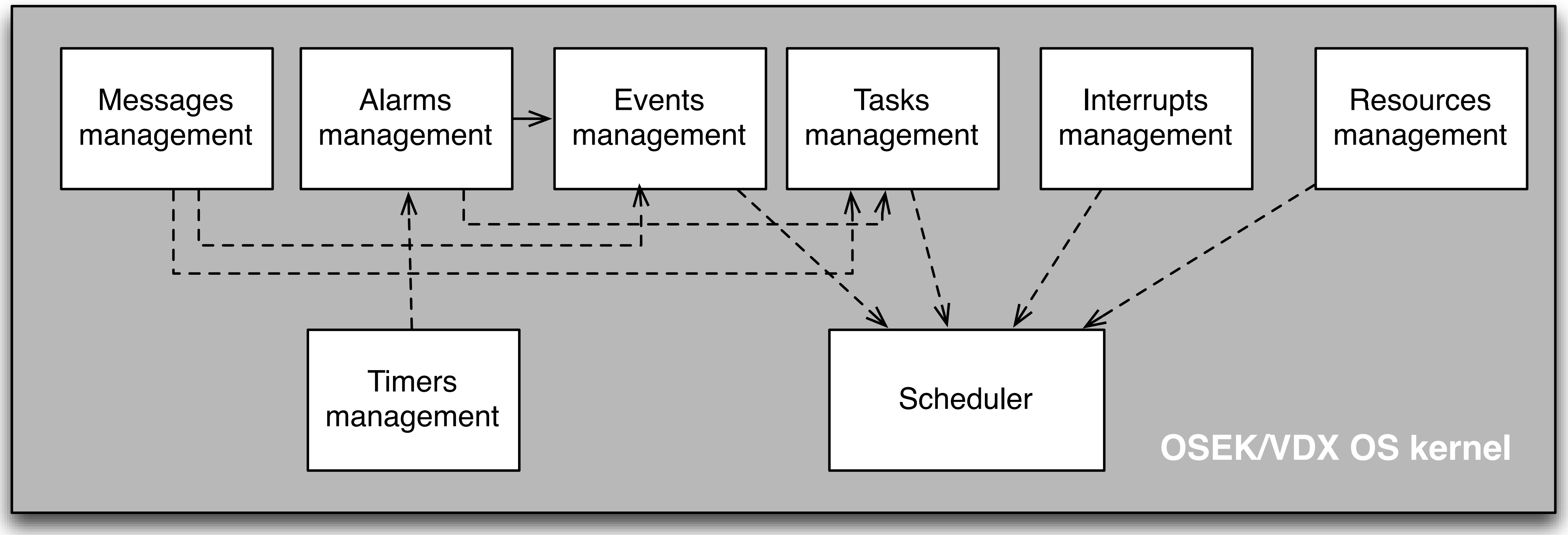
Where is OSEK compared to other OS



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

Architecture of OSEK/VDX OS



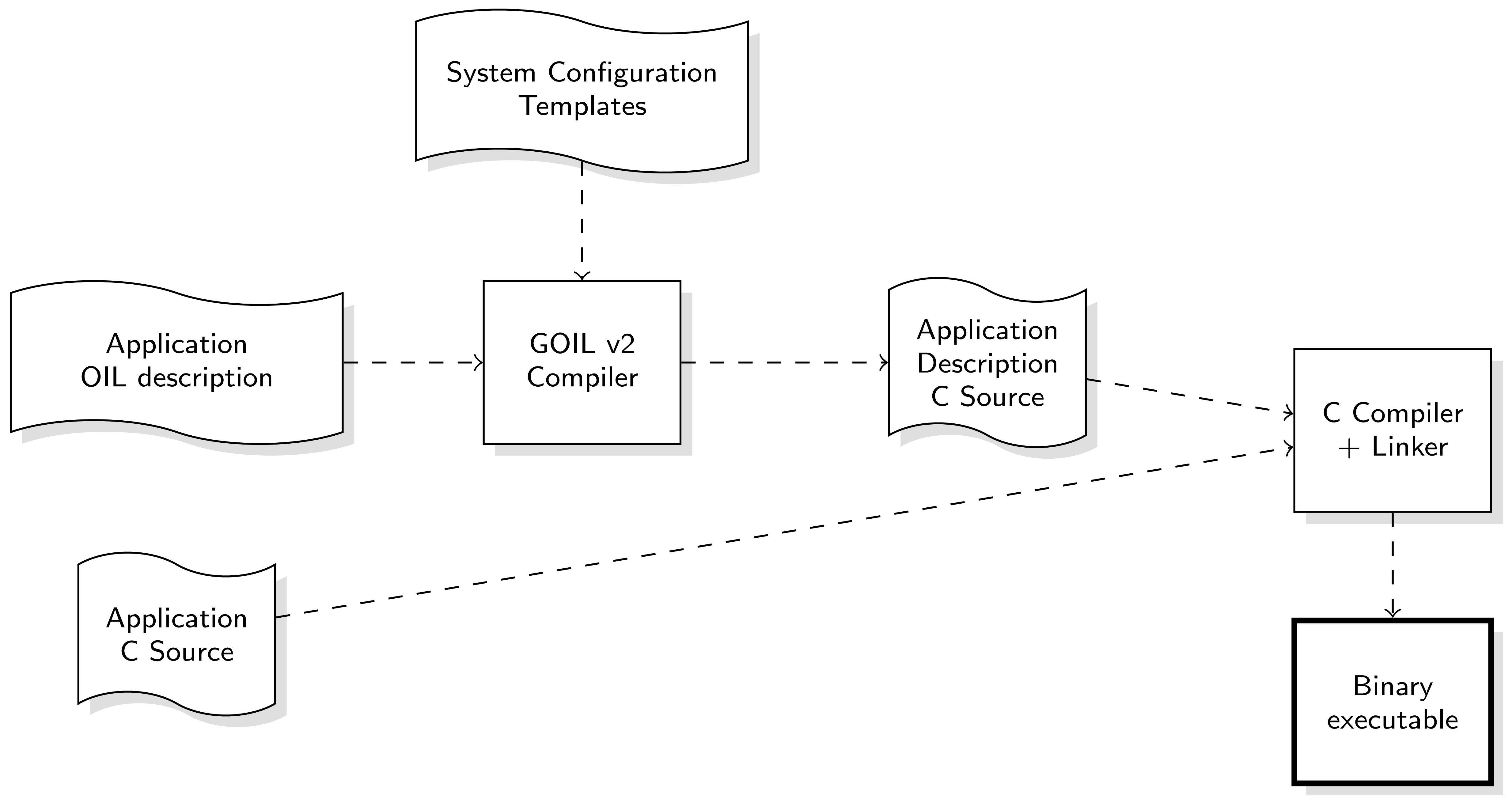
Development process of an OSEK OS + OIL Application

- 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


System Configuration Templates
Application OIL description
GOIL v2 Compiler
Application Description C Source
C Compiler + Linker
Application C Source
Binary executable

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).

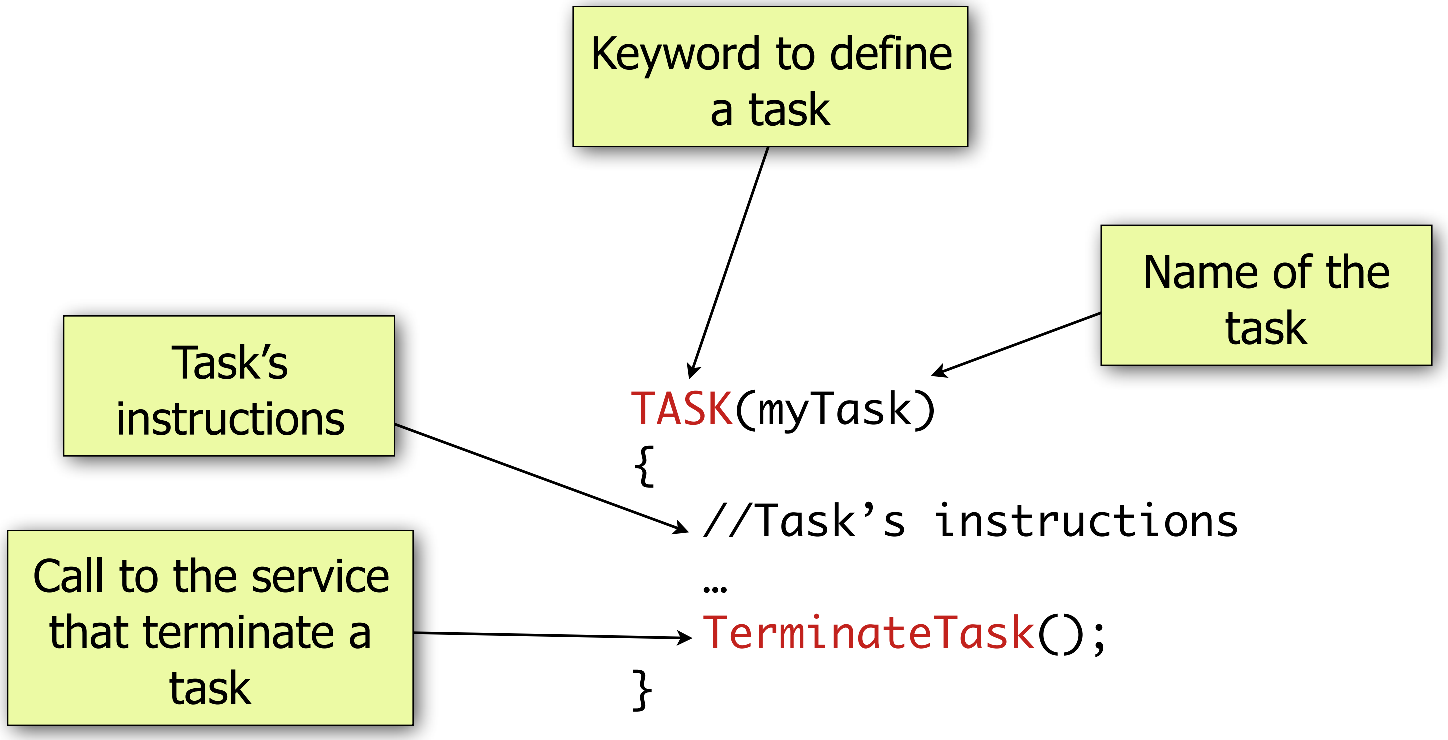
Services of OSEK

- 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!!

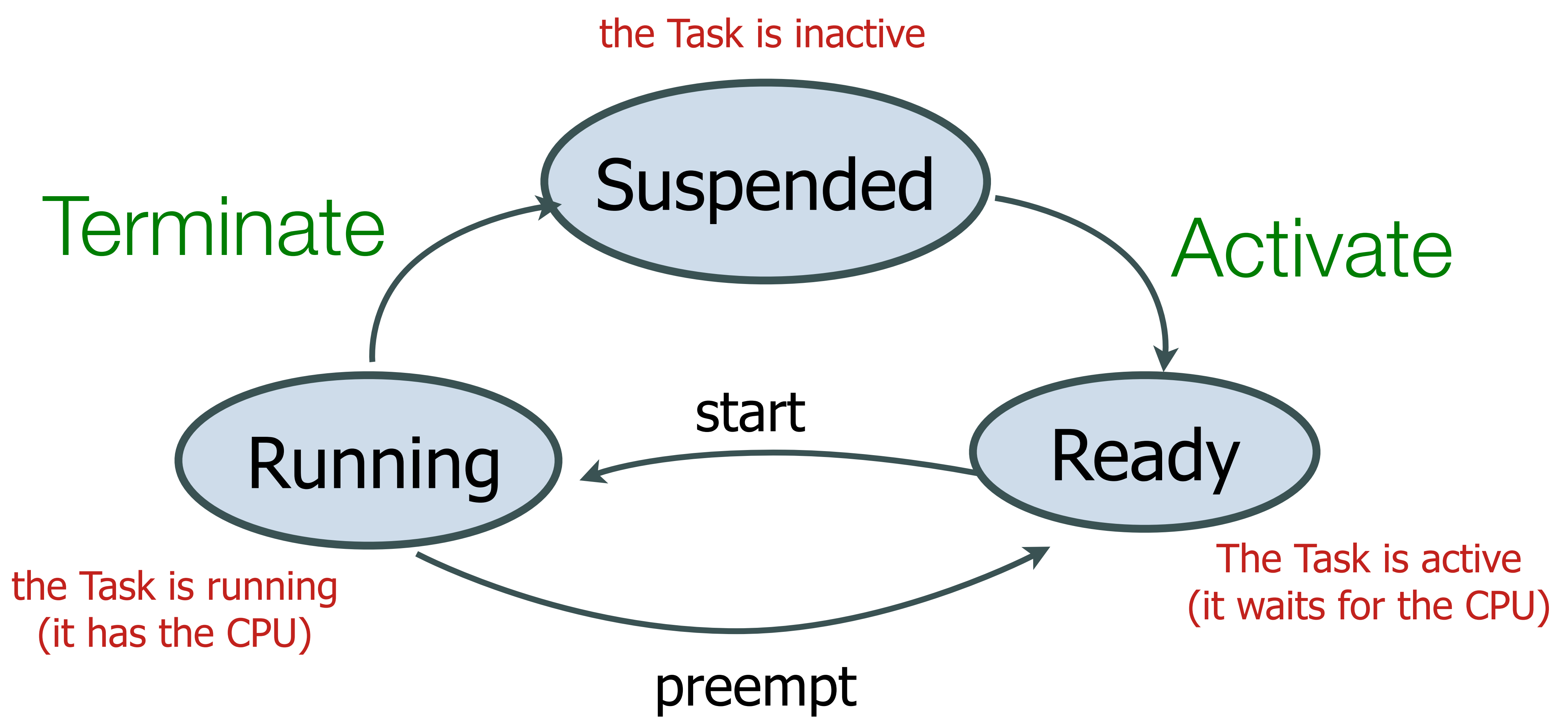
Tasks in OSEK

- 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)



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

Basic task states



OSEK scheduling policy

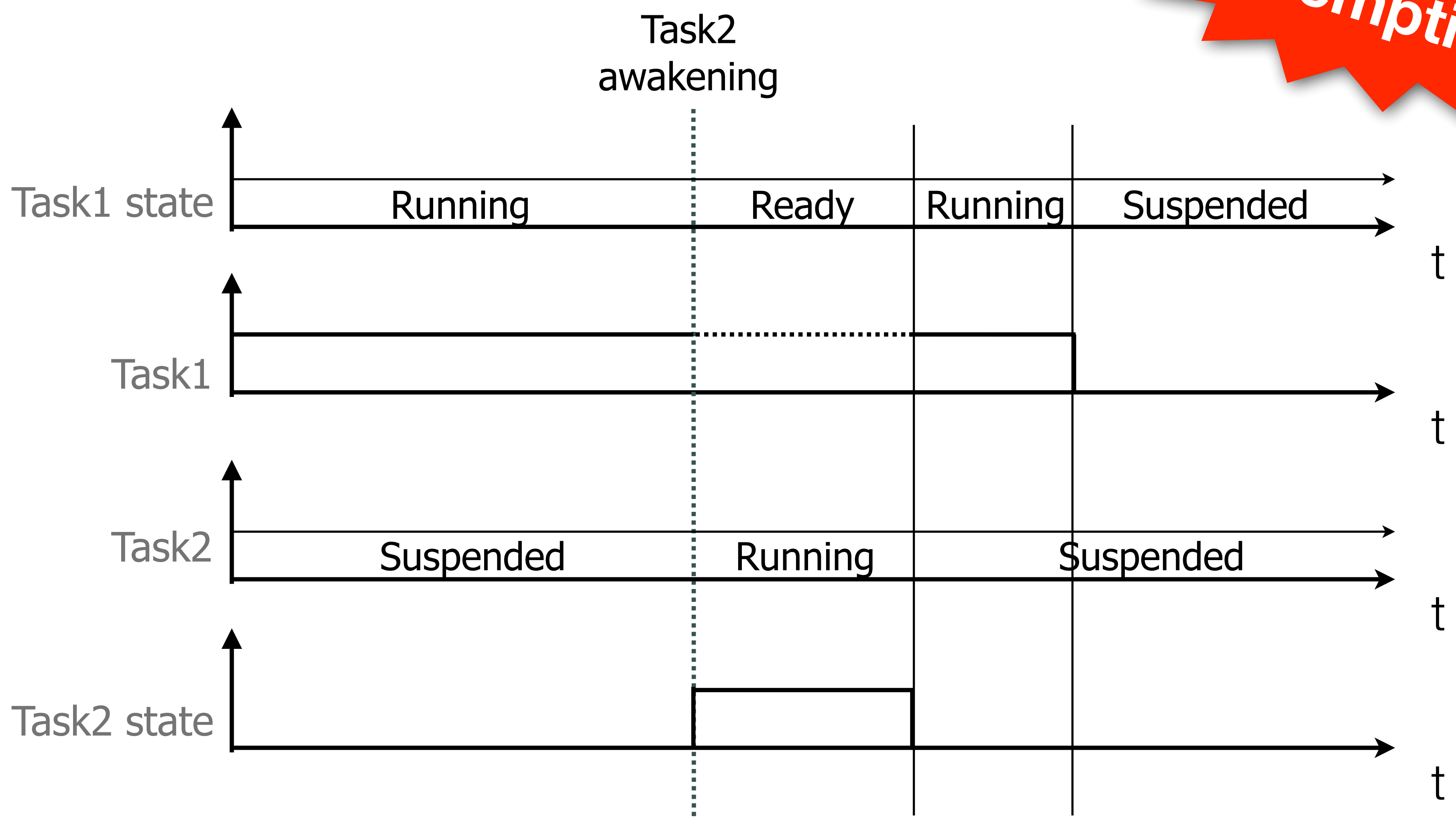
- Scheduling is done in-line
  - Scheduling is done dynamically during the execution of the application.
- Tasks have a fixed priority
  - The priority of a task is given at design stage;
  - The priority does not change (almost, taking and releasing resources may change the priority);
  - No round-robin. 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

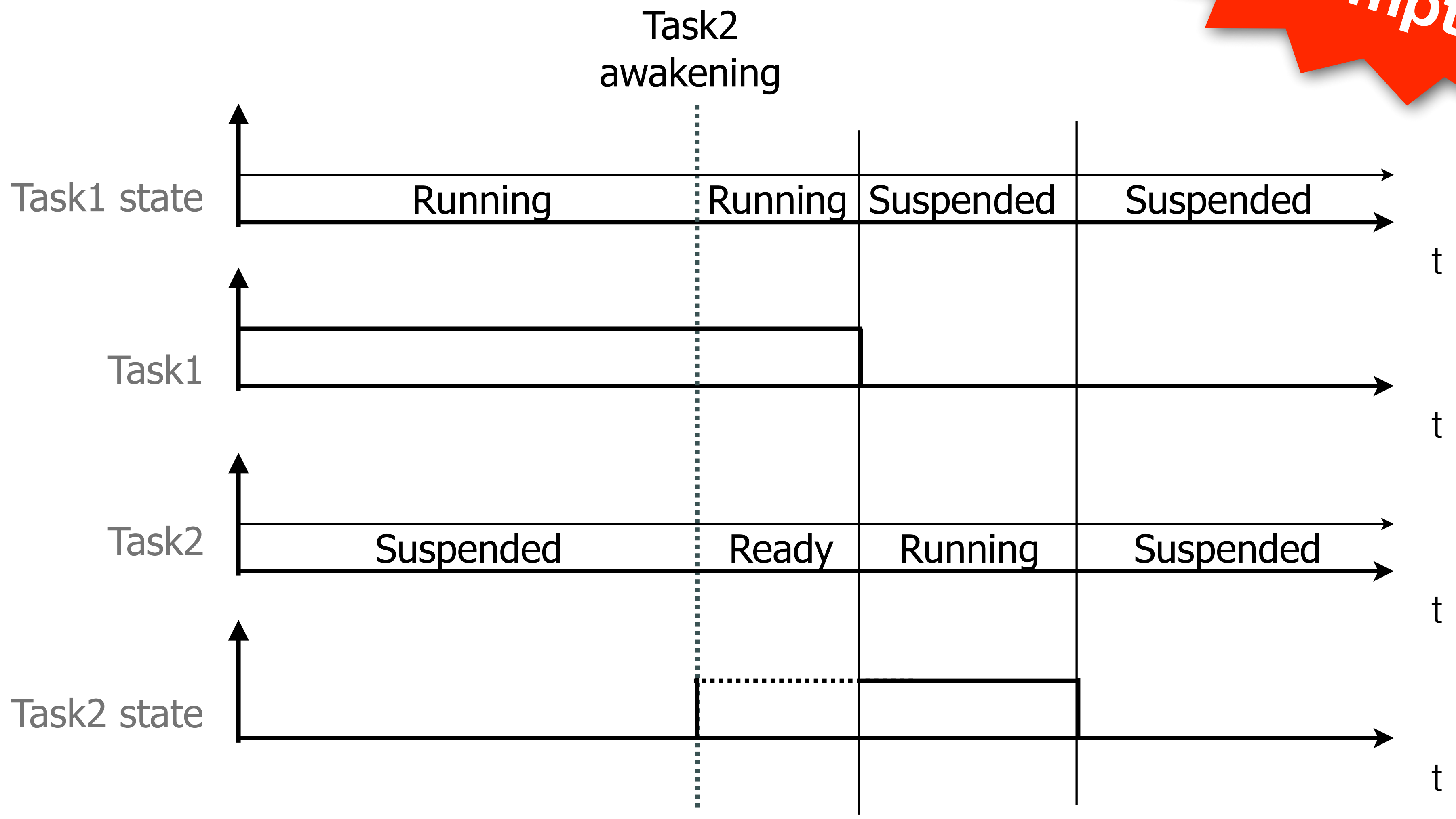
- Example: 2 tasks (Task1 and Task2).
  At start, Task1 runs. Then Task2 is activated



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

Scheduling modes

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



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).

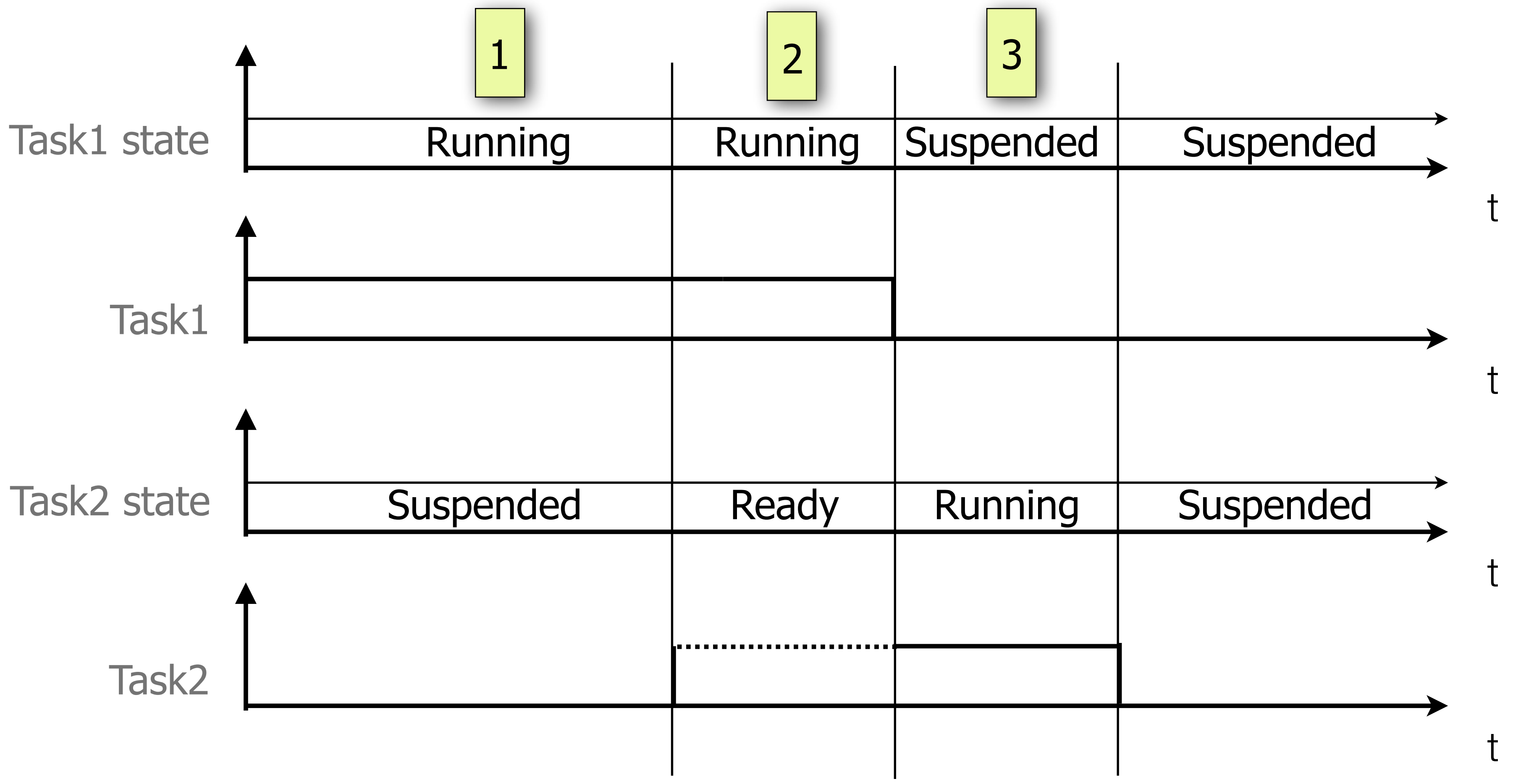
Tasks' services

- 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();
}
```



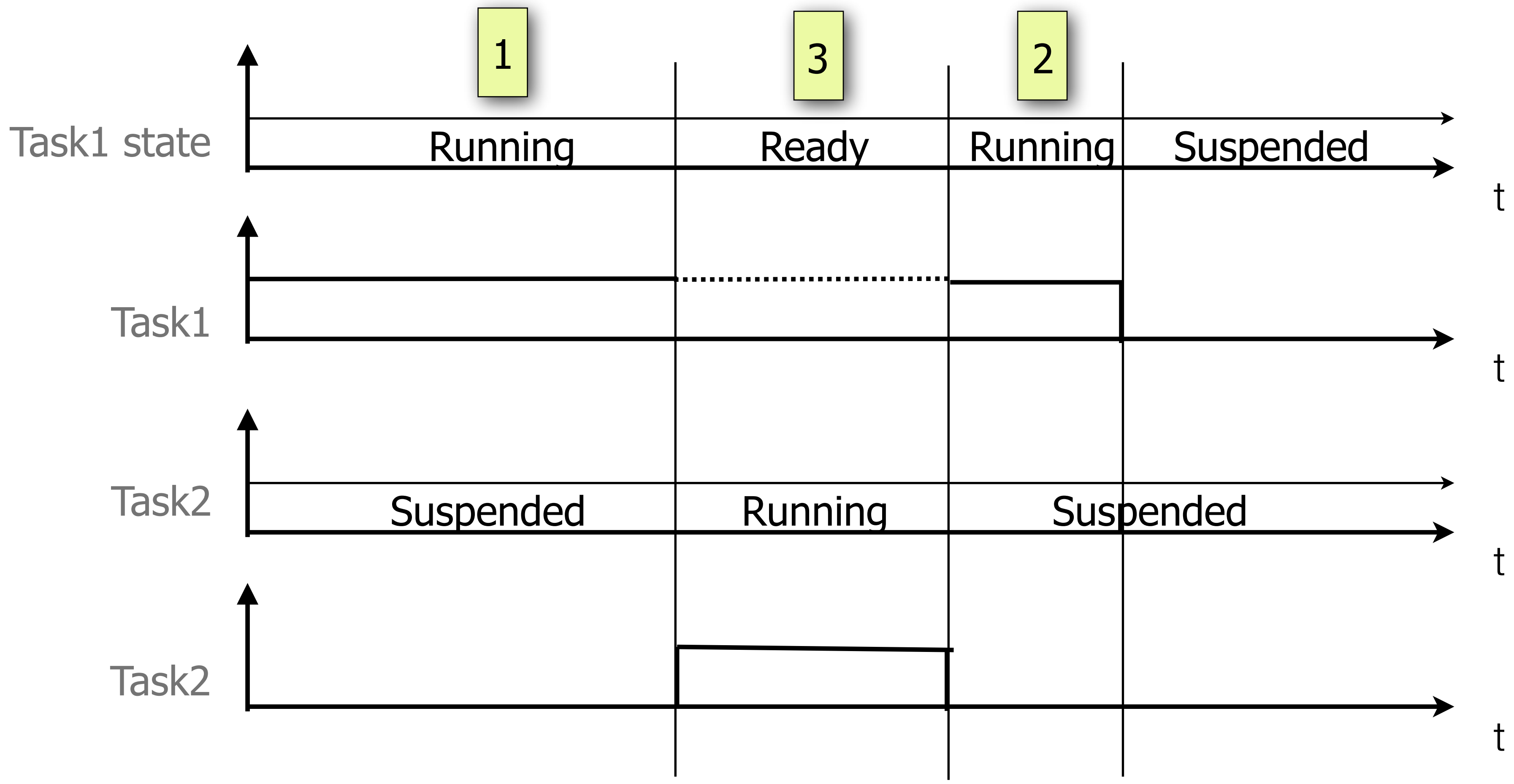
Tasks' services

- 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();
}
```



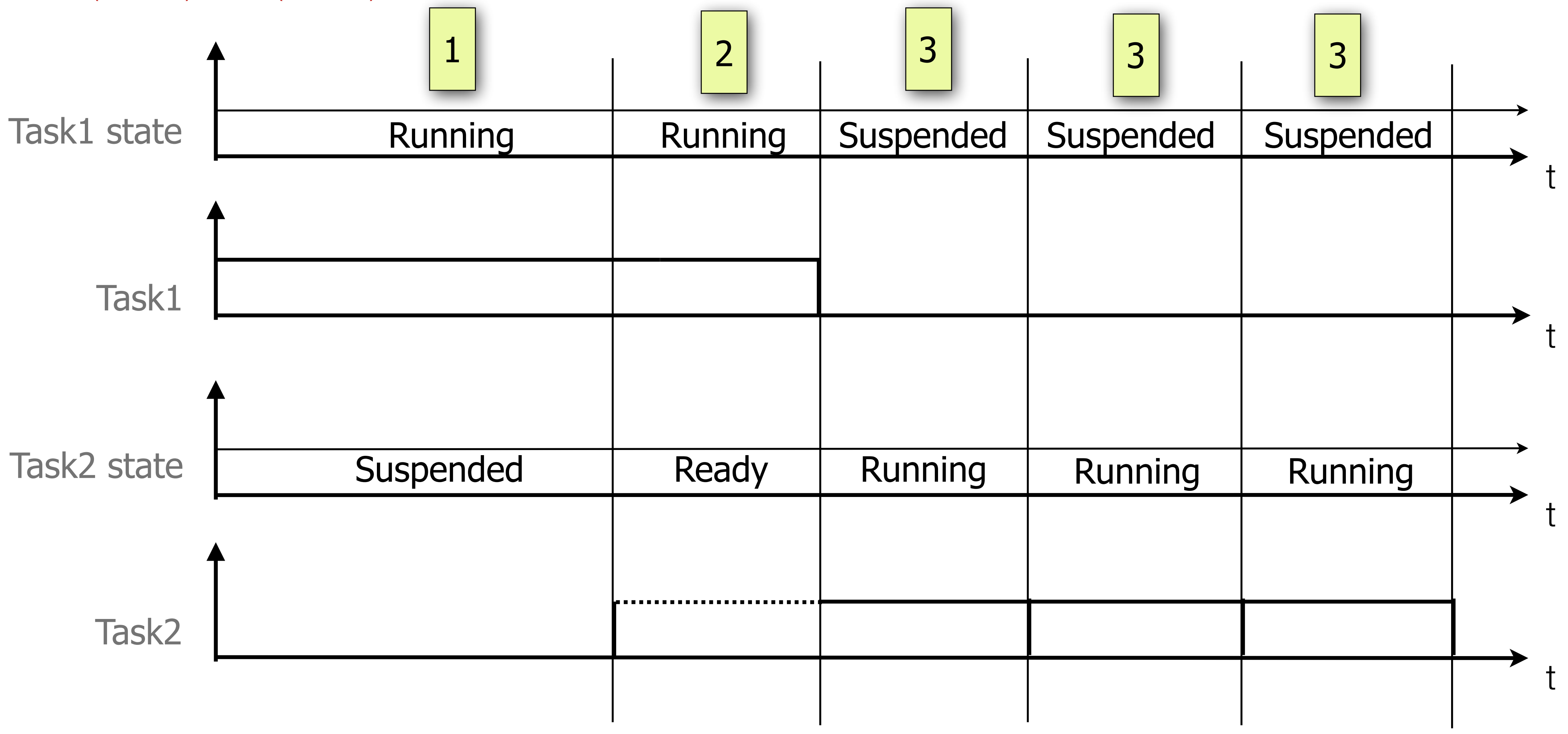
Tasks' services

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

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

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

Prio(Task1)≥Prio(Task2)



Tasks’ services

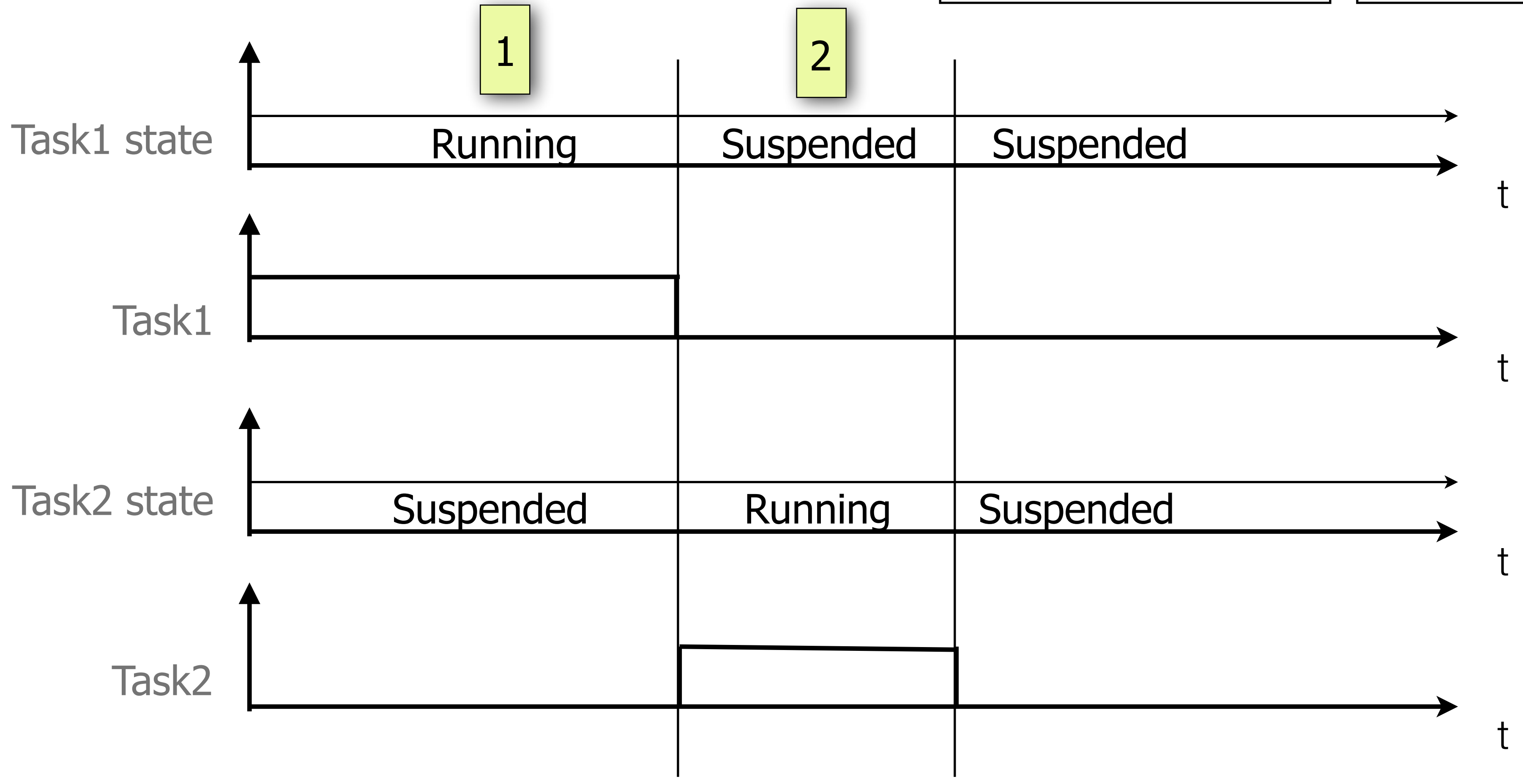
- **ChainTask** service:
  - StatusType ChainTask(TaskType <TaskId>);
  - The argument is the id of the task to activate;
  - StatusType is an error code:
    - E_OK: No error
    - E_OS_ID: invalid TaskId (no task with such an id)
    - E_OS_LIMIT: too many activations of the task
- This service puts 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)

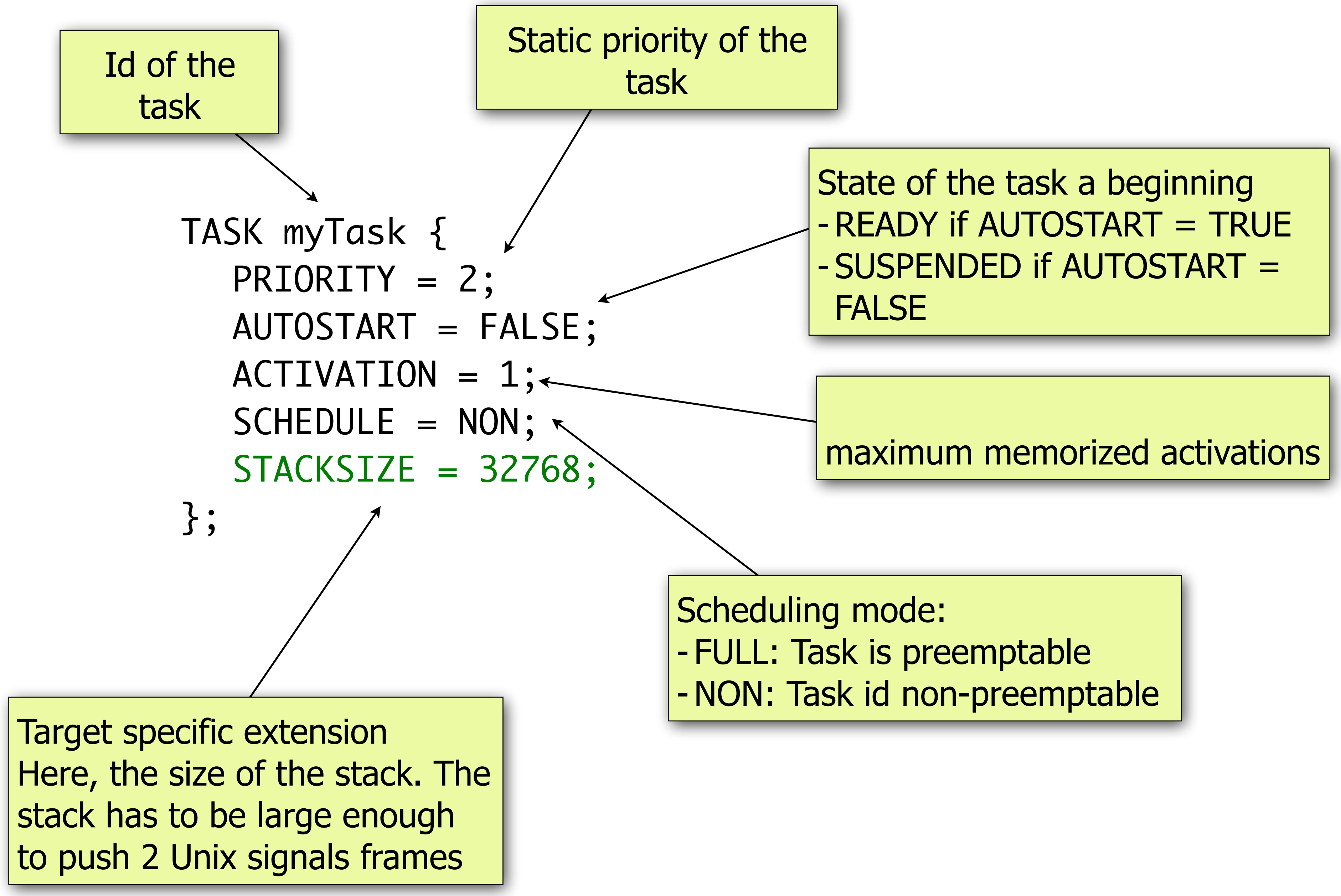
```
TASK(Task2)
{
2  …
   TerminateTask();
}
```

```
TASK(Task1)
{
1  …
   ChainTask(Task2);
}
```



Tasks' services

- OIL description of a task



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

Id of the task

Static priority of the task

State of the task a beginning
- READY if AUTOSTART = TRUE
- SUSPENDED if AUTOSTART = FALSE

maximum memorized activations

Scheduling mode:
- FULL: Task is preemptable
- NON: Task id non-preemptable

Target specific extension
Here, the size of the stack. The stack has to be large enough to push 2 Unix signals frames

Tasks' services

- OIL description of a task

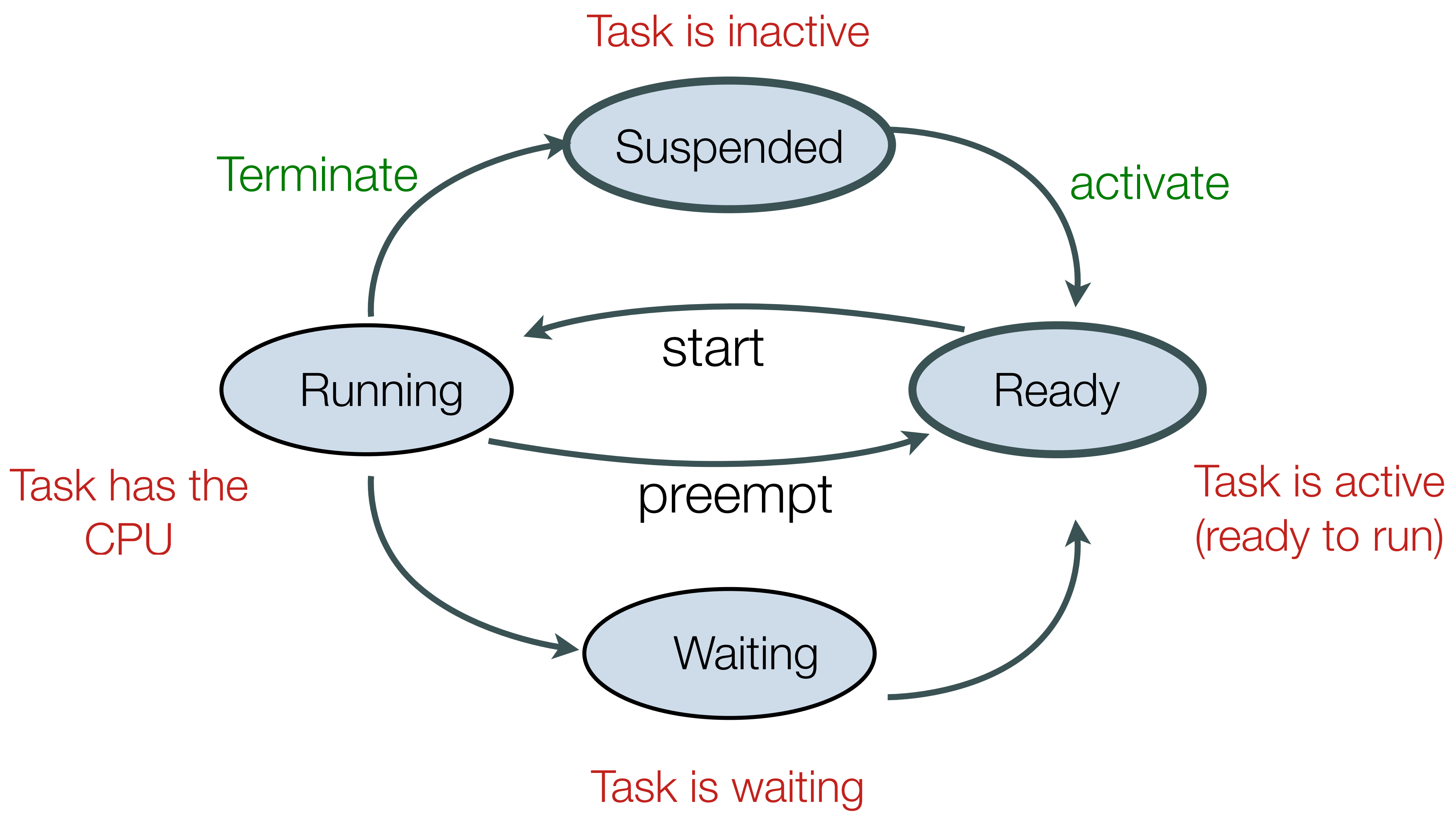
```
TASK myTask {
   PRIORITY = 2;
   AUTOSTART = TRUE {
       APPMODE = std;
   };
   ACTIVATION = 1;
   SCHEDULE = NON;
   STACKSIZE = 32768;
};
```

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

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 mabage 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 t[illegible]ariable event (A pointer to an EventMaskType is passed [illegible]e);
  - 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, my be called from a task or an ISR2.

Warning, a RefType is in fact a pointer to a Type

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

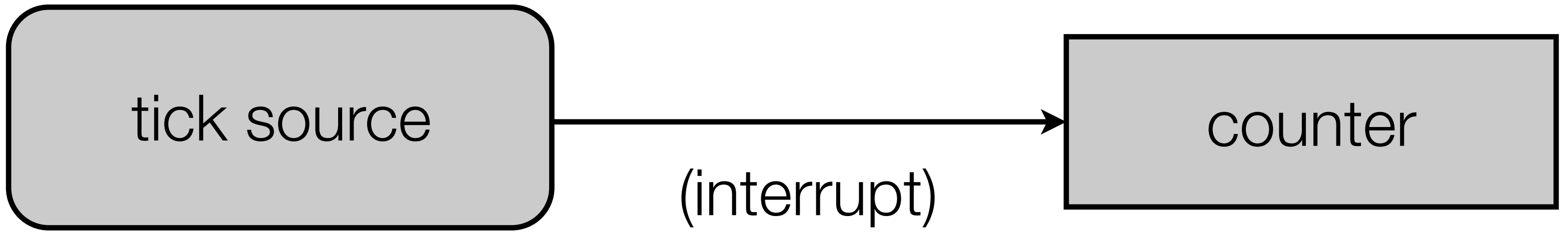
```
TASK(Task3)
{
   …
   SetEvent(myTask, ev2);
   …
   TerminateTask();
}
```

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.

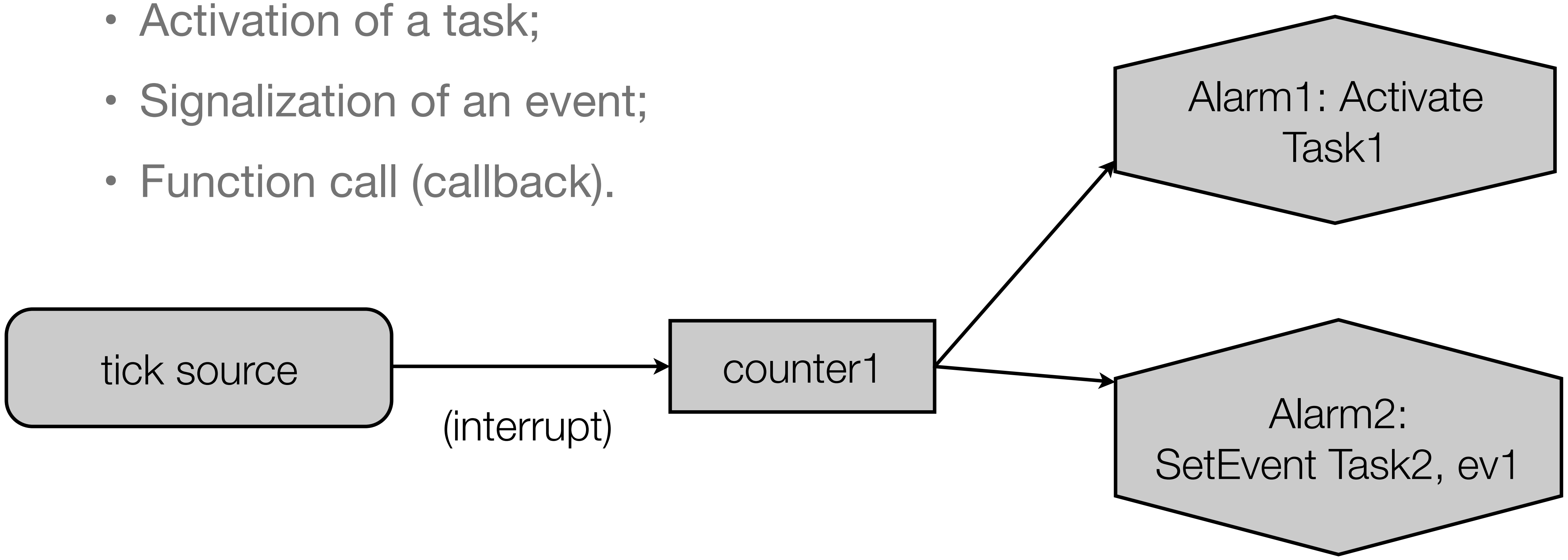
minimum interval between 2 triggering of the alarm.

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.

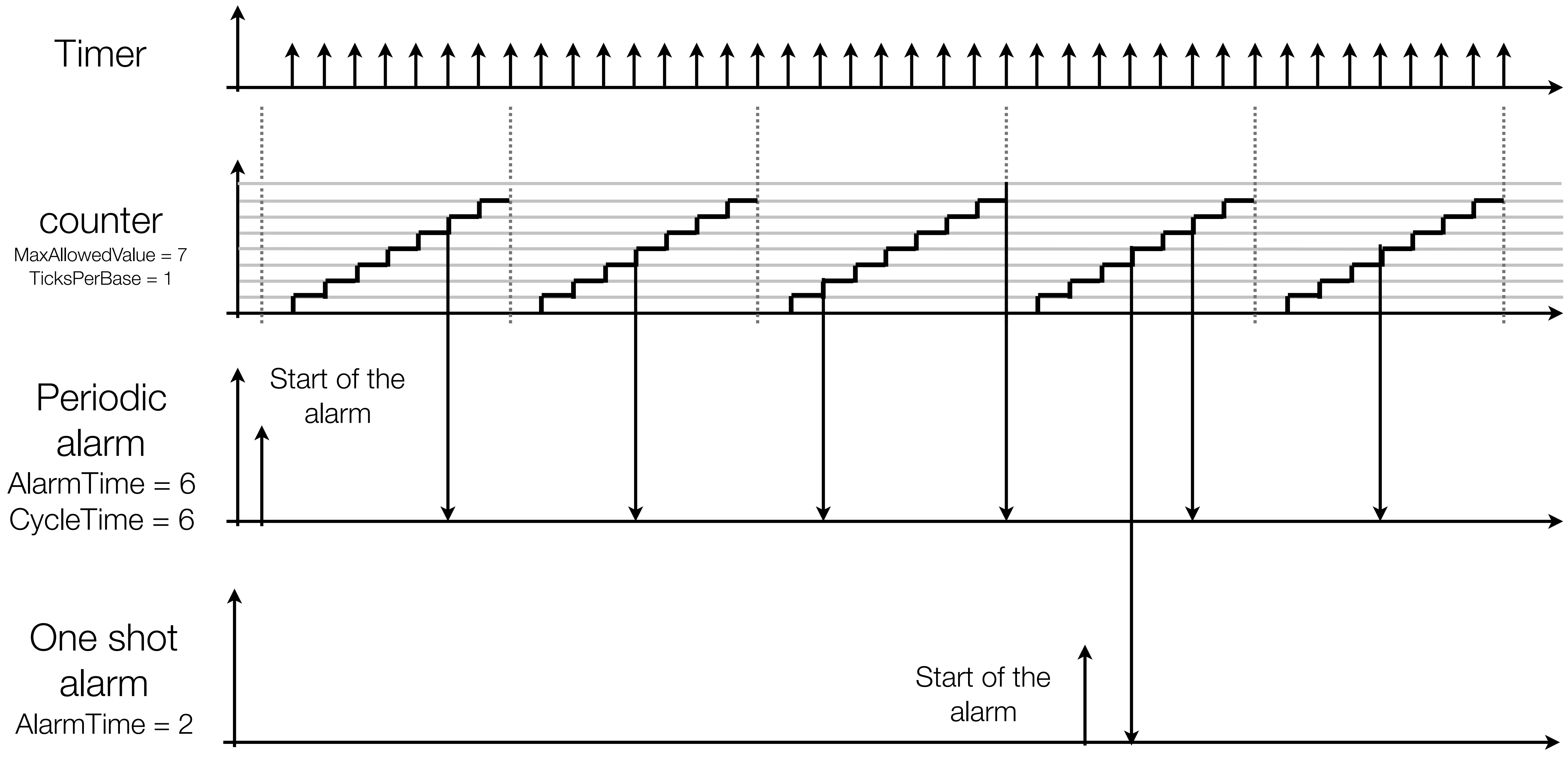
The Alarms

- 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 < 0 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 undelying 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

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.

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. <u>So its value in memory should always be up to date</u>
- 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

- 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

- OIL Description of a resource

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

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

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

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.