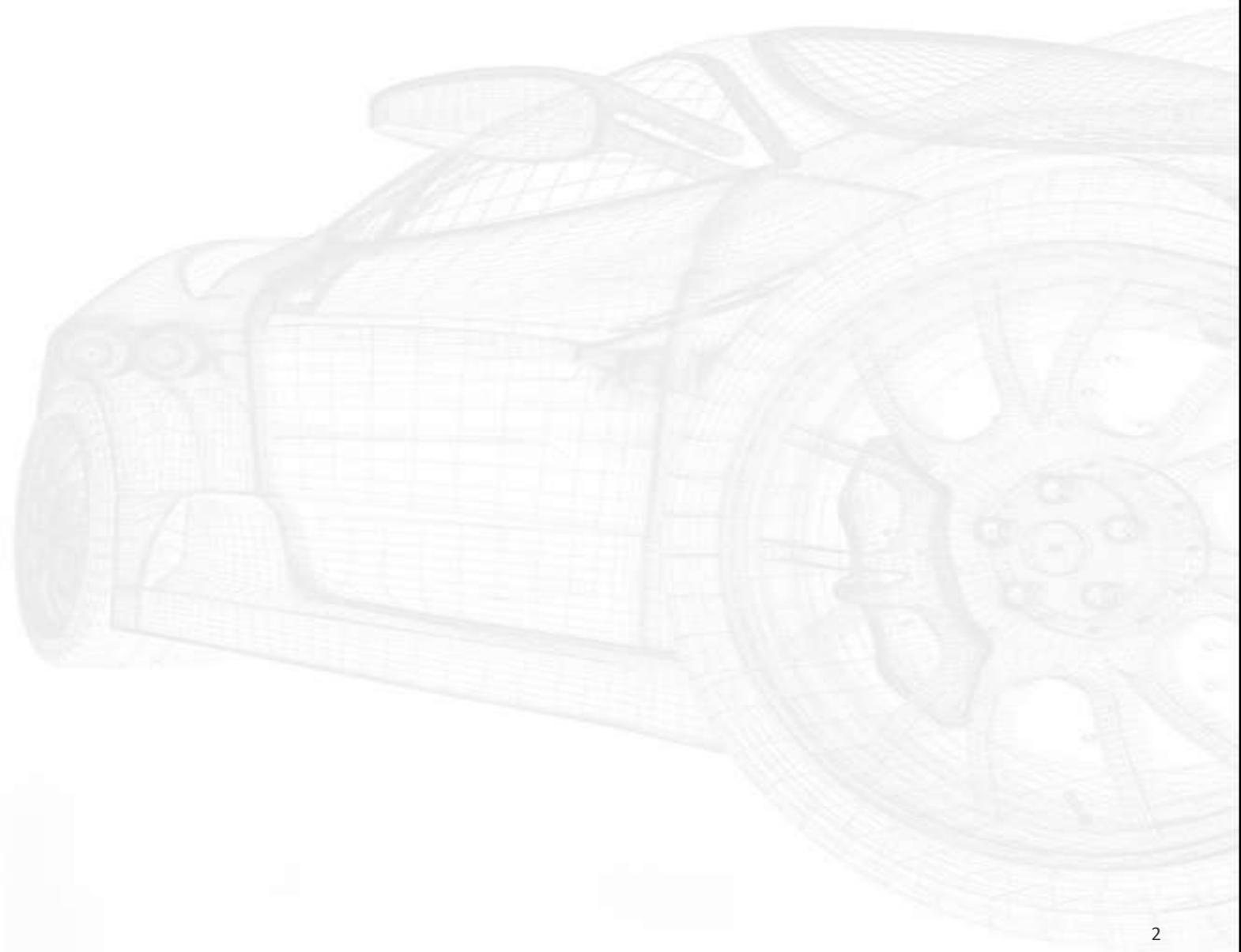
EB tresos classic AUTOSAR training - Operating system

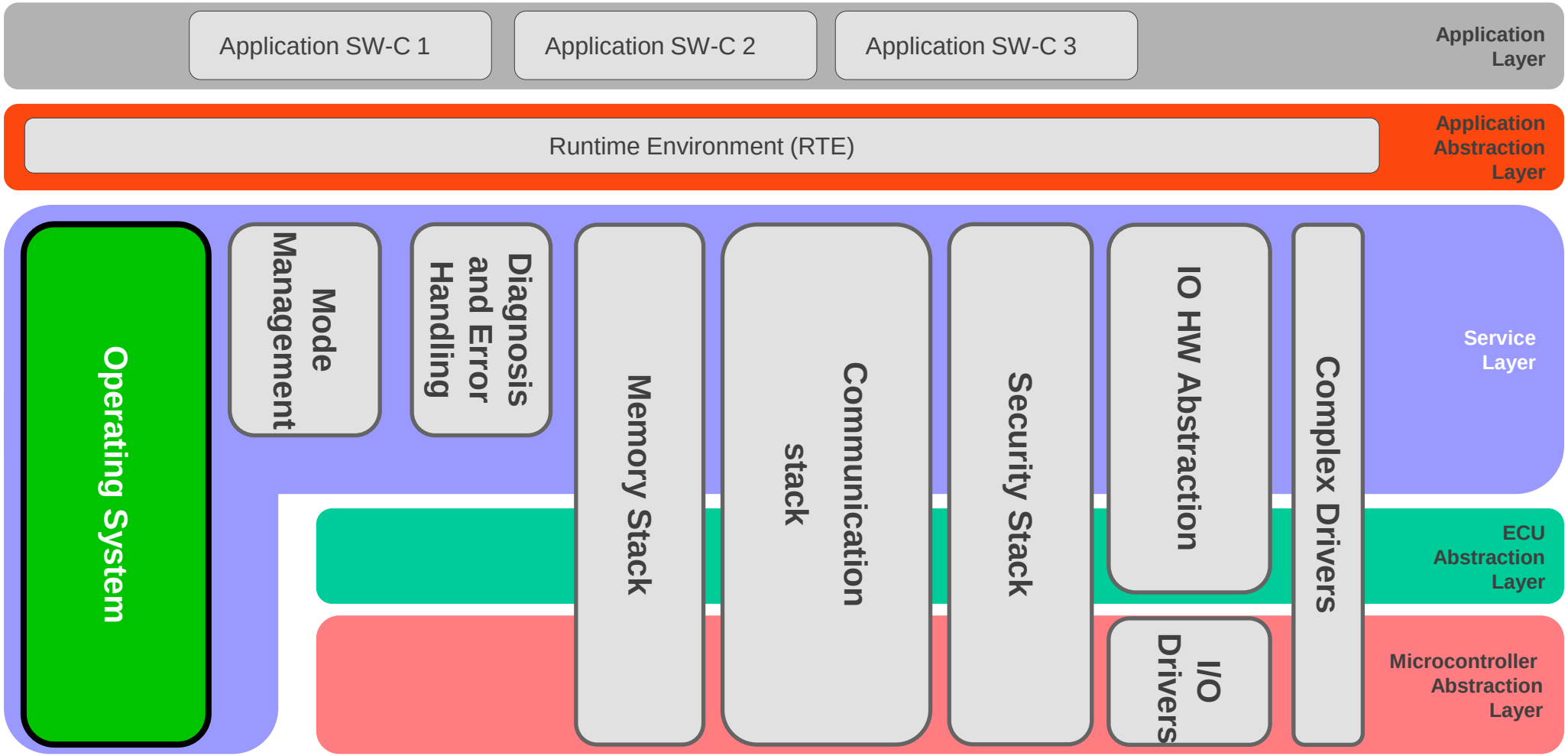
Chapter overview



- Introduction
- Scheduling
- Multi-core Support
- Critical Section Protection
- OS-Application Protection
- Hints and Tips
- EB tresos OS products

Introduction

Stack overview



Main characteristics

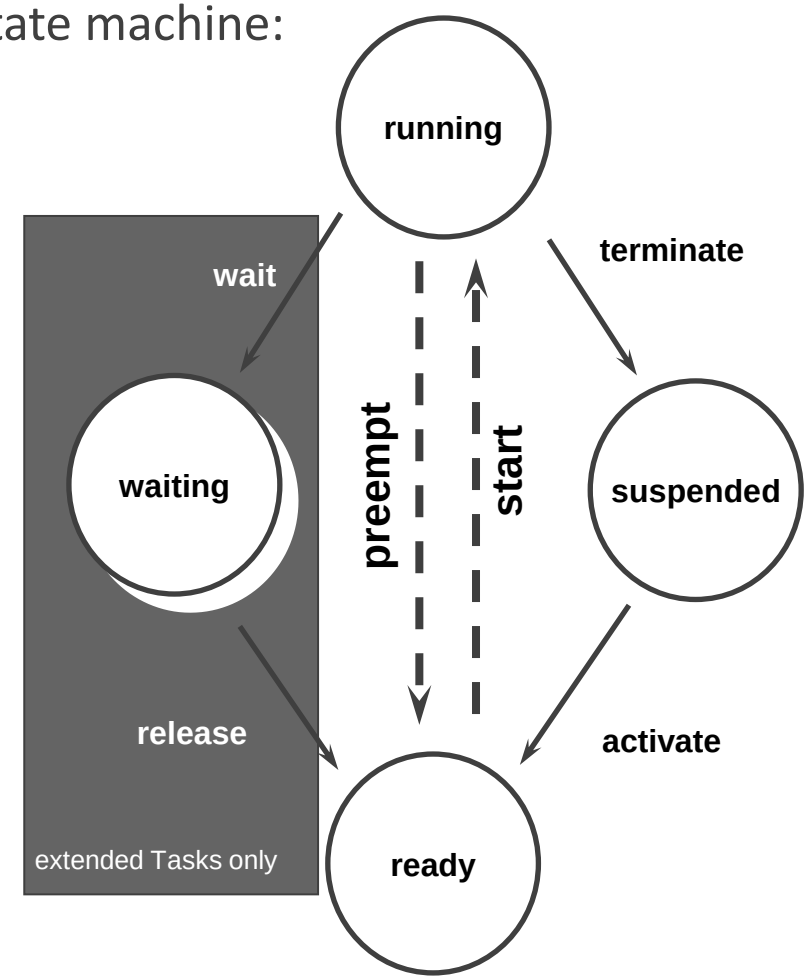
- AUTOSAR Os is a static operating system
  - No dynamic handling of system resources
  - AUTOSAR Os is configurable
  - AUTOSAR Os has to be individually configured and generated for each application
- Advantage: AUTOSAR Os can be optimized
- Configuration: EB tresos Studio

Scheduling

Tasks

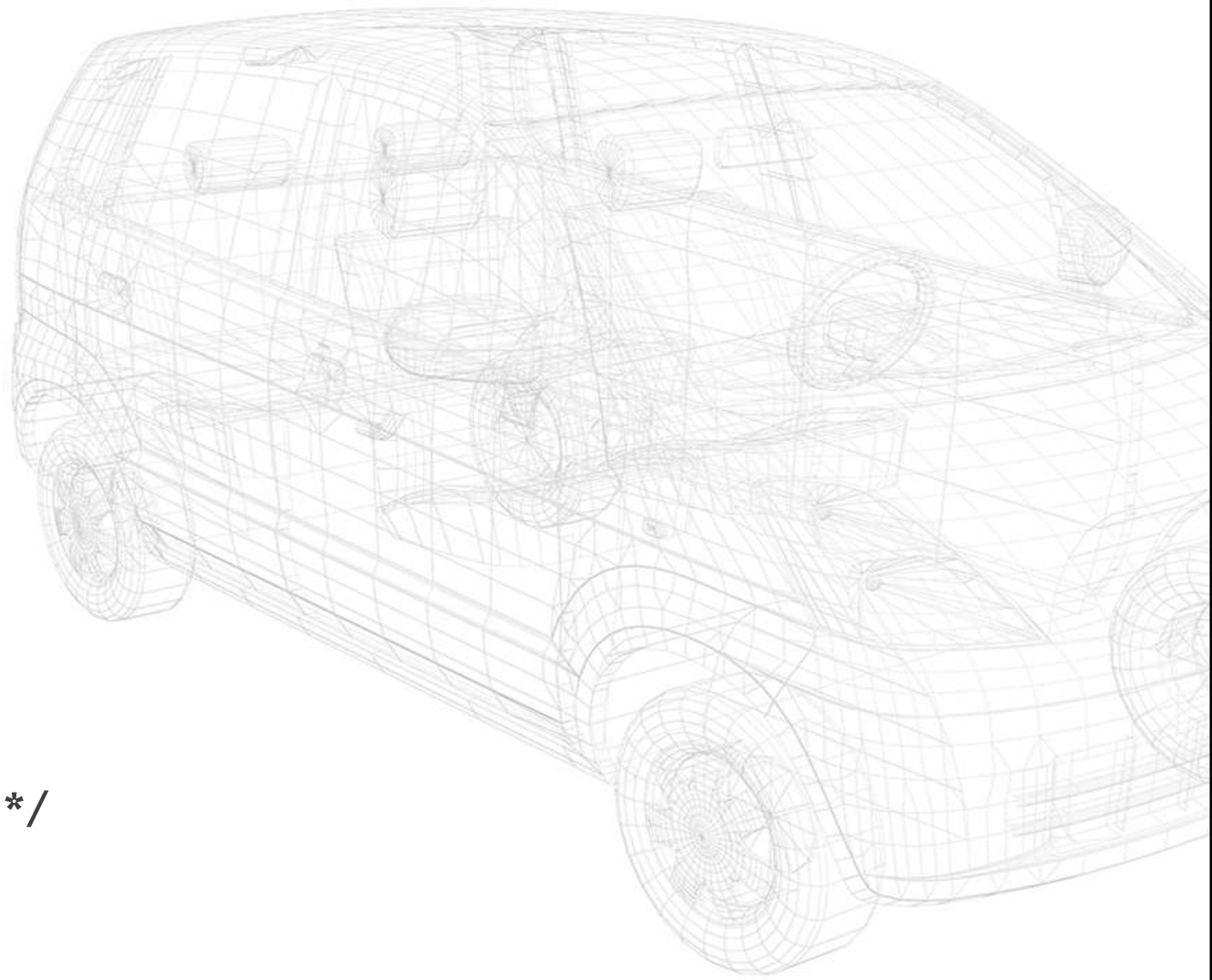
- Tasks are C functions that can be scheduled by the Os
- Typically, most parts of the application are executed within a Task
- Tasks must be activated to take part in the scheduling
  This can be done via API calls or other Os mechanisms
- Tasks have a configured priority
- There are **basic** and **extended** Tasks
  Difference: Extended Tasks support waiting for Events.

Task state machine:



Tasks: Example

```
DeclareTask(TaskA);

TASK(TaskA)
{
    StatusType status;
    TaskStateType stateB;

    /* do stuff */
    status = ActivateTask(TaskB);
    /* check return value */
    status = GetTaskState(TaskB, &stateB);
    /* check return value and do more stuff */

    TerminateTask();
}
```

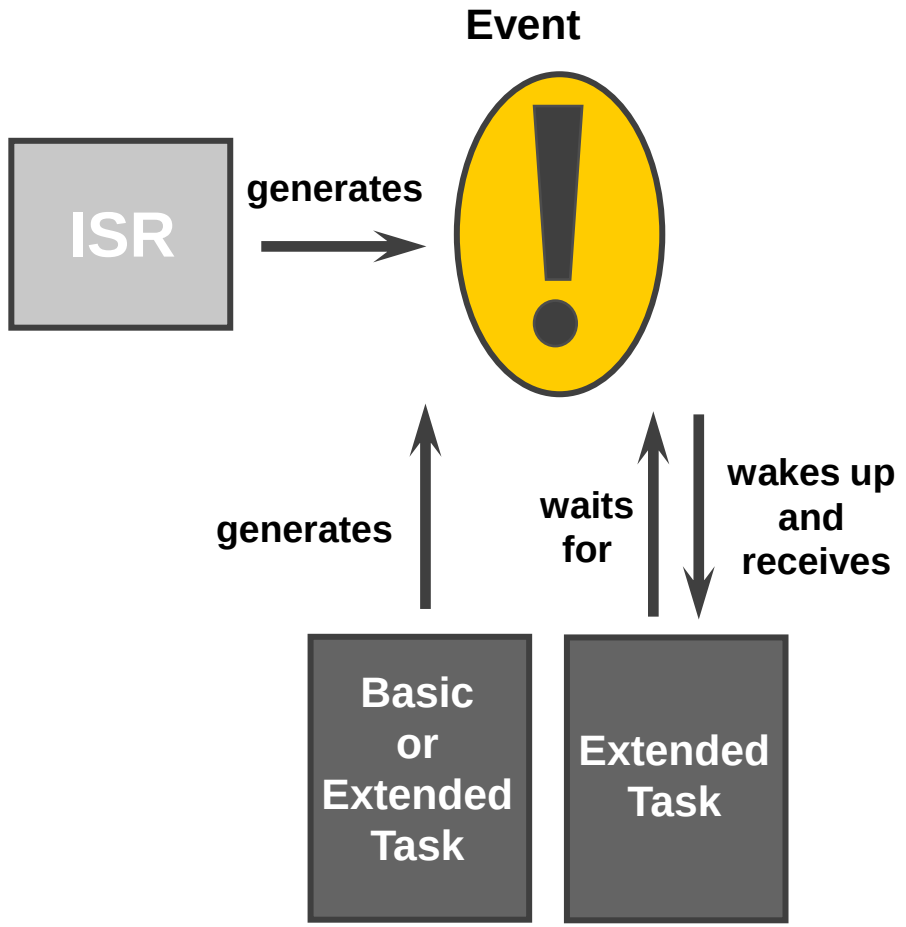


Interlude: Events

Events are a notification mechanism to notify extended Tasks:

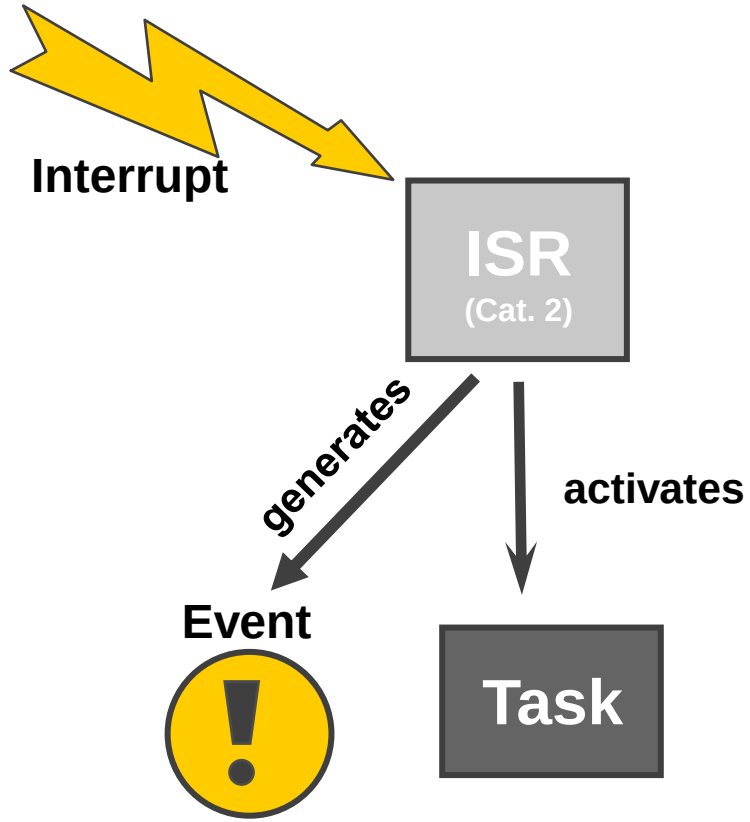
- Extended Tasks can wait for (one or several) Events. While waiting, the processor can execute other Tasks, …
- Events can be set from Tasks (basic or extended), ISRs or by the OS

Events are just notifications. They neither contain any data nor any additional information like the source of the event



Interrupt Service Routines (ISRs)

- ISRs are C functions to handle hardware events. They are typically not scheduled but directly triggered by hardware interrupt requests*
- ISRs preempt Tasks (unless the Task explicitly disables interrupts)
- Also ISR nesting is typically supported nowadays (ISRs can interrupt lower priority ISRs)
- AUTOSAR defines two ISR categories:
  - Category 1: Only few Os APIs are allowed. The ISR uses the current stack. The implementation depends on the compiler
  - Category 2: The Os provides an ISR frame which allows the access to additional Os APIs like activating tasks or setting events. The ISR has its own stack



* There are Os implementations that explicitly schedule ISRs. In this case, the ISRs have higher priorities than the Tasks.

Tasks Scheduling policy

Basic Task scheduling policy

The AUTOSAR Os schedules the Tasks according to the following properties:

- Among all Tasks that are ready to run, the one with the highest priority is taken
- If there are several Tasks with the same highest priority that are ready to run, the oldest one is taken

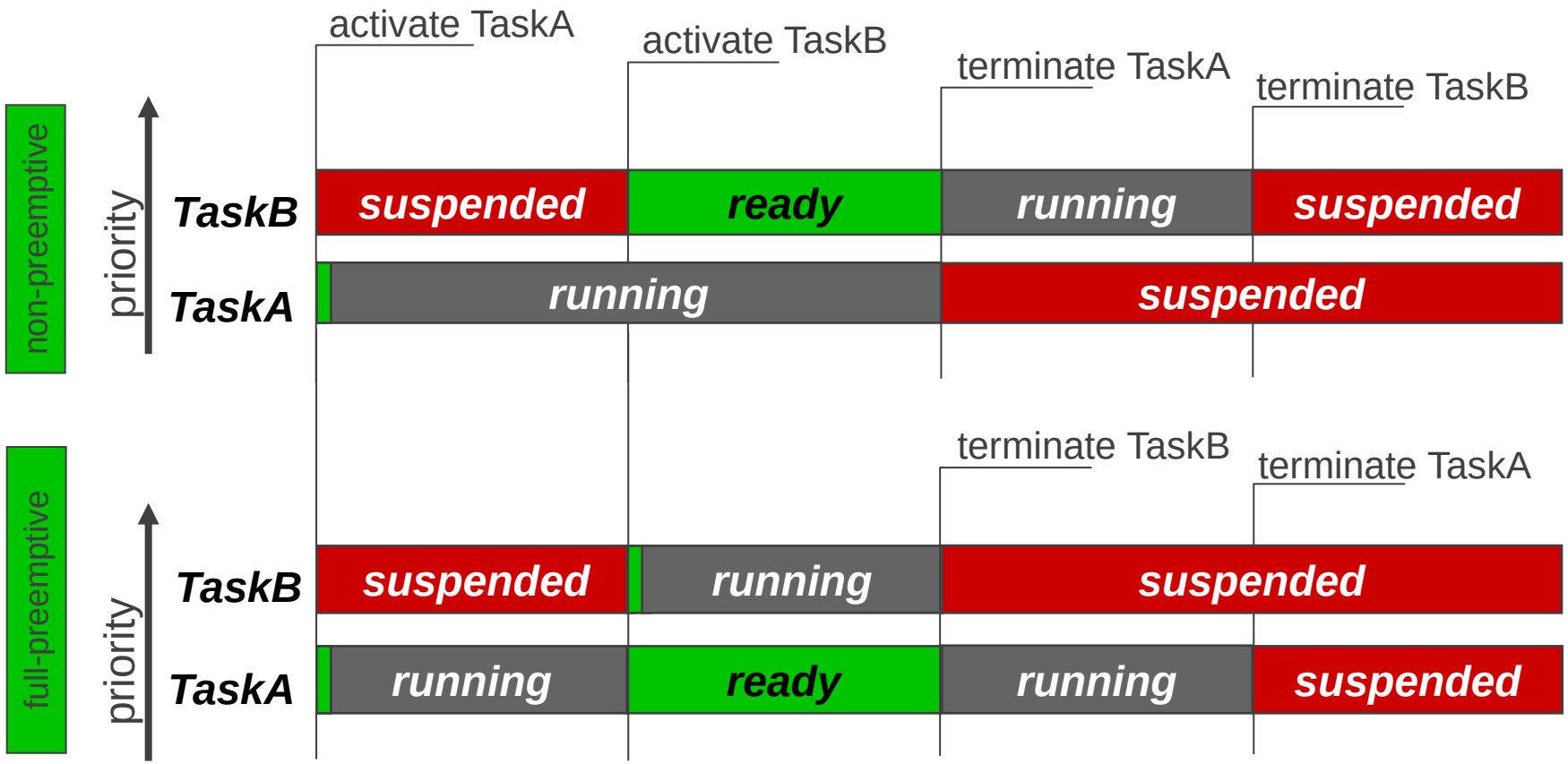
Advanced Task scheduling policies

Each Task can be configured in one of the two ways

- Non-preemptive
  - Such tasks will release the processor voluntarily
- Preemptive
  - Such tasks are preempted when a different task of higher priority is activated

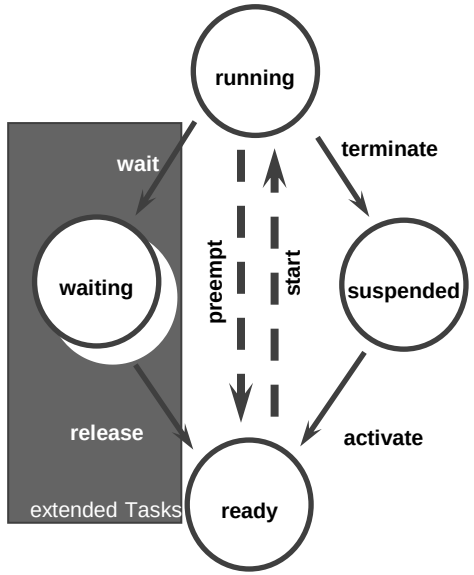
Non-preemptive Tasks can use the API Schedule() to allow Tasks with a higher priority to run

Scheduling: Example



Tasks: Example (continued)

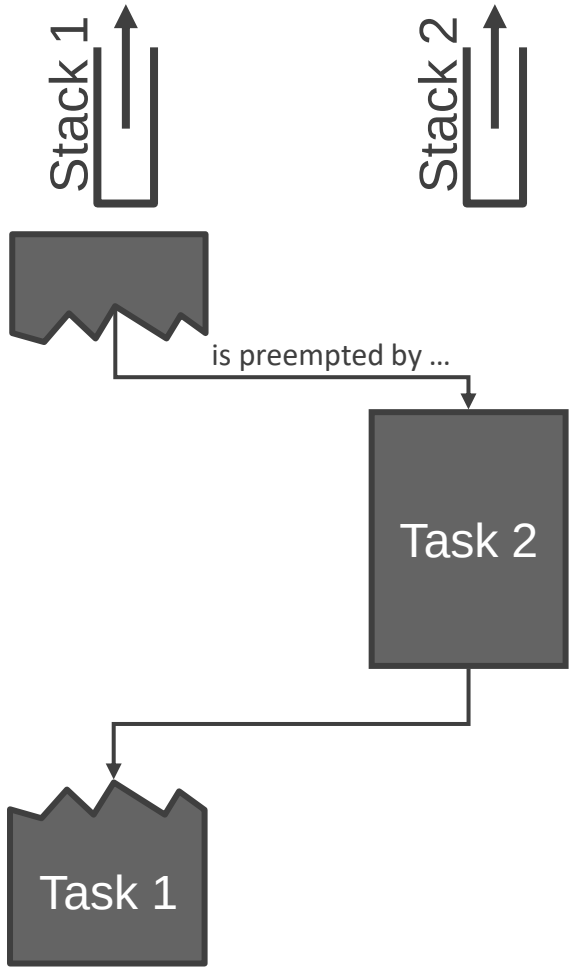
```
DeclareTask(TaskA); /* low priority */
DeclareTask(TaskB); /* high priority */

TASK(TaskA)
{
    StatusType status;
    TaskStateType stateB;

    /* do stuff */
    status = ActivateTask(TaskB);
    /* check return value */
    status = GetTaskState(TaskB, &stateB);
    /* check return value and do more stuff */

    TerminateTask();
}
```



What is the value of **`stateB`** after **`GetTaskState()`**?

Interlude: Task stacks

- Preempted tasks are resumed at the point, where they have been preempted
- Task stack: memory used e.g. for
  - local variables (compiler), or
  - to store data that shall be retained through function calls (compiler) or preemptions (Os)
- The AUTOSAR Os reserves memory for the Task stacks.
  Task stack sizes are configurable
- Possible optimization: Task stacks may be shared by several Tasks, if those Tasks can never preempt each other



Time triggered Task activation

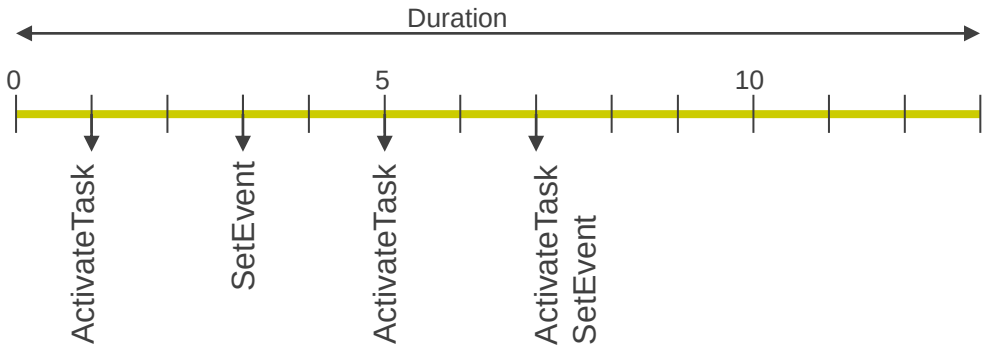
Counter: An AUTOSAR Os mechanism to count events, e.g. timer ticks

Alarm: A Mechanism to trigger an action (e.g. a task activation), when the attached Counter reaches a given value

ScheduleTable

- Predefined sequence of actions (expiry points) that are attached to a Counter
- If the Counter reaches the value for an expiry point, the corresponding action is triggered
- Several expiry point actions are possible, e.g. activating a Task or setting an Event

- ScheduleTable/Alarm modes: one-shot and periodic
- ScheduleTables can be synchronized to external sources



Summary

- Tasks are C functions, that can be scheduled concurrently by the AUTOSAR Os
- There are basic Tasks and extended Tasks. Extended Tasks support waiting for Events
- Tasks can be preemptive or non-preemptive
- Task scheduling is based on the priority. For Tasks with the same priority, the Task activation time decides which task shall be scheduled next
- ISRs are handlers for hardware events (interrupts)
- ISRs interrupt the execution of Tasks (unless interrupts are explicitly disabled within the running Task)

Multi-core Support

OS-Applications

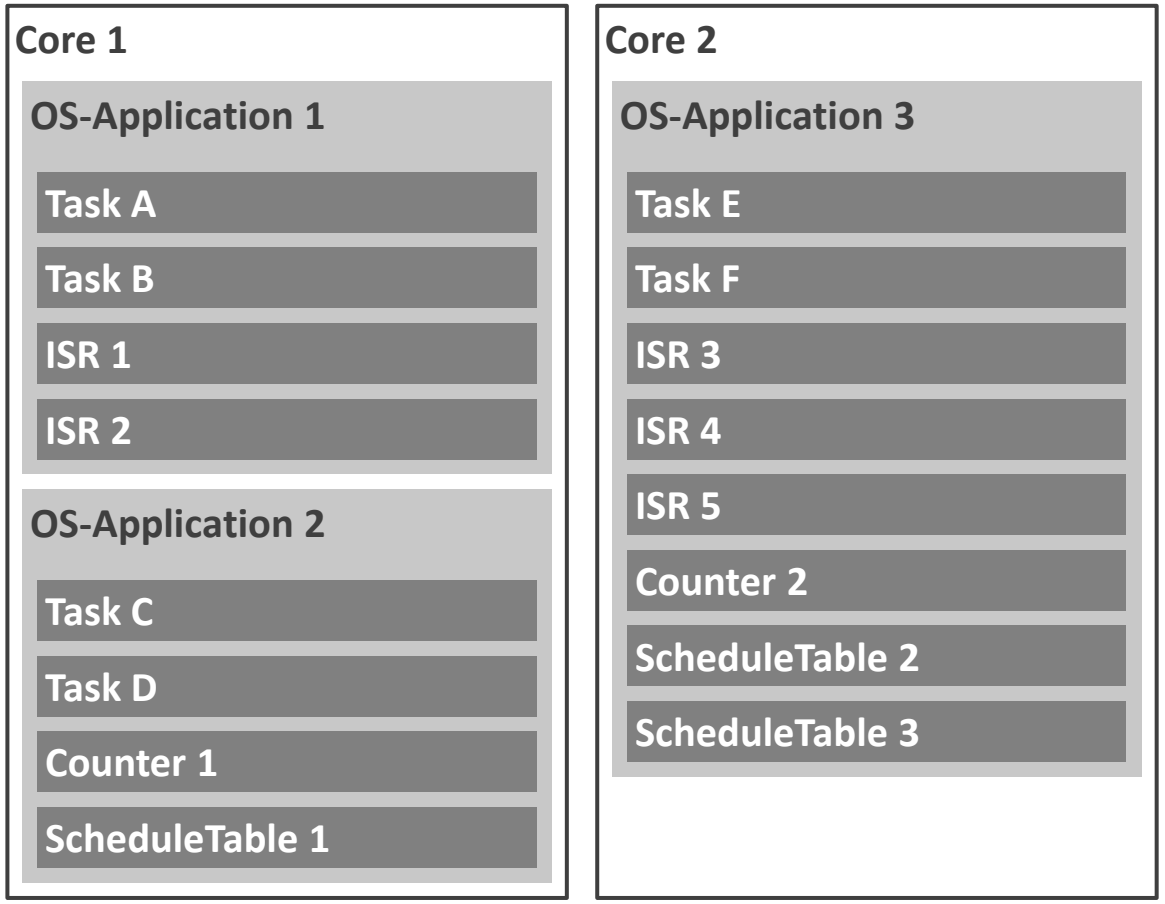
The AUTOSAR Os multi-core support is based on OS-Applications

OS-Application

- Group of AUTOSAR Os objects (Tasks, Counters, ISRs, …), that semantically belong together

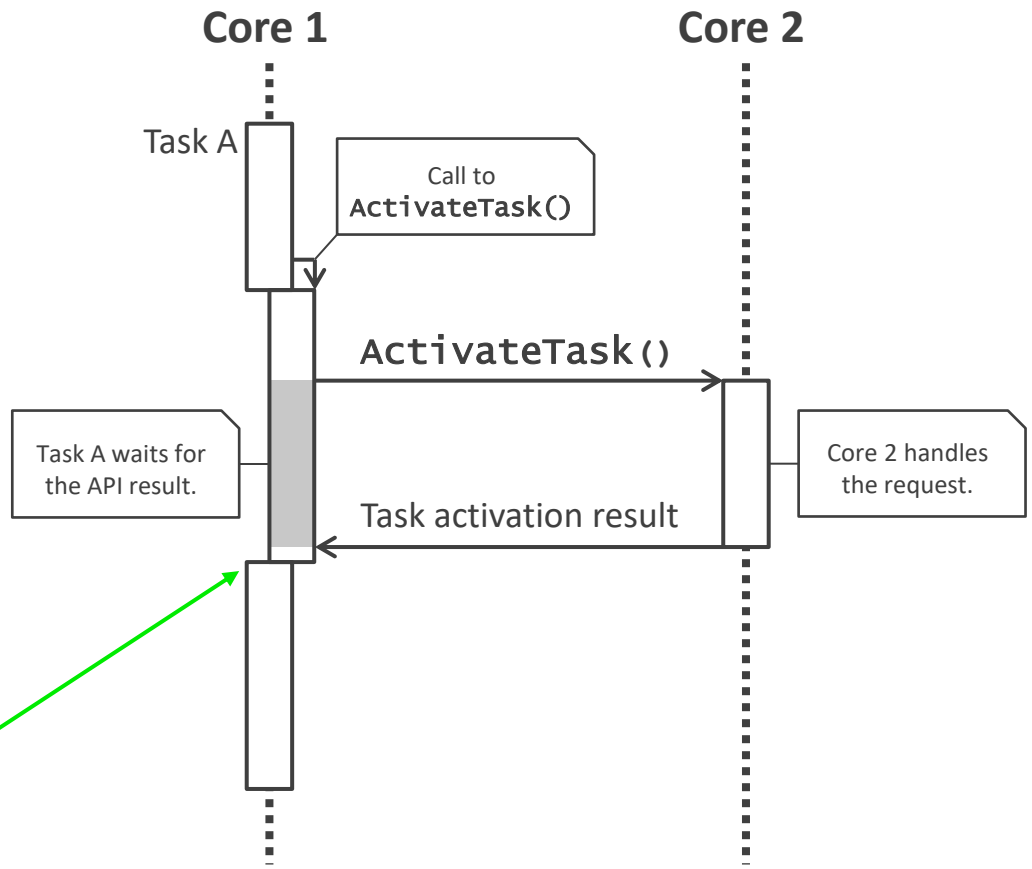
Application to Core Assignment

- Os objects are assigned to processor cores via their OS-Application
- The assignment is static: all objects of an OS-Application will always run on the core, to which the OS-Application is assigned



Cross-core Requests

- Os API calls are automatically forwarded to the correct core
- Os APIs wait for the result from the other core
- AUTOSAR 4.4. added two asynchronous Os APIs, one for activating a task and the other for setting an event
- Question: What is the state of the activated task on core 2 when Task A is continued on core 1?



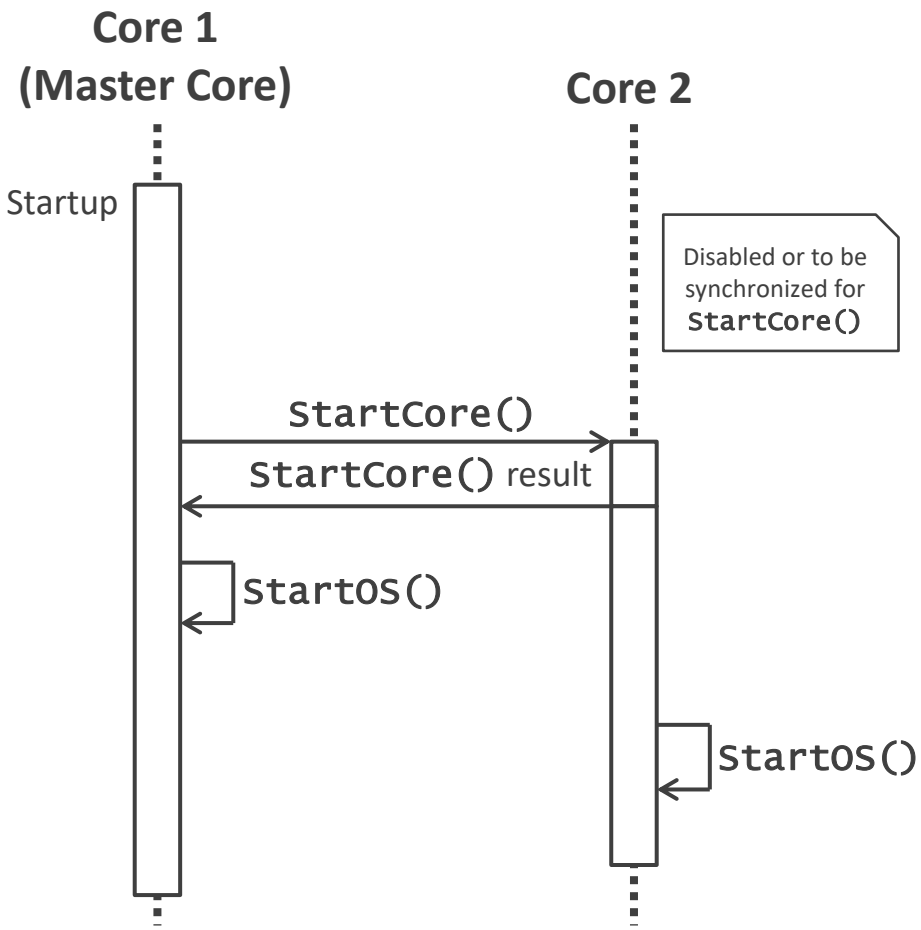
Startup

Single-core startup: **`StartOS()`**

Multi-core startup (according to AUTOSAR)

- The *Master Core* is the only core that is started automatically
- For other cores, **`StartCore()`** must be called.
- Call **`StartOS()`** on each core (including the *Master Core*), after the core was started via **`StartCore()`**

Consult the manual of your AUTOSAR Os for deviations and additional information.



Summary

- AUTOSAR Os objects are assigned to the different processor cores via OS-Applications
- The core assignment is static
- Requests are forwarded to the correct core
- APIs will wait for the result from the other core
- Multi-core startup is done as a master-slave system

Critical Section Protection

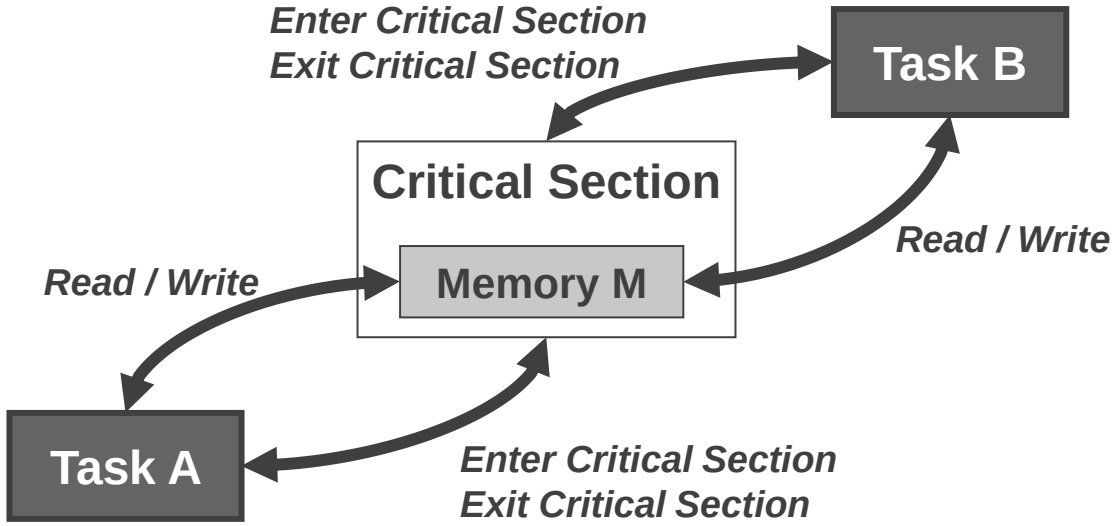
Overview

Critical Section Protection

Mechanisms to prevent concurrent access to shared resources (e.g. memory or peripherals)

The AUTOSAR Os provides three protection mechanisms:

- Resources
- Interrupt Locks (several variants)
- Spinlocks



Elektrobit

Single-core Protection Mechanisms

Resources

- Os mechanism to protect critical sections on a single core
- Critical sections can be protected against preemption by other Tasks and (category 2) ISRs
- Resources must be configured for each Task and (category 2) ISR that uses the shared resource

Interrupt Locks

- APIs to lock interrupts
- Different variants are available:
  - Category 2 ISRs only vs. all ISRs[1]
  - Nestable vs. non-nestable APIs
- Only affects the processor core which executes the APIs

Interrupt locks and Resources ensure that no other Task/ISR that uses the same shared resource is scheduled while a critical section is executed.[2]

[1] There might be some ISRs which are not locked. Check the documentation
[2] If an appropriate mechanism is chosen and configured and used correctly.

Multi-core Protection Mechanism

Spinlocks

- The only AUTOSAR Os multi-core protection mechanism
- Active waiting in a loop („spin“), while repeatedly checking if the lock is available
- No protection against core local interruptions

NOTE

- Combine Spinlocks with an Interrupt Lock or RES_SCHEDULER[1] to protect a section against interruptions on the local core[2]
- Spinlocks can easily result in deadlocks when being used incorrectly. Make sure to use them decently!

```
GetSpinlock(s)
{
    do {
        try to acquire spinlock s;
    } while (spinlock s was occupied);
}

ReleaseSpinlock(s)
{
    release spinlock s;
}
```

[1] RES_SCHEDULER is a Resource which blocks all Task scheduling.
[2] Directly supported in the configuration since AUTOSAR 4.1.

Summary

Mechanism	Advantages	Disadvantages
Resources	• No interrupt locking as long as no ISRs are involved • Fine grained control via the configuration	• Single-core only • Typically slower than Interrupt Locks[1] • No protection against category 1 ISRs • Misconfiguration possible
Interrupt Locks	• Typically faster than Resources[1] • Simple to understand and use	• Single-core only • Lock all interrupts of the corresponding type[2]
Spinlocks	• Multi-core protection mechanism • Efficient for small critical sections • Can be combined with an Interrupt Lock or RES_SCHEDULER[3]	• No protection against core-local interruption (unless combined with other locks) • Active waiting, therefore inefficient in case of long critical sections • Deadlocks easily possible

[1] Depends on the actual implementation.
[2] There might be exceptions.
[3] Since AUTOSAR 4.1, if supported.

OS-Application Protection

Trusted and Non-Trusted OS-Applications

Processors typically provide different modes that control the access to certain resources, e.g. to critical special purpose registers, certain assembler instructions or some peripheral registers.

Trusted OS-Application

Os objects of a trusted OS-Application may be executed in a privileged processor mode[1] and may have unrestricted[2] access to hardware resources as well as APIs.

Non-Trusted OS-Application

Os objects of a non-trusted OS-Application should be executed in a non-privileged processor mode[1] and have restricted access to hardware resources as well as APIs.

Trusted Function

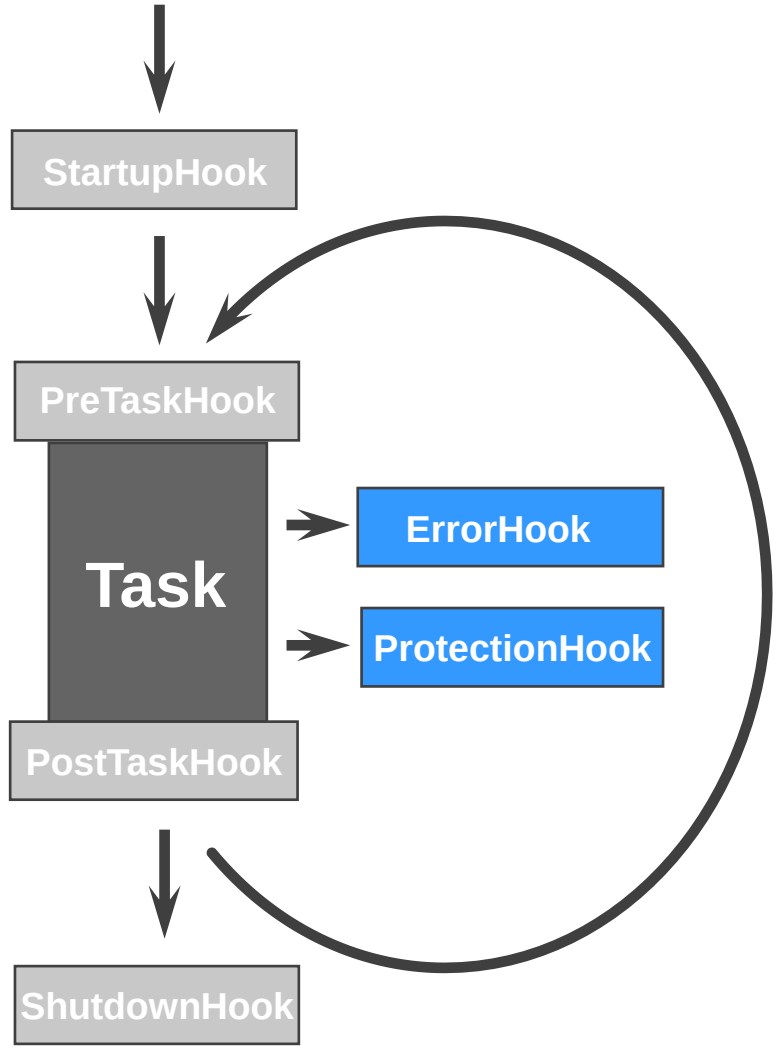
- Trusted Functions are provided by Trusted OS-Applications and executed in the privileged processor mode[1]
- Trusted Functions can be provided to Non-Trusted OS-Applications, to allow them access to restricted resources
- Trusted Functions cannot be provided to OS-Applications on different processor cores

[1] The used processor modes as well as the associated restrictions are hardware- and implementation-specific.
[2] Since AUTOSAR 4.2, memory protection can be activated for Trusted OS-Applications.

Interlude: Hooks

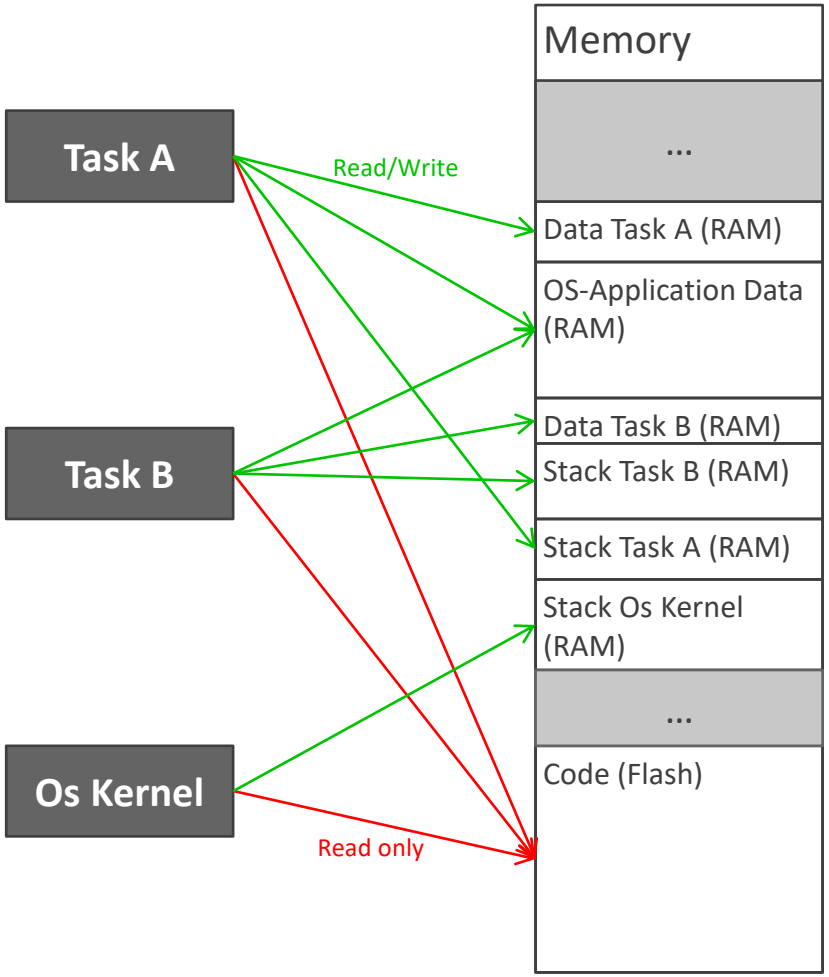
- Hooks are callback functions, which are called by the AUTOSAR Os in certain situations
- Hooks must be enabled in the configuration
- **`ErrorHook`**: Typically* called for errors, after which the Task/ISR can continue its execution
  Example: API called with invalid parameters
- **`ProtectionHook`**: Typically* called for errors, which must be handled before the Task/ISR can continue.
  Example: Memory protection exception

* Consult the AUTOSAR specification for details.



Memory Protection

- Prevents Tasks, ISRs, Hooks as well as the Os itself from accessing memory, that should not be accessible
- Hardware support (MPU or MMU) is necessary
- The **`ProtectionHook`** is called in case of a violation
- The configuration is hardware and implementation specific and tightly coupled with the linker script. Please consult the documentation for your Os on how to use memory protection correctly.



Further Protection Mechanisms

Service Protection: Accessing Applications

- For Os objects like Tasks, Resources, Counters, etc. the „accessing Applications“ are configured
- If such an object shall be used by an OS-Application, which is not configured as „accessing Application“ for that object, the access is prevented and the **`ErrorHook`*** is called

Service Protection: Invalid API parameters or calling contexts

- Invalid request is rejected, and an error is reported to the **`ErrorHook`***

Timing Protection

- Task and ISR time budget monitoring, as well as some other variants
- The **`ProtectionHook`*** is called in case of such a violation

* If enabled in the configuration.

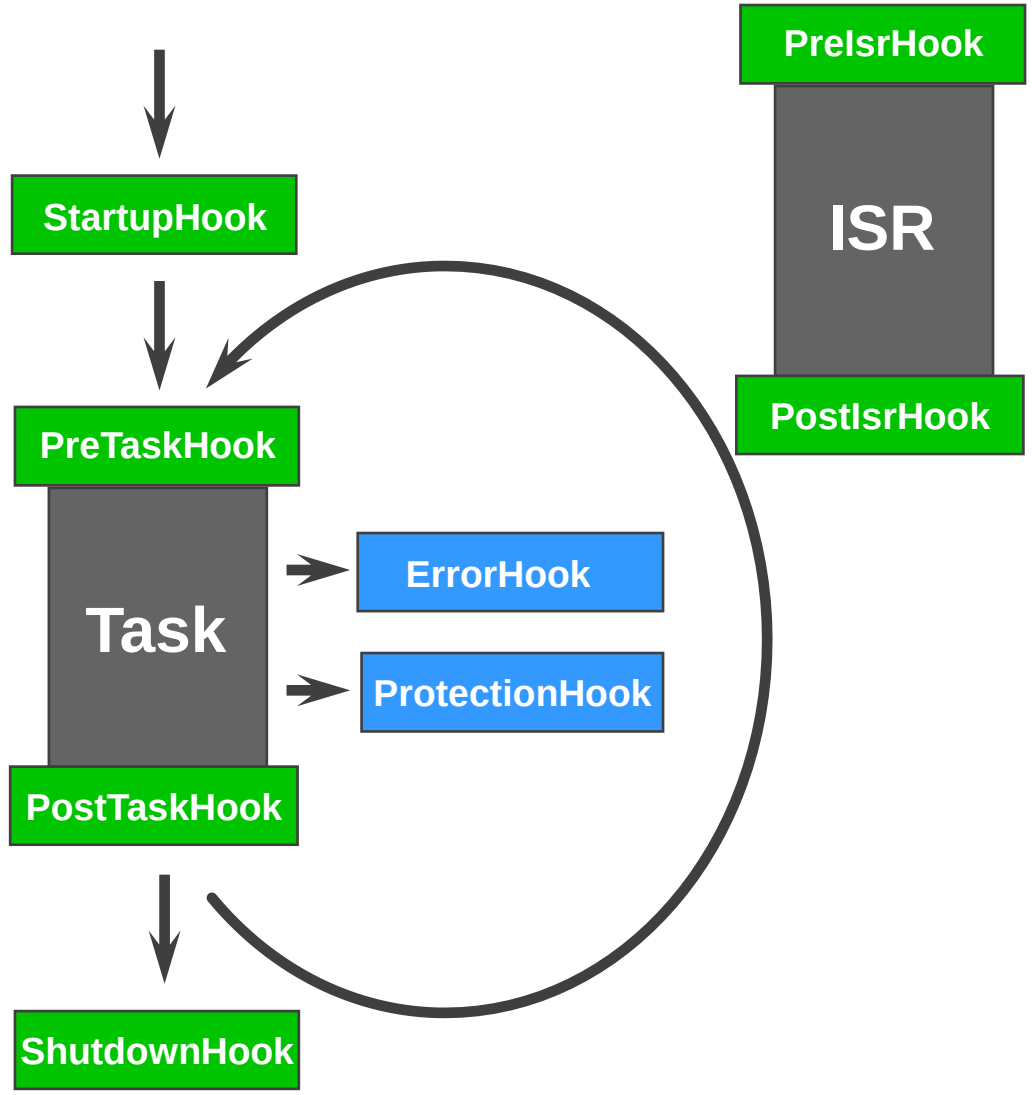
Summary

- Non-Trusted OS-Applications have restricted access rights. Use them whenever possible
- Memory protection prevents access to memory regions without proper permissions
- The **`ErrorHook`** and the **`ProtectionHook`** are your friends - enable them
- The Os provides further protection mechanisms with the service- and the timing protection

Hints and Tips

Hooks

- Hooks are callback functions, which are called by the AUTOSAR Os in certain situations
- If Hooks are enabled in the configuration, they must be defined by the user with the predefined names and prototypes
- Inside Hooks, not all Os API functions are available
- There are global as well as OS-Application specific Hooks. Consult your Os documentation for details and possible deviations from AUTOSAR.



Debugging

General Tips

- Enable extended error checks and check API return values
- Enable the **`ErrorHook`** and check that it is never called on your ECU
- Enable the **`ProtectionHook`** and implement it decently
- If „strange“ things happen, make sure that this is not caused by too small stack sizes

Additional Help

- The AUTOSAR Os supports the generation of an ORTI file. If your debugger supports ORTI, use that file to get easier access to Os internal information like task control blocks or stacks
- Check your documentation for extended debugging support like e.g. stack checking support

EB tresos OS Products

EB tresos classic AUTOSAR OS products

- EB tresos AutoCore OS
- EB tresos Safety OS

Both the above EB tresos OS products are

- AUTOSAR compatible
- Available for many Hardware derivatives (eg. Tricore, RH850, PA and ARM32 and ARM64) in single-core and multicore versions.

More info: https://www.elektrobit.com/products/ecu/eb-tresos/operating-systems/

EB tresos AutoCore OS

- EB tresos AutoCore OS is an embedded real-time operating system that is AUTOSAR (v4.0.3 or R19.11) compatible
- EB tresos AutoCore OS was developed for QM projects
- EB provides a safety application guide (SAG) with which EB tresos AutoCore OS can be used in safety projects up to ASIL-B*
- EB tresos AutoCore OS supports all major AUTOSAR features, including memory protection. Deviations, limitations, and restrictions are documented in the release notes.

* The actually achievable safety level depends on the hardware

EB tresos Safety OS - Overview

Goal:

- AUTOSAR compatible Os (v4.0.3; R19-11 is in preparation)
- Usable in safety projects up to level ASIL-D

Safety requirements:

- Provide spatial freedom from interference
- Support temporal freedom from interference

To ensure freedom from interference, EB tresos Safety OS differs from standard AUTOSAR Os implementations in a few points that are listed on the following slides

EB tresos Safety OS – Details (1)

Spatial freedom from interference: Memory protection and *threads*

- Tasks, ISRs, callout functions (hooks) as well as some Os APIs are executed in so called *threads*
- Each *thread* has its own memory protection configuration (independent from „trusted/non-trusted“) and processor mode (explicitly configurable)
- EB tresos Safety OS ensures that the environment is correctly set up when a *thread* is executed. This includes memory protection and processor mode
- The EB tresos Safety OS kernel has its own memory protection configuration and is expected not to have access to the data of other *threads* unless necessary

Temporal freedom from interference

- Execution budget monitoring is supported
- Other modules (i.e. TimE) are necessary for full temporal freedom from interference (e.g. deadline monitoring, alive supervision)

EB tresos Safety OS – Details (2)

- Safe start-up: Entry point is `MK_Entry2()`. With this function EB tresos Safety OS takes control of the CPU and ensures a safe execution environment
- Safe API usage: EB tresos Safety OS provides defined behavior for wrong or undefined API/feature usage (e.g. wrong calling contexts or missing TerminateTask() call)
- Simple Schedule Tables: A special form of a schedule table that can use a (EB tresos Safety OS specific) hardware ticker timer which increments the schedule table counter with a constant period
- Only functionality necessary to reach the safety requirements is qualified for use up to ASIL-D. Other functionality/API services are qualified as QM only!
- Does not support all AUTOSAR features. See the deviations provided in the EB tresos Safety OS user's guide for more information

Summary - EB treos OS Products

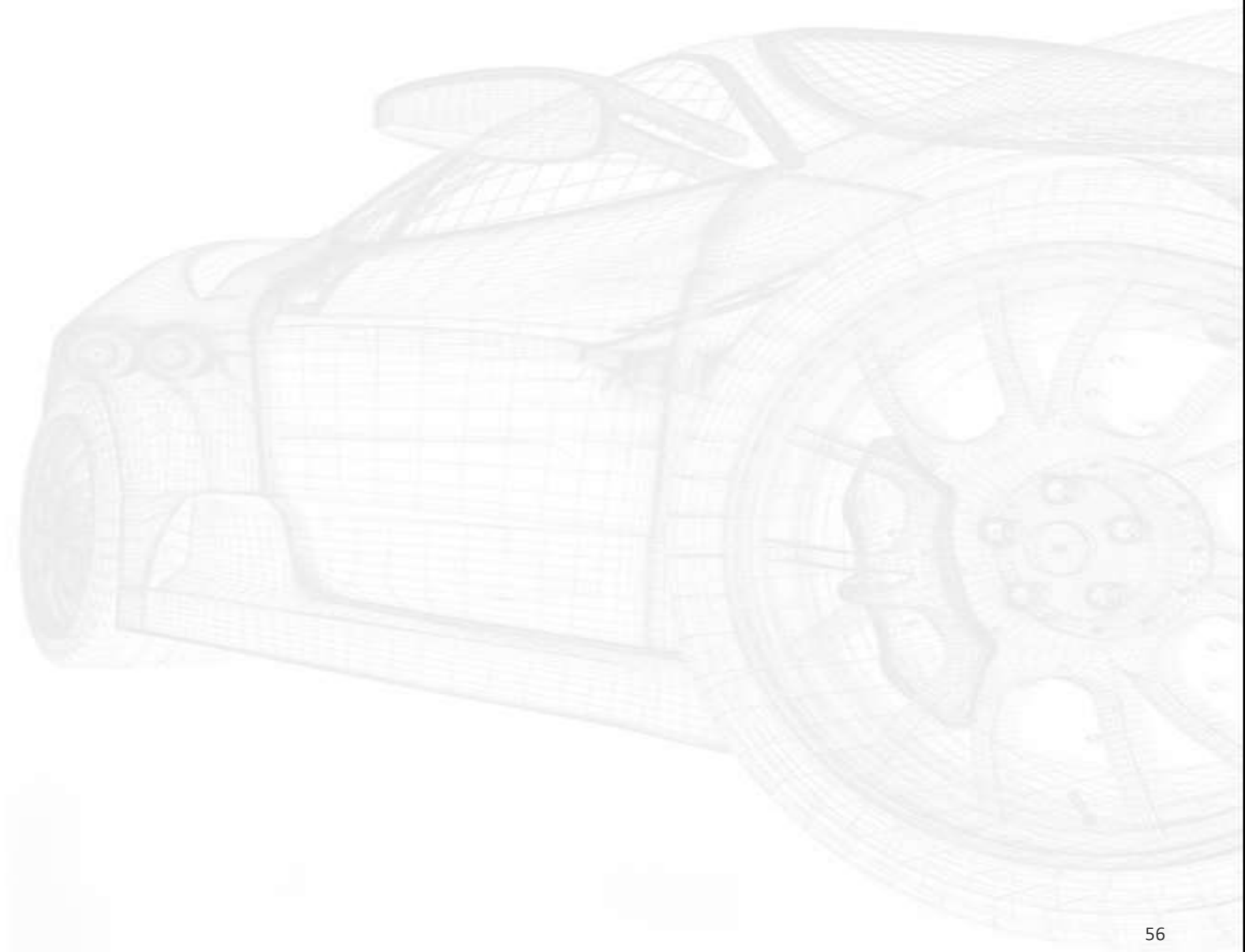
	AutoCore OS	Safety OS
Safety level	QM, with SAG* up to ASIL-B (ISO 26262)	Up to ASIL D (ISO 26262) / SIL 3 (IEC 61508)
Feature set	• Mostly full feature set with some deviations and restrictions • Some additional features like stack monitoring functionality	• Concentration on main features and safety related features • Some special features like simple schedule tables or asynchronous APIs
Startup	Standard AUTOSAR start up	Safe start-up via **MK_Entry2()**.
Memory Protection	Based on AUTOSAR 4.0.3 (trusted/non-trusted application)	Configurable per thread i.e. for each Task, ISR and callout function
Timing protection	Generally supported, some limitations might apply	Supports only execution budget monitoring

* SAG: Safety application guide

Summary

Summary

- Introduction
- Scheduling
- Multi-core Support
- Critical Section Protection
- OS-Application Protection
- Hints and Tips
- EB tresos OS products



Get in touch!

Elektrobit

sales@elektrobit.com
www.elektrobit.com