OSEK/VDX

Operating System

Version 2.1 revision 1

13. November 2000

Preface

OSEK/VDX is a joint project of the automotive industry. It aims at an industry standard for an open-ended architecture for distributed control units in vehicles.

For detailed information about OSEK project goals and partners, please refer to the “OSEK Binding Specification”.

This document describes the concept of a real-time operating system, capable of multitasking, which can be used for motor vehicles. It is not a product description which relates to a specific implementation.

This document also specifies the OSEK operating system - Application Program Interface.

General conventions, explanations of terms and abbreviations have been compiled in the additional inter-project "OSEK Overall Glossary".

Regarding implementation and system generation aspects please refer to the "OSEK Implementation Language" (OIL) specification.

Table of Contents

1 Introduction

The specification of the OSEK operating system is to represent a uniform environment which supports efficient utilisation of resources for automotive control unit application software. The OSEK operating system is a single processor operating system meant for distributed embedded control units.

1.1 System philosophy

Automotive applications are characterised by stringent real-time requirements. Therefore the OSEK operating system offers the necessary functionality to support event driven control systems.

The specified operating system services constitute a basis to enable the integration of software modules made by various manufacturers. To be able to react to the specific features of the individual control units as determined by their performance and the requirements of a minimum consumption of resources, the prime focus was not to achieve 100% compatibility between the application modules, but their direct portability.

As the operating system is intended for use in any type of control units, it must support time-critical applications on a wide range of hardware. A high degree of modularity and ability for flexible configuration are prerequisites to make the operating system suitable for low-end microprocessors and complex control units alike. These requirements have been supported by definition of "conformance classes" (see chapter 3.2, Conformance classes) and a certain capability for application specific adaptations.

For time-critical applications dynamic generation of system objects was left out. Instead, generation of system objects was assigned to the system generation phase. Error inquiries within the operating system are obviated to a large extent, so as not to affect the speed of the overall system unnecessarily. On the other hand, a system version with extended error inquiries has been defined. It is intended for the test phase and for less time-critical applications. Even at that stage defined uniform system appearance is ensured.

Standardised interfaces

The interface between the application software and the operating system is defined by system services. The interface is identical for all implementations of the operating system on various processor families.

System services are specified in an ISO/ANSI-C-like syntax, however the implementation language of the system services is not specified.

Scalability

Different conformance classes, various scheduling mechanisms and the configuration features make the OSEK operating system feasible for a broad spectrum of applications and hardware.

The OSEK operating system is designed to require only a minimum of hardware resources (RAM, ROM, CPU time) and therefore runs even on 8 bit microcontrollers.

Error checking

The OSEK operating system offers two levels of error checking, extended status for development phase and standard status for production phase.

The extended status allows for enhanced plausibility checks on calling operating system services. Due to the additional error checking it requires more execution time and memory space than the standard version. However, many errors can be found in a test phase. After all errors have been eliminated, the system can be recompiled with the standard version.

Portability of application software

One of the goals of OSEK is to support the portability and re-usability of application software. Therefore the interface between the application software and the operation system is defined by standardised system services with well-defined functionality. Use of standardised system services reduces the effort to maintain and to port application software and development cost.

Portability means the ability to transfer an application software module from one ECU to another ECU without bigger changes inside the application.

The application software lies on the operating system and in parallel on a application-specific Input/Output System interface which is not standardised in the OSEK specification. The application software module can have several interfaces. There are interfaces to the operating system for real time control and resource management, but also interfaces to other software modules to represent a complete functionality in a system and at least to the hardware, if the application has to work directly with microcontroller modules.

For better portability of application software, the OSEK defines a language for a standardised configuration information. This language "OIL" (OSEK Implementation Language) supports a portable description of all OSEK specific objects such as "tasks" and "alarms" etc.

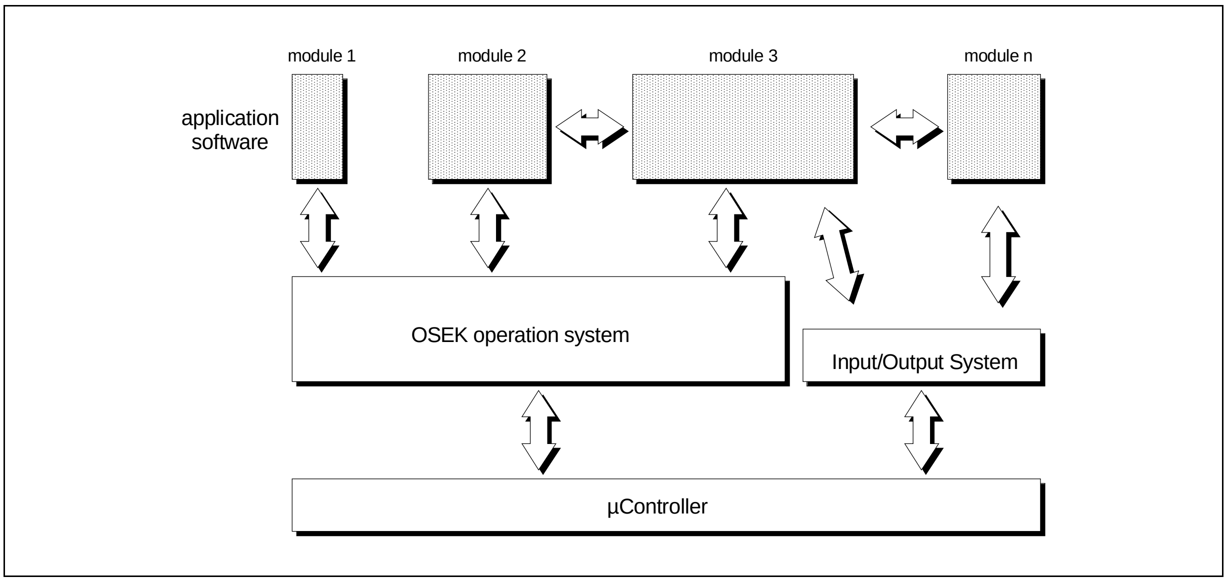


Figure 1–1 Software interfaces inside ECU[1]

During the process to port application software from one ECU to another ECU it is necessary to consider characteristics of the software development process, the development environment, and the hardware architecture of the ECU, for example:

[1] OSEK OS allows direct interfacing between application and the hardware.

- Software development guidelines
- File management system
- Data allocation and stack usage of the compiler
- Memory architecture of the ECU
- Timing behaviour of the ECU
- Different microcontroller specific interfaces e.g. ports, A/D converter, serial communication and watchdog timer
- Placement of the API calls

This means that the OSEK specifications are not enough to describe an OSEK implementation completely. The implementation has to supply specific documentation.

Support of Portability

The certification process ensures the conformance of different implementations to the specification. Chapter 13 of this specification collects implementation specific details which have to be regarded to increase portability of an application between various OSEK implementations. Herein only the operating system interface to the application is considered.

Special support for automotive requirements

Specific requirements for an OSEK operating system arise in the application context of software development for automotive control units. Requirements such as reliability, real-time capability, and cost sensitivity are addressed by the following features:

- The OSEK operating system is configured and scaled statically. The number of tasks, resources, and services required is statically specified by the user.
- The specification of the OSEK operating system supports implementations capable of running on ROM, i.e. the code could be executed from *Read-Only-Memory*.
- The OSEK operating system supports portability of application tasks.
- The specification of the OSEK operating system provides a predictable and documented behaviour to enable operating system implementations, which meet automotive real-time requirements.
- The specification of the OSEK operating system allows the implementation of predictable performance parameters.

1.2 Purpose of this document

The following description is to be regarded as a generic description which is mandatory for any implementation of the OSEK operating system. This concerns the general description of strategy and functionality, the interface of the calls, the meaning and declaration of the parameters and the possible error codes.

The specification leaves a certain amount of flexibility. On the one hand, the description is generic enough for future upgrades, on the other hand, there is some explicitly specified implementation-specific scope in the description.

Any implementation defines all implementation specific issues. The conformance classes supported by the implementation must be indicated precisely, and the issues identified as implementation-specific must be documented.

It is assumed that the description of the OSEK operating system is to be updated in the future, and will be adapted to extended requirements. Therefore, each implementation must specify which officially authorised version of the OSEK description has been used as a reference description. Officially authorised versions of the OSEK operating system description are named x.y. This document represents ”Version 2.1r1”.

Because this description is mandatory, definitions have only been made where the general system strategy is concerned. In all other respects, it is up to the system implementation to determine the optimal adaptation to a specific hardware type.

1.3 Structure of this document

In the following text, the specification chapters are described briefly:

Chapter 2, Summary

This chapter provides a brief introduction to the OSEK operating system concept.

Chapter 3, Architecture of the OSEK operating system

This chapter gives a survey about the design principles and the architecture of the OSEK operating system.

Chapter 4, Task management

This chapter explains the OSEK task management with the different task types and scheduling mechanisms.

Chapter 5, Interrupt processing

This chapter provides information about the OSEK interrupt strategy and the different types of interrupt service routines.

Chapter 6, Event mechanism

This chapter explains the event mechanism and the different behaviour depending on the scheduling.

Chapter 7, Resource management

This chapter describes the OSEK resource management and discusses the benefits and implementation of the OSEK priority ceiling protocol.

Chapter 8, Alarms

This chapter describes the two-stage concept to support time-based events (e.g. hardware-timer) as well as non-time-based events (e.g. angle measurement).

Chapter 9, Messages

The message handling for intra processor communication will be added to the OS specification. Full message handling is described in the OSEK COM specification.

The exact subset to be implemented is yet to be defined.

Chapter 10, Error handling, tracing and debugging

Description of the mechanisms to achieve centralised error-handling. This chapter also describes the services to initialise and shutdown the system.

Chapter 11, Description of system services

This chapter describes the conventions used for description.

Chapter 12, Specification of operating system services

This chapter describes all operating system services made available to the user. Structure of the description is identical for any service; it contains all the information the service user requires.

Chapter 13, Implementation and application specific topics,

This chapter provides a list of all operating system specific topics, including services, data types, and constants.

Chapter 14, Changes from specification 1.0 to 2.1 and 2.1r1

This chapter provides a survey of major changes in the operating system specification from version 1.0 to version 2.1 and 2.1r1.

Chapter 15, Index

List of all operating system services and figures.

Chapter 16, History

List of all official releases.

2 Summary

The OSEK operating system provides a pool of different services and processing mechanisms.

The OSEK operating system is built according to the user's configuration instructions at system generation time.

Four conformance classes are available to satisfy different requirements concerning functionality and capability of the OSEK operating system. Thus, the user can adapt the operating system to the control task and the target hardware. The operating system cannot be modified later at execution time.

Applications which have been written for a certain conformance class have to be portable to OSEK implementations of the same class. This is ensured by a definition of the services, their scope of capabilities, and the behaviour of each conformance class. Only if all the services of a conformance class are offered with the determined scope of capabilities, the operating system implementation conforms to OSEK.

The service groups are structured in terms of functionality.

Task management

- Activation and termination of tasks
- Management of task states, task switching

Synchronisation

The operating system supports two means of synchronisation effective on tasks:

- Resource management
  Access control for inseparable operations to jointly used (logic) resources or devices, or for control of a program flow.
- Event control
  Event management for task synchronisation.

Interrupt management

- Services for interrupt processing

Alarms

- Relative and absolute alarms

Intra processor message handling

- Services for exchange of data

Error treatment

- Mechanisms supporting the user in case of various errors

3 Architecture of the OSEK operating system

3.1 Processing levels

The OSEK operating system serves as a basis for application programs which are independent of each other, and provides their environment on a processor. The OSEK operating system enables a controlled real-time execution of several processes which appear to run in parallel.

The OSEK operating system provides a defined set of interfaces for the user. These interfaces are used by entities which are competing for the CPU. There are two types of entities:

- Interrupt service routines managed by the operating system
- Tasks (basic tasks and extended tasks)

The hardware resources of a control unit can be managed by operating system services. These operating system services are called by a unique interface, either by the application program or internally within the operating system.

OSEK defines three processing levels:

- Interrupt level
- Logical level for scheduler
- Task level

Within the task level tasks are scheduled (non, full or mixed pre-emptive) according to their user assigned priority. The run time context is occupied at the beginning of execution time and is released again once the task is finished.

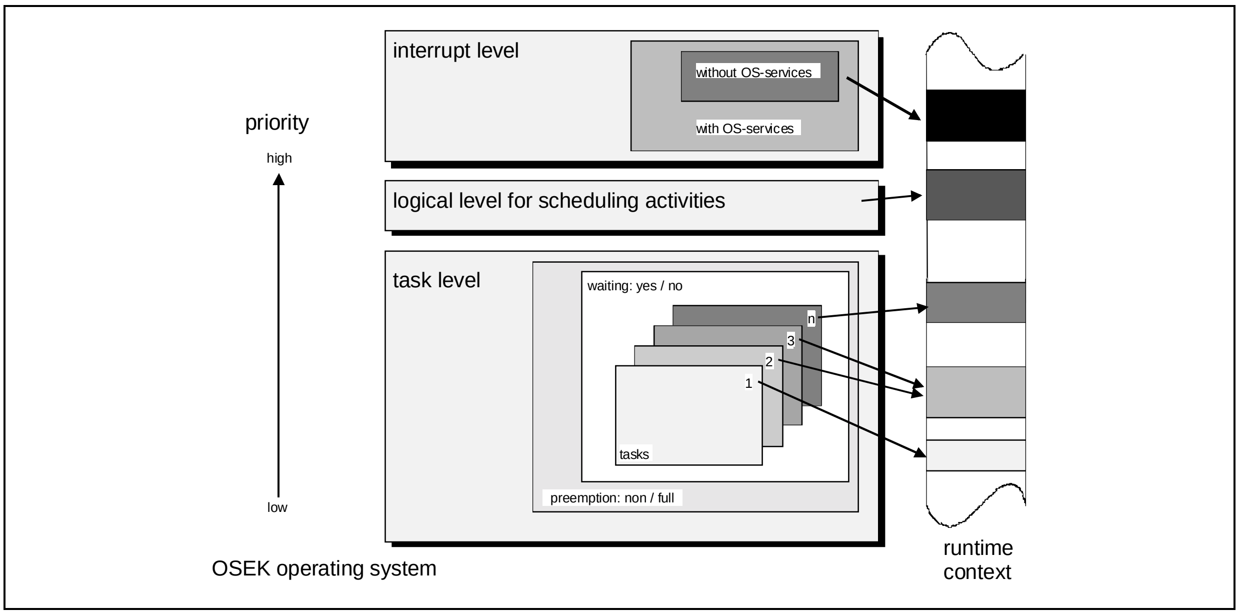


Figure 3–1 Processing levels of the OSEK operating system

The following priority rules have been established:

- Interrupts have precedence over tasks
- The interrupt processing level consists of one or more interrupt priority levels
- Interrupt service routines have a statically assigned interrupt priority level

- Assignment of interrupt service routines to interrupt priority levels is dependent on implementation and hardware architecture
- For task priorities and resource ceiling-priorities bigger numbers refer to higher priorities.
- The task's priority is statically assigned by the user (the meaning of task priorities is described in chapter 4.5)

Processing levels are defined for the handling of tasks and interrupt routines as a range of consecutive values.

Processing levels	Processed instance
k...m	Interrupt
j	Scheduler
0...i	Task

Figure 3–2 Processing levels of the OSEK operating system (table)

The following rule applies for the processing level :

$0 <= i < j < k <= m$

The operating system provides services and ensures compliance with the priority rules mentioned above.

Please note that assignment of a priority to the scheduler is only a logical concept which can be implemented without directly using priorities.

3.2 Conformance classes

Various requirements of the application software for the system, and various capabilities of a specific system (e.g. processor, memory) demand different features of the operating system. In the following description, these operating system features are described as "conformance classes" (CC).

Conformance classes exist to support the following objectives:

- To provide convenient groups of operating system features for easier understanding and discussion of the OSEK operating system.
- To allow partial implementations along pre-defined lines. These partial implementations may be certified as OSEK compliant.
- To create an upgrade path from classes of lesser functionality to classes of higher functionality with no changes to the application using OSEK related features.

The complete conformance class must be implemented to be certified. However, system generation needs only to link those system services that are required for a specific application. Conformance classes cannot be changed during execution.

Conformance classes are determined by the following attributes:

- Multiple requesting of task activation, as described in chapter 4.3
- Task types, as described in chapter 4.2
- Number of tasks per priority

All other OSEK features are mandatory if not explicitly stated otherwise.

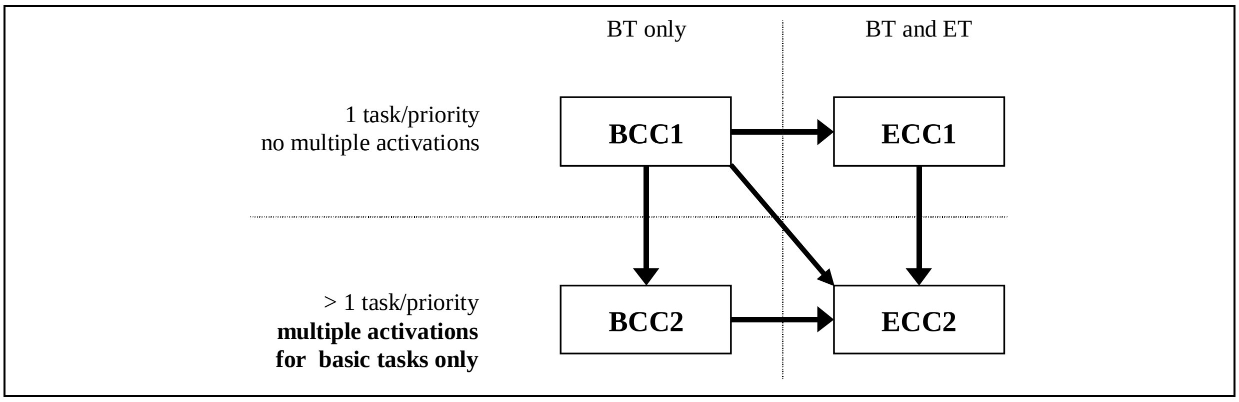


Figure 3–3 Restricted upward compatibility for conformance classes

The following conformance classes are defined:

- BCC1 (only basic tasks, limited to one activation request per task and one task per priority, while all tasks have different priorities)
- BCC2 (like BCC1, plus more than one task per priority possible and multiple requesting of task activation allowed)
- ECC1 (like BCC1, plus extended tasks)
- ECC2 (like ECC1, plus more than one task per priority possible and multiple requesting of task activation allowed for basic tasks)

The portability of applications can only be assumed if the minimum requirements are not exceeded. The minimum requirements for Conformance Classes are shown in the Figure 3–4.

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

Figure 3–4 The minimum requirements for Conformance Classes

[2] BT = Basic Task, ET = Extended Task

3.3 Relationship between OSEK OS and OSEKtime OS

OSEKtime OS is an operating system especially tailored to the needs of time triggered architectures. It allows OSEK OS to coexist with OSEKtime OS. Conceptually, OSEKtime assigns its idle time to be used by OSEK. OSEK OS interrupts and tasks have less importance (lower priority) than similar entities in OSEKtime OS.

The OSEK interfaces, and the definition of system calls, do not change if OSEK coexists with OSEKtime. There are minor exceptions with respect to system startup and shutdown due to the fact that OSEKtime is responsible for the overall system whereas OSEK is only locally responsible. These deviations are specifically mentioned within this specification.

On top of this, there is functionality defined within OSEKtime which imposes restrictions on the implementation of OSEK OS if it is intended to coexist with OSEKtime OS. For more information, please refer to the specification of the OSEKtime OS.

4 Task management

4.1 Task concept

Complex control software can conveniently be subdivided in parts executed according to their real-time requirements. These parts can be implemented by the means of tasks. A task provides the framework for the execution of functions. The operating system provides concurrent and asynchronous execution of tasks. The scheduler organises the sequence of task execution.

The OSEK operating system provides a task switching mechanism (scheduler), including an idle mechanism. (see chapter 4.4, Task switching mechanism).Two different task concepts are provided by the OSEK operating system:

- basic tasks
- extended tasks

Basic Tasks

Basic tasks only release the processor, if

- they terminate,
- the OSEK operating system switches to a higher-priority task, or
- interrupt occurs which cause the processor to switch to an interrupt service routine (ISR).

Extended Tasks

Extended tasks are distinguished from basic tasks by being allowed to use the operating system call *WaitEvent*, which may result in a *waiting* state (see chapter 6, Event mechanism, and chapter 12.5.3.4, WaitEvent). The *waiting* state allows the processor to be released and to be reassigned to a lower-priority task without the need to terminate the running extended task.

In view of the operating system, management of extended tasks is, in principle, more complex than management of basic tasks and requires more system resources.

4.2 Task state model

The following text describes the task states and the transitions between the states for both task types.

A task must be able to change between several states, as the processor can only execute one instruction of a task at any time, while several tasks may be competing for the processor at the same time. The OSEK operating system is responsible for saving and restoring task context in conjunction with task state transitions whenever necessary.

4.2.1 Extended tasks

Extended tasks have four task states:

running	In the *running* state, the CPU is assigned to the task, so that its instructions can be executed. Only one task can be in this state at any point in time, while all the other states can be adopted simultaneously by several tasks.
ready	All functional prerequisites for a transition into the *running* state exist, and the task only waits for allocation of the processor. The scheduler decides which *ready* task is executed next.

waiting	A task cannot continue execution because it has to *wait* for at least one event (see chapter 6, Event mechanism).
suspended	In the *suspended* state the task is passive and can be activated.

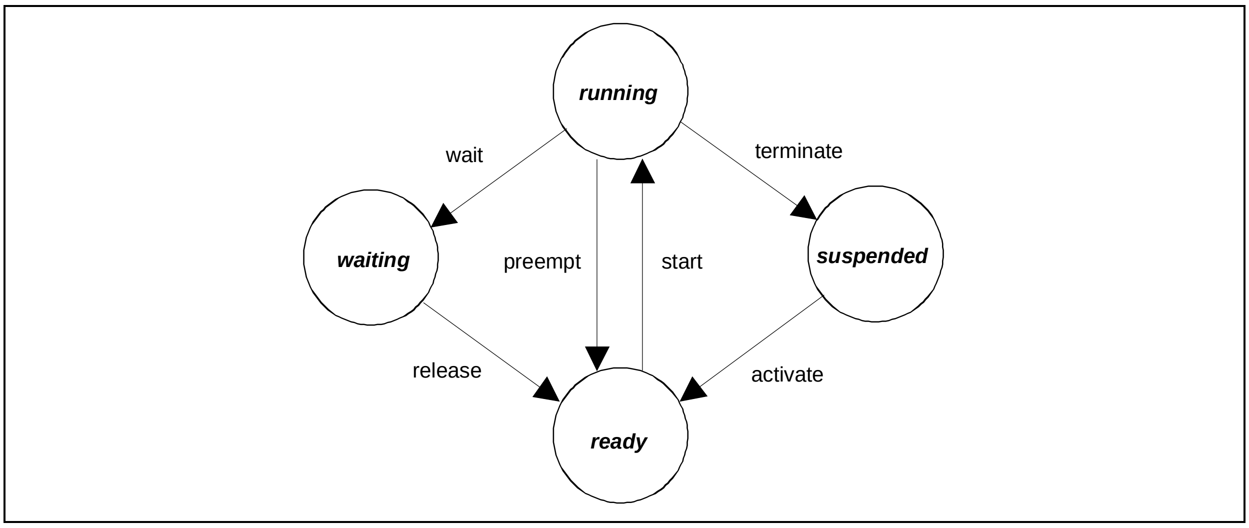


Figure 4–1 Extended task state model

Transition	**Former state**	**New state**	**Description**
activate	*suspended*	*ready*	A new task is set into the *ready* state by a system service. The OSEK operating system ensures that the execution of the task will start with the first instruction.
start	*ready*	*running*	A *ready* task selected by the scheduler is executed.
wait	*running*	*waiting*	The transition into the waiting state is caused by a system service. To be able to continue operation, the *waiting* task requires an event.
release	*waiting*	*ready*	At least one event has occurred which a task has *waited* for.
preempt	*running*	*ready*	The scheduler decides to start another task. The *running* task is put into the *ready* state.
terminate	*running*	*suspended*	The *running* task causes its transition into the *suspended* state by a system service.

Figure 4–2 States and status transitions for extended tasks

Termination of a task is only possible if the task terminates itself ("self-termination"). This restriction reduces complexity of an operating system. There is no provision for a direct transition from the *suspended* state into the *waiting* state. This transition is redundant and would add to the complexity of the scheduler.

4.2.2 Basic tasks

The state model of basic tasks is nearly identical to the extended tasks state model. The only exception is that basic tasks do not have a *waiting* state.

running	In the *running* state, the CPU is assigned to the task, so that its instructions can be executed. Only one task can be in this state at any point in time, while all the other states can be adopted simultaneously by several tasks.
ready	All functional prerequisites for a transition into the *running* state exist, and the task only waits for allocation of the processor. The scheduler decides which *ready* task is executed next.
suspended	In the *suspended* state the task is passive and can be activated.

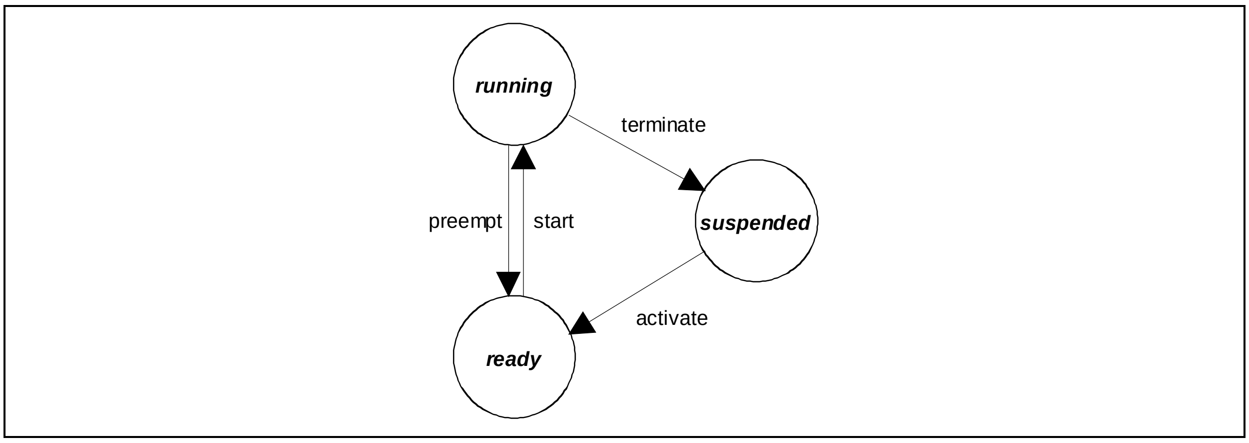


Figure 4–3 Basic task state model

Transition	Former state	New state	Description
activate	*suspended*	*ready*[3]	A new task is set into the *ready* state by a system service. The OSEK operating system ensures that the execution of the task will start with the first instruction.
start	*ready*	*running*	A *ready* task selected by the scheduler is executed.
preempt	*running*	*ready*	The scheduler decides to start another task. The *running* task is put into the *ready* state.
terminate	*running*	*suspended*	The *running* task causes its transition into the *suspended* state by a system service.

Figure 4–4 States and status transitions for basic tasks

4.2.3 Comparison of the task types

Basic tasks have no *waiting* state, and thus only comprise synchronisation points at the beginning and the end of the task. Parts of application with internal synchronisation points, have to be implemented by more than one basic tasks. An advantage of basic tasks is their moderate requirement regarding run time context (RAM).

An advantage of extended tasks is that they can handle a coherent job in a single task, no matter which synchronisation requests are active. Whenever current information for further processing is missing, the extended task switches over into the *waiting* state. It exits this state

[3] Task activation will not immediately change the state of the task in case of multiple activation requests. If the task is not suspended, the activation will only be recorded and performed later.

whenever corresponding events signal the receipt or the update of the desired data or events. Extended tasks also comprise more synchronisation points than basic tasks.

4.3 Activating a task

Task activations are performed using the operating system services *ActivateTask* or *ChainTask*. After activation the task is ready to execute from the first statement.

The OSEK operating system does not support C-like parameter passing when starting a task. Those parameters should be passed by message communication (see “Messages”) or by global variables.

Multiple requesting of task activation

Depending on the conformance class a basic task can be activated once or multiple times. "Multiple requesting of task activation" means that the OSEK operating system receives and records parallel activations of a basic task already activated.

The number of multiple requests in parallel is defined in a basic task specific attribute during system generation. If the maximum number of multiple requests has not been reached, the request is queued. The requests of basic task activations are queued per priority in activation order.

4.4 Task switching mechanism

Unlike conventional sequential programming, the principle of multitasking allows the operating system to execute various tasks concurrently. Therefore the scheduling policy has clearly to be defined (see chapter 4.6, Scheduling policy).

The entity deciding which task has to be started and the triggering of all necessary OSEK operating system internal activities is called scheduler. The scheduler is activated whenever a task switch is possible according to the implemented scheduling policy. The scheduler can be considered as a resource which can be occupied and released by tasks. Thus, a task can reserve the scheduler to avoid a task switch until it is released. For further details, please refer to chapter 7.3, Scheduler as a resource.

4.5 Task priority

The scheduler decides on the basis of the task priority (precedence) which is the next of the *ready* tasks to be transferred into the *running* state.

The value 0 is defined as the lowest priority of a task. Accordingly bigger numbers define higher priorities.

To enhance efficiency, a dynamic priority management is not supported. Accordingly the priority of a task is defined statically, i.e. it cannot be changed by the user at the time of execution. However, in particular cases the operating system can treat a task with a defined higher priority. In this context, please refer to chapter 7.5, OSEK Priority Ceiling Protocol.

Tasks of identical priority are supported in the conformance classes BCC2 and ECC2, see chapter 3.2, Conformance classes.

Tasks on the same priority level are started depending on their order of activation, whereby extended tasks in the *waiting* state do not block the start of subsequent tasks of identical priority.

A pre-empted task is considered to be the first task in the *ready* list of its current priority.

A task being released from the *waiting* state is treated like the newest task in the *ready* queue of its priority.

Figure 4–5 shows an example implementation of the scheduler using for each priority level. Several tasks of different priorities are in the *ready* state; i.e. three tasks of priority 3, one of priority 2 and one of priority 1, plus two tasks of priority 0. The task which has waited the longest time, depending on its order of requesting, is shown at the bottom of each queue. The processor has just processed and terminated a task. The scheduler selects the next task to be processed (priority 3, first queue). Before priority 2 tasks can be processed, all tasks of higher priority must have left the *running* and *ready* state, i.e. started and then removed from the queue either due to termination or due to transition into waiting state.

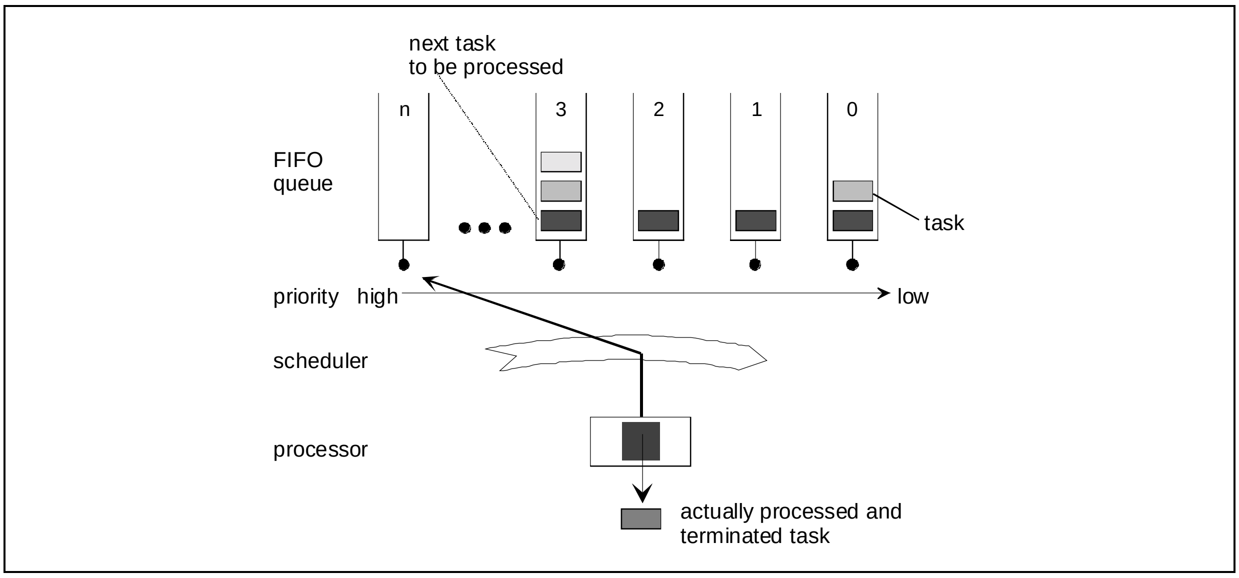


Figure 4–5 Scheduler: order of events

The following fundamental steps are necessary to determine the next task to be processed:

- The scheduler searches for all tasks in the *ready/running* state.
- From the set of tasks in the *ready/running* state, the scheduler determines the set of tasks with the highest priority.
- Within the set of tasks in the *ready/running* state and of highest priority, the scheduler finds the oldest task.

4.6 Scheduling policy

4.6.1 Non pre-emptive scheduling

The scheduling policy is described as non pre-emptive, if task switching is only performed via one of a selection of explicitly defined system services (explicit points of rescheduling).

Non pre-emptive scheduling imposes particular constraints on the possible timing requirements of tasks. Specifically the non pre-emptable section of a *running* task with lower priority delays the start of a task with higher priority up to the next point of rescheduling.

In Figure 4–6, task T2 with the lower priority delays task T1 with higher priority up to the next point of rescheduling (in this case termination of task T2).

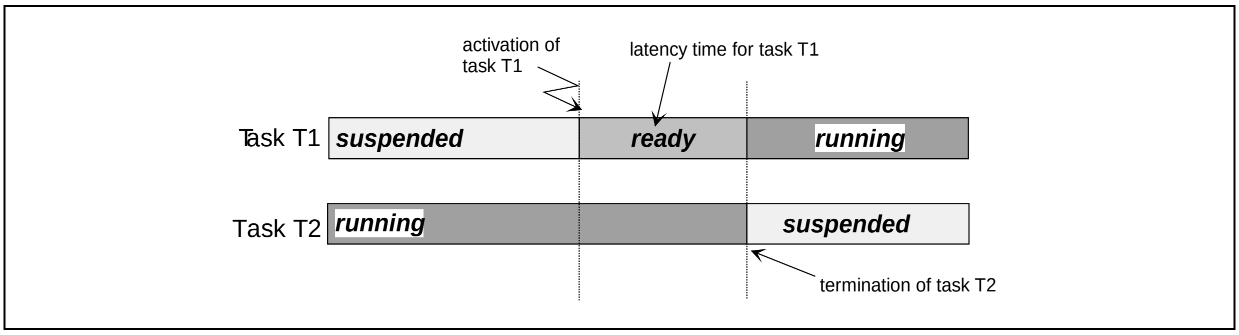


Figure 4–6 Non pre-emptive scheduling

Points of rescheduling

In the case of a non pre-emptive task, rescheduling will take place exactly in the following cases:

- Successful termination of a task (system service *TerminateTask*, see chapter 12.2.3.2).
- Successful termination of a task with explicit activation of a successor task (system service *ChainTask*, see chapter 12.2.3.3).
- Explicit call of scheduler (system service *Schedule*, see chapter 12.2.3.4).
- A transition into the *waiting* state takes place (system service *WaitEvent*, see chapter 12.5.3.4)[4].

Implementations of non pre-emptive systems may prescribe that operating system services which cause rescheduling may only be called at the highest task program level (not in task subfunctions). Consequently, a task switch at these points of scheduling only requires saving minimum task context (no stack, only few registers e.g. program counter and/or processor status).

4.6.2 Full pre-emptive scheduling

Full pre-emptive scheduling means that a task which is presently *running* may be rescheduled at any instruction by the occurrence of trigger conditions pre-set by the operating system. Full pre-emptive scheduling will put the *running* task into the *ready* state, as soon as a higher-priority task has got *ready*. The task context is saved so that the pre-empted task can be continued at the location where it was pre-empted.

With full pre-emptive scheduling the latency time is independent of the run time of lower priority tasks. Certain restrictions are related to the increased (RAM-) memory space required for saving the context, and the enhanced complexity of features necessary for synchronisation between tasks. As each task can theoretically be rescheduled at any location, access to data which are used jointly with other tasks must be synchronised.

In Figure 4–7, task T2 with the lower priority does not delay the scheduling of task T1 with higher priority.

[4] The call of WaitEvent does not lead to a *waiting* state if one of the events passed in the event mask to WaitEvent is already set. In this case WaitEvent does not lead to a rescheduling.

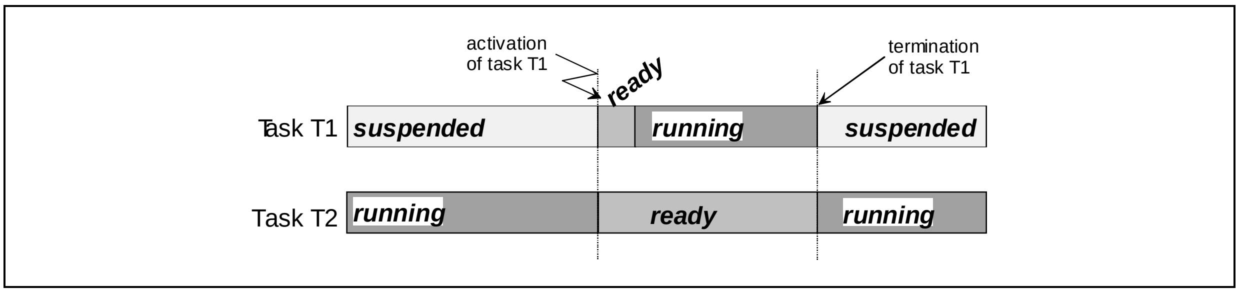


Figure 4–7 Full pre-emptive scheduling

In the case of a full pre-emptive system, the user must constantly expect pre-emption of the *running* task. If a task fragment must not be pre-empted, this can be achieved by blocking the scheduler temporarily via the system service *GetResource*.

Summarised, rescheduling is performed in all of the following cases:

- Successful termination of a task (system service *TerminateTask*, see chapter 12.2.3.2).
- Successful termination of a task with explicit activating of a successor task (system service *ChainTask*, see chapter 12.2.3.3).
- Activating a task at task level (e.g. system service *ActivateTask*, see chapter 12.2.3.1, message notification mechanism, alarm expiration, if task activation is defined, see chapter 8.2).
- Explicit *wait* call, if a transition into the *waiting* state takes place (extended tasks only, system service *WaitEvent*, see chapter 12.5.3.4).
- Setting an event to a *waiting* task at task level (e.g. system service *SetEvent*, see chapter 12.5.3.1, message notification mechanism, alarm expiration, if event setting defined, see chapter 8.2).
- Release of resource at task level (system service *ReleaseResource*, see chapter 11.3.3.2)
- Return from interrupt level to task level

During interrupt service routines no rescheduling is performed (see figure 3-1).

To enable portable applications to be written in spite of the different scheduling policies, the user can enforce a rescheduling via the system service *Schedule* at locations where he assumes a correct assignment of the CPU.

4.6.3 Mixed pre-emptive scheduling

If full pre-emptive and non pre-emptive tasks are mixed on the same system, the resulting policy is called "mixed pre-emptive" scheduling. In this case scheduling policy depends on pre-emption properties of running task. If the running task is non pre-emptive, then non pre-emptive scheduling is performed. If the running task is pre-emptive, then pre-emptive scheduling is performed.

The definition of a non pre-emptive task makes sense in a full pre-emptive operating system,

- if the execution time of the task is in the same magnitude of the time of a task switch,
- if RAM is to be used economically to provide space for saving the task context, or
- if the task must not be pre-empted.

Many applications comprise only few parallel tasks with a long execution time, for which a full pre-emptive operating system would be convenient, and many short tasks with a defined execution time where non pre-emptive scheduling would be more efficient. For this configuration, the mixed pre-emptive scheduling policy was developed as a compromise (see also the design hint in chapter 13.2.4).

4.6.4 Selecting the scheduling policy

The software developer or the system integrator determines the task execution sequence by configuring the task priorities and assigning the pre-emptibility as a task attribute.

We would like to point out expressly that the pre-emptibility of the system depends neither on the conformance class, nor on the task type. Above all, a full pre-emptive system may therefore contain basic tasks, and a non pre-emptive system extended tasks.

If an operating system service is running, pre-emption and context switch might be delayed until the completion of the service.

4.7 Termination of tasks

In the OSEK operating system, a task can only terminate itself ("self-termination").

The OSEK operating system provides the service *ChainTask* to ensure that a dedicated task activation is performed just after the termination of the running task. Chaining itself puts the task into the last element of the priority queue.

Each task has to terminate itself at the end of its code. Ending the task without a call to *TerminateTask* or *ChainTask* is strictly forbidden!

4.8 Application modes

Application modes are designed to allow an OSEK operating system to come up under different modes of operation. The minimum number of supported application modes is one. It is intended only for modes of operation that are totally mutually exclusive. An example of two exclusive modes of operation would be end-of-line programming and normal operation. Once the operating system has been started, it is not allowed to change the application mode.

The characteristics of application modes are:

- start up performance
- support of exclusive applications
- supported by all conformance classes

Scope of application modes

Many ECUs may execute completely independent applications as e.g. factory test, Flash programming or normal operation. The application mode is a means to structure the software running in the ECU according to those different conditions. Typically each application mode consists of an own set of tasks, ISRs and timing conditions, although there is no limitation to having a task or ISR running in different modes. Sharing a task/ISR between different modes is recommended if the same functionality is needed again, because checking the current application mode inside the task/ISR at runtime is very inefficient.

Having system generation and optimisation in mind, application modes are helpful to reduce the number of OS objects taken into consideration.

Switching between application modes at runtime is not a strong request from applications. It could be helpful e.g. if end-of-fabrication-test is designed as a separate mode. One reason why mode switching at runtime is not allowed is that normally timing constrains have to be met throughout the operation as for example the still-alive-protocol between main and supervisor processors.

4.8.1 Start up performance

The start up performance is a safety critical issue for ECUs in automotive applications since reset conditions may occur during normal operation. As a result the code used to determine the application mode should be very quick. It is recommended that only pin states, or similarly easy to assess conditions be used to determine the mode. The mode will be determined before the kernel is started and the resulting code is non-portable. It is clear that a lengthy or complicated starting procedure should be avoided.

4.8.2 Support of exclusive applications

Application modes allow independent development of totally separate systems.

For systems that are completely exclusive, this feature will allow a very clean mechanism for independent system development.

4.8.3 Supported by all conformance classes

Because the overhead of mode detection is minimal, there is no reason to restrict the feature of application modes to a subset of conformance classes. It is required for all classes. At start up, the user code using no system services (see Figure 10–2), will determine the mode and pass it as a parameter to the API-service *StartOS*[5]. This will allow the operating system to load the correct contexts, and other OS information to allow the execution of the correct applications.

There is no impact on the shutdown functionality.

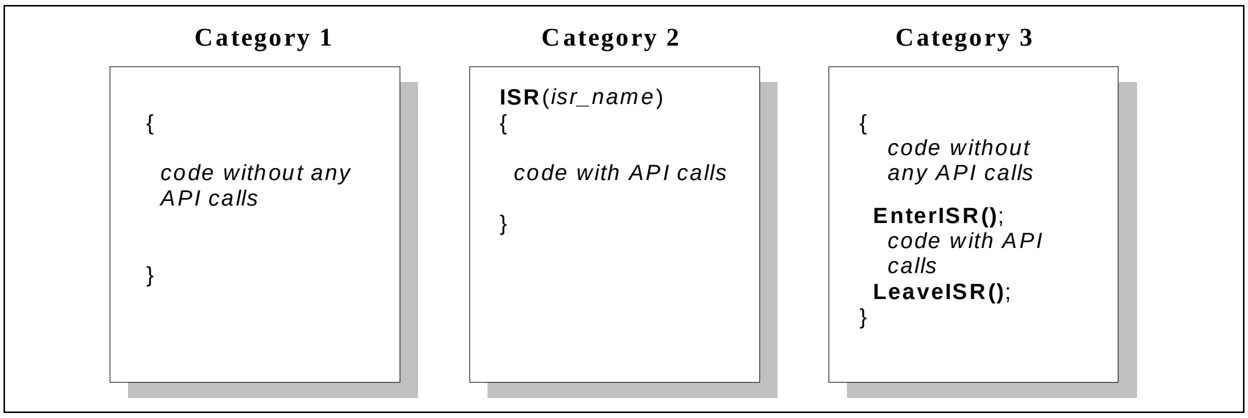
[5] In case of a system where OSEK and OSEKtime coexist, the application mode passed to OSEKtime is used.

5 Interrupt processing

The functions for processing an interrupt (Interrupt Service Routine: ISR) are subdivided into three ISR categories:

ISR category 1	The ISR does not use an operating system service. After the ISR is finished, processing continues exactly at the instruction where the interrupt has occurred, i.e. the interrupt has no influence on task management. ISRs of this category have the least overhead.
ISR category 2	The OSEK operating system provides an ISR-frame to prepare a run-time environment for a dedicated user routine. During system generation the user routine is assigned to the interrupt. From the applications' point of view, this category is the most comfortable one. Within an interrupt service routine of category 2, usage of OSEK operating system services is restricted according to Figure 5–2.
ISR category 3	Such ISRs can be used like category 1 ISRs. However, if the user needs to call system services, he has first to call *EnterISR*. After *EnterISR*, the ISR acts like an ISR of category 2. If *EnterISR* was called, a *LeaveISR* call is needed to return from the ISR. This category is the most flexible one. The services *EnterISR* and *LeaveISR* are provided as a part of the API. Between *EnterISR* and *LeaveISR* restrictions on OSEK operating system services are equal to category 2. Concerning the use of stack, registers and local variables outside and between *EnterISR* and *LeaveISR* implementation specific restrictions might apply. *LeaveISR* must be the last statement executed in the ISR.

The implementation of ISR categories 1 and 2 is mandatory, whereas ISR category 3 is optional.



```
{
  code without any
  API calls
}
```

```
ISR(isr_name)
{
  code with API calls
}
```

```
{
  code without
  any API calls
 EnterISR();
  code with API
  calls
 LeaveISR();
}
```

Figure 5–1 ISR categories of the OSEK operating system

Inside the ISR no rescheduling will take place. Rescheduling takes place on termination of the ISR category 2 or 3 if a pre-emptive task has been interrupted and if no other interrupt is active.
The implementation ensures that tasks are executed according to the OSEK scheduling points (see chapter 4.6.2 Full pre-emptive scheduling). To achieve this the implementation may prescribe restrictions concerning interrupt priority levels for ISRs of all categories and/or perform checks at configuration time (see chapter 13.2.3.2, Nested interrupts of different categories).

The maximum number of interrupt priorities depends on the controller used as well as on the implementation. The scheduling of interrupts is hardware dependent and not specified in OSEK. Interrupts are scheduled by hardware while tasks are scheduled by the scheduler. Regarding the interrupt priority levels there may be restrictions as described in 13.2.3.2. Interrupts can interrupt tasks (non and full pre-emptive tasks). If a task is activated from an interrupt routine the task is scheduled after the end of all active interrupt routines.

In interrupt service routines the following services of the OSEK operating system can be used:

Service	**called by Task**	**called by ISR category 2 and 3**
ActivateTask	**allowed**	**allowed**
TerminateTask	**allowed**	--
ChainTask	**allowed**	--
Schedule	**allowed**	--
GetTaskID	**allowed**	**allowed**
GetTaskState	**allowed**	**allowed**
EnterISR	--	**allowed**[6]
LeaveISR	--	**allowed**[6]
EnableInterrupt	**allowed**	**allowed**
DisableInterrupt	**allowed**	**allowed**
GetInterruptDescriptor	**allowed**	**allowed**
DisableAllInterrupts	**allowed**	**allowed**
EnableAllInterrupts	**allowed**	**allowed**
SuspendOSInterrupts	**allowed**	**allowed**
ResumeOSInterrupts	**allowed**	**allowed**
GetResource	**allowed**	**allowed**
ReleaseResource	**allowed**	**allowed**
SetEvent	**allowed**	**allowed**
ClearEvent	**allowed**	--
GetEvent	**allowed**	**allowed**
WaitEvent	**allowed**	--
GetAlarmBase	**allowed**	**allowed**
GetAlarm	**allowed**	**allowed**
SetRelAlarm	**allowed**	**allowed**
SetAbsAlarm	**allowed**	**allowed**
CancelAlarm	**allowed**	**allowed**
GetActiveApplicationMode	**allowed**	**allowed**
StartOS	--	--
ShutdownOS	**allowed**	**allowed**

Figure 5–2 API services allowed to be called by tasks and ISRs

[6] This service is allowed in ISR category 3 only.

Source related Disable/Enable interrupt API

Operating system services have been provided to enable and disable selected interrupt sources.

An interrupt source which has been disabled will stay disabled until it is re-enabled by the application.

Hint: Due to normal scheduling algorithms, interrupts or higher priority tasks may delay the time until an interrupt source is enabled. To keep the delay short, interrupts and tasks can be blocked out using resource management.

Fast Disable/Enable API-functions

OSEK offers fast functions to disable all interrupts (see chapter 12.3.2.6, EnableAllInterrupts and 12.3.2.7, DisableAllInterrupts), and to disable all interrupts of category 2 and 3 (see chapter 12.3.2.8, ResumeOSInterrupts and 12.3.2.9, SuspendOSInterrupts). Typical usage is to protect short critical sections. Operating system service calls are not allowed between disable and enable pairs. Exception: *SuspendOSInterrupts* and *ResumeOSInterrupts* are allowed to be nested.

6 Event mechanism

The event mechanism

- is a means of synchronisation
- is only provided for extended tasks
- initiates state transitions of tasks to and from the *waiting* state.

Events are objects managed by the operating system. They are not independent objects, but assigned to extended tasks. Each extended task has a definite number of events. This task is called the owner of these events. An individual event is identified by its owner and its name (or mask). When activating an extended task, these events are cleared by the operating system. Events can be used to communicate binary information to the extended task to which they are assigned. The meaning of events is defined by the application, e.g. signalling of an expiring timer, the availability of a resource, the reception of a message, etc.

Various options are available to manipulate events, depending on whether the dedicated task is the owner of the event or another task which does not necessarily have to be an extended task. All tasks can set any events of any not suspended extended task. Only the owner is able to clear its events and to *wait* for the reception (= setting) of its events.

Events are the criteria for the transition of extended tasks from the *waiting* state into the *ready* state. The operating system provides services for setting, clearing and interrogation of events, and for *waiting* for events to occur.

Any task or ISR can set an event for a not suspended extended task, and thus inform the extended task about any status change via this event.

The receiver of an event is an extended task in any case. Consequently, it is not possible for an interrupt service routine or a basic task to *wait* for an event. An event can only be cleared by the task which is the owner of the event. Extended tasks may only clear events they own, whereas basic tasks must not use the operating system service for clearing events.

An extended task in the *waiting* state is released to the *ready* state if at least one event for which the task is *waiting* has occurred. If a *running* extended task tries to *wait* for an event and this event has already occurred, the task remains in the *running* state.

Figure 6–1 explains synchronisation of extended tasks by setting events in case of full pre-emptive scheduling, where extended task T1 has the higher priority.

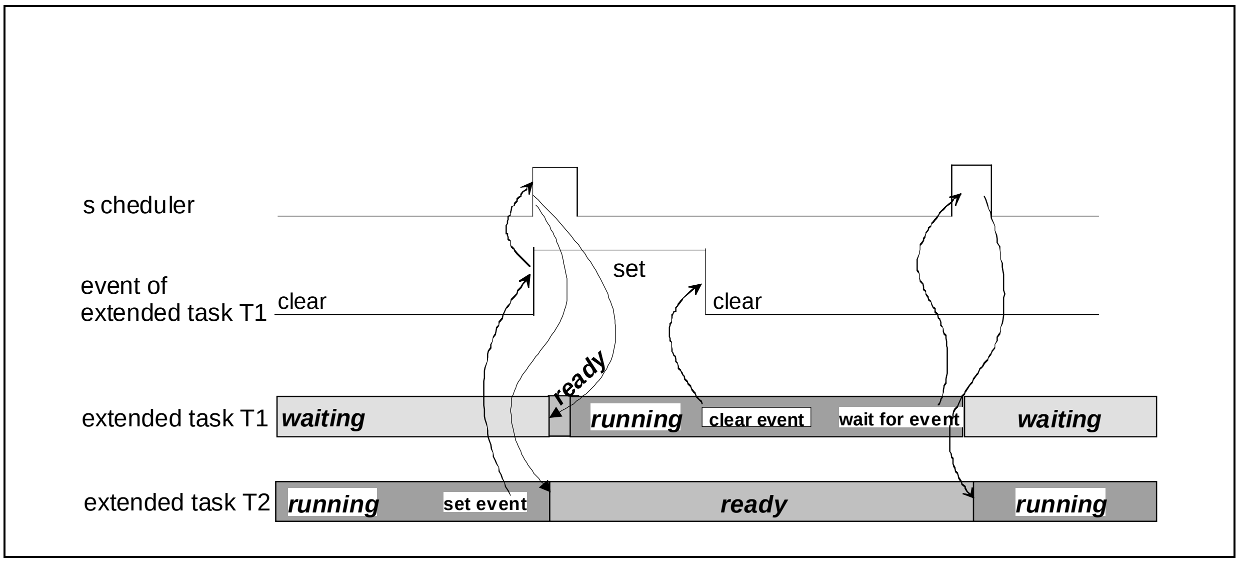


Figure 6–1 Full pre-emptive synchronisation of extended tasks

Figure 6–1 illustrates the procedures which are effected by setting an event: Task T1 *waits* for an event. Task T2 sets this event for T1. The scheduler is activated. Subsequently, T1 is transferred from the *waiting* state into the *ready* state. Due to the higher priority of T1 this results in a task switch, T2 being pre-empted by T1. T1 resets the event. Thereafter T1 *waits* for this event again and the scheduler continues execution of T2.

If non pre-emptive scheduling is supposed, rescheduling does not take place immediately after the event has been set (see Figure 6–2, where extended task T1 is of higher priority)

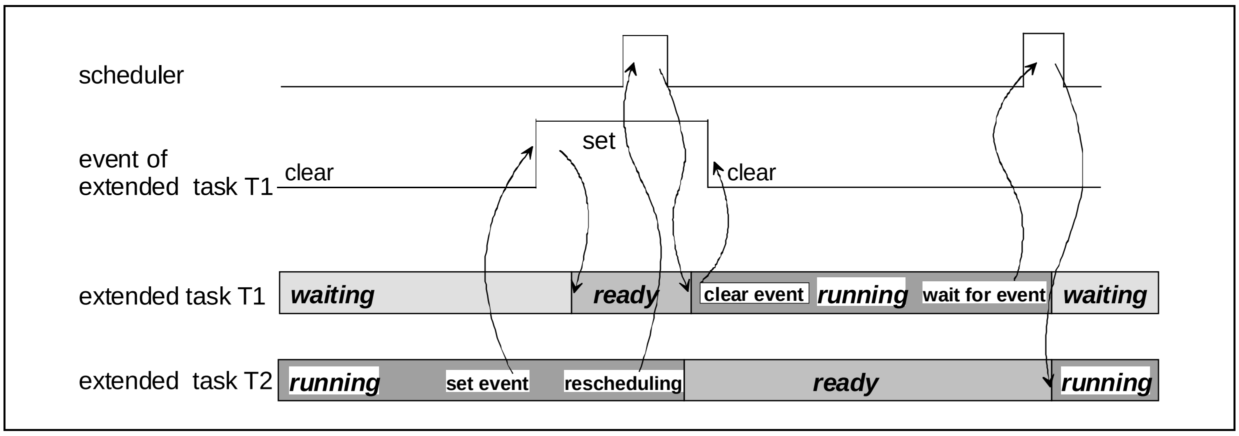


Figure 6–2 Non pre-emptive synchronisation of extended tasks

7 Resource management

The resource management is used to co-ordinate concurrent accesses of several tasks with different priorities to shared resources, e.g. management entities (scheduler), program sequences, memory or hardware areas.

The resource management is mandatory for all conformance classes.

The resource management can optionally be extended to co-ordinate concurrent accesses of tasks and interrupt routines.

Resource management ensures that

- two tasks cannot occupy the same resource at the same time.
- priority inversion can not occur.
- deadlocks do not occur by use of these resources.
- access to resources never results in a *waiting* state.

If the resource management is extended to the interrupt level it assures in addition that

- two tasks or interrupt routines cannot occupy the same resource at the same time.

The functionality of resource management is only required in the following cases:

- full pre-emptive tasks
- non pre-emptive tasks, if resources are also to remain occupied beyond a scheduling point
- non pre-emptive tasks, if the user intends to have the application code executed under other scheduling policies, too
- resource sharing between tasks and interrupt service routines
- resource sharing between interrupt service routines

If the user requires protection against interruptions not only caused by tasks, but also caused by interrupts, he can also use the operating system services to set and reset interrupt masks. Resetting interrupt masks does not cause rescheduling. (See chapter 5, Interrupt processing, and chapter 12.3, Interrupt handling).

7.1 Behaviour during access to occupied resources

OSEK OS prescribes the OSEK priority ceiling protocol (see chapter 7.5) Consequently, no situation occurs in which a task or an interrupt tries to access an occupied resource.

If the resource concept is used for task- and interrupt-coordination the OSEK operating system ensures also that an interrupt service routine is only processed if all resources which might be occupied by that interrupt service routine during its execution have been released.

Additionally, OSEK strictly forbids nested access to the same resource!

7.2 Restrictions when using resources

Neither *TerminateTask*, *ChainTask* nor *WaitEvent* must be called while a resource is occupied. Interrupt service routine must not be completed with a resource occupied.

In case of multiple resource occupation within one task, the user has to request and release resources following the LIFO principle (stack).

7.3 Scheduler as a resource

If a task has to protect itself against pre-emptions by other tasks, it can lock the scheduler. The scheduler is treated like a resource which is accessible to all tasks. Therefore a resource with a predefined name RES_SCHEDULER is generated.

Interrupts are received and processed independent of the state of the resource ‘scheduler’. However, it prevents the rescheduling of tasks.

7.4 General problems with synchronisation mechanisms

7.4.1 Explanation of priority inversion

A typical problem of common synchronisation mechanisms - e.g. the use of semaphores - is the problem relating to priority inversion.

This means that a lower-priority task delays the execution of higher-priority task. One solution to avoid priority inversion is to use the *OSEK Priority Ceiling Protocol* (see chapter 7.5).

Figure 7–1 illustrates sequencing of the common access of two tasks to a semaphore (in a full pre-emptive system, task T1 has the highest priority)
Task T4 which has a low priority, occupies the semaphore S1. T1 pre-empts T4 and requests the same semaphore. As the semaphore S1 is already occupied, T1 enters the waiting state. Now the low-priority T4 is interrupted and pre-empted by tasks with a priority between those of T1 and T4. T1 can only be executed after all lower-priority tasks have been terminated, and the semaphore S1 has been released again. Although T2 and T3 do not use semaphore S1, they delay T1 with their runtime.

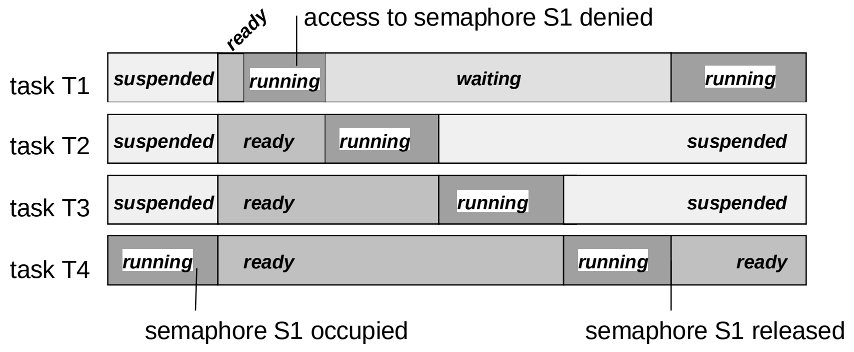


Figure 7–1 Priority inversion on occupying semaphores

7.4.2 Deadlocks

Another typical problem of common synchronisation mechanisms, such as the use of semaphores, is the problem of deadlocks. In this case deadlock means the impossibility of task execution due to infinite waiting for mutually locked resources.

The following scenario results in a deadlock (see Figure 7–2):
Task T1 occupies the semaphore S1 and subsequently cannot continue running, e.g. because it is waiting for an event. Thus, the lower-priority task T2 is transferred into the running state. It occupies the semaphore S2. If T1 gets ready again and tries to occupy semaphore S2, it enters the waiting state again. If now T2 tries to occupy semaphore S1, this results in a deadlock.

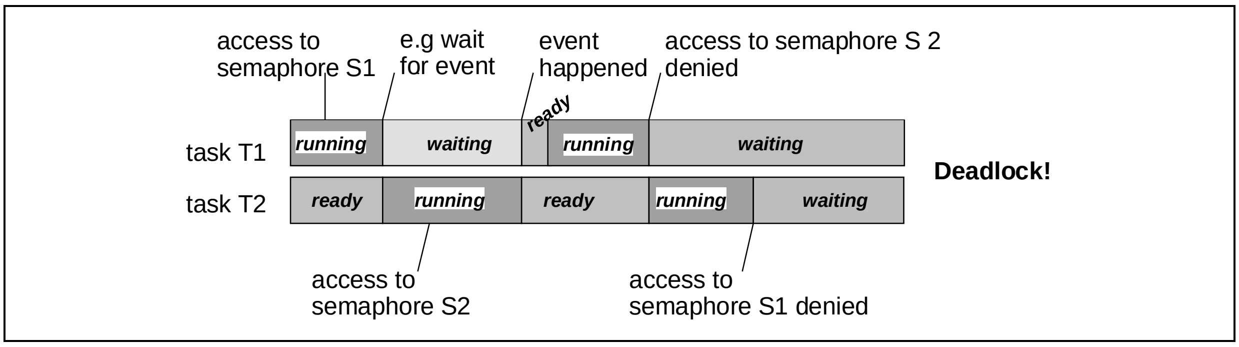


Figure 7–2 Deadlock situation using semaphores

7.5 OSEK Priority Ceiling Protocol

To avoid the problems of priority inversion and deadlocks the OSEK operating system requires following behaviour:

- At the system generation, to each resource its own ceiling priority will be statically assigned.
  The ceiling priority will be set at least to the highest priority of all tasks that access a resource. The ceiling priority must be lower than the lowest priority of all tasks that do not access the resource, and which have priorities higher than the highest priority of all tasks that access the resource.
- If a task requires a resource, and its current priority is lower than the ceiling priority of the resource, the priority of the task will be raised to the ceiling priority of the resource.
- If the task releases the resource, the priority of this task will be reset to the priority which was dynamically assigned before requiring that resource.

Priority ceiling results in a possible time delay for tasks with priorities equal or below the resource priority. This delay is limited by the maximum time the resource is occupied by any lower priority task.

Tasks which might occupy the same resource as the running task do not enter the *running* state, due to their lower or equal priority than the running task. If a resource occupied by a task is released, other task which might occupy the resource can enter the *running* state. For pre-emptive tasks this is a point of rescheduling.

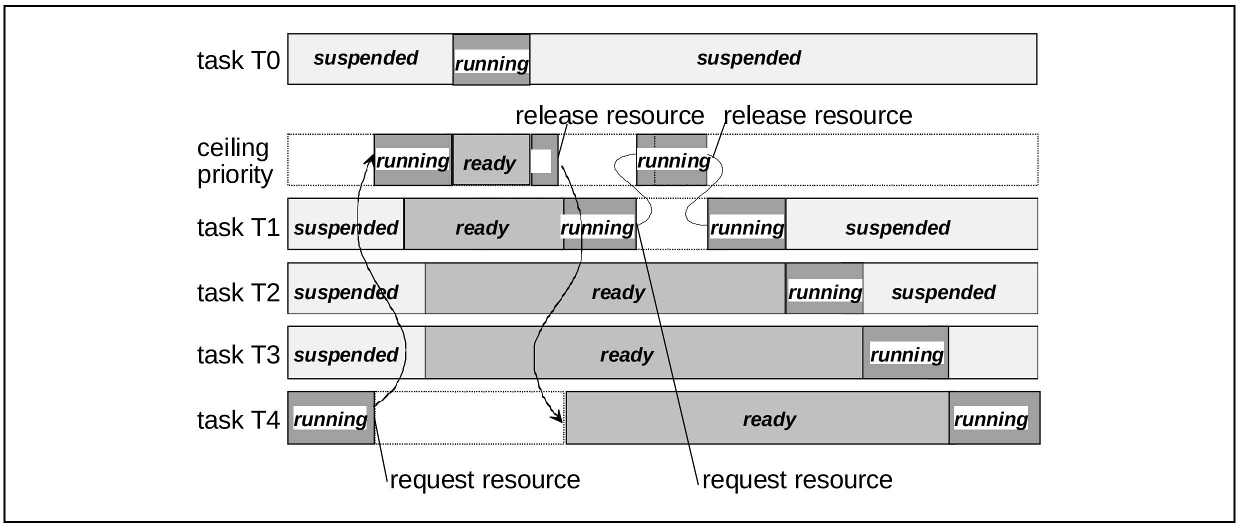


Figure 7–3 Resource assignment with priority ceiling between pre-emptive tasks.

The example shown in Figure 7–3 illustrates the mechanism of the priority ceiling. Task T0 has the highest, and task T4 the lowest priority. Task T1 and task T4 want to access the same resource. The system shows clearly that no unbounded priority inversion is entailed. The high-priority task T1 waits for a shorter time than the maximum duration of resource occupation by T4.

7.6 OSEK Priority Ceiling Protocol with extensions for interrupt levels

The extension of resource management to interrupt level is optional.

To determine the ceiling priority of resources which are used in interrupts, virtual priorities higher than all tasks priorities are assigned to interrupts. The calculated ceiling priority means for a resource which is only occupied by tasks a different handling than for a resource occupied by tasks and interrupt routines. The manipulation of software priorities and of hardware interrupt levels is up to the implementation.

- At the system generation, to each resource its own ceiling priority will be statically assigned.
  The ceiling priority will be set at least to the highest priority of all tasks and interrupt routines that access a resource. The ceiling priority must be lower than the lowest priority of all tasks or interrupt routines that do not access the resource, and which have at the same time higher priorities than the highest priority of all tasks or interrupt routines that access the resource.
- If a task or interrupt routine requires a resource, and its current priority is lower than the ceiling priority of the resource, the priority of the task or interrupt will be raised to the ceiling priority of the resource.
- If the task or interrupt routine releases the resource, the priority of this task or interrupt will be reset to the priority which was dynamically assigned before requiring that resource.

Tasks or interrupt routines which might occupy the same resource as the running task or interrupt routine has occupied do not run , due to their lower or equal priority than the running task or interrupt routine. If a resource occupied by a task is released, another task or interrupt routines which might occupy the resource could run. For pre-emptive tasks this is a point of rescheduling.

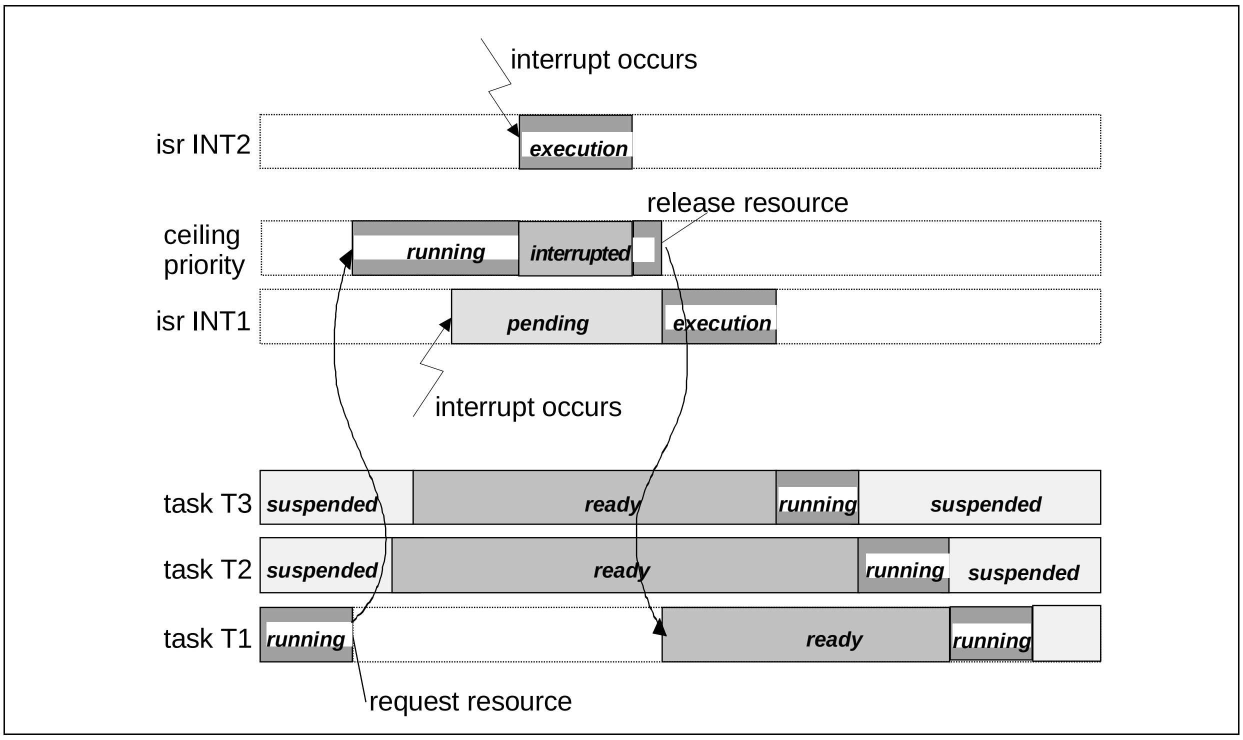


Figure 7-4 Resource assignment with priority ceiling between pre-emptive tasks and interrupt services routines.

The example shown in figure 7-4 describes the following scenario:

The pre-emptive task T1 is running and requests a resource shared with the interrupt service routine INT1. The task T1 activates the higher prior tasks T2 and T3. Because of OSEK Priority Ceiling Protocol the task T1 is still running. Interrupt INT1 occurs. Because of OSEK Priority Ceiling Protocol the task T1 is still running, the interrupt INT1 is pending. Interrupt INT2 occurs. The interrupt service routine INT2 interrupts the task T1 and it is executed. After INT2 is done the task T1 is continued. The task T1 releases the resource. The interrupt service routine INT1 is executed, the task T1 is interrupted. After INT1 is done the Task3 is running. After termination of task T3 the task T2 is running. After termination of task T2 the task T1 is continued.

The example below shown in figure 7-5 describes the following scenario:

The pre-emptive task T1 is running. The interrupt INT1 occurs. The task T1 is interrupted and the interrupt service routine INT1 is executed. The INT1 requests a resource shared with the interrupt service routine INT2. The higher prior interrupt INT2 occurs. Because of OSEK Priority Ceiling Protocol the INT1 is still executed, the INT2 is pending. The interrupt INT3 occurs. Because of higher priority than the INT1, the INT3 interrupts this interrupt service routine and is executed. The INT3 activates the task T2. After the INT3 is done the INT1 is continued. After the INT1 releases the requested resource the INT2 is executed because of higher priority than the INT1. After the INT2 is done the INT1 is continued. After the INT1 is done the task T2 is running because of higher priority than the task T1, the task T1 is ready. After the task T2 is terminated the task T1 is continued.

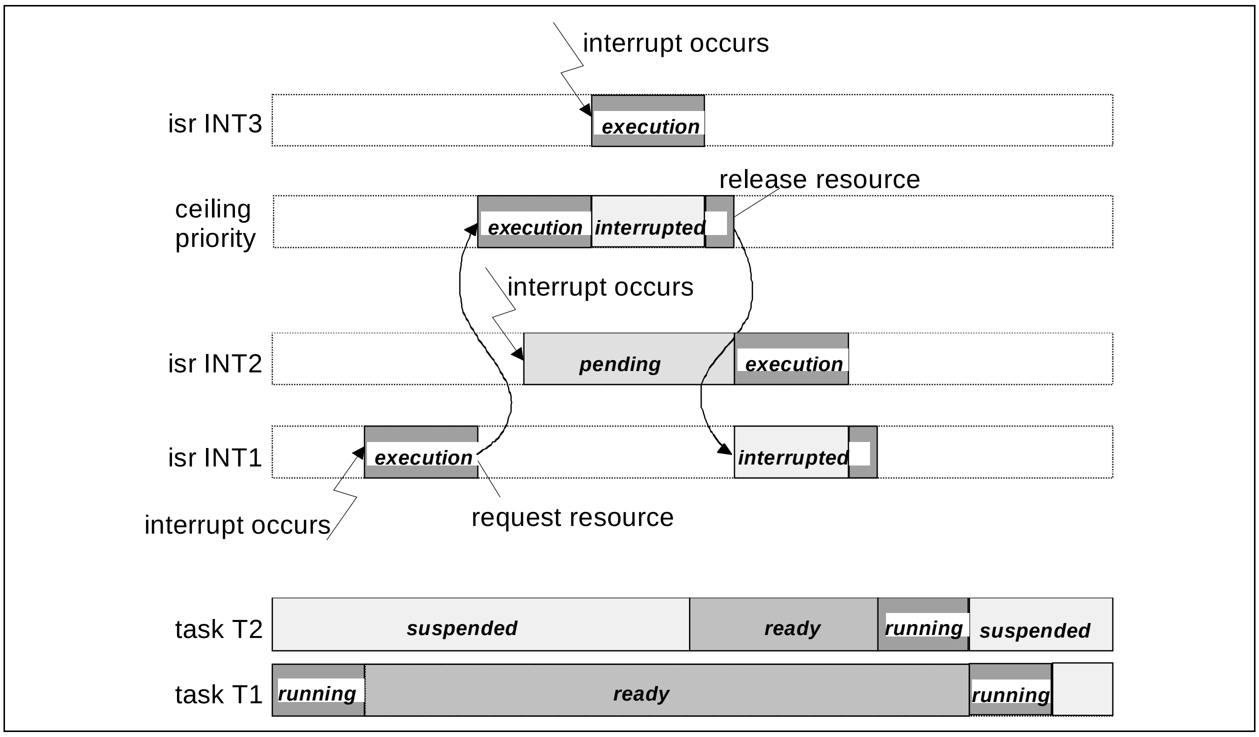


Figure 7-5 Resource assignment with priority ceiling between interrupt services routines

8 Alarms

The OSEK operating system provides services for processing recurring events. Such events may be for example timers which provide an interrupt at regular intervals, or encoders at axles which generate an interrupt in case of a constant change of a (camshaft or crankshaft) angle, or other regular application specific triggers.

The OSEK operating system provides a two-stage concept to process such events. The recurring events (sources) are registered by implementation specific counters. Based on counters, the OSEK operating system software offers alarm mechanisms to the application software.

8.1 Counters

A counter is represented by a counter value, measured in ”ticks”, and some counter specific constants.

The OSEK operating system does not provide a standardised API to manipulate counters directly.

The OSEK operating system takes care of the necessary actions of managing alarms when a counter is advanced and how the counter is advanced.

The OSEK operating system offers at least one counter which is derived from a (hardware or software) timer. The user can assume the existence of this counter.

8.2 Alarm management

The OSEK operating system provides services to activate tasks or set events when an alarm expires. An alarm will expire when a predefined counter value is reached. This counter value can be defined relative to the actual counter value (⇨ relative alarm) or as an absolute value (⇨ absolute alarm). Alarms can be defined to be either single alarms or cyclic alarms. Alarms may be for example the receipt of a number of timer interrupts, a specific angular position, or receiving a message. In addition the OS provides services to cancel alarms and to get the current state of an alarm.

More than one alarm can be attached to a counter.

An alarm is statically assigned at system generation time to:

- one counter
- one task

Depending on configuration this task will be activated, or an event will be set for this task when the alarm expires. Task activation and event setting when an alarm expires have the same properties as normal task activation and event setting.

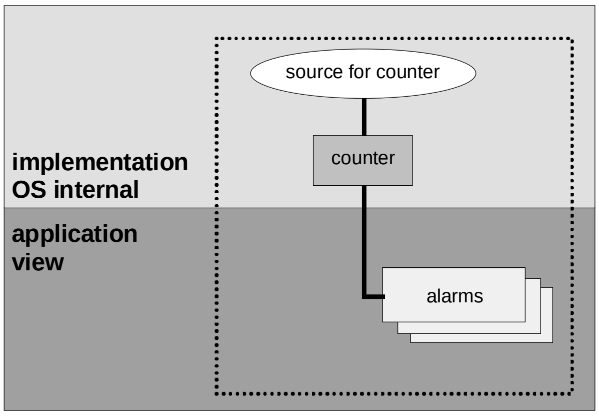


Figure 8–1 Layered model of alarm management

Counters and alarms are defined statically. The assignment of alarms to counters, as well as the action to be performed when an alarm expires, are defined statically, too.

Dynamic parameters are the counter value when an alarm has to expire, and the period for cyclic alarms.

9 Messages

For an OSEK implementation to be compliant, message handling for intra processor communication has to be offered. The minimum functionality required is CCCA as described in the OSEK COM specification. CCCA describes a communication conformance class specifically tailored to the needs of intra processor communication which supports unqueued messages. CCCB defines an extension which adds queued messages.

If an implementation offers even more functionality which is specified in other conformance classes described in the OSEK COM specification, the implementation must stick to syntax and semantic of the respective OSEK COM functionality.

Please note that for messages the rules stated in the OSEK COM specification are valid. For example, OSEK COM system interfaces do not call *ErrorHook*. However, if the OSEK COM functionality internally calls OS system function like *ActivateTask*, *ErrorHook* will be called if necessary from *ActivateTask*. For more details, refer to the OSEK COM specification.

10 Error handling, tracing and debugging

10.1 Hook routines

The OSEK operating system provides system specific hook routines to allow user-defined actions within the OS internal processing. The first parameter is fixed for all implementations of OSEK operating systems, additional parameters are optional and implementation dependent.

Those hook routines are

- called by the operating system, in a special context depending on the implementation of the operating system
- higher prior than all tasks
- not interrupted by category 2 and 3 interrupt routines
- using an implementation dependent calling interface.
- part of the operating system
- implemented by the user with user defined functionality
- standardised in interface per OSEK OS implementation, but not standardised in functionality (environment and behaviour of the hook routine itself), therefore usually hook routines are not portable
- are only allowed to use a subset of API functions
- optional (the implementation should omit calls to hook routines which do not exist)

In the OSEK operating system hook routines may be used for:

- system start-up (see chapter 10.3, System start-up).
  The corresponding hook routine (*StartupHook*) is called after the operating system start-up and before the scheduler is running.
- system shutdown (see chapter 10.4, System shutdown).
  The corresponding hook routine (*ShutdownHook*) is called when a system shutdown is requested by the application or by the operating system in case of a severe error.
- tracing or application dependent debugging purposes as well as user defined extensions of the context switch (see chapter 10.5, Debugging).
- error handling.

Each implementation of OSEK has to describe the interfaces and conventions for the hook routines.

If the application calls a not allowed API service in hook routines the behaviour is not defined. If an error is raised, the implementation should return an implementation specific error code.

Service	*Error Hook*	*PreTask Hook*	*PostTask Hook*	*Startup Hook*	*Shutdown Hook*
ActivateTask	--	--	--	**allowed**	--
TerminateTask	--	--	--	--	--
ChainTask	--	--	--	--	--
Schedule	--	--	--	--	--
GetTaskID	**allowed**[7]	**allowed**	**allowed**	--	--
GetTaskState	**allowed**	**allowed**	**allowed**	--	--
EnterISR	--	--	--	--	--
LeaveISR	--	--	--	--	--
EnableInterrupt	--	--	--	--	--
DisableInterrupt	--	--	--	--	--
GetInterruptDescriptor	**allowed**	**allowed**	**allowed**	--	--
DisableAllInterrupts	--	--	--	--	--
EnableAllInterrupts	--	--	--	--	--
SuspendOSInterrupts	--	--	--	--	--
ResumeOSInterrupts	--	--	--	--	--
GetResource	--	--	--	--	--
ReleaseResource	--	--	--	--	--
SetEvent	--	--	--	--	--
ClearEvent	--	--	--	--	--
GetEvent	**allowed**	**allowed**	**allowed**	--	--
WaitEvent	--	--	--	--	--
GetAlarmBase	**allowed**	**allowed**	**allowed**	--	--
GetAlarm	**allowed**	**allowed**	**allowed**	--	--
SetRelAlarm	--	--	--	--	--
SetAbsAlarm	--	--	--	--	--
CancelAlarm	--	--	--	--	--
GetActiveApplicationMode	**allowed**	**allowed**	**allowed**	**allowed**	**allowed**
StartOS	--	--	--	--	--
ShutdownOS	**allowed**	--	--	**allowed**	--

Figure 10–1 API services for hook routines

Most operating system services are not allowed for hook routines. This restriction is necessary to reduce system complexity.

10.2 Error handling

An error service is provided to handle temporarily and permanently occurring errors within the OSEK operating system. Its basic framework is predefined and has to be completed by the user. This gives the user a choice of efficient centralised or decentralised error handling.

[7] It may happen that currently no task is *running*. In this case the service returns the task ID *INVALID_TASK* (see chapter 12.2.3.5 GetTaskID).

Two different kinds of errors are distinguished:

- **Application errors**
  The operating system could not execute the requested service correctly, but assumes the correctness of its internal data.
  In this case, centralised error treatment is called. Additionally the operating system returns the error by the status information for decentralised error treatment. It is up to the user to decide what to do depending on which error has occured.
- **Fatal errors**
  The operating system can no longer assume correctness of its internal data.
  In this case the operating system calls the centralised system shutdown.

All those error services are assigned with a parameter that specifies the error.

The return value of the OSEK API-services has precedence over the output parameters. If an API service returns an error, the values of the output parameters are undefined.

The corresponding hook routine (*ErrorHook*) is called if a system service returns a StatusType value not equal to E_OK. The hook routine *ErrorHook* is not called if a system service is called from the *ErrorHook* itself (i.e., a recursive call of error hook never occurs). Any possibly occuring error by calling system services from the *ErrorHook* can only be detected by evaluating the return value.

ErrorHook also is called if an error is detected during task activation or event setting, for example upon alarm expiration or message arrival.

If a task is activated in the version with standard status, only "E_OK" is returned. Moreover, in a version with extended status, the additional return values "Task is invalid" or "Too many task activations", etc. can be returned. These extended return values must no longer occur in the target application at the time of execution, i.e. the corresponding errors are not intercepted in the run time version of the operating system.

10.3 System start-up

Initialisation after a processor reset is up to the implementation, but OSEK OS offers support for a standardised way of initialisation.

Interfaces for initialisation of hardware, operating system and application have to be clearly defined by the implementation.

OSEK OS does not force the application to define special tasks which have to be started after the operating system initialisation, but it allows the user to specify autostart-tasks during system generation.

After a reset of the CPU, hardware-specific application software is executed (no operating system context). The non-portable section ends with the detection of the application mode. For safety reasons this detection should not rely on system history.

In case of a system where OSEK OS and OSEKtime OS coexist (not reflected in Figure 10–2), the OSEKtime initialisation will always run first, and the remaining parts of the OSEK initialisation will be performed after OSEKtime enters the idle loop, which will cause OSEKtime to automatically call StartOS with the application mode already passed to OSEKtime as parameter.

Otherwise, the portable section of the application starts with the call to a function which starts up the operating system, i.e. *StartOS* with the application mode as a parameter. After the operating system is initialised (scheduler is not running), it calls the hook routine *StartupHook*, where the user can place the initialisation code for all his operating system dependent initialisation. In order to structure the initialisation code in *StartupHook* according to the started application mode, the service *GetActiveApplicationMode* is provided. After the return from that hook routine the operating systems enables the interrupts according to the *INITIAL_INTERRUPT_DESCRIPTOR*[8], and starts the scheduler. After that the system is running and executes user tasks.

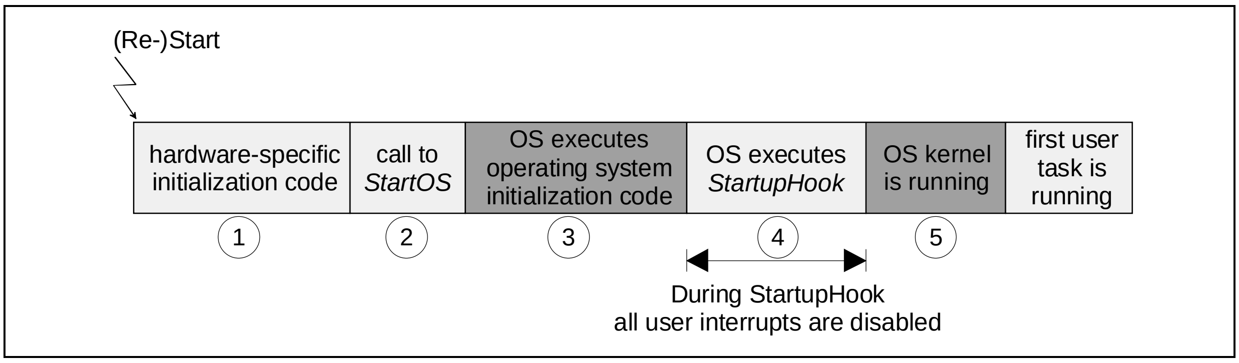

(Re-)Start
hardware-specific initialization code
call to StartOS
OS executes operating system initialization code
OS executes StartupHook
OS kernel is running
first user task is running
1
2
3
4
5
During StartupHook
all user interrupts are disabled

Figure 10–2 System start-up

(1) After a reset, the user is free to execute (non-portable) hardware specific code. Interrupts of category 2 and 3 are not allowed to run until the phase 5. The non-portable section ends by detection of the application mode.
(2) Call *StartOS* with the application mode as a parameter. This call starts the operating system (of OSEKtime is present, this is done automatically).
(3) The operating system performs internal start-up functions and
(4) calls the hook routine *StartupHook*, where the user may place initialisation procedures. During this hook routine, all user interrupts are disabled.
(5) The operating system enables user interrupts according to the *INITIAL_INTERRUPT_DESCRIPTOR*, and starts the scheduling activity. The *INITIAL_INTERRUPT_DESCRIPTOR* is statically assigned by the user.

10.4 System shutdown

The OSEK OS specification defines a service to shut down the operating system, *ShutdownOS*.

This service can be requested by the application or by the operating system due to a fatal error.

When *ShutdownOS* is called the operating system will call the hook routine *ShutdownHook* and shut down afterwards.

The user is usually free to define any system behaviour in *ShutdownHook* e.g. not to return from the routine. (See chapter 12.7.2.3, ShutdownOS). However, in case of a system where OSEK OS coexists with OSEKtime OS, there are restrictions with respect to functionality which may be performed in *ShutdownHook*. It is possible that only OSEK OS is shut down, whereas OSEKtime OS remains intact. Consequently, I/O devices which are handled within OSEKtime must not be reset in *ShutdownHook*, and *ShutdownHook* must return.

[8] The value of the *INITIAL_INTERRUPT_DESCRIPTOR* is defined by the user or by the implementation.

10.5 Debugging

Two hook routines (*PreTaskHook* and *PostTaskHook*) are called on task context switches.

These two hook routines may be used for debugging or time measurement (including context switch time). Therefore *PostTaskHook* is called after leaving the context of the old task, *PreTaskHook* is called before entering the context of a new task. However, the task is already/still in the running state, and GetTaskId will not return INVALID_TASK.

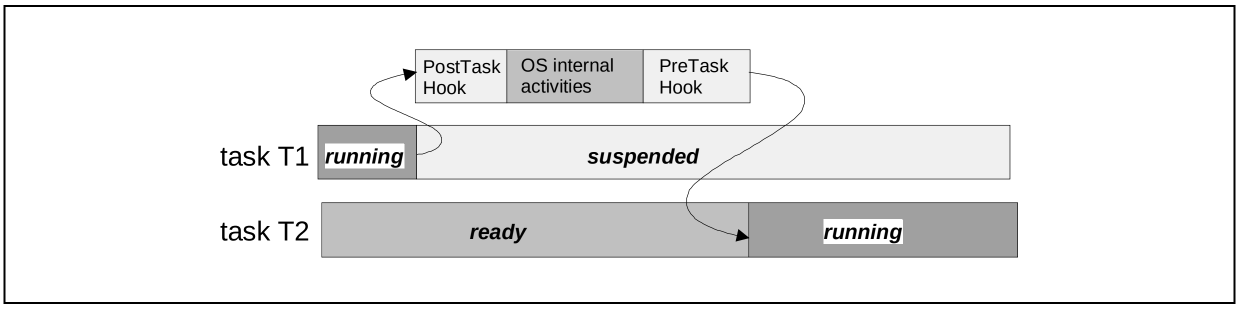


Figure 10–3 PreTaskHook and PostTaskHook

When *ShutdownOS* is called while a task is running *ShutdownOS* may or may not call *PostTaskHook.* If *PostTaskHook* is called it is undefined if it is called before or after *ShutdownHook.*

11 Description of system services

11.1 Definition of system objects

Within the OSEK operating system all system objects have to be determined statically by the user. The definition of the operating system objects is provided by the operating system supplier. The actual creation of the objects (unique names and specific characteristics) is done during the system generation phase. The declarations done in the application source are external references to those operating system objects. There are no system services available to dynamically create system objects. Declarations provide information that a system object is to be used which has been created at another location. The names are used as identifications within the system services.

Usually the scope of those names is like an external variable in C-language.

The creation of system objects within the source should be considered as an exception, due to loss of portability.

Internal representation of system objects is implementation specific. There are various alternatives for implementation of system objects. For example, a *TaskType* could be implemented either as a pointer to the data structure of the task or as an index to the corresponding list element. Application programmers cannot assume a specific representation.

The creation of system objects may require additional tools. They enable the user to add or to modify values which have been specified in definitions. Consequently, the system generation and the tools used to this effect are also implementation-specific.

11.2 Conventions

11.2.1 Type of calls

The system service interface is ISO/ANSI-C. Its implementation is normally a function call, but may also be solved differently, as required by the implementation - for example by macros of the C pre-processor. A specific type of implementation cannot be assumed.

11.2.2 Legitimacy of calls

System services are called from tasks, interrupt service routines, and hook routines. Depending on the system service, there may be restrictions regarding the availability. Further restrictions are imposed by the conformance classes.

11.2.3 Error characteristics

To keep the system efficient and fast, the OSEK operating system does not test all errors. If the application uses operating system services incorrectly, undefined system behaviour may result.

Most system services return a status to the user. The return status is E_OK if it was possible to execute the system service without any restrictions. If the system recognises an exceptional condition which restricts execution of the system service, a different status is returned.

A status other than E_OK may be information which is not considered to be an error ("warning"). An example is the return status of the system service *CancelAlarm*, which informs that the alarm to be cancelled has already expired. A user program is thus informed that e.g. a task activation has taken place which was not wanted. The detection of mild errors (warnings) is part of the system services.

If it is possible to exclude errors before run time, the run time version may omit checking of these errors. If the only possible return status is E_OK, the implementation is free not to return a status.

All return values of a system service are listed under the individual descriptions. The return status distinguishes between the ”standard” and ”extended” status. The ”standard” version fulfils the requirements of a debugged application system as described before. The "extended" version is considered to support testing of not yet fully debugged applications. It comprises extended error checking compared to the standard version.

The sequence of error checking within the operating system is not specified. Whenever multiple errors occur, it is implementation dependent which status is returned to the application.

In case of application errors, the OSEK operating system will call the hook routine *ErrorHook* if defined. The purpose of *ErrorHook* is to treat status information centralised.

In case of fatal errors, the system service does not return to the application, but activates *ShutdownOS*. An example is a non-detected incorrect parameter of a system service which generates an inconsistency in the system. The parameter passed to *ShutdownOS* is an implementation dependent system error code. System error codes occupy a range of numbers of their own and do not conflict with the states of the operating system services.

The functionality of *ShutdownOS* is implementation-specific. Possible implementations are to stop the application or to issue an assertion. The application itself can access *ShutdownOS* to shut down the operating system in a controlled fashion.

Calling of *ShutdownOS* is also recommended when processing non-assignable errors, for example "illegal instruction code". This is not mandatory because hardware support is necessary, which cannot be taken for granted.

12 Specification of operating system services

Structure of the description

Operating system services are arranged in logical groups. A coherent description is provided for all services of the task management, the interrupt management, etc.

The description of each logical group starts with data type definitions. A description of the group-specific constructional elements and system services follows. The last items are a description of constants, and of any additional conventions.

Constructional elements

The description of constructional elements contains the following fields:

Syntax:	Interface in C-like syntax.
Parameter (In):	List of all input parameters.
Description:	Explanation of the constructional element.
Particularities:	Explanation of restrictions relating to the utilisation.
Conformance:	Specifies the conformance classes where the constructional element is provided.

Service description

A service description contains the following fields:

Syntax:	Interface in C-like syntax.
Parameter (In):	List of all input parameters.
Parameter (Out):	List of all output parameters.
Description:	Explanation of the functionality of the operating system service.
Particularities:	Explanation of restrictions relating to the utilisation of the operating system service.
Status:	List of possible return values.
Standard:	• List of return values provided in the operating system's standard version. Special case: Service does not return.
Extended:	• List of additional return values in the operating system's extended version.
Conformance:	Specifies the conformance classes where the operating system service is provided.

The specification of operating system services uses the following naming conventions for data types:

...Type:	describes the values of individual data (including pointers).
...RefType:	describes a pointer to the ...Type (for call by reference).

12.1 Common datatypes

StatusType

This data type is used for all status information the API services offer. Naming convention: all errors for API services start with E_. Those reserved for the operating system will begin with E_OS_.

The normal return value is E_OK which is associated with the value 0.

The following error values are defined:

All errors of API services:

E_OS_ACCESS	= 1,
E_OS_CALLEVEL	= 2,
E_OS_ID	= 3,
E_OS_LIMIT	= 4,
E_OS_NOFUNC	= 5,
E_OS_RESOURCE	= 6,
E_OS_STATE	= 7,
E_OS_VALUE	= 8

If the only possible return status is E_OK, the implementation is free not to return a status, this is not separately stated in the description of the individual services.

Internal errors of the operating system:

These errors are implementation specific and not part of the portable section. The error names reside in the same name-space as the errors for API services mentioned above, i.e. the range of numbers must not overlap.

To show the difference in use, the names internal errors must start with E_OS_SYS_

Examples:

- E_OS_SYS_STACK
- E_OS_SYS_PARITY
- ... and other implementation-specific errors, which have to be described in the vendor-specific document.

The names and range of numbers of the internal errors of the OSEK operating system do not overlap the names and range of numbers of other OSEK services (i.e. communication and network management) or the range of numbers of the API error values.

12.2 Task management

12.2.1 Data types

TaskType

This data type identifies a task.

TaskRefType

This data type points to a variable of TaskType.

TaskStateType

This data type identifies the state of a task.

TaskStateRefType

This data type points to a variable of the data type TaskStateType.

12.2.2 Constructional elements

12.2.2.1 DeclareTask

Syntax:	DeclareTask (TaskIdentifier)
Parameter (In):	
-	Task identifier (C-identifier)
Description:	*DeclareTask* serves as an external declaration of a task. The function and use of this service are similar to that of the external declaration of variables.
Particularities:	-
Conformance:	BCC1, BCC2, ECC1, ECC2

12.2.3 System services

12.2.3.1 ActivateTask

Syntax:	StatusType ActivateTask (TaskType <TaskID>)
Parameter (In):	
TaskID	Task reference
Parameter (Out):	none
Description:	The task <TaskID> is transferred from the *suspended* state into the *ready* state[9]. The operating system ensures that the task code is being executed from the first statement.
Particularities:	The service may be called from interrupt level, from task level and the hook routine *StartupHook* (see Figure 10–1). Rescheduling after the call to *ActivateTask* depends on the place it is called from (ISR, non-preemptive task, preemptive task). If E_OS_LIMIT is returned the activation is ignored. When an extended task is transferred from suspended state into ready state all its events are cleared.
Status:	
Standard:	• No error, E_OK
Extended:	• Task <TaskID> is invalid, E_OS_ID • Too many task activations of <TaskID>, E_OS_LIMIT
Conformance:	BCC1, BCC2, ECC1, ECC2

[9] ActivateTask will not immediately change the state of the task in case of multiple activation requests. If the task is not suspended, the activation will only be recorded and performed later.

12.2.3.2 TerminateTask

Syntax:	StatusType TerminateTask (void)
Parameter (In):	none
Parameter (Out):	none
Description:	This service causes the termination of the calling task. The calling task is transferred from the *running* state into the *suspended* state[10].
Particularities:	The resources occupied by the task must have been released before the call to *TerminateTask*. If the resource is still occupied in standard status the behaviour is undefined. If the call was successful, *TerminateTask* does not return to the call level and the status can not be evaluated. If the version with extended status is used, the service returns in case of error, and provides a status which can be evaluated in the application. If the service *TerminateTask* is called successfully, it enforces a rescheduling. Ending a task function without call to *TerminateTask* or *ChainTask* is strictly forbidden and may leave the system in an undefined state.
Status:	
Standard:	No return to call level
Extended:	• Task still occupies resources, E_OS_RESOURCE • Call at interrupt level, E_OS_CALLEVEL
Conformance:	BCC1, BCC2, ECC1, ECC2

12.2.3.3 ChainTask

Syntax:	StatusType ChainTask (TaskType <TaskID>)
Parameter (In):	
TaskID	Reference to the sequential succeeding task to be activated.
Parameter (Out):	none
Description:	This service causes the termination of the calling task. After termination of the calling task a succeeding task <TaskID> is activated. Using this service, it ensures that the succeeding task starts to run at the earliest after the calling task has been terminated.
Particularities:	If the succeeding task is identical with the current task, this does not result in multiple requests. The task is not transfered to the suspended state. The resources occupied by the calling task must have been released before *ChainTask* is called. If the resource is still occupied in standard status the behaviour is undefined.

[10] In case of tasks with multiple activation requests, terminating the current instance of the task automatically puts the next instance of the same task into the *ready* state.

	If called successfully, *ChainTask* does not return to the call level and the status can not be evaluated.
	If the version with extended status is used, the service returns in case of error to the calling task, and provides a status which can then be evaluated in the application.
	If the service *ChainTask* is called successfully, this enforces a rescheduling.
	Ending a task function without call to *TerminateTask* or *ChainTask* is strictly forbidden and may leave the system in an undefined state.
	If E_OS_LIMIT is returned the activation is ignored.
	When an extended task is transferred from suspended state into ready state all its events are cleared.
Status:	
Standard:	• No return to call level
Extended:	• Task <TaskID> is invalid, E_OS_ID • Too many task activations of <TaskID>, E_OS_LIMIT • Calling task still occupies resources, E_OS_RESOURCE • Call at interrupt level, E_OS_CALLEVEL
Conformance:	BCC1, BCC2, ECC1, ECC2

12.2.3.4 Schedule

Syntax:	StatusType Schedule (void)
Parameter (In):	none
Parameter (Out):	none
Description:	If a higher-priority task is *ready*, the current task is put into the *ready* state, its context is saved and the higher-priority task is executed. Otherwise the calling task is continued.
Particularities:	In non pre-emptive tasks *Schedule* enables a processor assignment to other tasks in application-specific locations.
	This service has no influence on full pre-emptive tasks.
Status:	
Standard:	• No error, E_OK
Extended:	• Call at interrupt level, E_OS_CALLEVEL
Conformance:	BCC1, BCC2, ECC1, ECC2

12.2.3.5 GetTaskID

Syntax:	StatusType GetTaskID (TaskRefType <TaskID>)
Parameter (In):	none
Parameter (Out):	
TaskID	Reference to the task which is currently *running*
Description:	*GetTaskID* returns the information about the TaskID of the task which is currently *running*.

Particularities:		Allowed on task level, ISR level and in several hook routines (see Figure 10–1). This service is intended to be used by library functions and hook routines. If <TaskID> can't be evaluated (no task currently *running*), the service returns INVALID_TASK as TaskType.
Status:		
Standard:	•	No error, E_OK
Extended:	•	No error, E_OK
Conformance:		BCC1, BCC2, ECC1, ECC2

12.2.3.6 GetTaskState

Syntax:		StatusType GetTaskState (TaskType <TaskID>, TaskStateRefType <State>)
Parameter (In):		
TaskID		Task reference
Parameter (Out):		
State		Reference to the state of the task <TaskID>
Description:		Returns the state of a task (*running*, *ready*, *waiting*, *suspended*) at the time of calling *GetTaskState*.
Particularities:		The service may be called from interrupt service routines, task level, and some hook routines (see Figure 10–1). Within a full pre-emptive system, calling this operating system service only provides a meaningful result if the task runs in an interrupt disabling state at the time of calling. When a call is made from a task in a full pre-emptive system, the result may already be incorrect at the time of evaluation. When the service is called for a task, which is multiply activated, the state is set to *running* if any instance of the task is running.
Status:		
Standard:	•	No error, E_OK
Extended:	•	Task <TaskID> is invalid, E_OS_ID
Conformance:		BCC1, BCC2, ECC1, ECC2

12.2.4 Constants

RUNNING	•	Constant of data type TaskStateType for task state *running*.
WAITING	•	Constant of data type TaskStateType for task state *waiting*.
READY	•	Constant of data type TaskStateType for task state *ready*.
SUSPENDED	•	Constant of data type TaskStateType for task state *suspended*.
INVALID_TASK	•	Constant of data type TaskType for a not defined task.

12.2.5 Naming convention

The operation system must be able to assign the entry address of the task function to the name of the corresponding task for identification. With the entry address the operating system is able to call the task.

Within the application, a task is defined according to the following pattern:

```
        TASK (TaskName)
        {
        }
```

With the macro `TASK` the user may use the same name for "task identification" and "name of task function".

The task identification will be generated from the `TaskName` during system generation time.[11]

12.3 Interrupt handling

12.3.1 Data types

IntDescriptorType

Data type for logical interrupt masks.

IntDescriptorRefType

Reference to the logical interrupt mask, this data type usually is implemented as "pointer to IntDescriptorType".

12.3.2 System services

12.3.2.1 EnterISR

Syntax:	void EnterISR (void)
Parameter (In):	none
Parameter (Out):	none
Description:	*EnterISR* establishes the conditions needed to request OS services in an interrupt service routine category 3 (see particularities). Inside *EnterISR* the following functions are executed if needed: • Registration of the switching to the interrupt level inside the operating system. • Switch of the current context (e.g. to the ISR stack).
Particularities:	*EnterISR* establishes in ISRs category 3 the possibility to use operating system services. It is necessary to place *EnterISR* before the first call of an operating system service. The detailed implementation of *EnterISR* depends on the target system. It is explicitly allowed to use system specific variations.

[11] The pre-processor could for example generate the name of the task function by using the pre-processor symbol sequence ## to add a string „Func“ to the task name:

```
 #define TASK(TaskName) StatusType Func ## TaskName(void)
```

With this macro, `TASK(MyTask)` has the entry function `FuncMyTask`

	The call to this service is only allowed in ISRs category 3, but the specification does not force an error status. For example some microcontrollers can not perform the test "called outside from ISR". But a system analysis tool may check whether the call is performed within task level.
	This service is a counterpart of LeaveISR service (see Chapter 5).
Status:	
Standard:	none
Extended:	none
Conformance:	BCC1, BCC2, ECC1, ECC2

12.3.2.2 LeaveISR

Syntax:	void LeaveISR (void)
Parameter (In):	none
Parameter (Out):	none
Description:	*LeaveISR* is the counterpart of *EnterISR* and resets the conditions to request operating system services in an ISR category 3. *LeaveISR* may only be called after *EnterISR* has been called.
	This function does not imply the return from ISR although it has to be the last statement executed in the ISR.
Particularities:	The call to this service is only allowed in ISRs category 3.
	The detailed implementation of *LeaveISR* depends on the target system. It is explicitly allowed to use system specific variations.
Status:	
Standard:	none
Extended:	none
Conformance:	BCC1, BCC2, ECC1, ECC2

12.3.2.3 EnableInterrupt

Syntax:	StatusType EnableInterrupt (IntDescriptorType <Descriptor>)
Parameter (In):	
Descriptor	Hardware dependent parameter for selections of interrupt sources to enable. In <Descriptor>, a "1" means "enable".
Parameter (Out):	none
Description:	This service allows enabling of several interrupt sources simultaneously.
Particularities:	The service may be called from an ISR and from the task level, but not from hook routines.
	To save the current state of interrupt sources the application must use *GetInterruptDescriptor* before.
	The implementation has to adapt this service to the target hardware.

	If not all requested interrupt sources are disabled, this service is nevertheless executed for the disabled interrupt sources and returns E_OS_NOFUNC in Extended Status.
Status:	
Standard:	• No error, E_OK
Extended:	• At least one of the interrupt sources was not disabled, E_OS_NOFUNC
Conformance:	BCC1, BCC2, ECC1, ECC2

12.3.2.4 DisableInterrupt

Syntax:	StatusType DisableInterrupt (IntDescriptorType <Descriptor>)
Parameter (In):	
Descriptor	Hardware dependent parameter for selections of interrupt sources to disable. In <Descriptor>, a "1" means "disable".
Parameter (Out):	none
Description:	This service allows disabling of several interrupt sources simultaneously.
Particularities:	The service may be called from an ISR and from the task level, but not from hook routines. To save the current state of interrupt sources the application must use *GetInterruptDescriptor* before. The implementation has to adapt this service to the target hardware. If not all requested interrupt sources are enabled, this service is nevertheless executed for the enabled interrupt sources and returns E_OS_NOFUNC in Extended Status.
Status:	
Standard:	• No error, E_OK
Extended:	• At least one interrupt source was not enabled, E_OS_NOFUNC
Conformance	BCC1, BCC2, ECC1, ECC2

12.3.2.5 GetInterruptDescriptor

Syntax:	StatusType GetInterruptDescriptor (IntDescriptorRefType <Descriptor>)
Parameter (In):	none
Parameter (Out):	
Descriptor	Reference to current status of interrupt sources. In <Descriptor> all interrupt sources, which are enabled, are marked by "1", “0” otherwise.
Description:	Query of interrupt status
Particularities:	The service may be called from an ISR, task level, and some hook routines (see Figure 10–1). The implementation has to adapt this service to the target hardware.

Status:

Standard:	•	No error, E_OK
Extended:	•	none

Conformance: BCC1, BCC2, ECC1, ECC2

12.3.2.6 EnableAllInterrupts

Syntax: void EnableAllInterrupts (void)

Parameter (In):
Descriptor none

Parameter (Out): none

Description: This service restores the state saved by DisableAllInterrupts.

Particularities: The service may be called from an ISR and from the task level, but not from hook routines.

This service is a counterpart of DisableAllInterrupts service, which must have been called before, and its aim is the completion of the critical section of code. No API service calls are allowed within this critical section.

The implementation should adapt this service to the target hardware providing a minimum overhead. Usually this service enables recognition of interrupts by the central processing unit.

Status:

Standard:	•	none
Extended:	•	none

Conformance: BCC1, BCC2, ECC1, ECC2

12.3.2.7 DisableAllInterrupts

Syntax: void DisableAllInterrupts (void)

Parameter (In):
Descriptor none

Parameter (Out): none

Description: This service allows disabling of all interrupts supported by the hardware. The state before is saved for the EnableAllInterrupts call.

Particularities: The service may be called from an ISR and from the task level, but not from hook routines.

This service is intended to start a critical section of the code. This section must be finished by calling the EnableAllInterrupts service. No API service calls are allowed within this critical section.

The implementation should adapt this service to the target hardware providing a minimum overhead. Usually this service disables recognition of interrupts by the central processing unit.

Note that this service does not support nesting. If nesting is needed for critical sections e.g. for libraries

	SuspendOSInterrupts and ResumeOSInterrupts should be used.
Status:	
Standard:	• none
Extended:	• none
Conformance:	BCC1, BCC2, ECC1, ECC2

12.3.2.8 ResumeOSInterrupts

Syntax:	void ResumeOSInterrupts (void)
Parameter (In):	
Descriptor	none
Parameter (Out):	none
Description:	This service restores the recognition status of interrupts saved by the SuspendOSInterrupts service.
Particularities:	The service may be called from an ISR and from the task level, but not from hook routines. This service is the counterpart of SuspendOSInterrupts service, which must have been called before, and its aim is the completion of the critical section of code. No API service calls beside SupendOSInterrupts/ ResumeOSInterrupts are allowed within this critical section. The implementation should adapt this service to the target hardware providing a minimum overhead. In case of nesting pairs of the calls SuspendOSInterrupts and ResumeOSInterrupts the interrupt recognition status saved by the first call of SuspendOSInterrupts is restored by the last call of the ResumeOSInterrupts service.
Status:	
Standard:	• none
Extended:	• none
Conformance:	BCC1, BCC2, ECC1, ECC2

12.3.2.9 SuspendOSInterrupts

Syntax:	void SuspendOSInterrupts (void)
Parameter (In):	
Descriptor	none
Parameter (Out):	none
Description:	This service saves the recognition status of interrupts of categories 2 and 3 and disables the recognition of these interrupts.
Particularities:	The service may be called from an ISR and from the task level, but not from hook routines. This service is intended to protect a critical section of code. This section must be finished by calling the ResumeOSInterrupts service. No API service calls beside

SupendOSInterrupts/ResumeOSInterrupts are allowed within this critical section.

The implementation should adapt this service to the target hardware providing a minimum overhead.

It is intended only to disable interrupts of category 2 and 3. However if this is not possible in an efficient way more interrupts may be disabled.

Status:

Standard:	•	none
Extended:	•	none

Conformance: BCC1, BCC2, ECC1, ECC2

12.3.3 Constants

`INITIAL_INTERRUPT_DESCRIPTOR`

- Constant of data type IntDescriptorType (see chapter 10.3, System start-up).

12.3.4 Naming convention

Within the application, an interrupt service routine of category 2 is defined according to the following pattern:

```
ISR (FuncName)
{
}
```

The keyword `ISR` is evaluated by the system generation to clearly distinguish between functions and interrupt service routines in the source code.

For category 1 and 3 interrupt service routines no naming conventions are prescribed, their definition is implementation specific.

12.4 Resource management

12.4.1 Data types

ResourceType

Data type for a resource.

12.4.2 Constructional elements

12.4.2.1 DeclareResource

Syntax:	DeclareResource (ResourceIdentifier)
Parameter (In):	
-	Resource identifier (C-identifier)
Description:	*DeclareResource* serves as an external declaration of a resource. The function and use of this service are similar to that of the external declaration of variables.
Particularities:	-
Conformance:	BCC1, BCC2, ECC1, ECC2

12.4.3 System services

12.4.3.1 GetResource

Syntax:	StatusType GetResource (ResourceType <ResID>)
Parameter (In):	
ResID	Reference to resource
Parameter (Out):	none
Description:	This call serves to enter critical sections in the code that are assigned to the resource referenced by <ResID>. A critical section must always be left using *ReleaseResource*.
Particularities:	The OSEK priority ceiling protocol for resource management is described in chapter 7.5.
	Nested resource occupation is only allowed if the inner critical sections are completely executed within the surrounding critical section (strictly stacked, see chapter 7.2, Restrictions when using resources). Nested occupation of one and the same resource is also forbidden!
	Corresponding calls to *GetResource* and *ReleaseResource* should appear within the same function on the same function level.
	Services which put the *running* task into the state *suspended* or *waiting* must not be used in critical sections (i.e. *TerminateTask*, *ChainTask* and *WaitEvent*).
	Generally speaking, critical sections should be short.
	The service may be called from an ISR and from task level (see Figure 10–1).
Status:	
Standard:	• No error, E_OK
Extended:	• Resource <ResID> is invalid, E_OS_ID
	• Attempt to get resource which is already occupied by any task or ISR, or the statically assigned priority of the calling task or interrupt routine is higher than the calculated ceiling priority, E_OS_ACCESS
Conformance:	BCC1, BCC2, ECC1, ECC2

12.4.3.2 ReleaseResource

Syntax:	StatusType ReleaseResource (ResourceType <ResID>)
Parameter (In):	
ResID	Reference to resource
Parameter (Out):	none
Description:	*ReleaseResource* is the counterpart of *GetResource* and serves to leave critical sections in the code that are assigned to the resource referenced by <ResID>.

Particularities: For information on nesting conditions, see particularities of *GetResource*.

The service may be called from an ISR and from task level (see Figure 10–1).

Status:

Standard:
- No error, E_OK

Extended:
- Resource <ResID> is invalid, E_OS_ID
- Attempt to release a resource which is not occupied by any task or ISR, or another resource has to be released before E_OS_NOFUNC
- Attempt to release a resource which has a lower ceiling priority than the statically assigned priority of the calling task or interrupt routine, E_OS_ACCESS

Conformance: BCC1, BCC2, ECC1, ECC2

12.4.4 Constants

RES_SCHEDULER
- Constant of data type ResourceType (see chapter 7, Resource management).

12.5 Event control

12.5.1 Data types

EventMaskType

Data type of the event mask.

EventMaskRefType

Reference to an event mask.

12.5.2 Constructional elements

12.5.2.1 DeclareEvent

Syntax: DeclareEvent (EventIdentifier)

Parameter (In):

Event identifier (C-identifier)

Description: *DeclareEvent* serves as an external declaration of an event. The function and use of this service are similar to that of the external declaration of variables.

Particularities: -

Conformance: ECC1, ECC2

12.5.3 System services

12.5.3.1 SetEvent

Syntax: StatusType SetEvent (TaskType <TaskID>
EventMaskType <Mask>)

Parameter (In):	
TaskID	Reference to the task for which one or several events are to be set.
Mask	Mask of the events to be set
Parameter (Out):	none
Description:	The service may be called from an interrupt service routine and from the task level, but not from hook routines. The events of task <TaskID> are set according to the event mask <Mask>. Calling *SetEvent* causes the task <TaskID> to be transferred to the *ready* state, if it was *waiting* for at least one of the events specified in <Mask>.
Particularities:	Any events not set in the event mask remain unchanged.
Status:	
Standard:	• No error, E_OK
Extended:	• Task <TaskID> is invalid, E_OS_ID • Referenced task is no extended task, E_OS_ACCESS • Events can not be set as the referenced task is in the *suspended* state, E_OS_STATE
Conformance:	ECC1, ECC2

12.5.3.2 ClearEvent

Syntax:	StatusType ClearEvent (EventMaskType <Mask>)
Parameter (In)	
Mask	Mask of the events to be cleared
Parameter (Out)	none
Description:	The events of the extended task calling *ClearEvent* are cleared according to the event mask <Mask>.
Particularities:	The system service *ClearEvent* is restricted to extended tasks which own the event.
Status:	
Standard:	• No error, E_OK
Extended:	• Call not from extended task, E_OS_ACCESS • Call at interrupt level, E_OS_CALLEVEL
Conformance:	ECC1, ECC2

12.5.3.3 GetEvent

Syntax:	StatusType GetEvent (TaskType <TaskID> EventMaskRefType <Event>)
Parameter (In):	
TaskID	Task whose event mask is to be returned.
Parameter (Out):	
Event	Reference to the memory of the return data.
Description:	This service returns the current state of all event bits of the task <TaskID>, **not** the events that task is *waiting* for.

	The service may be called from interrupt service routines, task level and some hook routines (see Figure 10–1). The current status of the event mask of task <TaskID> is copied to <Event>.
Particularities:	The referenced task must be an extended task.
Status:	
Standard:	• No error, E_OK
Extended:	• Task <TaskID> is invalid, E_OS_ID • Referenced task <TaskID> is not an extended task, E_OS_ACCESS • Referenced task <TaskID> is in the *suspended* state, E_OS_STATE
Conformance:	ECC1, ECC2

12.5.3.4 WaitEvent

Syntax:	StatusType WaitEvent (EventMaskType <Mask>)
Parameter (In):	
Mask	Mask of the events *waited* for.
Parameter (Out):	none
Description:	The state of the calling task is set to *waiting*, unless at least one of the events specified in <Mask> has already been set.
Particularities:	This call enforces the rescheduling, if the *wait* condition occurs. This service may be called from the extended task owning the event.
Status:	
Standard:	• No error, E_OK
Extended:	• Calling task is not an extended task, E_OS_ACCESS • Calling task occupies resources, E_OS_RESOURCE • Call at interrupt level, E_OS_CALLEVEL
Conformance:	ECC1, ECC2

12.6 Alarms

12.6.1 Data types

TickType

This data type represents count values in ticks.

TickRefType

This data type points to the data type TickType.

AlarmBaseType

This data type represents a structure for storage of counter characteristics. The individual elements of the structure are:

`maxallowedvalue`	• Maximum possible allowed count value in ticks

`ticksperbase`	• Number of ticks required to reach a counter-specific (significant) unit.
`mincycle`	• Smallest allowed value for the cycle-parameter of SetRelAlarm/SetAbsAlarm) (only for systems with extended status).

All elements of the structure are of data type TickType.

AlarmBaseRefType

This data type points to the data type AlarmBaseType.

AlarmType

This data type represents an alarm object.

12.6.2 Constructional elements

12.6.2.1 DeclareAlarm

Syntax:	DeclareAlarm (AlarmIdentifier)	
Parameter (In):		
	Alarm identifier (C-identifier)	
Description:	*DeclareAlarm* serves as external declaration of an alarm element.	
Particularities:	Conformance:	BCC1, BCC2, ECC1, ECC2

12.6.3 System services

12.6.3.1 GetAlarmBase

Syntax:	StatusType GetAlarmBase (AlarmType <AlarmID>, AlarmBaseRefType <Info>)
Parameter (In):	
AlarmID	Reference to alarm
Parameter (Out):	
Info	Reference to structure with constants of the alarm base.
Description:	The system service *GetAlarmBase* reads the alarm base characteristics. The return value <Info> is a structure in which the information of data type AlarmBaseType is stored.
Particularities:	Allowed on task level, ISR, and in several hook routines (see Figure 10–1).
Status:	
Standard:	• No error, E_OK
Extended:	• Alarm <AlarmID> is invalid, E_OS_ID
Conformance:	BCC1, BCC2, ECC1, ECC2

12.6.3.2 GetAlarm

Syntax:	StatusType GetAlarm (AlarmType <AlarmID> TickRefType <Tick>)
Parameter (In):	
AlarmID	Reference to an alarm
Parameter (Out):	
Tick	Relative value in ticks before the alarm <AlarmID> expires.
Description:	The system service *GetAlarm* returns the relative value in ticks before the alarm <AlarmID> expires.
Particularities:	It is up to the application to decide whether for example a *CancelAlarm* may still be useful. If <AlarmID> is not in use, <Tick> is not defined. Allowed on task level, ISR, and in several hook routines (see Figure 10–1). Status:
Standard:	• No error, E_OK • Alarm <AlarmID> is not used, E_OS_NOFUNC
Extended:	• Alarm <AlarmID> is invalid, E_OS_ID
Conformance:	BCC1, BCC2, ECC1, ECC2

12.6.3.3 SetRelAlarm

Syntax:	StatusType SetRelAlarm (AlarmType <AlarmID>, TickType <increment>, TickType <cycle>)
Parameter (In):	
AlarmID	Reference to the alarm element
increment	Relative value in ticks
cycle	Cycle value in case of cyclic alarm. In case of single alarms, cycle has to be zero.
Parameter (Out):	none
Description:	The system service occupies the alarm <AlarmID> element. After <increment> ticks have elapsed, the task assigned to the alarm <AlarmID> is activated or the assigned event (only for extended tasks) is set.
Particularities:	The behaviour of <increment> equal to 0 is up to the implementation. If the relative value <increment> is very small, the alarm may expire, and the task may become *ready* before the system service returns to the user. If <cycle> is unequal zero, the alarm element is logged on again immediately after expiry with the relative value <cycle>. The alarm <AlarmID> must not already be in use. To change values of alarms already in use the alarm has to be cancelled first. If the alarm is already in use, this call will be ignored and the error E_OS_STATE is returned.

Allowed on task level and in ISR, but not in hook routines.

Status:

Standard:	• No error, E_OK
	• Alarm <AlarmID> is already in use, E_OS_STATE
Extended:	• Alarm <AlarmID> is invalid, E_OS_ID
	• Value of <increment> outside of the admissible limits (lower than zero or greater than **`maxallowedvalue`**), E_OS_VALUE
	• Value of <cycle> unequal to 0 and outside of the admissible counter limits (less than **`mincycle`** or greater than **`maxallowedvalue`**), E_OS_VALUE

Conformance: BCC1, BCC2, ECC1, ECC2; Events only ECC1, ECC2

12.6.3.4 SetAbsAlarm

Syntax: StatusType SetAbsAlarm (AlarmType <AlarmID>,
TickType <start>,
TickType <cycle>)

Parameter (In):

AlarmID	Reference to the alarm element
start	Absolute value in ticks
cycle	Cycle value in case of cyclic alarm. In case of single alarms, cycle has to be = zero.

Parameter (Out): none

Description: The system service occupies the alarm <AlarmID> element. When <start> ticks are reached, the task assigned to the alarm <AlarmID> is activated or the assigned event (only for extended tasks) is set.

Particularities: If the absolute value <start> is very close to the current counter value, the alarm may expire, and the task may become *ready* before the system service returns to the user.

If the absolute value <start> already was reached before the system call, the alarm will only expire when the absolute value <start> will be reached again, i.e. after the next overrun of the counter.

If <cycle> is unequal zero, the alarm element is logged on again immediately after expiry with the relative value <cycle>.

The alarm <AlarmID> must not already be in use.

To change values of alarms already in use the alarm has to be cancelled first.

If the alarm is already in use, this call will be ignored and the error E_OS_STATE is returned.

Allowed on task level and in ISR, but not in hook routines.

Status:

Standard:	• No error, E_OK
	• Alarm <AlarmID> is already in use, E_OS_STATE

Extended:	• Alarm <AlarmID> is invalid, E_OS_ID • Value of <start> outside of the admissible counter limit (less than zero or greater than **`maxallowedvalue`**), E_OS_VALUE • Value of <cycle> unequal to 0 and outside of the admissible counter limits (less than **`mincycle`** or greater than **`maxallowedvalue`**), E_OS_VALUE
Conformance:	BCC1, BCC2, ECC1, ECC2; Events only ECC1, ECC2

12.6.3.5 CancelAlarm

Syntax:	StatusType CancelAlarm (AlarmType <AlarmID>)
Parameter (In):	
AlarmID	Reference to an alarm
Parameter (Out):	none
Description:	The system service cancels the alarm <AlarmID>.
Particularities:	Allowed on task level and in ISR, but not in hook routines. Status:
Standard:	• No error, E_OK • Alarm <AlarmID> not in use, E_OS_NOFUNC
Extended:	• Alarm <AlarmID> is invalid, E_OS_ID
Conformance:	BCC1, BCC2, ECC1, ECC2

12.6.4 Constants

There always exists at least one counter which is a time counter (system counter). To facilitate programming of this counter, the return values of the call *GetAlarmBase* are also defined as constants.

`OSMAXALLOWEDVALUE`	• Maximum possible allowed value of the system counter in ticks.
`OSTICKSPERBASE`	• Number of ticks required to reach specific unit of the system counter.
`OSMINCYCLE`	• Minimum allowed number of ticks for a cyclic alarm.

Additionally the following constant is supplied:

`OSTICKDURATION`	• Duration of a tick of the system counter in nanoseconds.

12.7 Operating system execution control

12.7.1 Data types

AppModeType

This data type represents the application mode.

12.7.2 System services

12.7.2.1 GetActiveApplicationMode

Syntax	AppModeType GetActiveApplicationMode (void)
Description:	This service returns the current application mode. It may be used to write mode dependent code.
Particularities:	See chapter 4.8 for a general description of application modes. Allowed for task, ISR and all hook routines.
Conformance:	BCC1, BCC2, ECC1, ECC2

12.7.2.2 StartOS

Syntax	void StartOS (AppModeType <Mode>)
Parameter (In):	
Mode	application mode
Parameter (Out):	none
Description:	The user can call this system service to start the operating system in a specific mode, see chapter 4.8, Application modes.
Particularities:	Only allowed outside of the operating system, therefore implementation specific restrictions may apply. See also chapter 10.3, System start-up, especially with respect to systems where OSEK and OSEKtime coexist. This call does not need to return.
Conformance:	BCC1, BCC2, ECC1, ECC2

12.7.2.3 ShutdownOS

Syntax	void ShutdownOS (StatusType <Error>)
Parameter (In):	
Error	error occurred
Parameter (Out):	none
Description:	The user can call this system service to abort the overall system (e.g. emergency off). The operating system also calls this function internally, if it has reached an undefined internal state and is no longer ready to run. If a ShutdownHook is configured the hook routine *ShutdownHook* is always called (with <Error> as argument) before shutting down the operating system. If *ShutdownHook* returns, further behaviour of ShutdownOS is implementation specific. In case of a system where OSEK OS and OSEKtime OS coexist, *ShutdownHook* must return. <Error> must be a valid error code supported by OSEK OS. In case of a system where OSEK OS and OSEKtime OS coexist, <Error> might also be a value accepted by OSEKtime OS. In this case, if enabled by an OSEKtime configuration parameter, OSEKtime OS will be shut down after OSEK OS shutdown.

Particularities:	After this service the operating system is shut down. Allowed at task level, ISR level, in *ErrorHook* and *StartupHook*, and also called internally by the operating system. If the operating system calls *ShutdownOS* it never uses E_OK as the passed parameter value.
Conformance:	BCC1, BCC2, ECC1, ECC2

12.7.3 Constants

OSDEFAULTAPPMODE • Default application mode, always a valid parameter to *StartOS*.

12.8 Hook routines

The specification allows for implementation specific additional parameters in hook routines. In the following description only mandatory parameters are listed.

12.8.1 ErrorHook

Syntax	void ErrorHook (StatusType <Error>)
Parameter (In):	
Error	error occurred
Parameter (Out):	none
Description:	This hook routine is called by the operating system at the end of a system service which returns StatusType not equal E_OK. It is called before returning to the task level. This hook routine is called when an alarm expires and an error is detected during task activation or event setting. The ErrorHook is not called, if a system service called from ErrorHook does not return E_OK as status value. Any error by calling of system services from the *ErrorHook* can only be detected by evaluating the status value.
Particularities:	See chapter 10.1 for general description of hook routines.
Conformance:	BCC1, BCC2, ECC1, ECC2

12.8.2 PreTaskHook

Syntax	void PreTaskHook (void)
Parameter (In):	none
Parameter (Out):	none
Description:	This hook routine is called by the the operating system before executing a new task, but after the transition of the task to the *running* state (to allow evaluation of the TaskID by *GetTaskID*).
Particularities:	See chapter 10.1 for general description of hook routines.
Conformance:	BCC1, BCC2, ECC1, ECC2

12.8.3 PostTaskHook

Syntax	void PostTaskHook (void)
Parameter (In):	none
Parameter (Out):	none
Description:	This hook routine is called by the operating system after executing the current task, but before leaving the task's *running* state (to allow evaluation of the TaskID by *GetTaskID*).
Particularities:	See chapter 10.1 for general description of hook routines.
Conformance:	BCC1, BCC2, ECC1, ECC2

12.8.4 StartupHook

Syntax	void StartupHook (void)
Parameter (In):	none
Parameter (Out):	none
Description:	This hook routine is called by the operating system at the end of the operating system initialisation and before the scheduler is running. At this time the application can start tasks, initialise device drivers etc.
Particularities:	See chapter 10.1 for general description of hook routines.
Conformance:	BCC1, BCC2, ECC1, ECC2

12.8.5 ShutdownHook

Syntax	void ShutdownHook (StatusType <Error>)
Parameter (In):	
Error	error occurred
Parameter (Out):	none
Description:	This hook routine is called by the operating system when the OS service *ShutdownOS* has been called. This routine is called during the operating system shut down.
Particularities:	*ShutdownHook* is a hook routine for user defined shutdown functionality, see chapter 10.4.
Conformance:	BCC1, BCC2, ECC1, ECC2

13 Implementation and application specific topics

This chapter is not normative nor mandatory. It provides information for implementers and application programmers.

13.1 Implementation hints.

OSEK specifies an operating system interface and its functionality. Implementation aspects are not prescribed. There is no restriction on the implementation of the operating system as long as the implementation corresponds to any of the defined conformance classes.

13.1.1 Aspects of implementation

The range of automotive applications varies greatly such that no performance characteristics of the operating system implementation can be specified, i.e. as to the execution time and memory space required.

As a result,

- the OSEK operating system can be implemented with various degrees of efficiency.
- The linker needs only to link those objects and services of the operating system which are actually used.
- the operating system used in a product (e.g. in a control unit's EPROM) cannot be described as OSEK operating system, but as an operating system which conforms to an OSEK operating system conformance class.
- the tool environment of the operating system configuration and initialisation is not part of the operating system specification and therefore implementation-specific.
- commercial systems which provide the user with all OSEK operating system specific services and their functionalities via an OSEK adaptation layer, are also OSEK operating system compliant. They are compliant irrespective of their actual suitability for control units as regards the memory space they require and their processing speed.

The conformance class selected for an application software is determined by the needs on functionality and flexibility.

The real-time behaviour of the application software used with a specific hardware is also defined by the quality of implementation.

13.1.2 Parameters of implementation

The operating system vendor provides a list of parameters specifying the implementation. Detailed information is required concerning the functionality, performance and memory demand. Furthermore the basic conditions to reproduce the measurement of those parameters have to be mentioned, e.g. functionality, target CPU, clock speed, bus configuration, wait states etc.

13.1.2.1 Functionality

- Maximum number of tasks
- Maximum number of not suspended tasks
- Maximum number of priorities

- Number of tasks per priority (for BCC2 and ECC2)
- Upper limit for number of task activations (must be "1" for BCC1 and extended tasks)
- Maximum number of events per task
- Limits for the number of alarm objects (per system / per task)
- Limits for the number of nested resources (per system / per task)
- Lowest priority level used internally by the OS

13.1.2.2 Hardware resources

- RAM and ROM requirement for each of the operating system components
- Size for each linkable module
- Application dependent RAM and ROM requirements for operating system data (e.g. bytes RAM per task, RAM required per alarm, ...)
- Execution context of the operating system (e.g. size of OS internal tables)
- Timer units reserved for the OS
- Interrupts, traps and other hardware resources occupied by the operating system

13.1.2.3 Performance

- Total execution time for each service[12]
- OS start-up time (beginning of *StartOS* until execution of first task in standard mode) without invoking hook routines
- Interrupt latency[13] for ISRs of category 1, 2 and 3
- Task switching times for all types of switching[14]
- Base load of system without applications running

All performance figures shall be stated as minimum and maximum (worst case) values.

13.1.2.4 Configuration of run time context

A run time context is assigned to each task. This refers to all memory resources of the task which are occupied at the beginning of the execution time, and which are released again once the task is terminated. Typically the run time context consists of some registers, a task control block and a certain amount of stack to operate.

Depending on the design of tasks (e.g. type and pre-emptibility) and depending on the scheduling mechanism (non-, mixed- or full pre-emptive) the run time context may have

[12] The time of execution may depend on the current state of the system, e.g. there are different execution times of "SetEvent" depending on the state of the task (waiting or ready). Therefore comparable results have to be extracted from a common benchmark procedure.

[13] Time between interrupt request and execution of the first instruction of user code inside the ISR. A comparison of interrupt latencies of ISRs from category 1 to ISRs from category 2 or 3 specifies the operating system overhead.

[14] Should be measured from the last user instruction of the preceding task to the first user instruction of the following task so that all overhead is covered. Task switching types are different for: normal termination of a task, termination forced by *ChainTask*(), preemptive task switch, task activation when OS idle task is *running*, alarm triggered task activation and task activations from ISRs of types 2 and 3.

different sizes. Tasks which can never pre-empt each other may be executed in the same run time context in order to achieve an efficient utilisation of the available RAM space.

The operating system vendor should provide information about the implemented handling of the run time context (e.g. one context per task or one context per priority level). Considering this information the user may optimise the design of his application regarding RAM requirements versus run time efficiency.

13.2 Application design hints

The purpose of this chapter is to provide additional information about possible problems which might arise when designing applications for the OSEK operating system. Not all of the consequences for the system design can be mentioned in the specification itself. Other design hints result from the experience of current ECU applications.

13.2.1 Resource management

Some aspects are mentioned in this chapter in order to guarantee a proper handling of all resources.

13.2.1.1 Occupation in LIFO order

Each access to a resource should be encapsulated with calls to the services *GetResource* and *ReleaseResource*. Resources have to be released in reversed order of their occupation. The following code sequence is incorrect because function *foo* is not allowed to release resource *res_1*.

```
TASK(incorrect)
{
     GetResource( res_1 );
     /* some code accessing resource res_1 */
     ...
     foo();
     ...
     ReleaseResource( res_2 );
}

void foo()
{
     GetResource( res_2 );
     /* code accessing resource res_2 */
     ...
     ReleaseResource( res_1 );
}
```

Nested resource occupations is allowed. The occupation of resources has to be performed in strict LIFO order (stack principle). If the code accessing the resource as shown above is pre-empted by a task with higher priority (higher than the ceiling priority of the resource), another resource might be requested in that task leading to a nested resource occupation which conforms to the LIFO order.

13.2.1.2 Call level of API-services

The OSEK API-services *GetResource* and *ReleaseResource* should be called from the same functional call level. If function *foo* is corrected concerning the LIFO order of resource occupation like:

```
void foo( void )
{
        ReleaseResource( res_1 );
        GetResource( res_2 );
        /* some code accessing resource res_2 */
        ...
        ReleaseResource( res_2 );
    }
```

there still may be a problem because *ReleaseResource(res_1)* is called on a different level than *GetResource(res_1)*. Calling the API services from different call levels might cause problems in some implementations.

13.2.1.3 Resources still occupied at task termination

The access to a resource should be encapsulated directly by the calls of *GetResource* and *ReleaseResource*. Otherwise one might miss to release the resource and possibly terminate the task.

```
    GetResource( res_1 );
    ...
    switch ( condition )
    {
        case CASE_1 :
            do_something1();
            ReleaseResource( res_1 );
            break;
        case CASE_2 :             /* !!! WRONG: no release of   */
                                  /*    resource here !!!       */
            do_something2();
            break;
        default:
            do_something3();
            ReleaseResource( res_1 );
    }
    ...
```

If in standard status of the operating system a task terminates without releasing all of the occupied resources the resulting behaviour is not defined by the specification. Depending on the implementation of the operating system the resource may be locked forever since further accesses are rejected by the operating system.

13.2.2 Placement of API calls

For the same reasons as above mentioned in chapter 13.2.1.2 the placement of API services *TerminateTask* and *ChainTask* is crucial for the operating system. Both services are used to terminate the *running* task. Calling these services from a subroutine level of the task, the operating system is responsible for a correct treatment of the stack when terminating the task. One solution could be to store the position of the stack pointer at the entry point of the *running* task and restore that value after terminating the task.

13.2.3 Interrupt service routines

The user has to be aware of some possible error cases when using ISRs of category 1, 2 and 3 as described in chapter 5.

13.2.3.1 Local variables in ISRs of category 3

In ISRs of category 3 the user is allowed to write application code before the operating system context is entered using the service *EnterISR*. If *EnterISR* switches to a different stack, automatic variables defined in the preceding application code might be no longer accessible in the operating system context.

The application code at the beginning of the ISR might not be portable between different compilers when using local variables. This is because the convention for register usage is not always the same for compilers from different manufacturers.

13.2.3.2 Nested interrupts of different categories

Since all interrupts are of higher priority than the task levels, the processing of interrupts has to be terminated before the system returns to task level. If an ISR of category 2 interrupts an ISR of category 1 the system will continue processing of ISR1 after ISR2 terminates. Having tasks activated or events set from interrupt level in ISR2 the operating system is not invoked after termination of ISR1 in order to perform a rescheduling.

Please note that, in this respect, an ISR3, before *EnterISR* is called, acts like an ISR category 1, afterwards like an ISR category 2.

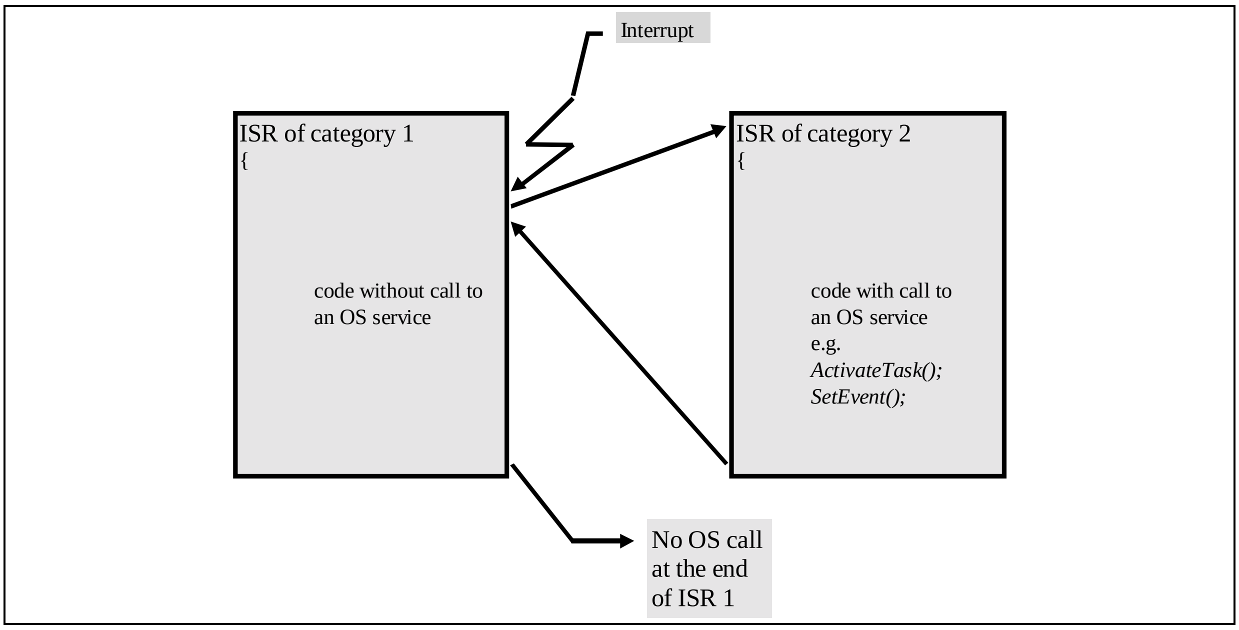

Interrupt
ISR of category 1
{
code without call to an OS service
ISR of category 2
{
code with call to an OS service e.g. *ActivateTask(); SetEvent();*
No OS call at the end of ISR 1

Figure 13–1 Nested interrupts

Because ISRs of category 1 (or category 3 before *EnterISR*) do not run under control of the operating system the OS has no possibility to perform a rescheduling when the ISR terminates. Thus any activities corresponding to the calls of the operating system in the interrupting ISR2 (or ISR3 after *EnterISR*) are unbounded delayed until the next rescheduling point.

As a result of the problems discussed above, each system should set up rules to avoid these problems. There may be specific implementations which can avoid these problems, or the application might have specific properties such that these problems can not occur (e.g. in non pre-emptive systems). The rules must therefore take into account both the specific implementations and the applications.

However, for maximal application portability, an easy rule of thumb which always works is the following:

- all interrupts of category 1 have to have a higher or equal hardware priority compared with interrupts of category 2.
- all interrupts of category 3 have to share one hardware priority not higher than the lowest category 1 interrupt priority, and not lower than the highest category 2 interrupt priority.

13.2.3.3 Direct manipulation of interrupt levels

Direct manipulation of interrupt levels is not portable and restricted by the implementation.

13.2.4 Priority and pre-emption

Tasks are scheduled by the operating system according to their priority. A task is declared as being pre-emptive / non pre-emptive (see chapter 4.6.3). The application has to treat these two task attributes in a consistent manner to avoid conflicts in the run-time behaviour of the system. Care has to be taken because non pre-emptive tasks of lower priority delay tasks of higher priority.

Typically the pre-emption of a task is assigned when designing, whereas priority is configured during system integration. Because many people are involved in larger software projects, the development process has to be co-ordinated precisely. To achieve a well-defined run-time behaviour of the system this co-ordination is crucial.

13.2.5 Parameter to pass to ShutdownOS

The parameter passed to *ShutdownOS* is also passed to the *ShutdownHook*. If the operating system calls *ShutdownHook*, the passed parameter is an implementation dependent error value. If the user calls *ShutdownOS* he has to use one of the existing OSEK OS error numbers. If OSEKtime and OSEK coexist, an OSEKtime OS error number can also be passed.

It is strongly recommended to use the error number described in the implementation documentation. If no specific error number for *ShutdownOS* is defined, it is possible to use E_OK and to distinguish this way between operating system calls of *ShutdownOS* and application calls.

13.2.6 Error handling

Errors in the application software are typically caused by:

- Errors on handling the operating system, i.e. incorrect configuration / initialisation / dimensioning of the operating system or violations of restrictions regarding the operating system service.
- Error in software design, e.g. inappropriate choice of task priorities, unprotected critical sections, incorrect scaling of time, inefficient conceptual design of task organisation

Test of implementation

Breakpoints, traces and time stamps can be integrated individually into the application software.

Example: The user can set time stamps enabling him to trace the program execution at the following locations before calling operating system services:

- When activating or terminating tasks.
- When setting or clearing events in the case of extended tasks.
- At explicit points of the schedule.
- At the beginning or the end of ISRs.
- When occupying and releasing resources or at critical locations.

Time monitoring

The operating system needs not include a time monitoring feature which ensures that each or only, e.g. the lowest-priority task has been activated in any case after a defined maximum time period.

The user can optionally use hook routines or establish a watchdog task that takes "one-shot displays" of the operating system status.

Constructional elements

Constructional elements (e.g. DeclareTask) were introduced in OSEK OS as means to create references to system objects used in the application. Like external declarations constructors would be placed at the beginning of source files. With respect to the implementation they can be implemented as macros. With the definition of OIL most implementations do not need them any more. However they are still kept for compatibility.

13.2.7 Errors and warnings

Most of the error values of system services point to application errors. However, in some special cases error values indicate warnings which might come up during normal operation. These cases are:

• *EnableInterrupt, DisableInterrupt*	E_OS_NOFUNC	(extended)
• *GetAlarm*	E_OS_NOFUNC	(standard)
• *SetAbsAlarm, SetRelAlarm*	E_OS_STATE	(standard)
• *CancelAlarm*	E_OS_NOFUNC	(standard)

Especially when implementing a central error handling using *ErrorHook*, this has to be taken into account.

13.3 Implementation specific tools

When buying or writing portable code one has to be aware of the different implementation tools on the market. This has an impact, on what kind of documentation has to go in parallel with the code.

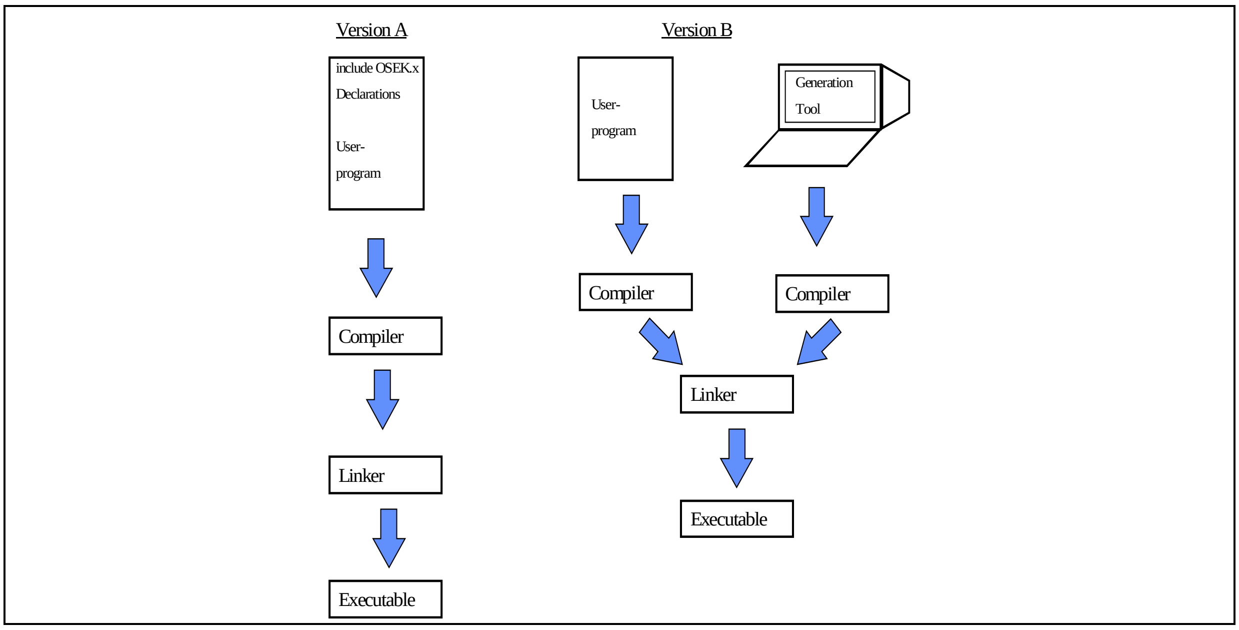


Figure 13–2 Implementation specific tools

The example here shows two possible implementations of a tool chain:

- Version A, with all declarations related to task properties etc. within the code
- Version B, using a separate generation tool for these task properties etc.

For definitions which should be supplied with portable code please consult the OIL specification.

14 Changes from specification 1.0 to 2.1r1

14.1 Changes from specification 1.0 to 2.0r1

This chapter mentions all changes in the concept and the API of the OSEK operating system, with explanation for the reason of change.

14.1.1 Conceptual changes

14.1.1.1 Conformance classes

This chapter refers to chapter 3.2 Conformance classes.

The OSEK OS specification version 2.0 now supports only four conformance classes instead of five (as in version 1.0). Also the CCs are renamed, so for example ECC1 (version 1.0) has other features than ECC1 (version2.0). The experience of working with version 1.0 has shown that the four CCs of version 2.0 will better meet application requirements.

Changes in detail are:

- Multiple requesting of task activation for extended tasks is not supported. That is only allowed for basic tasks.
- The number of multiple requesting of task activation is an attribute of the basic task and no requirement of the conformance class.
- The conformance classes of version 2.0 are no longer strictly upward compatible.

14.1.1.2 Messages

Specification version 2.0 does not support communication via messages. All message services are part of the communication specification and therefore described in the OSEK COM specification.

14.1.1.3 Multiple requesting of task activation

This chapter refers to chapter 4.3, Activating a task.

In version 1.0 the order of activation in case of multiple request was not explicitly defined but up to the implementation. In version 2.0 it is clearly defined that the activations are queued in a FIFO structure according to the order of requesting.

14.1.1.4 Application modes

This chapter refers to chapter 4.8, Application modes.

For some applications it should be useful to have different application modes depending on external conditions.

14.1.1.5 Counters

The API for counters has been removed (see chapter 8.1, Counters). In version 1.0 access to counters was allowed for the application. This feature is strongly depending on the underlying hardware. Therefore the API services for counters are cancelled in version 2.0. The API services for alarms are still available.

14.1.1.6 Hook routines

This chapter refers to chapter 10.1 Hook routines.

The naming of hook routines changed from OSxxxx to xxxxHook.

In version 2.0 two additional hook routines *StartupHook* (see chapter 12.8.4) and *ShutdownHook* (see chapter 12.8.5) are introduced. This feature offers the possibility of user defined start-up and shutdown.

14.1.1.7 OS execution control

In version 2.0 of the OSEK OS specification two new API services are introduced, *StartOS* (see chapter 12.7.2.1) and *ShutdownOS* (see chapter 12.7.2.3). With this two services, the user can start-up and shutdown the overall system.

14.1.2 Clarifications

14.1.2.1 Scheduling of non pre-emptive tasks

When a non pre-emptive task is pre-empted by calling the scheduler, the task context is saved. If the task is assigned to the processor again, the task will continue at the point of pre-emption and will not be restarted from the beginning.

14.1.2.2 Services available on which level

In version 2.0 two tables are specifying which service is available on interrupt level, on task level and in which hook routine.

14.1.2.3 Interrupt processing

In version 2.0 the ISR category 3 is mandatory and not optional any more.

14.1.2.4 Priority ceiling

This chapter refers to chapter 7.5, OSEK Priority Ceiling Protocol.

In version 2.0, the ceiling priority of a resource is defined exactly as:
a) identical or higher to the highest task priority with access to this resource (e.g. TaskX)
and
b) lower than the priority off all other of higher priority than that task (TaskX).

14.1.2.5 Types and constants

In version 2.0 the type TaskType is specified. The following types are defined:

• TaskType:	identifies a task
• TaskRefType:	points to a variable of TaskType
• TaskStateType:	identifies the state of a task
• TaskStateRefType:	points to a variable of TaskStateType

14.1.2.6 Naming conventions

In version 2.0 the macro `TASK` has got a new meaning (see chapter 12.2.5). This change was necessary because the old version of `TASK` had a drawback; the user was forced to define a name for the *task function* he was not allowed to use as *task name*

```
TASK TaskFuncName (void)
{ /* Task function for the Task "TaskName" */
  /* The name "TaskFuncName" must NOT be used as a task name */
}
```

14.1.3 Changes of the documentation

14.1.3.1 Document structure

The specification documentation of version 1.0 consists of two documents, the "concept" and the "API". In version 2.0 these two papers are integrated into this one, called OSEK OS specification.

14.1.3.2 New chapters

Portability of application software (paragraph in chapter 1.1)

This new chapter regards aspects of portability of OSEK software.

Implementation and application specific topics (see chapter 13)

This new chapter gives hints for implementing an OSEK operating system.

14.1.3.3 Removed chapters

Chapter messages

The message concept is described in the OSEK COM specification. Therefore the message parts are removed.

System generation

All questions of system generation are described in an extra paper called **OIL specification** (OIL = **O**SEK **I**mplementation **L**anguage). Several references to that paper are made throughout this document.

14.2 Changes from specification 2.0r1 to 2.1 and 2.1r1

Most changes appeared from 2.0r1 to 2.1. Changes from 2.1 to 2.1r1 are specifically marked.

A lot of wording within the document has been changed for clarification and to improve readability (2.1 and 2.1r1). The document structure was also changed for the same reason. These changes are not explicitly mentioned in this section, but only changes in the concept and the API of the OSEK operating system.

14.2.1 Behaviour of ChainTask/TerminateTask with allocated resources is undefined.

In 2.0r1 the behaviour was not undefined but only the occupation of the resource was. As this is a clear application error resulting in unsafe behaviour it was not considered useful to define part of the behaviour in case of serious errors.

14.2.2 GetTaskID is allowed in ISRs.

As GetTaskState was allowed in ISRs and hook routines, and GetTaskID was already allowed in hook routines, it seemed inconsistent and problematic not to allow it in ISRs.

14.2.3 Interrupt handling has been clarified and extended.

- Support for interrupts of category 3 is optional.
- Clarification that EnableInterrupt/DisableInterrupt manipulates interrupt sources and that the InterruptDescriptor is global.
- Added functions DisableAllInterrupts/EnableAllInterrupts.
- Added functions SuspendOSInterrupts/ResumeOSInterrupts.
- Optional extension of resources to interrupts (including the concept of interrupt priorities).

14.2.4 Error checking of GetResource/ReleaseResource have been modified.

The definition in 2.0r1 was incomplete and the extension of the resource concept to ISRs required this change.

14.2.5 Added constant OSTICKSPERBASE.

There have been constants for two of the three values returned by GetAlarmBase for a single system counter. The missing third one was added for completeness.

14.2.6 ShutdownOS is allowed in ISRs and certain hook routines.

ShutdownOS is meant to be called by the application in case of fatal errors. As such errors are likely to be discovered in ISRs or hooks (e.g. ErrorHook) it was considered dangerous to prevent the application from immediately shutting down the operating system.

14.2.7 Behaviour of ShutdownOS after ShutdownHook returns is implementation defined.

Version 2.0r1 of the specification was inconsistent in this point.

14.2.8 Added constant OSDEFAULTAPPMODE.

This constant was added to increase portability of applications.

14.2.9 ErrorHook is never called recursively.

Recursive calling of ErrorHook possibly leads to unbounded recursion and was considered too dangerous.

14.2.10 Local Messages added to specification.

Intra processor message handling (refer to conformance class CCCA/CCAB as defined in the OSEK Communication Specification) has been added.

14.2.11 Startup/shutdown when OSEK and OSEKtime coexist (2.1r1)

In case OSEK OS coexists with OSEKtime, restrictions have been added to the startup and the shutdown procedure of the system. Especially, *ShutdownHook* must return.

15 Index

15.1 List of figures

16 History

Version	Date	Remarks	
1.0	11. Sept. 1995	Authors:	
		Thomas Wollstadt	Adam Opel AG
		Wolfgang Kremer	BMW AG
		Jochem Spohr	Daimler-Benz AG
		Stephan Steinhauer	Daimler-Benz AG
		Thomas Thurner	Daimler-Benz AG
		Karl Joachim Neumann	University of Karlsruhe
		Helmar Kuder	Mercedes-Benz AG
		François Mosnier	Renault SA
		Dietrich Schäfer-Siebert	Robert Bosch GmbH
		Jürgen Schiemann	Robert Bosch GmbH
		Reiner John	Siemens AG
2.0	02. June 1997	Authors:	
		Wolfgang Kremer	BMW AG
		Salvatore Parisi	Centro Ricerche Fiat
		Andree Zahir	ETAS GmbH & Co KG
		Stephan Steinhauer	Daimler-Benz AG
		Jochem Spohr	ATM Computer GmbH
		Jan Söderberg	Delco
		Piero Mortara	Magneti Marelli
		Helmar Kuder	Mercedes-Benz AG
		Bob France	Motorola SPS
		Kenji Suganuma	Nippondenso co., ltd
		Stefan Poledna	Robert Bosch AG
		Gerhard Göser	Siemens Automotive SA
		Georg Weil	Siemens Automotive SA
		Alain Calvy	Siemens Automotive SA
		Karl Westerholz	Siemens Semiconductors
		Jürgen Meyer	Softing GmbH
		Ansgar Maisch	University of Karlsruhe
2.0 revision 1	15. October 1997	Authors see version 2.0	

2.1	22. May 2000	Authors:	
		Manfred Geischeder	BMW
		Klaus Gresser	BMW
		Adam Jankowiak	DaimlerChrysler
		Jochem Spohr	DaimlerChrysler
		Andree Zahir	ETAS
		Markus Schwab	Infineon
		Erik Svenske	Mecel
		Maxim Tchervinsky	Motorola
		Ken Tindell	NRTA
		Gerhard Göser	Siemens Automotive
		Carsten Thierer	University of Karlsruhe
		Winfried Janz	Vector Informatik
		Volker Barthelmann	3Soft
2.1 revision 1	13. November 2000	Authors:	
		OSEK OS WG/OSEKtime WG	
		compiled by: Jochem Spohr	DaimlerChrysler