OSEK/VDX

Time-Triggered Operating System

Version 1.0

July 24th 2001

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 time-triggered real-time operating system (OSEKtime). It is not a product description which relates to a specific implementation. This document also specifies the OSEKtime 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 OSEKtime operating system (OSEKtime OS) is to represent a uniform functioning environment which supports efficient utilisation of resources for automotive control unit application software. The OSEKtime operating system is a single processor operating system meant for distributed embedded control units.

1.1 System Philosophy

The objective of the OSEKtime working group is to specify a time-triggered operating system with a fault-tolerant communication layer as a standardised run-time environment for highly dependable real-time software in automotive electronic control units. The operating system must implement the following properties:

- predictability (deterministic, a priori known behaviour even under defined peak load and fault conditions),
- clear, modular concept as a basis for certification,
- dependability (reliable operation through fault detection and fault tolerance),
- support for modular development and integration without side-effects (composability), and
- compatibility to the OSEK/VDX.

The OSEKtime operating system supports static scheduling and offers all basic services for real-time applications, i.e., interrupt handling, dispatching, system time and clock synchronisation, local message handling, and error detection mechanisms.

All services of OSEKtime are hidden behind a well-defined API. The application interfaces to the OS and the communication layer only via this API.

For a particular application the OSEKtime operating system can be configured such that it only comprises the services required for this application. Thus the resource requirements of the operating system are as small as possible.

OSEKtime also comprises a fault-tolerant communication layer that supports real-time communication protocols and systems and is described in FTCom specification.

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 OSEKtime 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. The description is generic enough for future upgrades.

It is assumed that the description of the OSEKtime 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 OSEKtime operating system description has been used as a reference description.

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 essential specification chapters are described briefly:

Chapter 2, Summary

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

Chapter 3, Architecture of the OSEKtime Operating System

This chapter gives a survey about the design principles and the architecture of the OSEKtime-triggered operating system.

Chapter 4, Task Management

This chapter contains a description of the attributes of a task, the task states and transitions, and the special case of task deadline monitoring supported in OSEKtime.

Chapter 5, Interrupt Processing

This chapter contains a short description of the interrupt handling mechanism within OSEKtime OS.

Chapter 6, Synchronisation

This chapter contains a description of local and global times, together with the possibilities to handle the start-up of a system under various conditions.

Chapter 7, System Start-up and Shutdown in a mixed OSEKtime / OSEK/VDX System

This chapter contains a description of system Start-up and Shutdown in a mixed OSEKtime / OSEK/VDX system.

Chapter 8, Inter-Task Communication

The main topic of this chapter is the definition of inter-task communication. Full message handling is described in OSEKtime FTCom.

Chapter 9, **Error Handling**

This chapter contains a description of the error handling..

Chapter 10, Specification of OSEKtime Operating System Services

This chapter contains a description of the services provided by the operating system.

Chapter 11, Index

List of all operating system services and figures.

Chapter 12, History

List of all versions.

2 Summary

The OSEKtime operating system provides the necessary services to support distributed fault-tolerant highly dependable real-time applications (e.g., start-up of the system, message handling, state message interface, interrupt processing, synchronisation and error handling).

The operating system is built according to the user's configuration instructions at system generation time. The operating system cannot be modified later at execution time.

The service groups are structured in the terms of functionality.

Task management

- Management of task states, task switching
- Scheduling policy
- Deadline monitoring

Interrupt management

- Services for interrupt processing

System time and start-up

- Synchronisation of system time
- Services for system start-up

Intra processor message handling

- Services for exchange of data

Error treatment

- Mechanisms supporting the user in case of various errors

Event mechanisms, alarms, and resources

- Event mechanisms, alarms, and resources are restricted for the use to OSEK/VDX tasks and are not allowed for time-triggered tasks. So event mechanism, alarms, and resources are not part of the OSEKtime specification as the normal OSEK/VDX OS mechanisms apply.

2.1 Architecture of a OSEKtime System

The operating system is responsible for the on-line management of the CPUs resources, management of time and task scheduling. The FTCom layer is responsible for the communication between nodes, error detection and fault-tolerance functionality within the domain of the communication subsystem. Figure 2-1 shows the architecture of a OSEKtime system. Application software and FTCom layer are executed under control of the operating system. OSEK-NM describes node-related (local) and network-related (global) management methods. The global NM component is optional and described in the OSEK/VDX NM specification.

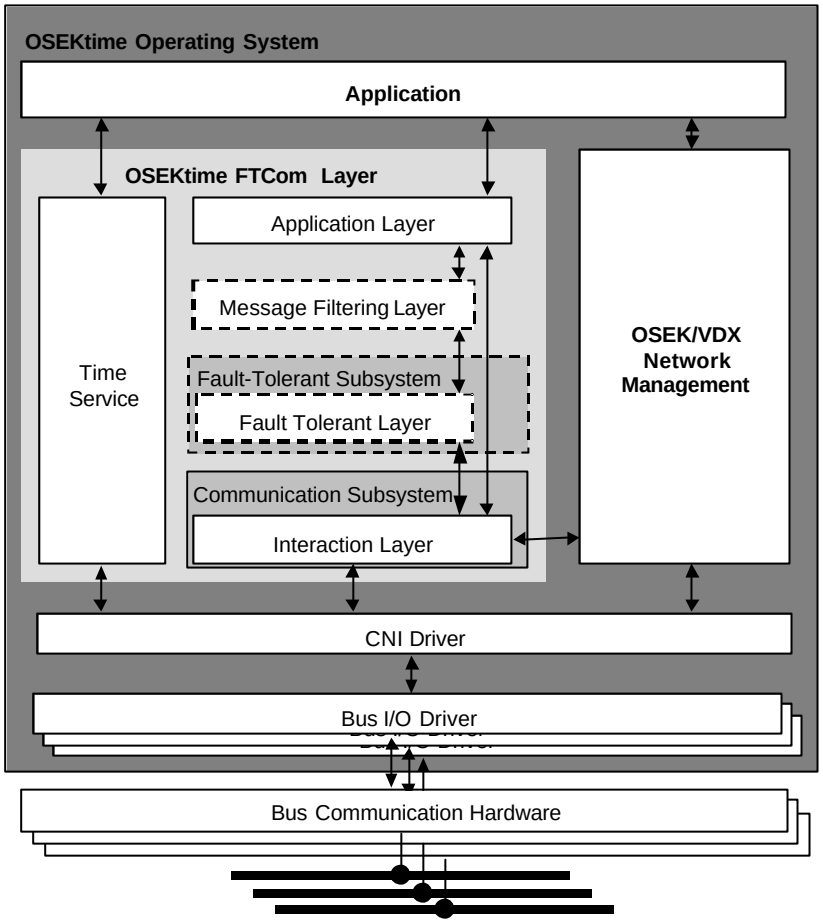

Figure 2-1: Architecture of an OSEKtime system

3 Architecture of the OSEKtime Operating System

This chapter describes the architecture of the OSEKtime operating system.

3.1 Processing Levels

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

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

- Interrupt services routines managed by the operating system
- 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.

OSEKtime defines two processing levels:

- Interrupt level
- TT Task level

Figure 3-1 shows the interrupt and task level model of the OSEKtime OS:

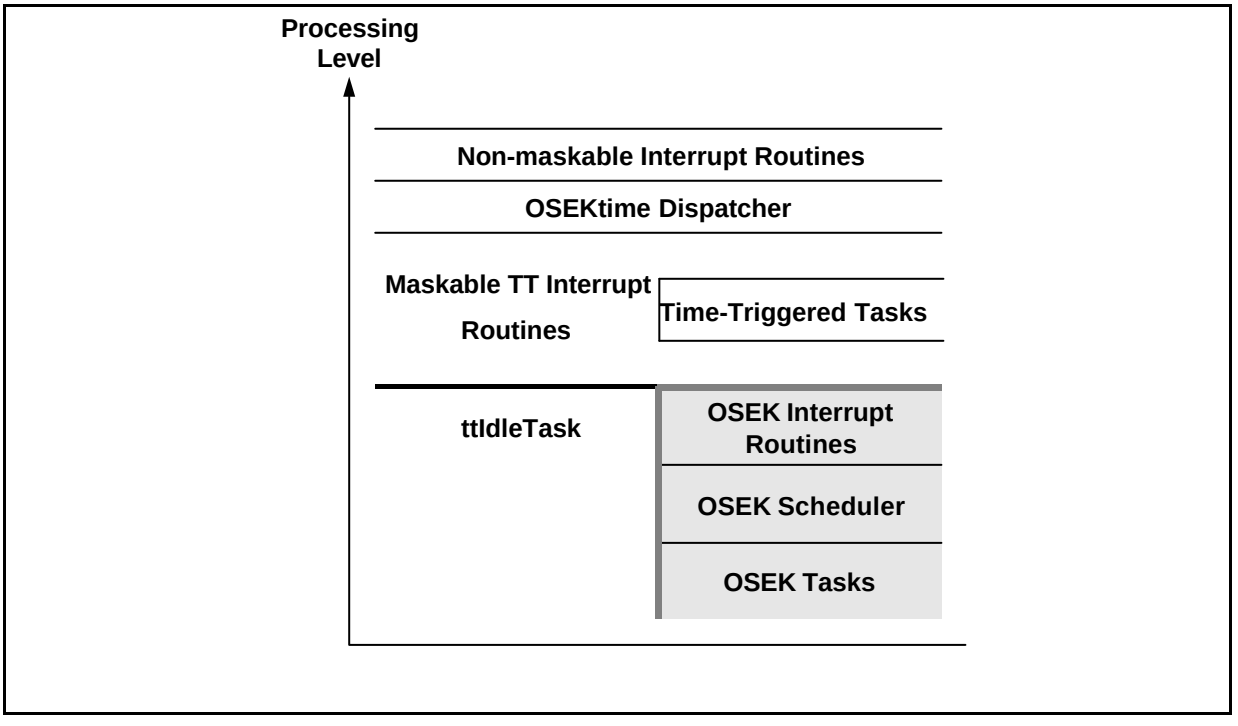

Figure 3-1: Processing levels

Beside the OSEKtime subsystem it is possible to include a full OSEK/VDX OS subsystem in the kernel. In a highly dependable application, however, the OSEK/VDX subsystem can only be used, if the following restriction is fulfilled:

- The complete OSEKtime must have a higher processing level than the OSEK/VDX subsystem.

The following requirements should be fulfilled:

- The microcontroller should provide a sufficient number of interrupt levels for the implementation of the above model.
- For highly dependable applications memory protection mechanisms either in hardware or software should be used.

Only if these requirements are fulfilled it can be guaranteed that tasks of the OSEK/VDX subsystem cannot interfere with a highly dependable time-triggered task located in the OSEKtime subsystem.

In case no OSEK/VDX subsystem is required or the above requirements cannot be met, OSEKtime can also be implemented without an OSEK/VDX subsystem without compromising the functionality of the OSEKtime subsystem. In this case the OSEKtime subsystem must offer an idle (background) task.

In the model a mixture of interrupt routines and time-triggered tasks is possible, i.e., the time-triggered tasks can have precedence over interrupt service routines.

Non-preemptive OSEK/VDX tasks are allowed but they are non-preemptable by OSEK/VDX tasks only. The OSEKtime dispatcher that activates time-triggered tasks according to the dispatcher table preempts even non-preemptive OSEK/VDX tasks in order to realise deterministic timing behaviour for all time-triggered tasks.

Additionally time-triggered tasks and OSEK/VDX tasks do not share common resources. The inter-task communication is done by local message handling as defined in chapter 8.

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

4 Task Management

4.1 Task Concept

Tasks are executed sequentially starting at the entry point and running to the exit point (see Figure 4-1). Internal loops are allowed but one must be able to determine the Worst Case Execution Time (WCET). Blocking (i.e., waiting for an external event) of tasks is *not* supported. The beginning of task execution is linked to an activation event. In a time-triggered application activation events originate from the dispatcher table only.

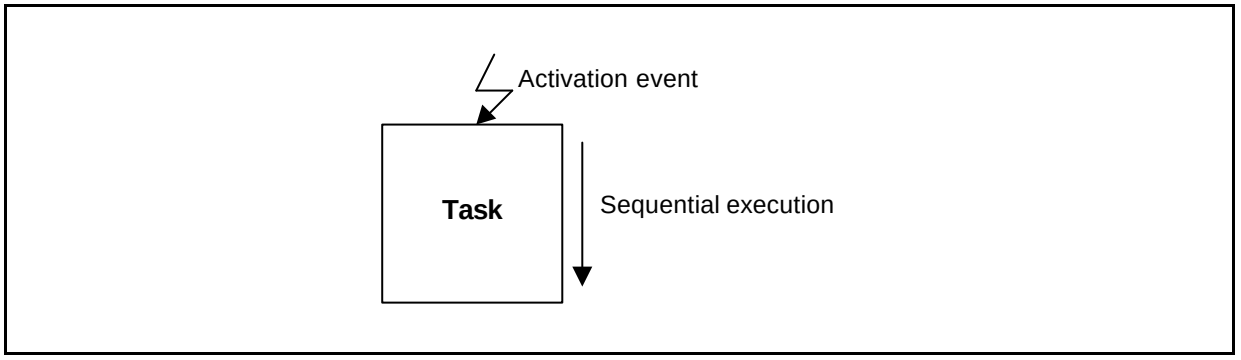

Figure 4-1: Task model

4.2 Task State Model

A task must be able to change between the states running, suspended and preempted, as the processor can only execute one instruction of a task at any time static scheduling is supported by OS. Task activation times are stored in the dispatcher table. The OSEKtime operating system is responsible for starting the tasks at the right time and the monitoring of the deadlines. Time-triggered tasks always preempt the execution of other time-triggered tasks.

4.2.1 Time-triggered Tasks

Time-triggered tasks only have three 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.

preempted: In the *preempted* state the instructions of a task are not executed. A task enters this state from the running state only. The only allowed succeeding state of this state is the running state. A task enters this state from the running state if and only if another task changes from the suspended state to the running state, triggered by the dispatcher. A task leaves this state if its preempting task changes from the running state to the suspended state.

suspended: In the *suspended* state the task is passive and can be activated.

Time-triggered task state transitions are shown on Figure 4-2.

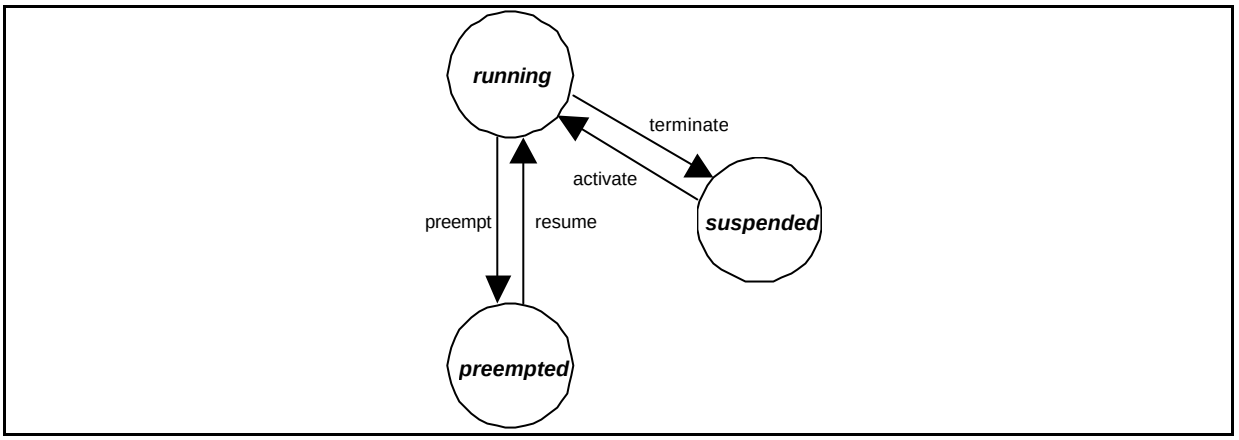

Figure 4-2: Time-triggered task model

States and status transitions for time-triggered tasks are shown in Table 4-1.

Transition	Former state	New state	Description
activate	*suspended*	*running*	A new task is set into the *running* state by the OSEKtime Dispatcher as a result of an OSEKtime Dispatcher tick.
resume	*preempted*	*running*	The last *preempted* task is resumed.
preempt	*running*	*preempted*	The OSEKtime Dispatcher decides to start another task as a result of an OSEKtime Dispatcher tick. The *running* task is put into the *preempted* state.
terminate	*running*	*suspended*	The *running* task causes its transition into the *suspended* state as a result of task completion.

Table 4-1: States and status transitions for time-triggered tasks

4.3 Activating a Task

Task activations are performed by the OSEKtime dispatcher as a result of OSEKtime dispatcher ticks (TT interrupts). OSEKtime dispatcher invocation events are defined in an offline generated dispatcher table. A task can be activated more than once during one dispatcher round, but is not allowed to preempt itself.

4.4 Scheduling Policy – time-triggered Activation

The OSEKtime OS is based on preemptive scheduling. Static scheduling is supported. From the task timing characteristics (such as offsets, worst case execution times and deadlines) an external

scheduling tool generates a dispatcher table. The time-triggered tasks can preempt each other. No blocking mechanisms through events or resource management like in OSEK/VDX OS are allowed.

The dispatcher activates the tasks in a strictly sequential order, which is stored in the dispatcher table. A complete execution of the dispatcher table is called dispatcher round. A task (except the idle task) cannot run during the end of one dispatcher round and the start of the consecutive dispatcher round.

In the dispatcher table all task activations are pre-planned. The dispatcher table is executed cyclically providing a periodic task execution scheme. This guarantees that no internal intermediate calculation results of a task are exposed to other tasks. The dispatcher is initiated by an interrupt, the interrupt source is the local logical time, which is synchronised with the global time, when a global time is available (see Chapter 6).

Time-triggered tasks do not have a static priority that is configurable by the user. The dispatcher always activates a new time-triggered task according to the dispatcher table. If another time-triggered task is running at that activation time it is always preempted and remains preempted until termination of the newly activated task. This scheduling policy is referred to as stack-based scheduling. The stack-based scheduling policy requires that the off-line scheduling tool constructs a dispatcher table in such a way that all deadlines are met and no stack overflows will occur.

The offline defined dispatcher table guarantees precedence relations between tasks (such as user-defined task sequences and offline resource constraints). In order to guarantee precedence relations, the dispatcher table prevents unexpected task preemptions.

Figure 4-3 shows an example of time-triggered task activation. The time-triggered tasks 1-3 (TT1-3) are started at their activation time. If no time-triggered task is running OSEK/VDX OS tasks will be executed. The states of the *Idle Task* in Figure 4-3 are corresponding to OSEK/VDX tasks.

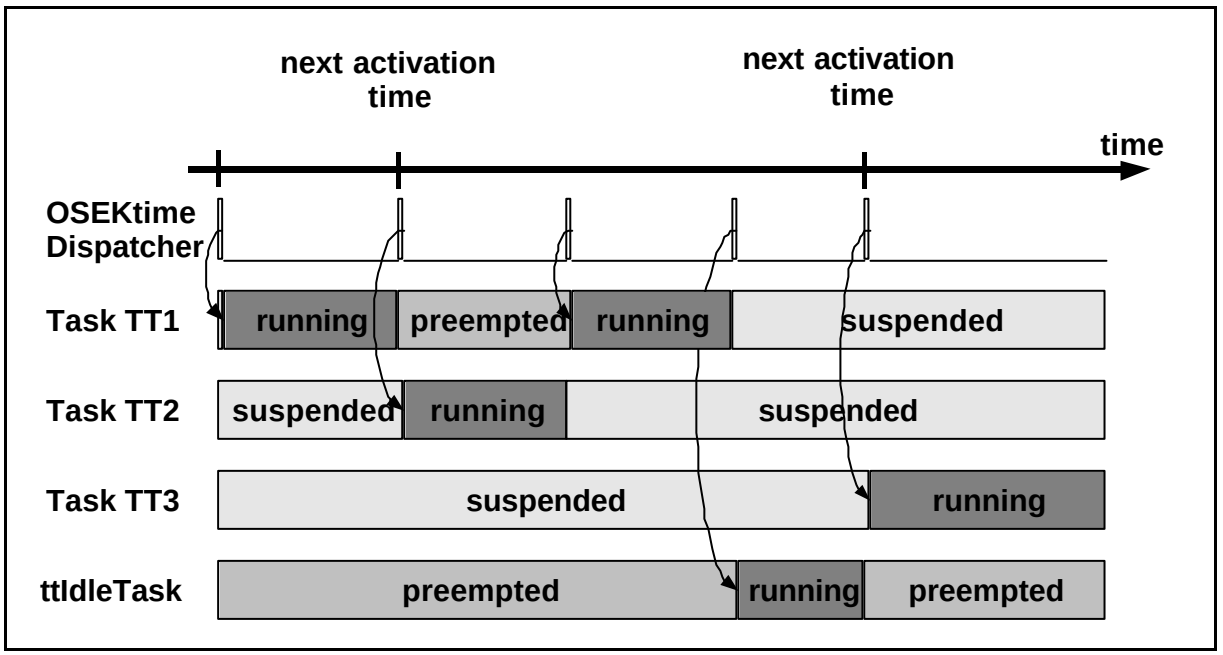

Figure 4-3: Time-triggered scheduling

4.5 Termination of Tasks

In the OSEKtime operating system, a time-triggered task has to terminate itself before its deadline occurs.

4.6 Deadline Monitoring

An important attribute of a task in a real-time system is its deadline, i.e., the point in time when the task execution must be finished. Task deadline violation must be checked for each task during runtime.

The deadline monitoring is done by the dispatcher. A special dispatcher table entry called “Deadline Monitoring” is used to check task deadline. This entry indicates that the dispatcher should be started and the deadline monitoring should be performed. If a task violates its deadline, error handling is initiated (see chapter 9).

Each Task deadline has to be checked by one of the following mechanisms:

- *Stringent task deadline monitoring:* A “Deadline Monitoring” entry is added in the dispatcher table exactly at the point of time when the task deadline is expired.
- *Non-stringent task deadline monitoring:* A “Deadline Monitoring” entry is added in the dispatcher table at a convenient point (after the task deadline expiration), but not later than the end of the current dispatcher round. In this case, the “Deadline Monitoring” entry may be grouped with an entry associated with an activation of a new task in order to enhance performance characteristics.

The task deadline monitoring method is a task-level attribute configurable by the user.

4.7 ttIdleTask

The first task started by the OSEKtime dispatcher is always a task with the predefined name: *ttIdleTask*. It has the following special properties:

- it is not registered in a dispatching round and therefore not periodically restarted (however, it may be restarted under special conditions),
- it can be interrupted by all interrupts handled by OSEKtime (normally, interrupts are grouped to either be able to interrupt OSEKtime tasks, or not be able to do so, see Figure 3-1),
- no deadline is defined for *ttIdleTask* and
- because it is started first, it will always run if there is no other task ready.
- *ttIdleTask* never returns.

With these properties, *ttIdleTask* acts as the idle task of the OSEKtime OS.

A default *ttIdleTask* has to be supplied as part of an OSEKtime OS. The user can replace this task by a task covering his special needs as long as it conforms to the restrictions stated above.

In case of a mixed OSEKtime/OSEK system, *ttIdleTask* is supplied by the OSEK manufacturer.

4.8 Application Modes

The concept of application modes allows the efficient management of different processing states in the application software. An application mode is defined by a dispatcher table. The length of all dispatcher rounds must be equal. Application modes can be, for example, initialisation, normal operation and shutdown.

The operating system is started with the application mode passed as parameter by calling the system service *ttStartOS*. The switching between different dispatcher tables during runtime, without losing synchronisation is performed by the system service *ttSwitchAppMode* (see chapter 10.5.3). The actual switch happens at the end of the current dispatcher round.

5 Interrupt Processing

The OSEKtime operating system provides an ISR-frame to prepare a run-time environment for a dedicated user routine. The contents of function will be assigned to this ISR during configuration of the OS. Within an interrupt service routine, usage of OSEKtime operating system services is restricted according to Table 10-1.

The operating system must provide means to define intervals in time where each interrupt may occur at most once. This property must be enforced by the operating system during runtime.

Interrupts will be disabled when they get serviced (see Figure 5-1). Reenabling of interrupts is done at particular points in time defined offline and controlled from the dispatcher table (IEE_1 ... IEE_n).

When supported by processors or other devices, alternative means of interrupt control are permitted.

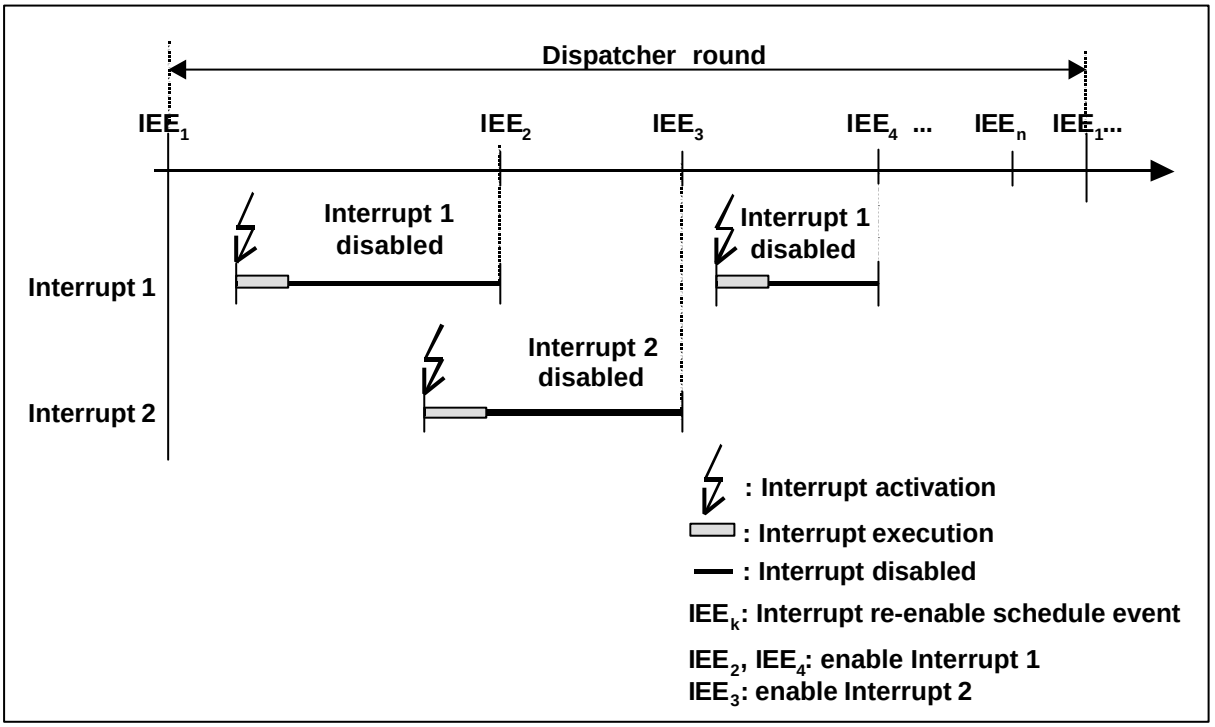

Figure 5-1: Interrupt re-enable schedule event

Non-maskable interrupts should be used with special care because such interrupts may delay OSEKtime OS dispatching interrupts. If it is not possible to guarantee MINT of non-maskable interrupts, the usage of non-maskable interrupts should be prevented.

Nested interrupt support is an implementation specific and hardware specific feature, there are no restrictions connected with nested interrupt implementation in an OSEKtime system.

The application must not enable or disable interrupts during runtime. The operating system must enable all interrupts for which an ISR is specified in the configuration, and disable all other interrupts.

A list of allowed API calls in ISRs can be found in chapter 10.

6 Synchronisation

6.1 Synchronisation of System Time

Each ECU operates with a local time that increments according to the local clock source.

If a global time is available by the synchronisation layer the synchronisation mechanism will be executed:

- At system start-up and after loosing the synchronisation with the global time base.
- During normal operation (no temporary loss of the global time) the adjustment is done repeatedly (e.g., at every end of the dispatching table).

The synchronisation of the local time has to be done in the ground state.

The synchronisation of the local time *can* be done by setting the local time to the value of the global time.

For the application a system call is available (*ttGetOSSyncStatus*) to detect if the local time is synchronous to the global one or not.

The global time is provided by the *synchronisation layer* which has the knowledge about the start of every dispatcher round. The *synchronisation layer* will in general be provided by FTCom. If FTCom is not used the functionality of the *synchronisation layer* as described in FTCom must be provided by an additional software module.

The operating system offers the *ttSyncTimes* API service which is used by the *synchronisation layer* to provide the operating system with the current global time and the value of the global time at the start of the current dispatcher round.

For a detailed description of *ttSyncTimes* refer to the API specification (see chapter 10).

6.2 Start-up and Resynchronisation

Requirements on the local and global time:

- It must be possible to represent one complete dispatcher round without an overflow of the global time. One overflow during one dispatcher round must be considered.
- The value domain of the global time must be configurable and must be identical at the OSEKtime OS and at the Synchronisation Layer.
- The time values have to have the granularity of the global time, the rate between local and global time must be defined as a constant.

6.2.1 Synchronisation Methods

At system start-up and after loosing the synchronisation with the global time base the following three scenarios (see Figure 6-1) are possible:

- *Synchronous start-up*: The ECU does not execute the time-triggered tasks before a global time is available.
- *Asynchronous start-up - hard synchronisation:* The ECU has to perform time-triggered task execution according to the local time without waiting for the synchronisation to the global time. The synchronisation of the local time to the global time is done at the end of a dispatcher round (end of the dispatcher table) by delaying the start of the next dispatcher round.
- *Asynchronous start-up - smooth synchronisation:* The ECU has to perform time-triggered task execution according to the local time without waiting for the synchronisation to the global time. The synchronisation of the local time to the global time is done during several dispatcher rounds by limiting the delay of the start of the next dispatcher round according to pre-defined configuration parameters. The system is considered to be synchronised as long as the delay does not exceed such a limit.

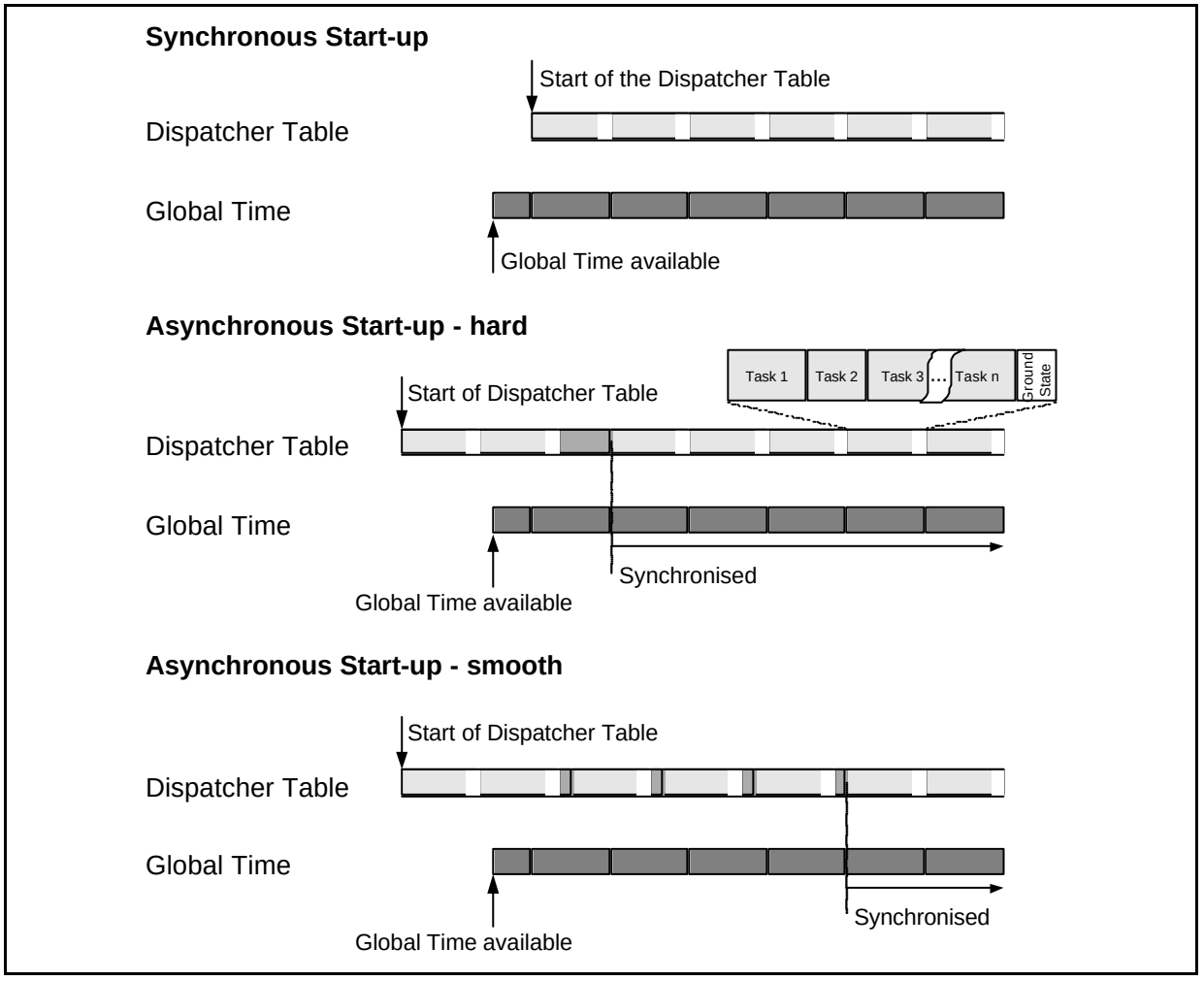

Figure 6-1: Start-up

6.2.2 Synchronisation

When the synchronisation layer calls *ttSyncTimes*, the operating system is able to calculate the drift between the local time and the global time. At the end of the dispatcher round, this difference can be used to extend or shorten the last ground state of the dispatcher round. In the case of smooth synchronisation this adjustment may be limited by some configuration parameters.

After a synchronisation loss, there is no explicit notification of the application. The application can detect a synchronisation loss by the API call *ttGetOSSyncStatus*.

7 System Start-up and Shutdown in a mixed OSEKtime / OSEK/VDX System

Implementations may combine OSEKtime and OSEK OS systems if the OSEK OS subsystem does not interfere with the OSEKtime system. The entire OSEKtime has to have priority over OSEK OS and neither OSEK OS tasks nor ISRs must delay any OSEKtime tasks or ISRs.

Also, the OSEK OS functions for disabling interrupts/interrupt-sources have to be local to OSEK OS and must not affect any interrupts used by OSEKtime. In such a combined system no *ttIdleTask* is used. The OSEK OS subsystem will run during the idle-times of the OSEKtime system. The OSEKtime system can query the state of the OSEK OS subsystem by using the API service *ttGetOSEKOSState.*

The current version of start-up and shutdown describes only a one vendor solution of a combined OSEKtime / OSEK/VDX system, therefore a detailed interface specification between OSEKtime and OSEK/VDX is not necessary.

7.1 Start-up of mixed OSEKtime and OSEK/VDX OS Systems

The start-up of a combined OSEK OS/OSEKtime system is initiated by calling the API service *ttStartOS*. This will also initiate the start-up of the OSEK OS system. However, the OSEKtime system has to be started first within a defined start-up time. The start-up of the OSEK OS system must not cause unbounded delay of this start-up time, even if start-up of the OSEK OS subsystem fails.

OSEKtime start-up completes first and OSEK OS tasks and ISRs will not be activated before the *ttIdleTask* of an OSEKtime-only system would start to run.

7.2 Shutdown of mixed OSEKtime and OSEK/VDX OS Systems

Two types of shutdown-procedures are supported. A local shutdown of the OSEK OS subsystem which does not affect the OSEKtime system and a global shutdown of the entire system.

The local shutdown will not affect the OSEKtime system. The *ShutdownHook* (which has to return) will be called and the OSEK OS system will stop running, i.e. no OSEK OS tasks or ISRs will be executed. OSEKtime is able to notice a local shutdown by calling the API service *ttGetOSEKOSState.*

The global shutdown will immediately shut down both operating systems. *ttShutdownHook* will be called and if it does return, the entire system will be shut down. The OSEK OS *ShutdownHook* is not called to guarantee a shutdown-time for OSEKtime. If necessary the OSEK OS *ShutdownHook* may be called within *ttShutdownHook*.

The local shutdown can be initiated by calling *ShutdownOS* from the OSEK OS subsystem.

Global shutdown can be initiated by calling *ttShutdownOS* either from the OSEKtime system or from the OSEK OS subsystem.

It is recommended to provide a configuration option which prevents OSEK OS from initiating a global shutdown.

8 Inter-Task Communication

For an OSEKtime implementation to be compliant, message handling for inter-task communication (or intra-processor communication) has to be offered. The minimum functionality to be supported is the inter-task communication as described in the OSEKtime FTCom specification.

If an implementation offers even more functionality which is specified in the FTCom specification the implementation must stick to syntax and semantic of the OSEKtime FTCom functionality.

For more details, refer to the OSEKtime FTCom specification.

9 Error Handling

The error handling of OSEKtime is equivalent to OSEK/VDX.

9.1 Error Handling for Deadline Violation

If a deadline violation is detected the dispatcher calls the *ttErrorHook* routine (see Figure 9-1) that has to be programmed by the user (Error code: TT_E_OS_DEADLINE). Thus, the user can implement an application specific error handling for deadline violations. After executing the *ttErrorHook*, *ttShutdownOS* is called from the operating system. The operating system will shut down and call the hook routine *ttShutdownHook* (Error code: TT_E_OS_DEADLINE).

The user is free to define any system behaviour in *ttShutdownHook* e.g. not to return from the routine. If *ttShutdownHook* returns, the operating system jumps to the instruction following the initial call to *ttStartOS*.

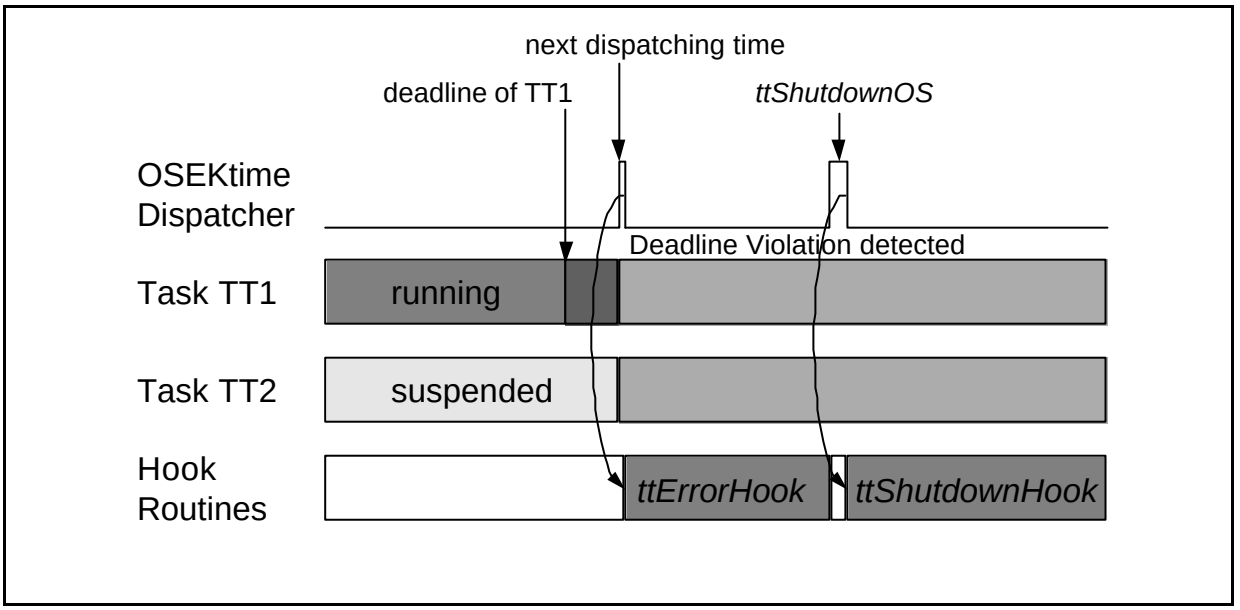

Figure 9-1: Deadline Monitoring

10 Specification of OSEKtime Operating System Services

This chapter is structured according to the original OSEK/VDX OS-specification. Sections 10.3 to 10.7 include a classification of all OSEKtime OS system services. The table in Section 10.8 gives an overview which services and elements could be called within which kind of tasks.

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.

Structure of the Description

OSEKtime 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 and a description of constants. A description of the group-specific constructional elements and system services follows. The last items are additional conventions.

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.

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).

10.1 Common Data Types

ttStatusType

This data type is used for all status information the API services offer. The normal return value is TT_E_OS_OK which is associated with the value of OSEK's E_OK.

The following error values are defined:

All errors of API services:

- TT_E_OS_ID: corresponds to E_OS_ID OSEK OS error code
- TT_E_OS_DEADLINE: task deadline violation - additional OSEKtime OS error code

If the only possible return status is TT_E_OS_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 OSEKtime 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 TT_E_OS_SYS_

Examples:

- TT_E_OS_SYS_STACK
- TT_E_OS_SYS_SCHEDOVERFLOW
- ... 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 OSEKtime OS do not overlap the names and range of numbers of other OSEK/VDX services (i.e. OSEK/VDX OS and OSEK/VDX COM/NM) or the range of numbers of the API error values according to the OSEK/VDX binding specification.

10.2 General Naming Conventions

The following prefixes are used for all OSEKtime OS constructional elements, data types, constants, error codes and system services:

- "tt" prefix is used for constructional elements, data types and system services;
- "TT_E_OS_" prefix is used for error codes;
- "TT" prefix is used for constants.

This is to ensure that no name clashes occur.

10.3 Task Management

10.3.1 Data Types

ttStatusType

This data type is identical with StatusType in the binding specification.

ttTaskType

This data type identifies a task.

ttTaskRefType

This data type points to a variable of ttTaskType.

ttTaskStateType

This data type identifies the state of a task.

ttTaskStateRefType

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

10.3.2 Constants

`TT_RUNNING`	Constant of data type ttTaskStateType for task state *running*.
`TT_PREEMPTED`	Constant of data type ttTaskStateType for task state *preempted*.
`TT_SUSPENDED`	Constant of data type ttTaskStateType for task state *suspended*.
`TT_INVALID_TASK`	Constant of data type ttTaskType for a not defined task.

10.3.3 Constructional Elements

No constructional elements which may be called during runtime are supported.

10.3.4 System Services

10.3.4.1 ttGetTaskID

Syntax:	ttStatusType	ttGetTaskID (ttTaskRefType <TaskID>)
Parameter (In):	none	
Parameter (Out):	TaskID	reference to the task which is currently running.
Description:	*ttGetTaskID* returns the information about the TaskID of the task which is currently in the *running* state.	
Particularities:	Allowed on task level, ISR level and *ttErrorHook* hook routine. This service is intended to be used by library functions and in the *ttErrorHook* hook routine.	
Status:	No error, TT_E_OS_OK	

10.3.4.2 ttGetTaskState

Syntax:	ttStatusType	ttGetTaskState (ttTaskType <TaskID>, ttTaskStateRefType <State>)
Parameter (In):	TaskID	Reference to the task
Parameter (Out):	State	Reference to the state of the task <TaskID>
Description:	Returns the state of a task (*running*, *preempted*, *suspended*) at the time of calling *ttGetTaskState* by the output parameter State.	
Particularities:	The service may be called from interrupt service routines, task level, and *ttErrorHook* hook routine. This service should be used with special care in time-triggered systems because the task state may be changed during *ttGetTaskState* service execution and the result may already be incorrect at the time of evaluation.	
Status:	No error, TT_E_OS_OK, <TaskID> is invalid, TT_E_OS_ID	

10.3.5 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:

```
ttTASK (TaskName)
{
}
```

With the macro `ttTASK` 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.

For the definition of the ttIdleTask the same pattern is used.

10.4 Interrupt Handling

Within the application, an interrupt service routine is defined according to the following naming convention:

```
ttISR (FuncName)
{
}
```

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

10.5 Operating System Execution Control

10.5.1 Data types

ttAppModeType

This data type represents the application mode.

ttAppModeRefType

Reference to the application mode of data type ttAppModeType.

ttOSEKOSStateType

This data type represents the state of an OSEK OS subsystem.

ttOSEKOSStateRefType

Reference to the state of an OSEK OS subsystem of data type ttOSEKOSStateType.

10.5.2 Constants

`TT_OS_DEFAULTAPPMODE`

Default application mode, always a valid parameter to *ttStartOS*. Constant of data type ttAppModeType.

`TT_OS_OSEKOSUP`

There is an OSEK OS subsystem running. Constant of data type ttOSEKOSStateType.

`TT_OS_OSEKOSDOWN`

There is no OSEK OS subsystem or it has not yet been started or has been shut down. Constant of data type ttOSEKOSStateType.

10.5.3 System Services

10.5.3.1 ttSwitchAppMode

Syntax:	ttStatusType	ttSwitchAppMode(ttAppModeType <Mode>)
Parameter (In):	Mode	application mode
Parameter (Out):	none	
Description:	This service performs switching between different dispatcher tables during runtime, without losing synchronisation. The new application mode is set by the parameter <Mode>. The actual switch happens at the end of the current dispatcher round. The length of the dispatcher round is not changed.	
Particularities:	Allowed for time-triggerd tasks and ISRs.	
Status:	No error, TT_E_OS_OK	

10.5.3.2 ttGetActiveApplicationMode

Syntax:	ttStatusType	ttGetActiveApplicationMode (ttAppModeRefType <Mode>)
Parameter (In):	none	
Parameter (Out):	Mode	Reference to the active application mode of data type ttAppModeType.
Description:	This service returns the current application mode. It may be used to write mode dependent code.	
Particularities:	Allowed for task, ISR and all hook routines.	
Status:	No error, TT_E_OS_OK	

10.5.3.3 ttGetOSEKOSState

Syntax:	ttOSEKOSStateType	ttGetOSEKOSState (ttOSEKOSStateRefType <State>)
Parameter (In):	none	
Parameter (Out):	Mode	Reference to the state of an OSEK OS subsystem of data type OSEKOSStateType.
Description:	This service returns the state of an OSEK OS subsystem.	
Particularities:	Allowed for task, ISR and all hook routines.	
Status:	No error, TT_E_OS_OK	

10.5.3.4 ttStartOS

Syntax:	void	ttStartOS (ttAppModeType <Mode>)
Parameter (In):	Mode	application mode
Parameter (Out):	none	
Description:	This system service starts the operating system in a specific mode.	
Particularities:	Only allowed outside of the operating system, therefore implementation specific restrictions may apply. This call does not need to return.	
Status:	none	

10.5.3.5 ttShutdownOS

Syntax:	void	ttShutdownOS (ttStatusType <Error>)
Parameter (In):	Error	error occured
Parameter (Out):	none	
Description:	The user can call this system service to abort the overall system (e.g. emergency off). The OSEKtime operating system also calls this function internally, if it has reached an undefined internal state and is no longer ready to run (for example if stack overflow has been detected). The hook routine *ttShutdownHook* is always called (with <Error> as argument) before shutting down the operating system.	
Particularities:	After this service the OSEKtime operating system is shut down. Allowed at task level, ISR level, in *ttErrorHook* and *ttStartupHook* hook routines, and also called internally by the operating system. If the operating system calls *ttShutdownOS* it never uses TT_E_OS_OK as the passed parameter value. Depending on the configuration this service might also be called by an OSEK OS subsystem.	
Status:	none	

10.6 Hook Routines

The usage of all hook routines is mandatory.

10.6.1 ttErrorHook

Syntax:	void	ttErrorHook (ttStatusType <Error>)
Parameter (In):	Error	error occured
Parameter (Out):	none	
Description:	This hook routine is called by the OSEKtime operating system at the end of a system service which returns *ttStatusType* not equal TT_E_OS_OK (e.g. *ttGetTaskState* system service). It is called before returning to the task or ISR level. Also this hook routine is called when task deadline violation is detected. The *ttErrorHook* is not called if a system service called from t*tErrorHook* does not return TT_E_OS_OK as status value. Any error by calling of system services from the *ttErrorHook* can only be detected by evaluating the status value.	
Particularities:	-	
Status:	none	

10.6.2 ttStartupHook

Syntax:	void	ttStartupHook (void)
Parameter (In):	none	
Parameter (Out):	none	
Description:	This hook routine is called by the OSEKtime operating system at the end of the operating system initialisation and before the scheduler is running. At this time the application can initialise device drivers etc.	
Particularities:	-	
Status:	none	

10.6.3 ttShutdownHook

Syntax:	void	ttShutdownHook (ttStatusType <Error>)
Parameter (In):	Error	error occured
Parameter (Out):	none	
Description:	This hook routine is called by the OSEKtime operating system when the OS service *ttShutdownOS* has been called. This routine is called during the operating system shutdown.	
Particularities:	*ttShutdownHook* is a hook routine for user defined shutdown functionality.	
Status:	none	

10.7 Synchronisation

10.7.1 Data Types

ttSynchronizationStatusType

This data type represents the synchronisation status of the OSEKtime operating system.

ttSynchronizationStatusRefType

Reference to the status of the OSEKtime operating system of data type ttSynchronizationStatusType.

ttTickType

This data type defines the data type for the count value (count value in ticks).

10.7.2 Constants

`TT_SYNCHRONOUS`

Constant of data type ttSynchronizationStatusType for *synchronous* system state.

`TT_ASYNCHRONOUS`

Constant of data type ttSynchronizationStatusType for *asynchronous* system state.

10.7.3 System Services

10.7.3.1 ttGetOSSyncStatus

Syntax:	ttStatusType	ttGetOSSyncStatus (ttSynchronisationStatusRefType <Status>)
Parameter (In):	none	
Parameter (Out):	Status	Synchronisation status of the system: synchronous or asynchronous system time.
Description:	This service returns the synchronization status of the system. If the system is synchronised, Status will refer to a value equal to TT_SYNCHRONOUS. If the system is not yet synchronised, Status will refer to a value of data type ttSynchronisationStatusType equal to TT_ASYNCHRONOUS.	
Particularities:	-	
Status:	No error, TT_E_OS_OK.	

10.7.3.2 ttSyncTimes

Syntax:	ttStatusType	ttSyncTimes (ttTickType <GlobalTime>, ttTickType <ScheduleTime>)
Parameter (In):	GlobalTime	The current network-wide synchronised time.
	ScheduleTime	The value of the global time at start of the last dispatching table.
Parameter (Out):	None	
Description:	This service provides the operating system with the current global time. It is used to calculate the difference between global and local time and perform synchronisation as needed.	
Particularities:	Allowed for tasks and ISRs	
Status:	No error, TT_E_OS_OK.	

10.8 API Services

In TT-Tasks, interrupt service routines (ISRs) and hook routines the following OSEKtime system services and constructional elements can be used (see Table 10-1):

Service	*Tasks*	*ttIdle Task*	*ISRs*	*ttStartup Hook*	*ttShutdown Hook*	*ttError Hook*
ttGetOSEKOSState	allowed	allowed	allowed	allowed	allowed	allowed
ttGetTaskID	allowed	allowed	allowed	--	--	allowed
ttGetTaskState	allowed	allowed	allowed	--	--	allowed
ttGetActiveApplication Mode	allowed	allowed	allowed	allowed	allowed	allowed
ttSwitchAppMode	allowed	allowed	allowed	--	--	--
ttStartOS	--	--	--	--	--	--
ttShutdownOS	allowed	allowed	allowed	--	--	allowed
ttSyncTimes	allowed	--	allowed	--	--	--
ttGetOSSyncStatus	allowed	allowed	allowed	--	allowed	allowed

Table 10-1: API services

11 Index

11.1 List of Services, Data Types, and Constants

11.2 List of Figures

11.3 List of Tables

12 History

Version	Date	Remarks	
1.0	July 24th 2001	Authors:	
		Volker Barthelmann	3SOFT
		Anton Schedl	BMW
		Elmar Dilger	Bosch
		Thomas Führer	Bosch
		Bernd Hedenetz	DaimlerChrysler
		Jens Ruh	DaimlerChrysler
		Matthias Kühlewein	DaimlerChrysler
		Emmerich Fuchs	DeComSys
		Yaroslav Domaratsky	Motorola
		Andreas Krüger	Motorola, since 04/01 Audi
		Patrick Pelcat	Peugeot Citroen
		Martin Glück	TTTech
		Stefan Poledna	TTTech
		Thomas Ringler	University of Stuttgart
		Brian Nash	Wind River
		Tim Curtis	Wind River