OSEK OS

Michael Kunz

March 18, 2009

This document introduces the OSEK/VDX OS specification, which is an established standard in automotive industry. The realtime capabilities of the OSEK/VDX OS scheduler and resource management will be discussed with focus on timing guarantees and function constraints for a typical RMS based system.

Contents

1 Introduction

This document is about OSEK/VDX Operating System specification [1], in short OSEK OS, which is an established standard in automotive industry. It focuses on the structure and architecture of OSEK OS, especially in the scheduling (section 2.6) and resource management (section 2.7). Furthermore it will explain the realtime capabilities on the example of implementing a rate monotonic scheduling (RMS) and finally discuss the results.
The paper consists of two parts: the actual specification and description of OSEK OS followed by the discussion of realtime properties by the means of RMS. The former provides general information about the OSEK/VDX group, introduces the concepts used by the OS including task concepts (section 2.3) and conformance classes (section 2.5). It furthermore describes what is handled as a resource and how those are managed.
The second part (section 3) will clarify if an OSEK/VDX system can be realtime and which properties can be guaranteed in which way based on the previously given description.

2 OSEK OS

2.1 OSEK/VDX Organisation

OSEK/VDX is a joint project of the German OSEK group and the French VDX group formed in 1994.
In May 1993, OSEK was founded by members of the German automotive industry, namely BMW, Bosch, DaimlerChrysler, Opel, Siemens, VW and the IIIT of the University of Karlsruhe as coordinator. OSEK is the abbreviation of the German term "**O**ffene **S**ysteme und deren Schnittstellen für die **E**lektronik im **K**raftfahrzeug", which stands for "Open Systems and the Corresponding Interfaces for Automotive Electronics".
VDX (**V**ehicle **D**istributed e**X**ecutive) was a similar project launched by PSA and Renault in France and joined OSEK in 1994.
The project was created to work against recurring expenses in development and various management of control unit software as well as incompatible interfaces and protocols of control units produced by different manufactures.
This is intended to be achieved by creating a set of abstract and application independent interfaces which do not rely on any specific hardware or network. Furthermore, it is supposed to adjust optimally to the particular application by being efficiently scalable and configurable. Besides, it should be possible to verify that the functionality of an implementation conforms to the interface specification.

2.2 OSEK OS Overview

The OSEK Operating System is an interface specification defining system services to use within applications. The specification itself does not describe any particular implementation, leaving aspects open which shall be defined within the specific documentation of the implementation.
OSEK OS is focused on single processor systems in specially distributed embedded control units of any type. To achieve high portability, the services of the operating system are defined in an ISO/ANSI-C-like syntax, but the implementation language of the system is not specified.
To keep the implementation simple and predictable, dynamic creation of system objects is not supported. Every system object, including tasks and resources, has to be specified during system creation. The whole operating system is therefore configured and scaled statically, using the OSEK/VDX Implementation Language (OIL).
The system is based on an event driven control mechanism, which means that actions of the system are only triggered by certain events of the tasks or the hardware.

2.3 Task concept

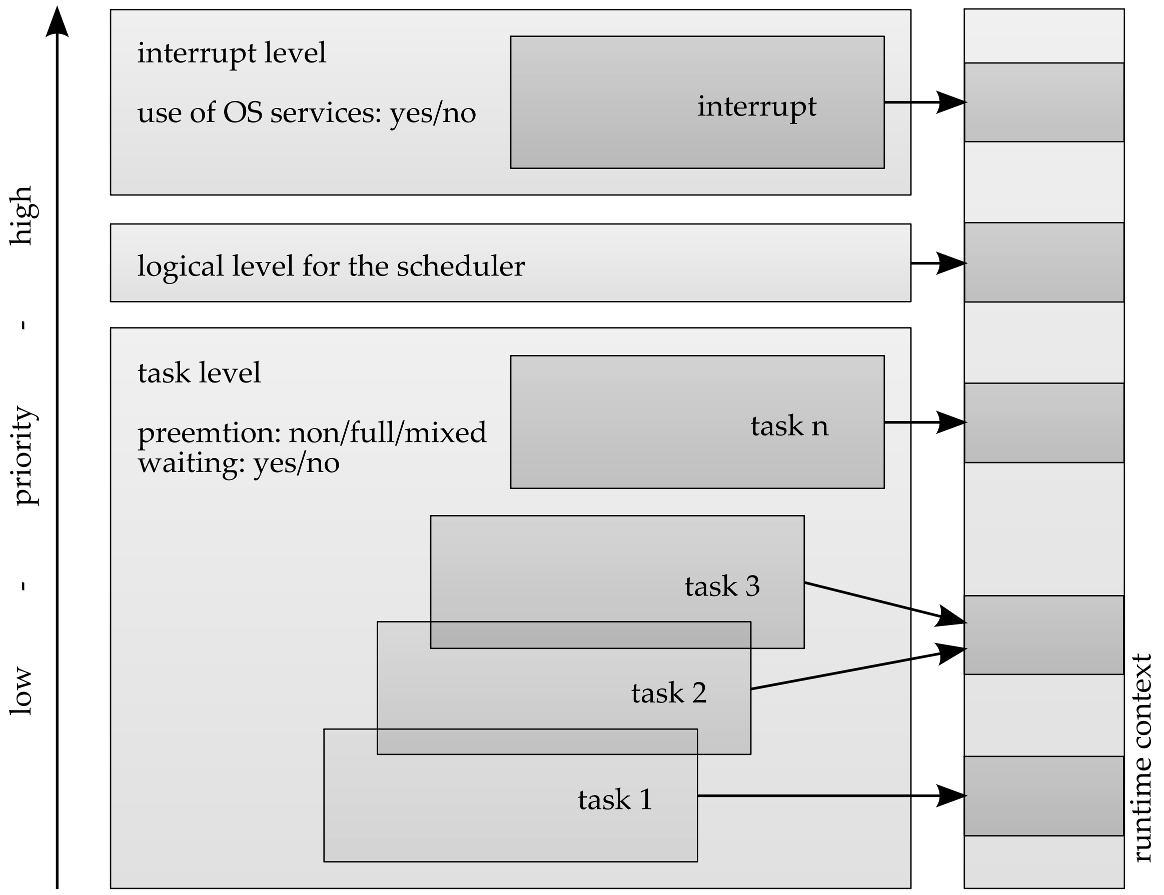

Figure 1: Processing levels inside OSEK OS

A task is a framework for a specific functionality within complex control software, which should be executed according to its timing requirements. Typical requirements of a task are that it gets started as soon as possible as well as it completes within a given interval based on the effect or result it produces in the environment. The OSEK/VDX Operating System serves as a basis for application programs which are independent of each other and provides their environment on a processor. The task can be executed asynchronously or concurrently, so that they appear to run simultaneously. The system provides a task switching mechanism, the scheduler, supporting preemptive and non-preemptive scheduling as well as an idle-mechanism, running while no task or system functionality is active. Besides the tasks, there are interrupts managed by the operating system and competing for the CPU. Therefore OSEK OS defines three processing levels, see figure 1.
The interrupt level can contain one or more interrupt priorities which are statically assigned to interrupt service routines and depend on the implementation and hardware architecture. Within the task level, the tasks are scheduled according to their policy and priority, which are statically assigned by the user during system creation. Concerning task priorities, higher numbers refer to higher priority, but interrupts are always preferred over tasks.

2.4 Basic and Extended Task models

OSEK OS distinguishes between two task types: *Basic Tasks* and *Extended Tasks*. The main difference is that extended tasks are allowed to wait for events whereas basic tasks are not. A task has to change between different states, because the processor can only execute one instruction of one task at a time. While one task is executed, others are competing for the processor, and the operating system is responsible for saving and restoring a task's context whenever state transitions are necessary. A basic task can only enter three states, shown in figure 2. A task is

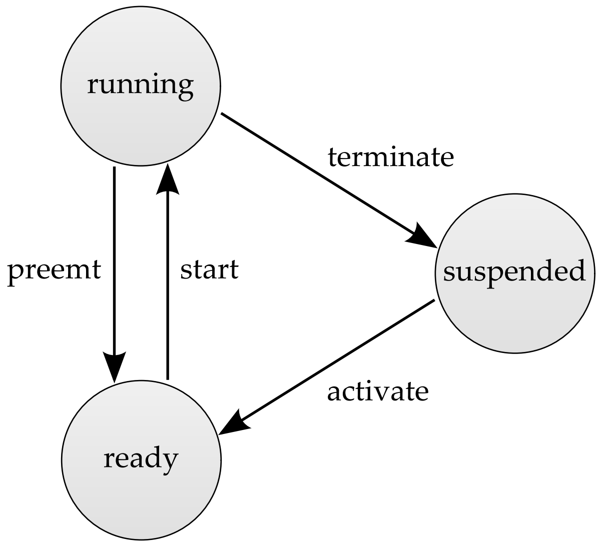

Figure 2: task states of basic tasks

in the suspended state when it was not activated or has terminated. This state exists because tasks cannot be created during execution of the system and must be defined statically during system design.
Tasks can only be terminated when they are in the running state, meaning they can only terminate themselves which was a decision of OSEK design, made to keep the task state model and implementation as simple as possible. A task usually ends its execution with a call to the *TerminateTask* system service, but there is another function, *ChainTask*, which terminates the task and activates another one. The *ChainTask* service can be used to easily produce an asynchronous execution of various tasks. The extended task model has the additional waiting state (figure 3).

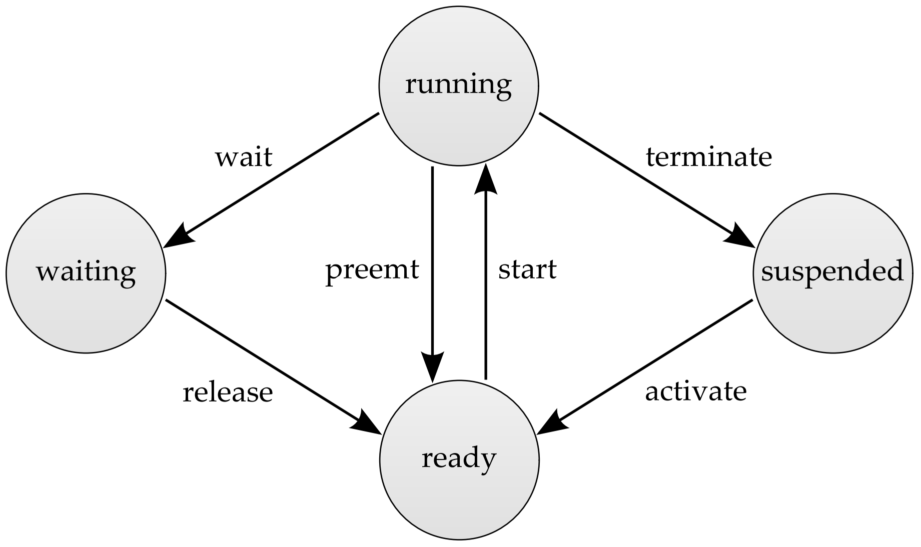

Figure 3: task states of extended tasks

2.4.1 Events

Activated extended tasks may own events which they are allowed to wait for until other tasks or interrupt service routines signal them. An event therefore transports binary data to its owning task which can be used for synchronization. Only the owning extended task may wait for or reset the event. If the event is set at the moment the task wants to wait for it, execution is continued without interruption.

2.5 Conformance Classes

Any OSEK OS conformant implementation must be compatible with one or more of the four *Conformance Classes*. They exist to allow better scaling of the system through partial implementation along defined lines. A class represents a convenient subset of functionality of the OSEK OS specification. The classes are in relation to each other, so that one with higher functionality includes all services of classes with lesser functionality. Therefore it is possible to upgrade an implementation to any higher class without any changes to the applications. Conformance classes (figure 4) are determined by three attributes:

- support of extended tasks or basic tasks only
- multiple requesting for activation of a basic task
- count of tasks per priority

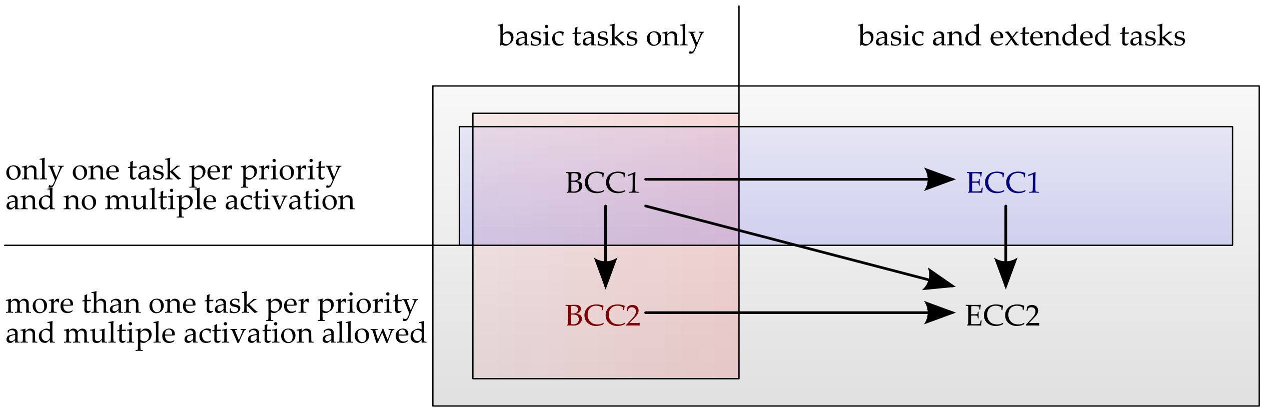
basic tasks only
basic and extended tasks
only one task per priority and no multiple activation
more than one task per priority and multiple activation allowed
BCC1
ECC1
BCC2
ECC2

Figure 4: Relation and update path of conformance classes

2.6 Scheduling

The OSEK OS *Scheduler* uses static priorities assigned to the tasks by the system designer. The active tasks are managed in separate time sorted queues per priority. The scheduler selects a ready task with the highest priority to be started next. If there are extended tasks in waiting state, they will be ignored and the next element in the queue will be considered to be scheduled if queues and another task are present.
Rescheduling only takes place at specific points during execution of tasks and is not initiated by the operating system itself, but is a reaction to a specific event. During interrupt processing no rescheduling is performed. The OSEK OS scheduler supports three types of scheduling policies which description follows.

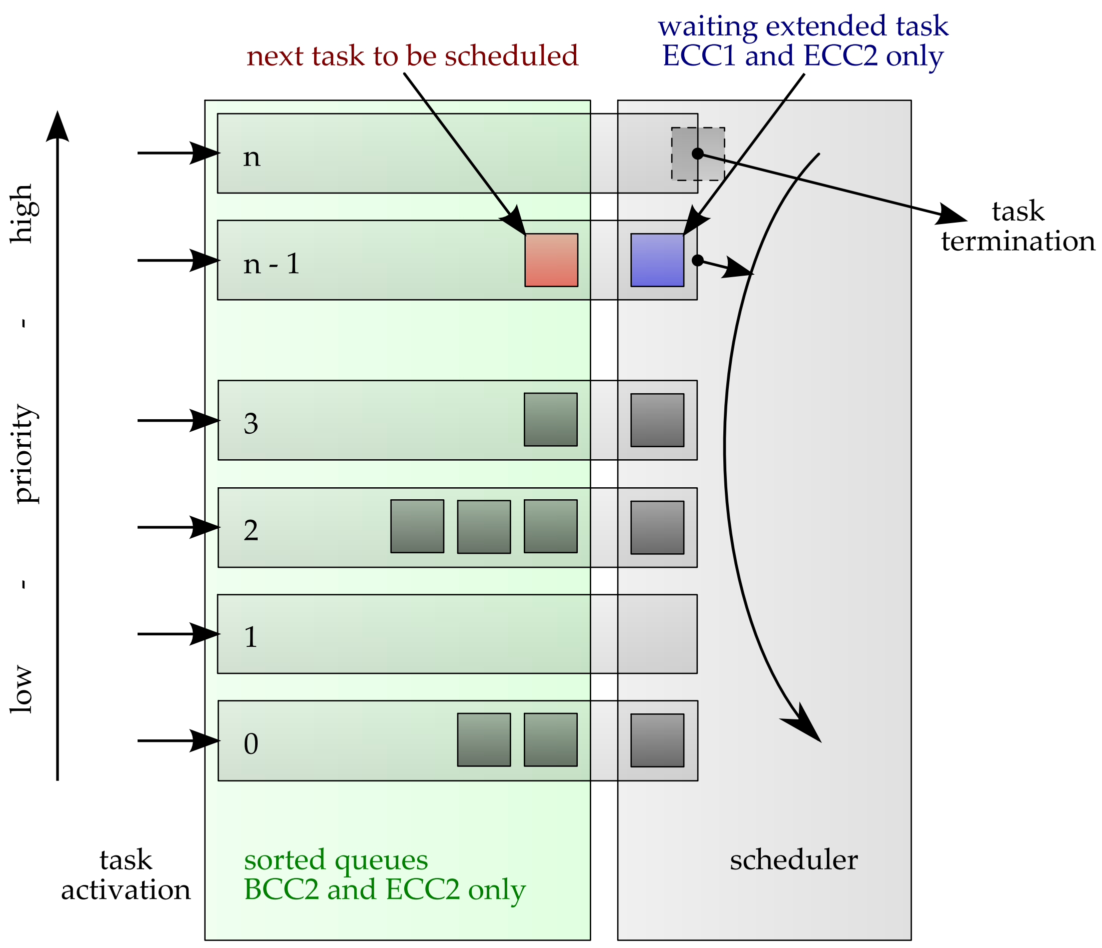

Figure 5: General scheduling in OSEK OS

2.6.1 Non-Preemptive Scheduling

With *Non-Preemptive Scheduling*, tasks only release the CPU for rescheduling when they perform one of the following actions:

- The task terminates itself with or without explicitly activating a successor task with *TerminateTask* or *ChainTask*.
- An explicit call to the service *Schedule* occurs which then activates the scheduler.
- Waiting for a non-set event is done by an extended task with *WaitEvent*, and therefore transition into waiting state is performed.

Of course interrupt service routines can always interrupt the execution of the running task, but after completion of the interrupt handling no rescheduling is performed and the previously running task continues. The running task can release the CPU and explicitly invoke the scheduler with a call of the system service *Scheduler* at points of execution where it is appropriate. That service allows cooperative scheduling between non-preemptive tasks.

2.6.2 Full Preemptive Scheduling

The *Full Preemptive Scheduling* policy means that the currently running task may be rescheduled at any instruction by the occurrence of any system event. Likely the preemption can occur

at any instruction the scheduler needs to save a more complex task context than with non-preemptive scheduling.
The scheduler is invoked in case one of the following situations occurs:

- The task terminates itself with or without explicitly activating a successor task with *TerminateTask* or *ChainTask*.
- A task gets activated by the current task.
- An extended task enters the waiting state by a call of *WaitEvent* which only happens if the event was not set at this moment.
- The running task sets an event of an extended task that is in the waiting state.
- Return from interrupt level with the completion of an interrupt service routine because ISR may activate tasks or set events.
- Release of a resource at task level.

All of these are actions that may induce a task to change its state, and therefore there is no need for full preemptive scheduled tasks to use the system service *Schedule*, but it is necessary to ensure the correct task is assigned to the CPU when the application is written in spite of different scheduling policies for portable applications.

2.6.3 Mixed Preemptive Scheduling

The scheduling policy is called mixed preemptive scheduling if tasks with full preemptive and tasks with non-preemptive policy are both existent on the same system. The scheduler is invoked depending on the policy of the currently running task which means if the task is non-preemptable, non-preemptive scheduling is performed and if the task uses full preemptive scheduling policy, rescheduling can occur on any specified event.
This policy is a compromise between non and full preemptive scheduling and was introduced to fulfil the usual needs of control applications which consist of only a few parallel tasks with long execution times for which a full preemptable system would be best and many short tasks with defined execution times for which a non-preempting system would be more efficient.

2.6.4 Grouping Tasks

A task group is a set of tasks which do not preempt each other but are preemptable by tasks of higher priority. This behaviour can be achieved in two different ways. It is possible to either give all the tasks of one group the same priority. This will need an OSEK OS implementation conformant to BCC2 or ECC2 which require many more resources for the queues of the scheduler than BCC1 and ECC1 do.
The second possibility are so called task groups which are tasks of various priorities. They share one logical internal system resource which every task requires for execution. This allows the execution of the current task of the group without interruption from other tasks of the group, while the task is still preemptable by tasks with higher priority. For further details on resources see section 2.7.

2.6.5 Critical Sections

In OSEK OS there are different ways to propagate a critical section. First of all is the use of task groups to prevent preemption by a set of other tasks as described before. The critical section is achieved against every task which occupies the logical resource of this group but the task can

still be preempted by tasks with higher priority.
Another very similar way is the use of the logical system resource RES_SCHEDULER instead of a group resource. While this resource is occupied by a task, no scheduler will be activated and therefore no other task will preempt the critical section. For this kind of critical section it is sufficient that only the task with the critical operation requests the resource, but this method blocks all tasks whereas task groups do not.
Another aspect of critical sections are interrupts. OSEK OS distinguishes between interrupt service routines which make use of system services and those which do not. The former are called interrupts of category 2 or OS-Interrupts. The latter are interrupts of category 1. To achieve the (additional) exclusion of interrupts from the critical operations, the system provides services for disabling and re-enabling all interrupts or only those of category 2.

2.7 Resource Management - Priority Ceiling

There are various problems to be solved by the resource management. It has to coordinate concurrent access of tasks with different priorities to shared resources. A resource can be everything that must or could be managed by the functionality of the resource management which includes management entities itself (scheduler), program sequences, memory areas and hardware objects. There are several constraints to be ensured during the distribution of resources:

- Two tasks cannot occupy the same resource at the same time.
- Priority inversion cannot occur.
- No deadlock is possible when using these resources.
- Request of a resource by a task does not result in a blocked state.

2.7.1 Priority Ceiling

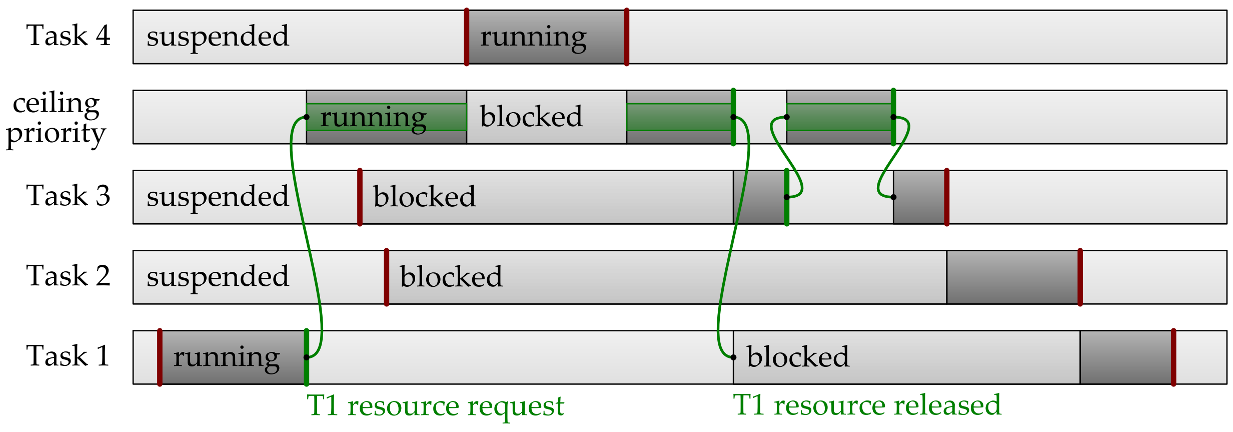

Figure 6: Example of priority ceiling

The *Priority Ceiling Protocol* resolves the problems of priority inheritance as well as deadlocks and fulfils the requirements of resource management and operates as follows.
Every OSEK OS application is a statically scaled and specified system which includes tasks with static priorities and all resources. These definitions are extended by the priority ceiling protocol by the means of assigning an additional ceiling priority to every resource. The ceiling priority of a resource is set according to the highest priority of every task requesting the resource. The priority must be at least the numerical value of the highest task requesting this resource but it should not be larger than priorities of tasks never requesting the resource.

If a task occupies a resource the task's priority is temporarily raised to the ceiling priority of the resource and thus the task cannot be preempted by another task requesting this resource. The task shall not terminate, wait for an event or exclusively call the scheduler while accessing a resource.
If a task releases a resource, its priority is lowered to the highest ceiling priority of the remaining owned resources or to its initially assigned task priority. OSEK OS assumes a LIFO order of resource demands and matching releases. This simplifies the determination of the resulting task priority equal to the one the task had before it occupied the resource.
Resulting from the priority ceiling protocol every request can be fulfilled instantly because the requesting task could not be selected by the scheduler if the resource is occupied.
High priority tasks can be blocked by low priority tasks due to priority ceiling. The maximum time the task can possibly be blocked is the maximum time of resource occupation by a low priority task of a resource with greater or equal ceiling priority than the task in question.

2.7.2 Resources within Interrupt Service Routines

An optional extension to the resource management in OSEK OS is to allow resource usage in interrupt service routines of category 2. This implies that interrupts must be included into the priority ceiling protocol, which means they need a virtual priority above every task. Furthermore, the execution of an interrupt service routine needs to be delayed until the priority of the running task is lower than the virtual priority of the interrupt which means all needed resources of the ISR are available. That extension supposes that it is possible and acceptable to delay interrupt processing as well as to block high priority tasks by tasks with lower priority, which receive the high ceiling priority of the resources.

2.8 Other Features

2.8.1 Counters and Alarms

Counters are OSEK OS implementation specific objects, such as timers or sensors, of the hardware represented as an integer value. It is the responsibility of the implementation how counters are manipulated. At least one counter, which is derived from a software or hardware timer, is present in every system.
An alarm is an action which is executed when a counter reaches a specific value. These actions can be task activation, execution of a callback routine on interrupt level or setting of an event whereas the underlying value of the counter can be specified as either absolute or relative. Alarms are specified during system creation and are assigned to one counter and one task or callback. One counter can activate more than one alarm whereas an alarm can be recurrent, too. OSEK OS provides services to manage the alarms which means setting and cancelling them as well as checking their state.

2.9 Other parts of OSEK/VDX

The specifications published by OSEK/VDX also cover other important parts of automotive systems such as communication and network management.
The OSEC COM specification [2] describes the communication over a network or bus and supports filters, multicast and deals with different data representations of communication partners. Furthermore, it is used for communication between tasks of the same OSEK OS instance. Various transmission modes are provided, including periodic, triggered and mixed modes, as well as deadline monitoring and notifications about message delivery or timeout.
Another operating system specification by OSEK/VDX is the Time-Triggered Operating System (ttOS) [3] providing time table based execution of the system and therefore leaves planning

and scheduling in the responsibility of the system creator. The ttOS supports different application modes which yield the execution of different time tables. An OSEK OS instance can exist within a ttOS as replacement for the idle task.
Furthermore, a fault tolerant communication layer specification [4] exists which defines structures for multiple transmission of messages with support for voters/agreements in the case of failures. Additionally, it supports algorithms for global time determination and synchronisation with external sources.
Finally, a Network Management system [5] is described, which monitors the state of the network and existing nodes. The monitoring can be performed directly, with dedicated messages, or indirectly by the monitoring of application messages. The information of the system state can be reported to the operating system(s) of each participating node.

3 Discussion

3.1 Description

The most important fact of any realtime system is the ability to guarantee timing constraints, especially the completion of tasks and resource allocation. Due to the complexity and flexibility of the OSEK system, it is hard to analyse it as a whole. Because of that, only special arrangements are considered.

3.2 Resources

Normally, the work with resources needs special care, whereby effects such as deadlocks and priority inversion must not occur to ensure the correct execution of any realtime application. OSEK OS implements the priority ceiling protocol (see section 2.7) which deals with these effects. Furthermore, a request for any resource must be fulfilled within a determinable time interval. The priority ceiling protocol ensures that every request can be fulfilled instantly when the task is running, but it achieves this guarantee by the cost of more complex time demand while scheduling a task.

3.3 Tasks

The OSEK OS scheduler is very flexible and complex, but works with static priorities. For realtime applications with periodic tasks and static priorities the RMS (rate monotonic scheduling) is optimal. A scheduler is considered optimal, if it produces a legal schedule of a taskset under the condition there exists a legal schedule at all. It is possible to realise RMS within an OSEK OS system.

3.3.1 Explanation of RMS

RMS is a preemptive scheduling method of periodic tasks with static priorities. The scheduler selects the task with highest priority from the set of ready tasks. The priority of a task results from its period, the higher the period the lower the priority.

3.3.2 RMS with OSEK OS

In an ordinary case, the use of conformance classes BCC1 or ECC1 (see section 2.5) is sufficient because every task gets its own priority with RMS. Assuming a system of n periodic tasks with periods $t_{p,i}$ which are ordered according to their period, starting with the biggest $t_{p,i}$. The first task gets OSEK priority 1, the second gets priority 2 and so on until the n-th task with shortest period gets the highest OSEK priority n.
Periodicity of RMS tasks can be created using recurrent, relative alarms on the timer counter (see section 2.8.1) which will simply activate the task. For additional deadline monitoring, a callback can be used on the alarm to activate the task and check the return value of the service call or even do more complex deadline checking.

3.3.3 Time demand analysis

Time demand analysis describes how often a task can be blocked by higher tasks with their execution time. It iterates along time interval which blocking can occur in and therefore describes how long a task may need until completion. The maximum response time for task i (the task with i-th lowest priority) can be calculated as follows:

$$\begin{aligned}t^{(l)}_{maxresp,i} &= t_{e,i} + \sum_{k=i+1}^{n} \left\lceil \frac{t^{(l-1)}_{maxresp,i}}{t_{p,k}} \right\rceil \cdot t_{e,k} \\ t^{(0)}_{maxresp,i} &= t_{e,i}\end{aligned}$$

This does not consider the additional blocking which may occur due to priority ceiling. The protocol ensures that a task may be blocked only once due to priority ceiling of a task. The additional blockade can only be caused by tasks of lower priority which use resources with a resource priority $p_{r,i}$ at least as high as the task priority $p_{t,k} = k$.
Assume the following resource set by each task:

$$R_k = \{i\text{: resource } i \text{ is used by task k}\}$$

The resulting block time for a task is the maximum length of each possible resource occupations $t_{r,k}$:

$$t_{b,i} = \max_{\forall k,r \in R_k:\ k<i,\ p_{res,r} \geq i} t_{r,k}$$

The modified iteration formula takes the blocking time as an additional addend:

$$\begin{aligned}t^{(l)}_{maxresp,i} &= t_{e,i} + t_{b,i} + \sum_{k=i+1}^{n} \left\lceil \frac{t^{(l-1)}_{maxresp,i}}{t_{p,k}} \right\rceil \cdot t_{e,k} \\ t^{(0)}_{maxresp,i} &= t_{e,i} + t_{b,i}\end{aligned}$$

The tasks can be scheduled with RMS if nothing violates their deadline, which means the time demand is lower or equal to the deadline.
Wang Lei, Wu Zhaohui and Zhao Mingde [6] provide an extended worst-case time analysis for OSEK conformant systems, additionally covering non-preemptive tasks, context switches and priority based communication on a CAN-bus.

3.3.4 Behaviour with overloaded tasksets

In overloaded systems the tasks with lowest priority are impaired whereas high priority task can continue normal execution. While using RMS in such a system it is determinable how many tasks are affected. This can be done by checking the time demand of each task starting with the highest priority until the first task violates its deadline. This task and every task with lower priority will be affected.

3.4 Memory usage

Memory usage in a system depends on many factors such as system object count and size, task stack and heap memory/variables as well as compiler specific stack and heap layout. Because compiler usage is not specified in OSEK/VDX and tasks are in the responsibility of the application programmer, the only part left to discuss is the memory usage of the system. All system objects are static and always present so that the operating system itself needs a certain fixed amount of memory. Furthermore it is possible to reduce the needed resources to a minimum by shrinking scheduler queue sizes and priority level count to a minimal value. As further optimization, a system compatible to conformance classes BCC1 or BCC2 (section 2.5), thus only consisting of basic tasks, can share a single stack with all tasks, which will massively reduce context information of a task.

3.5 Conclusion

After all, OSEK OS is very flexible, which makes it hard to ensure realtime operation, but with some restrictions of the full OSEK/VDX operating system functionality it is possible to use it as a basis for realtime applications. Non-critical applications can still use the extended functionality of the system, such as events, but those should be background tasks and run with lowest priority so that they do not interfere with the critical operations.
Another interesting fact is the rapid development of multi-core processors even for the embedded systems market, which may make it impossible to use OSEK OS for such devices in the future.

References

[1] OSEK/VDX. Operating system version 2.2.3. http://portal.osek-vdx.org/files/pdf/specs/os223.pdf, February 2005.

[2] OSEK/VDX. Communication version 3.0.3. http://portal.osek-vdx.org/files/pdf/specs/osekcom303.pdf, July 2004.

[3] OSEK/VDX. Time triggered operating system version 1.0. http://portal.osek-vdx.org/files/pdf/specs/ttos10.pdf, July 2001.

[4] OSEK/VDX. Fault-tolerant communication version 1.0. http://portal.osek-vdx.org/files/pdf/specs/ftcom10.pdf, July 2001.

[5] OSEK/VDX. Network management version 2.5.3. http://portal.osek-vdx.org/files/pdf/specs/nm253.pdf, July 2004.

[6] Wang Lei; Wu Zhaohui; Zhao Mingde. Worst-case response time analysis for osek/vdx compliant real-time distributed control systems. http://ieeexplore.ieee.org/iel5/9304/29570/01342819.pdf, 2004.

[7] Robert Baumgartl. Script echtzeitsysteme - abschnitt 3 scheduling. http://rtg.informatik.tu-chemnitz.de/docs/ezs/ezs-03-sched.pdf, 2008.

[8] Robert Baumgartl. Script echtzeitsysteme - abschnitt 4 ressourcen. http://rtg.informatik.tu-chemnitz.de/docs/ezs/ezs-04-ress.pdf, 2008.