INTERNATIONAL STANDARD

ISO 17356-3

First edition
2005-11-01

Road vehicles — Open interface for embedded automotive applications —

Part 3: **OSEK/VDX Operating System (OS)**

Véhicules routiers — Interface ouverte pour applications automobiles embarquées —

Partie 3: Système d'exploitation OSEK/VDX

Reference number
ISO 17356-3:2005(E)

Contents

Page

Foreword v
Introduction vi
1 Scope 1
2 Normative references 1
3 Architecture of the operating system "OS" 1
3.1 Processing levels 1
3.2 Conformance classes 3
3.3 Relationship between OS and OSEKtime OS 4
4 Task management 5
4.1 Task concept 5
4.2 Task state model 5
4.3 Activating a task 8
4.4 Task switching mechanism 8
4.5 Task priority 8
4.6 Scheduling policy 9
4.7 Termination of tasks 12
5 Application modes 12
5.1 General 12
5.2 Scope of application modes 12
5.3 Start-up performance 13
5.4 Support for application modes 13
6 Interrupt processing 13
6.1 General 13
7 Event mechanism 14
8 Resource management 16
8.1 General 16
8.2 Behaviour during access to occupied resources 16
8.3 Restrictions when using resources 17
8.4 Scheduler as a resource 17
8.5 General problems with synchronization mechanisms 17
8.6 Priority Ceiling Protocol 18
8.7 Priority Ceiling Protocol with extensions for interrupt levels 19
8.8 Internal resources 21
9 Alarms 22
9.1 General 22
9.2 Counters 22
9.3 Alarm management 22
9.4 Alarm-callback routines 23
10 Messages 24
11 Error handling, tracing and debugging 24
11.1 Hook routines 24
11.2 Error handling 25
11.3 System start-up 26
11.4 System shutdown 28
11.5 Debugging 28
12 Description of system services 29

12.1 Definition of system objects 29
12.2 Conventions 29

13 Specification of OS services 31
13.1 Basics 31
13.2 Common data types 32
13.3 Task management 33
13.4 Interrupt handling 38
13.5 Resource management 41
13.6 Event control 43
13.7 Alarms 46
13.8 OS execution control 50
13.9 Hook routines 51

14 Implementation- and application-specific topics 54
14.1 General 54
14.2 Implementation hints 54
14.3 Application design hints 56
14.4 Implementation-specific tools 60

Foreword

ISO (the International Organization for Standardization) is a worldwide federation of national standards bodies (ISO member bodies). The work of preparing International Standards is normally carried out through ISO technical committees. Each member body interested in a subject for which a technical committee has been established has the right to be represented on that committee. International organizations, governmental and non-governmental, in liaison with ISO, also take part in the work. ISO collaborates closely with the International Electrotechnical Commission (IEC) on all matters of electrotechnical standardization.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of technical committees is to prepare International Standards. Draft International Standards adopted by the technical committees are circulated to the member bodies for voting. Publication as an International Standard requires approval by at least 75 % of the member bodies casting a vote.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO shall not be held responsible for identifying any or all such patent rights.

ISO 17356-3 was prepared by Technical Committee ISO/TC 22, *Road vehicles*, Subcommittee SC 3, *Electrical and electronic equipment*.

ISO 17356 consists of the following parts, under the general title *Road vehicles — Open interface for embedded automotive applications*:

— *Part 1: General structure and terms, definitions and abbreviations terms*

— *Part 2: OSEK/VDX specifications for binding OS, COM and NM*

— *Part 3: OSEK/VDX Operating System (OS)*

— *Part 4: OSEK/VDX Communication (COM)*

— *Part 5: OSEK/VDX Network Management (NM)*

— *Part 6: OSEK/VDX Implementation Language (OIL)*

Introduction

0.1 System philosophy

Automotive applications are characterized by stringent real-time requirements. Therefore, the operating system (OS) offers the necessary functionality to support event-driven control systems.

The specified OS 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 OS is intended for use in any type of control units, it supports time-critical applications on a wide range of hardware. A high degree of modularity and ability for flexible configuration are prerequisites to making the OS suitable for low-end microprocessors and complex control units alike. These requirements have been supported by definition of “conformance classes” (see 3.2) 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.

0.1.1 Standardized interfaces

The interface between the application software and the OS is defined by system services. The interface is identical for all implementations of the OS 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.

0.1.2 Scalability

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

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

0.1.3 Error checking

The OS 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 OS 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.

0.1.4 Portability of application software

One of the goals of ISO 17356 is to support the portability and re-usability of application software. Therefore, the interface between the application software and the OS is defined by standardized system services with well-defined functionality. Use of standardized 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 without bigger changes inside the application. The standardized interface (service calls, type definitions and constants) to the operating system supports the portability on source code level. Exchange of object code is not addressed by ISO 17356.

The application software lies on the operating system and in parallel on an application-specific Input/Output System interface which is not standardized in ISO 17356. The application software module can have several interfaces. There are interfaces to the OS 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 works directly with microcontroller modules.

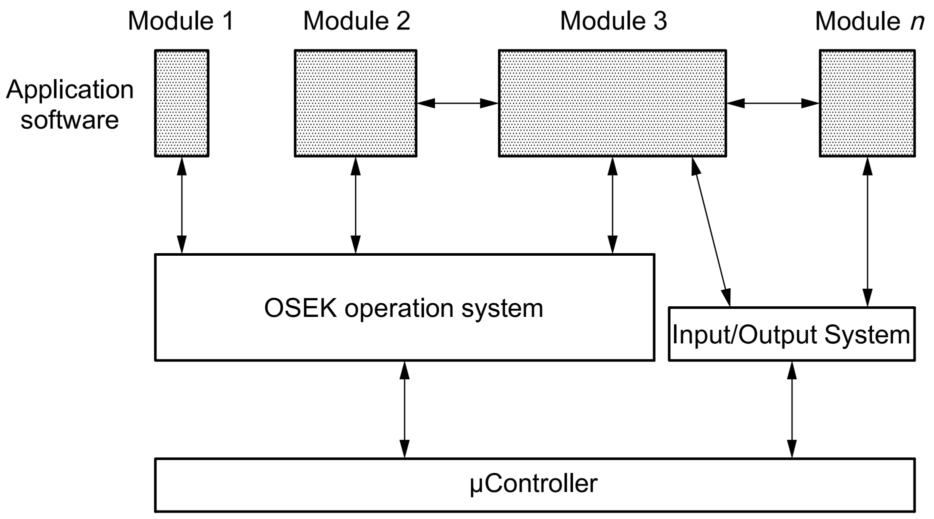

Figure 1 — Software interfaces inside ECU[1)]

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

— 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; and

— placement of the API calls.

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

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

0.1.5 Support of portability

The certification process ensures the conformance of different implementations to the specification. Clause 14 of this International Standard collects implementation specific details which should be regarded to increase portability of an application between various implementations. Herein, only the OS interface to the application is considered.

0.1.6 Special support for automotive requirements

Specific requirements for the OS arise in the application context of software development for automotive control units. The following features address requirements such as reliability, real-time capability and cost sensitivity:

— The OS is configured and scaled statically. The user statically specifies the number of tasks, resources and services required.

— The specification of the OS supports implementations capable of running on ROM, i.e. the code could be executed from *Read-Only-Memory*.

— The OS supports portability of application tasks.

— The specification of the OS provides a predictable and documented behaviour to enable OS implementations, which meet automotive real-time requirements.

— The specification of the OS allows the implementation of predictable performance parameters.

0.2 Purpose of this document

The following description is to be regarded as a generic description which is mandatory for any implementation of the OS. 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.

This part of ISO 17356 leaves a certain amount of flexibility. On the one hand, the description is generic enough for future upgrades; on the other hand, part of the description is explicitly specified and implementation-specific.

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

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.

0.3 Structure of this document

0.3.1 General

In the following text, the clauses of this International Standard are described briefly:

0.3.2 Clause 3 — Architecture of the operating system "OS"

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

0.3.3 Clause 4 — Task management

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

0.3.4 Clause 5 — Application modes

This clause describes application modes and how they are supported.

0.3.5 Clause 6 — Interrupt processing

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

0.3.6 Clause 7 — Event mechanism

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

0.3.7 Clause 8 — Resource management

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

0.3.8 Clause 9 — Alarms

This clause 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).

0.3.9 Clause 10 — Messages

The message handling for intra-processor communication is added to ISO 17356-3. Full message handling is described in ISO 17356-4. The exact subset to be implemented is described in ISO 17356-4.

0.3.10 Clause 11 — Error handling, tracing and debugging

This clause describes the mechanisms to achieve centralized error handling. It also describes the services to initialize and shut down the system.

0.3.11 Clause 12 — Description of system services

This clause describes the conventions used for description.

0.3.12 Clause 13 — Specification of operating system services

This clause 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.

0.3.13 Clause 14 — Implementation and application-specific topics

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

0.4 Summary

0.4.1 General

The OS provides a pool of different services and processing mechanisms. It is built according to the user's configuration instructions at system generation time.

Four conformance classes are described, meant to satisfy different requirements concerning functionality and capability of the OS. Thus, the user can adapt the OS to the control task and the target hardware. It is not possible to modify the OS later at execution time.

Applications which have been written for a certain conformance class are portable to 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 does the OS implementation conform to ISO 17356-3. The service groups are structured in terms of functionality.

0.4.2 Task management

Task management includes:

— activation and termination of tasks; and

— management of task states and task switching.

0.4.3 Synchronization

The OS supports the following means of synchronization 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; and

— event management for task synchronization.

0.4.4 Interrupt management

This includes services for interrupt processing.

0.4.5 Alarms

Alarms can be either relative or absolute.

0.4.6 Intra-processor message handling

This includes services for exchange of data.

0.4.7 Error treatment

This includes mechanisms supporting the user in case of various errors.

Road vehicles — Open interface for embedded automotive applications —

Part 3: OSEK/VDX Operating System (OS)

1 Scope

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.

This part of ISO 17356 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. It also specifies the operating system Application Program Interface (API).

General conventions, explanations of terms and abbreviations have been compiled in ISO 17356-1. ISO 17356-6 describes implementation and system generation aspects.

The specification of the OS represents a uniform environment which supports efficient utilization of resources for automotive control unit application software. The OS is a single processor operating system meant for distributed embedded control units.

2 Normative references

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO 17356-1, *Road vehicles — Open interface for embedded automotive applications — Part 1: General structure and terms, definitions and abbreviations terms*

ISO 17356-2, *Road vehicles — Open interface for embedded automotive applications — Part 2: OSEK/VDX specifications for binding OS, COM and NM*

ISO 17356-6, *Road vehicles — Open interface for embedded electronic equipment — Part 6: OSEK/VDX Implementation Language (OIL)*

3 Architecture of the operating system "OS"

3.1 Processing levels

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

The OS 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; and

— tasks (basic tasks and extended tasks).

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

The OS defines three processing levels:

— interrupt level;

— logical level for scheduler; and

— task level.

Within the task level, tasks are scheduled (non, full or mixed preemptive scheduling) 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.

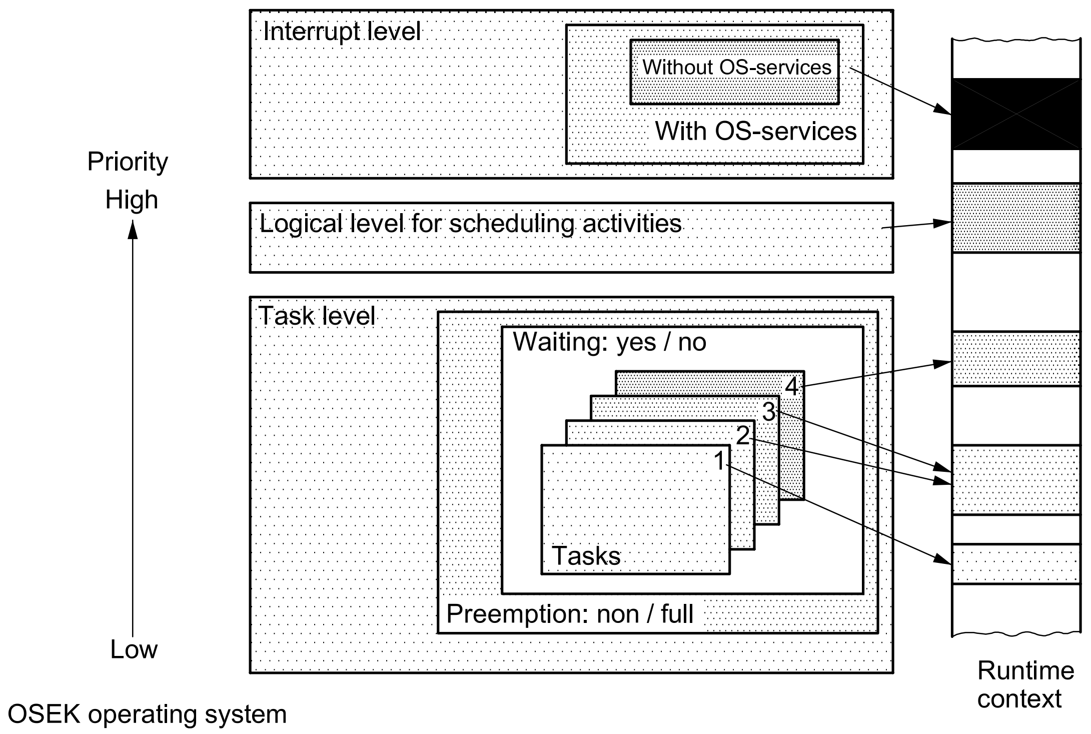

Figure 2 — Processing levels of the operating system

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 4.5.)

Processing levels are defined for the handling of tasks and interrupt routines as a range of consecutive values. Mapping of operating system priorities to hardware priorities is implementation-specific.

NOTE Assignment of a priority to the scheduler is only a logical concept which can be implemented without directly using priorities. Additionally, it does not prescribe any rules concerning the relation of task priorities and hardware interrupt levels of a specific microprocessor architecture.

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 OS. In the following description, these OS features are described as "conformance classes" (CC).

Conformance classes exist to support the following objectives:

— to provide convenient groups of OS features for easier understanding and discussion of the OS;

— to allow partial implementations (which may be certified as compliant) along pre-defined lines; and

— to create an upgrade path from classes of lesser functionality to classes of higher functionality with no changes to the application using related features.

To be certified, the complete conformance class shall be implemented. However, system generation needs only to link those system services that are required for a specific application. It shall not be possible to change conformance classes during execution.

Conformance classes are determined by the following attributes:

— multiple requesting of task activation, as described in 4.3;

— task types, as described in 4.2.4; and

— number of tasks per priority.

All other features are mandatory if not explicitly stated otherwise.

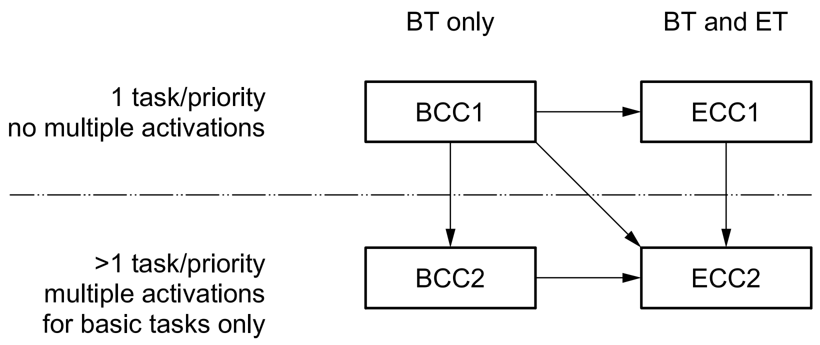

Figure 3 — Restricted upward compatibility for conformance classes

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); and

— 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 Table 1.

Table 1 — The minimum requirements for Conformance Classes

	BCC1	**BCC2**	**ECC1**	**ECC2**
Multiple requesting of task activation	no	yes	Basic task (BT): no Extended task (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		16	
Resources	RES_SCHEDULER	8 (including RES_SCHEDULER)		
Internal resources	2			
Alarm	1			
Application Mode	1			

3.3 Relationship between OS and OSEKtime OS

OSEKtime OS (www.osek-vdx.org) is an OS especially tailored to the needs of time-triggered architectures. It allows ISO 17356-3 to coexist with OSEKtime OS. Conceptually, OSEKtime assigns its idle time to be used by ISO 17356-3. OS interrupts and tasks have less importance (lower priority) than similar entities in OSEKtime OS.

The OS interfaces, and the definition of system calls, do not change if the OS 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 the OS is only locally responsible. These deviations are specifically mentioned within this part of ISO 17356.

On top of this, there is functionality defined within OSEKtime which imposes restrictions on the implementation of the 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

4.1.1 General

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

The OS provides a task switching mechanism (see scheduler, 4.4), including a mechanism which is active when no other system or application functionality is active. This mechanism is called idle-mechanism. Two different task concepts are provided by the OS:

— basic tasks; and

— extended tasks.

4.1.2 Basic tasks

Basic tasks only release the processor if:

— they terminate;

— the OS switches to a higher-priority task; or

— an interrupt occurs which causes the processor to switch to an interrupt service routine (ISR).

4.1.3 Extended tasks

Extended tasks are distinguished from basic tasks by being allowed to use the OS *WaitEvent*, which may result in a *waiting* state (see Clause 7 and 13.6.3.4). 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 OS, management of extended tasks is in principal more complex than management of basic tasks, and requires more system resources.

4.2 Task state model

4.2.1 General

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

A task shall 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 OS is responsible for saving and restoring task context in conjunction with task state transitions whenever necessary.

4.2.2 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 shall *wait* for at least one event (see Clause 7).

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

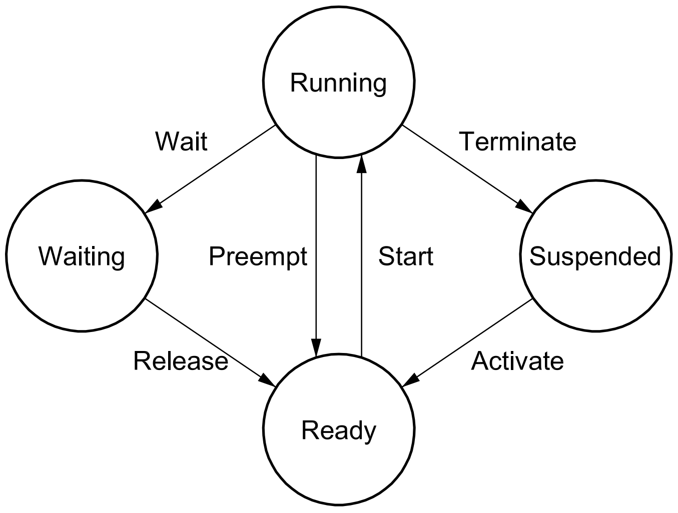

Figure 4 — Extended task state model

Table 2 — States and status transitions for extended tasks

Transition	Former state	New state	Description
activate	*suspended*	*ready*	A new task is set into the *ready* state by a system service. The operating system ensures that the execution of the task starts 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.

Termination of a task is only possible if the task terminates itself (“self-termination”). This restriction reduces complexity of an OS. 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.

4.2.3 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. Basic tasks consist of the following 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.

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

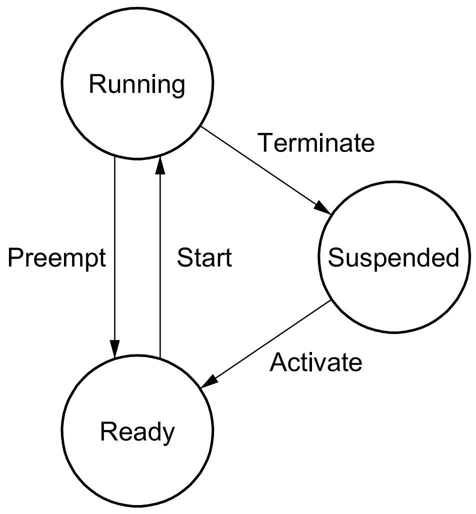

Figure 5 — Basic task state model

Table 3 — States and status transitions for basic tasks

Transition	Former state	New state	Description
activate	*suspended*	*ready*	A new task is set into the *ready* state by a system service. The operating system ensures that the execution of the task starts 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.
NOTE 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.			

4.2.4 Comparison of the task types

Basic tasks have no *waiting* state, and thus only comprise synchronization points at the beginning and the end of the task. Application parts with internal synchronization points shall be implemented by more than one basic task. 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 synchronization requests are active. Whenever current information for further processing is missing, the extended task switches over into the *waiting* state. It exits this state whenever corresponding events signal the receipt or the update of the desired data or events. Extended tasks also comprise more synchronization points than basic tasks.

4.3 Activating a task

4.3.1 General

Task activation is performed using the OS services *ActivateTask* or *ChainTask*. After activation, the task is ready to execute from the first statement.

The OS does not support C-like parameter passing when starting a task. Those parameters should be passed by message communication (see Clause 10) or by global variables.

4.3.2 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 OS 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 OS to execute various tasks concurrently. Therefore, the scheduling policy has to be defined clearly (see 4.6).

The entity deciding which task shall be started and the triggering of all necessary OS internal activities is called the “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 (see 8.4).

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. the user shall not be able to change it at the time of execution. However, in particular cases the OS can treat a task with a defined higher priority (see 8.6).

Tasks of identical priority are supported in the conformance classes BCC2 and ECC2 (see 3.2).

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 preempted task is considered to be the first (oldest) task in the *ready* list of its current priority.

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

Figure 6 shows an example implementation of the scheduler using a queue 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 shall 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.

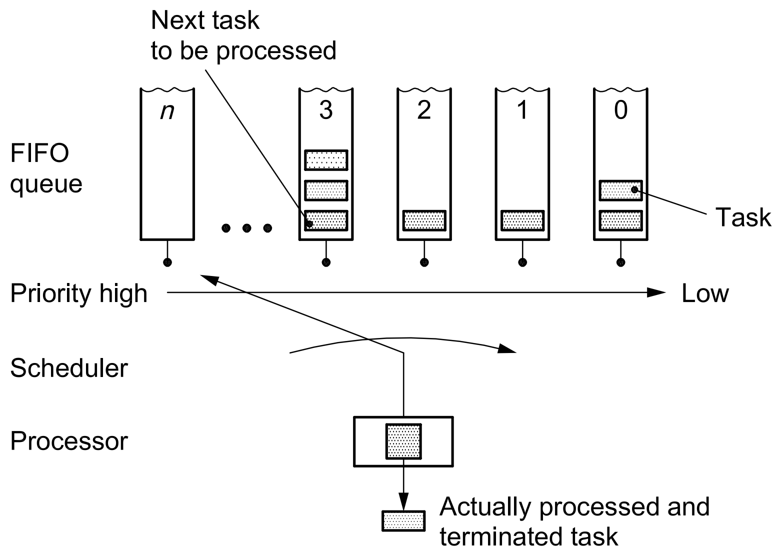

Figure 6 — Scheduler: order of events

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 Full preemptive scheduling

Full preemptive 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 OS. Full preemptive scheduling puts the *running* task into the *ready* state, as soon as a higher-priority task has gotten *ready*. The task context is saved so that the preempted task can be continued at the location where it was preempted.

With full preemptive 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 synchronization between tasks. As each task can be theoretically rescheduled at any location, access to data which are used jointly with other tasks shall be synchronized.

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

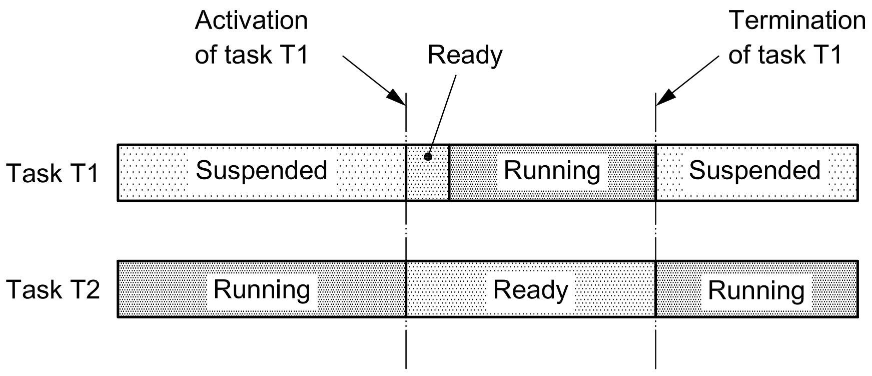

Figure 7 — Full preemptive scheduling

In the case of a full preemptive system, the user shall constantly expect preemption of the *running* task. If a task fragment shall not be preempted, this can be achieved by blocking the scheduler temporarily via the system service *GetResource*.

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

— successful termination of a task (system service *TerminateTask*, see 13.3.3.2);

— successful termination of a task with explicit activating of a successor task (system service *ChainTask*, see 13.3.3.3);

— activating a task at task level (e.g. system service *ActivateTask*, see 13.3.3.1, message notification mechanism, alarm expiration; if task activation is defined, see 9.3);

— explicit *wait* call if a transition into the *waiting* state takes place (extended tasks only, system service *WaitEvent*, see 13.6.3.4);

— setting an event to a *waiting* task at task level (e.g. system service *SetEvent*, see 13.6.3.1, message notification mechanism, alarm expiration; if event setting defined, see 9.3);

— release of resource at task level (system service *ReleaseResource*, see 13.5.3.2); and

— return from interrupt level to task level.

During interrupt service routines, no rescheduling is performed (see Figure 2).

Applications using the scheduling strategy “full preemptive scheduling” do not need the system service *Schedule*, but other scheduling policies make use of this system service. 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/she assumes a correct assignment of the CPU.

4.6.2 Non-preemptive scheduling

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

Non-preemptive scheduling imposes particular constraints on the possible timing requirements of tasks. Specifically, the non-preemptable 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 8, 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).

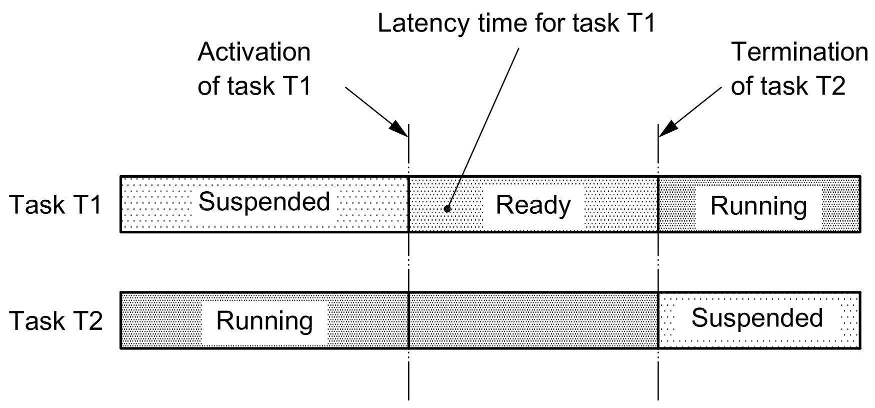

Figure 8 — Non-preemptive scheduling

4.6.3 Points of rescheduling

In the case of a non-preemptable task, rescheduling shall take place exactly in the following cases:

— Successful termination of a task (system service *TerminateTask*, see 13.3.3.2).

— Successful termination of a task with explicit activation of a successor task (system service *ChainTask*, see 13.3.3.3).

— Explicit call of scheduler (system service *Schedule*, see 13.3.3.4).

— A transition into the *waiting* state takes place (system service *WaitEvent*, see 13.6.3.4).

NOTE 1 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.

Implementations of non-preemptive systems may prescribe that OS services which cause rescheduling may only be called at the highest task program level (not in task subfunctions).

NOTE 2 A task switch at these points of scheduling usually requires saving of less task context information.

4.6.4 Groups of tasks

The OS allows tasks to combine aspects of preemptive and non-preemptive scheduling by defining groups of tasks. For tasks which have the same or lower priority as the highest priority within a group, the tasks within the group behave like non-preemptable tasks: rescheduling shall only take place at the points of rescheduling described in 4.6.2. For tasks with a higher priority than the highest priority within the group, tasks within the group behave like preemptable tasks (see 4.6.1).

Subclause 8.8 describes the mechanism of defining groups by using internal resources. Non-preemptable tasks are the most common usage of the concept of internal resources; they are tasks with a special internal resource of highest task priority assigned.

4.6.5 Mixed preemptive scheduling

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

The definition of a non-preemptable task makes sense in a full preemptive 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 is not to be preempted.

Many applications comprise only a few parallel tasks with a long execution time, for which a full preemptive OS would be convenient and many short tasks with a defined execution time where non-preemptive scheduling would be more efficient. For this configuration, the mixed preemptive scheduling policy was developed as a compromise (see also the design hint in 14.3.5).

4.6.6 Selecting the scheduling policy

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

The task type (basic or extended) is independent from the task's scheduling type (preemptable or non-preemptable). A full preemptive system may therefore contain basic tasks, and a non-preemptive system extended tasks.

If an OS service is running, preemption and context switch might be delayed until the completion of the service.

4.7 Termination of tasks

In the OS, a task can only terminate itself ("self-termination"). The OS 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 shall terminate itself at the end of its code. When the task is ended, a call shall be made to *TerminateTask* or *ChainTask;* failure to do so causes undefined behaviour.

5 Application modes

5.1 General

Application modes are designed to allow an OS 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 OS has been started, it shall not be allowed to change the application mode.

5.2 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 and is a clean mechanism for development of totally separate systems. Typically, each application mode uses its own subset of all tasks, ISRs, alarms and timing conditions, although there is no limitation to having a task or ISR running in different modes. Sharing a task/ISR/alarm between different modes is recommended if the same functionality is needed again. If the functionality is not exactly the same, there is a trade-off between runtime and resources: either the application mode shall be dynamically checked, or separate tasks shall be defined.

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

5.3 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. At start-up, the user code using no system services (see Figure 19) shall determine the mode and pass it as a parameter to the API service *StartOS*.

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

It is recommended that only pin states, or similarly easy to assess conditions, be used to determine the mode. The mode shall be determined before the kernel is started and the resulting code is non-portable. A lengthy or complicated starting procedure should be avoided.

The application mode passed to *StartOS* allows the OS to autostart the correct subset of tasks and alarms. The assignment of autostart tasks and alarms to application modes shall be made statically in the ISO 17356-6 file.

5.4 Support for application modes

There is no restriction of application modes to a subset of conformance classes. It is required for all classes. There is no impact on the shutdown functionality. Switching between application modes at runtime is not supported.

6 Interrupt processing

6.1 General

Interrupt (Interrupt Service Routine: ISR) is subdivided into two ISR categories:

- **ISR category 1:** The ISR does not use an operating system service. (Exceptions are some system services to enable and disable interrupts; see Table 4.) 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 OS 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.

Within interrupt service routines, usage of OS services is restricted according to Table 4.

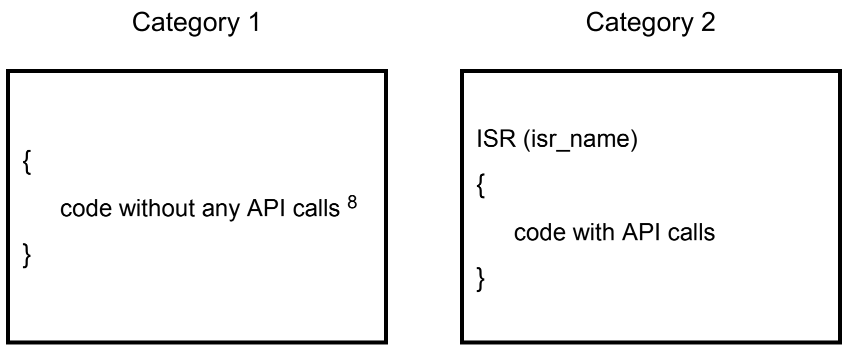

Figure 9 — ISR categories of the operating system

Inside the ISR, no rescheduling shall take place. Rescheduling takes place on termination of the ISR category 2 if a preemptable task has been interrupted and if no other interrupt is active.

The implementation ensures that tasks are executed according to the scheduling points (see 4.6.1). 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 14.3.4.1).

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 ISO 17356. Interrupts are scheduled by hardware while tasks are scheduled by the scheduler. Regarding the interrupt priority levels, there may be restrictions as described in 14.3.4.1. Interrupts can interrupt tasks (preemptable and non-preemptable 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 system services listed in Table 4 can be used.

Fast Disable/Enable API-functions

ISO 17356 offers fast functions to disable all interrupts (see 13.4.2.1, *EnableAllInterrupts*, 13.4.2.2, *DisableAllInterrupts*, 13.4.2.3, *ResumeAllInterrupts* and 13.4.2.4, *SuspendAllInterrupts*), and to disable all interrupts of category 2 (see 13.4.2.5, *ResumeOSInterrupts* and 13.4.2.6, *SuspendOSInterrupts*). Typical usage is to protect short critical sections. An interrupt routine shall not return from an interrupt within such protected critical sections, i.e. a “suspend/disable” shall have a matching “resume/enable”. The only OS service calls allowed between Suspend- and Resume- pairs are further *SuspendOSInterrupts*/*ResumeOSInterrupts* – pairs or *SuspendAllInterrupts*/*ResumeAllInterrupts* – pairs.

7 Event mechanism

The event mechanism:

— is a means of synchronization;

— is only provided for extended tasks; and

— initiates state transitions of tasks to and from the *waiting* state.

Events are objects managed by the OS. 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 OS. 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 is not necessarily an extended task. All tasks can set any events of any non-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 OS provides services for setting, clearing and interrogation of events and for waiting for events to occur.

Any task or ISR of category 2 can set an event for a non-suspended extended task, and thus inform the extended task about any status change via this event.

The receiver of an event shall always be an extended task. 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 shall not use the OS 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 10 explains synchronization of extended tasks by setting events in case of full preemptive scheduling, where extended task T1 has the higher priority.

Figure 10 illustrates the procedures which are affected 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 preempted by T1. T1 resets the event. Thereafter, T1 waits for this event again and the scheduler continues execution of T2.

If non-preemptive scheduling is supposed, rescheduling does not take place immediately after the event has been set (see Figure 11, where extended task T1 is of higher priority).

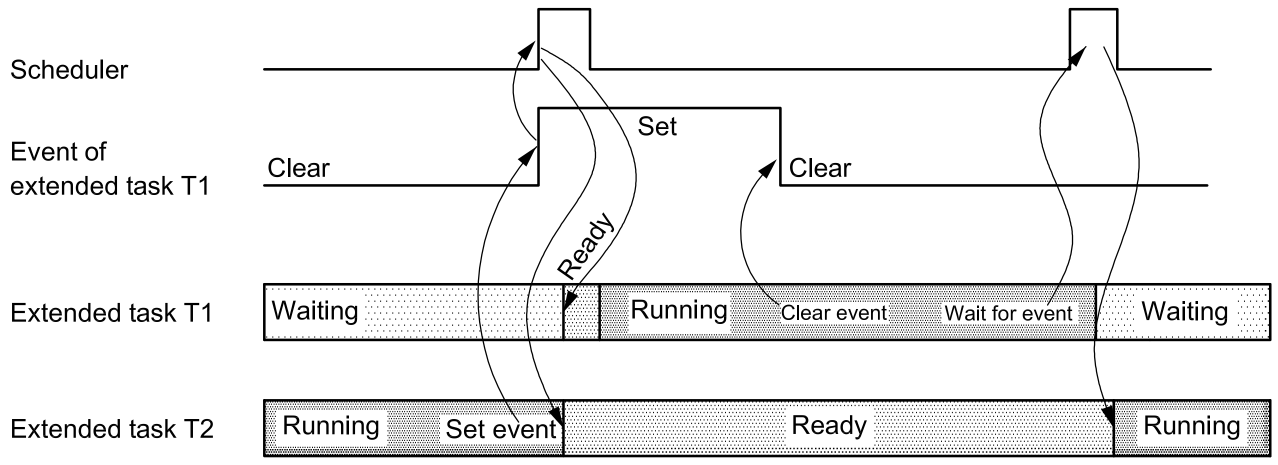

Figure 10 — Synchronization of preemptable extended tasks

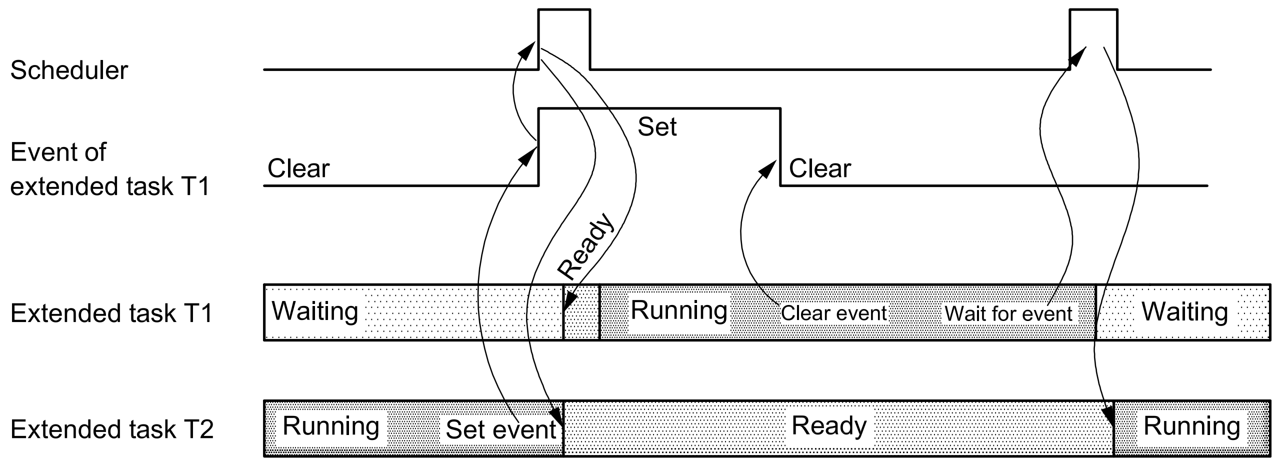

Figure 11 — Synchronization of non-preemptable extended tasks

8 Resource management

8.1 General

The resource management is used to coordinate 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 coordinate concurrent accesses of tasks and interrupt service routines.

Resource management ensures that:

— two tasks cannot occupy the same resource at the same time;

— priority inversion cannot occur;

— deadlocks do not occur by use of these resources; and

— 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 useful in the following cases:

— preemptable tasks;

— non-preemptable tasks, if the user intends to have the application code executed under other scheduling policies, too;

— resource sharing between tasks and interrupt service routines; and

— resource sharing between interrupt service routines.

If the user requires protection against interruptions not only caused by tasks, but also caused by interrupts, he/she can also use the OS services to enable/disable interrupts which do not cause rescheduling (see Clause 6 and 13.4).

8.2 Behaviour during access to occupied resources

OS prescribes the priority ceiling protocol (see 8.6) Consequently, no situation occurs in which a task or an interrupt tries to access an occupied resource.

If the resource concept is used for coordination of tasks and interrupts, the OS ensures also that an interrupt service routine is only processed if all resources which might be occupied by that interrupt service during its routine execution have been released.

ISO 17356 strictly forbids nested access to the same resource. In the rare cases when nested access is needed, it is recommended to use a second resource with the same behaviour as the first resource. The ISO 17356-6 language especially supports the definition of resources with identical behaviour (so-called 'linked resources').

8.3 Restrictions when using resources

TerminateTask, ChainTask, Schedule, *WaitEvent* shall not be called while a resource is occupied. Interrupt service routine shall not be completed with a resource occupied.

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

8.4 Scheduler as a resource

If a task protects itself against preemptions 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 independently of the state of the resource RES_SCHEDULER. However, it prevents the rescheduling of tasks.

8.5 General problems with synchronization mechanisms

8.5.1 Explanation of priority inversion

A typical problem of common synchronization 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. ISO 17356 prescribes the *Priority Ceiling Protocol* (see 8.6) to avoid priority inversion.

Figure 12 illustrates sequencing of the common access of two tasks to a semaphore (in a full preemptive system, task T1 has the highest priority) task T4, which has a low priority, occupies the semaphore S1. T1 preempts 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 preempted 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.

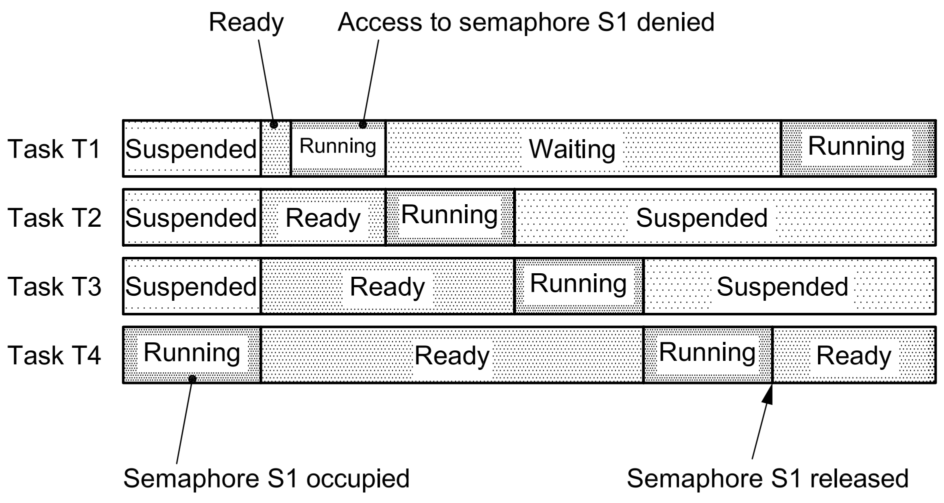

Figure 12 — Priority inversion on occupying semaphores

8.5.2 Deadlocks

Another typical problem of common synchronization 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 13):

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 T2 then tries to occupy semaphore S1, this results in a deadlock.

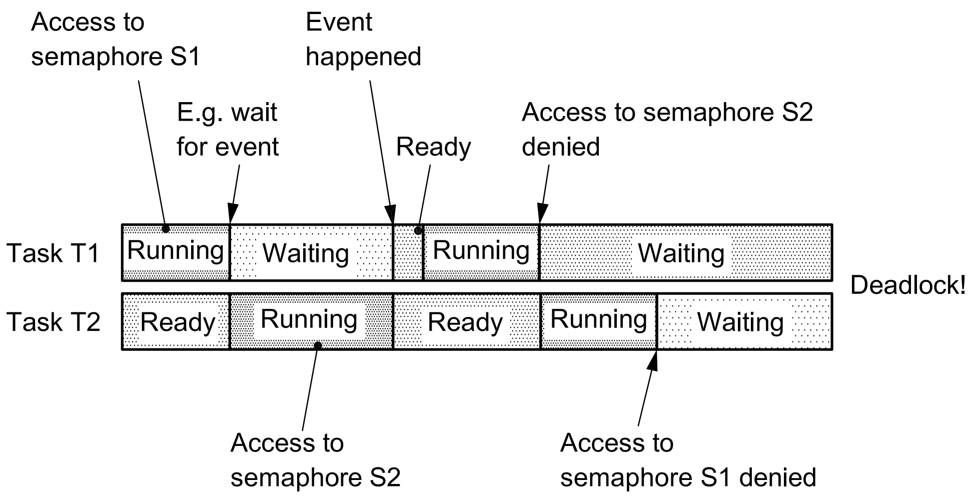

Figure 13 — Deadlock situation using semaphores

8.6 Priority Ceiling Protocol

To avoid the problems of priority inversion and deadlocks, the OS requires the following behaviour:

— At the system generation, each resource shall be statically assigned its own ceiling priority. The ceiling priority shall be set at least to the highest priority of all tasks that access a resource or any of the resources linked to this resource. The ceiling priority shall 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 shall be raised to the ceiling priority of the resource.

— If the task releases the resource, the priority of this task shall 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, another task which might occupy the resource can enter the *running* state. For preemptable tasks, this is a point of rescheduling.

The example shown in Figure 14 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.

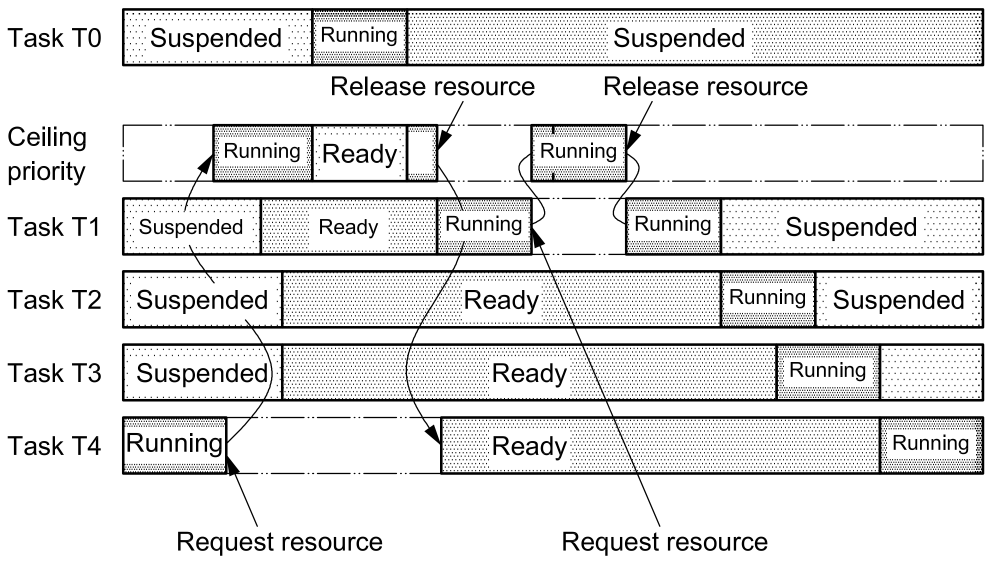

Figure 14 — Resource assignment with priority ceiling between preemptable tasks

8.7 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 shall be assigned to interrupts. The manipulation of software priorities and of hardware interrupt levels is up to the implementation.

— At the system generation, each resource shall be statically assigned its own ceiling priority. The ceiling priority shall be set at least to the highest priority of all tasks and interrupt routines that access a resource or any of the resources linked to this resource. The ceiling priority shall 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 shall 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 shall 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 routine which might occupy the resource can run. For preemptable tasks, this is a point of rescheduling if the new priority of the task is not the virtual priority of an interrupt.

The example shown in Figure 15 describes the following scenario:

The preemptable 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 Priority Ceiling Protocol, the task T1 is still running. Interrupt INT1 occurs. Because of Priority Ceiling Protocol, the task T1 is still running and 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 task 3 is running. After termination of task T3, the task T2 is running. After termination of task T2 the task T1 is continued.

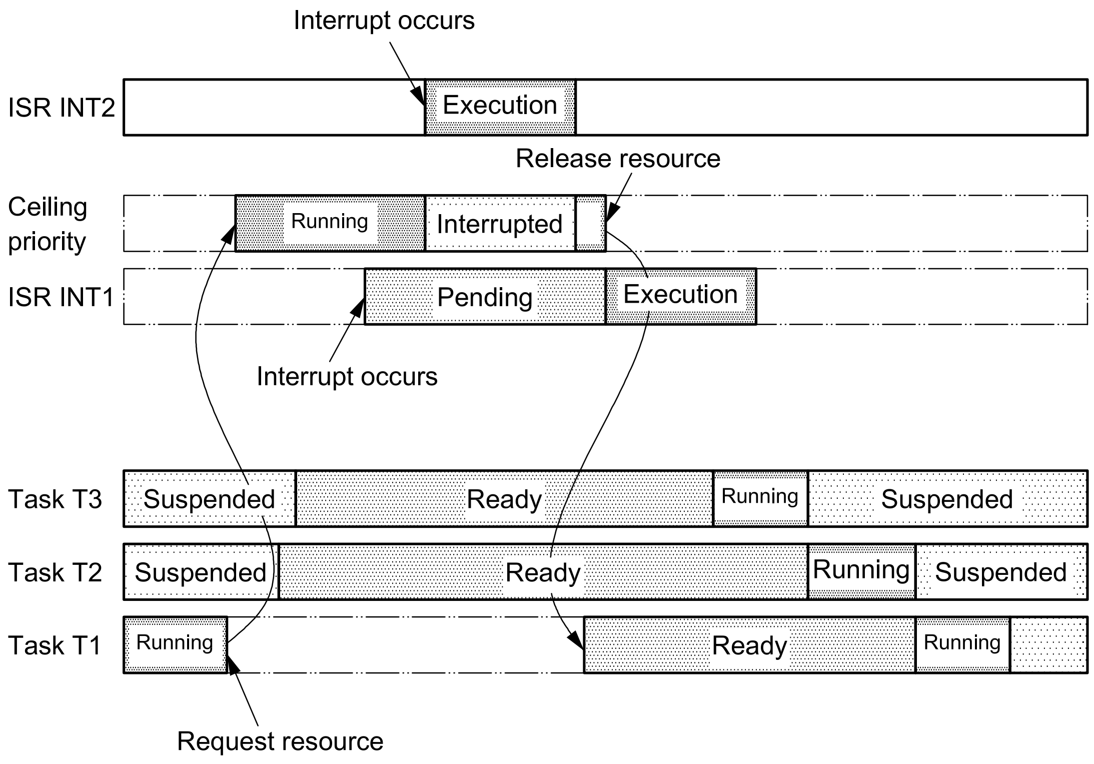

Figure 15 — Resource assignment with priority ceiling between preemptable tasks and interrupt services routines

The example shown in Figure 16 describes the following scenario:

The preemptable 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 Priority Ceiling Protocol, the INT1 is still executed and 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, and the task T1 is ready. After the task T2 is terminated, the task T1 is continued.

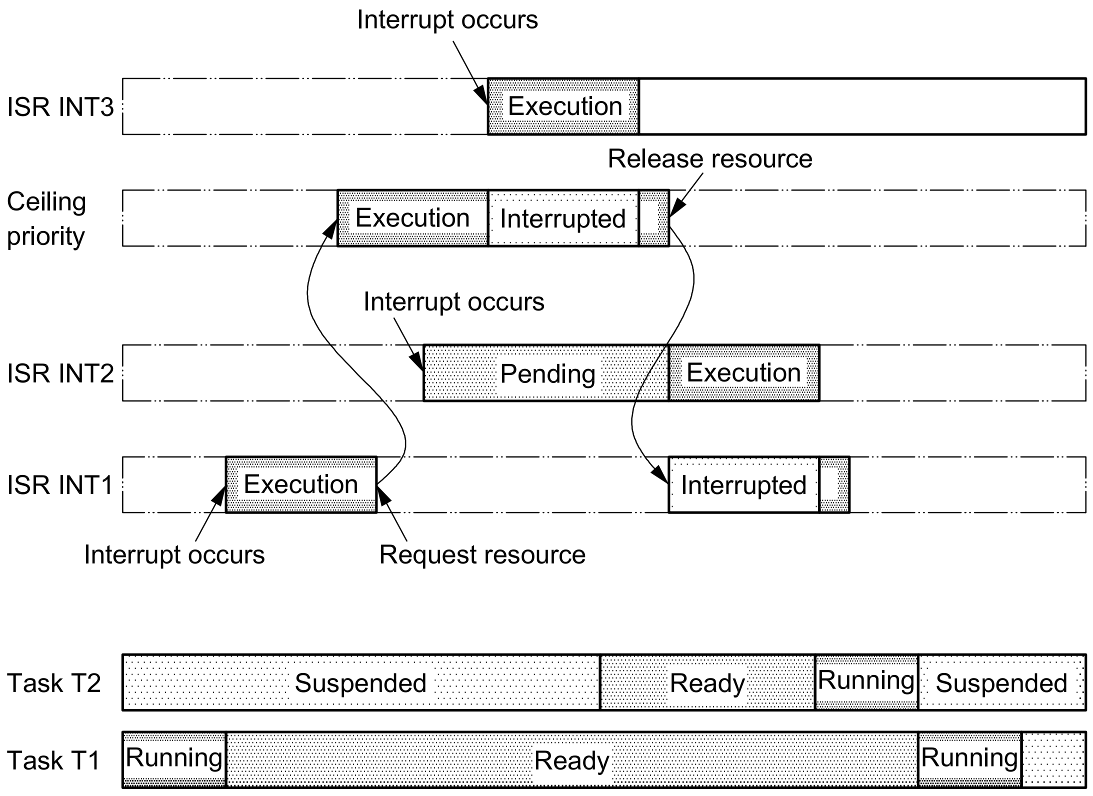

Figure 16 — Resource assignment with priority ceiling between interrupt services routines

8.8 Internal resources

Internal resources are resources which are not visible to the user and therefore can not be addressed by the system functions *GetResource* and *ReleaseResource*. Instead, they are managed strictly internally within a clearly defined set of system functions. Besides, the behaviour of internal resources is exactly the same as standard resources (priority ceiling protocol, etc.).

Internal resources are restricted to tasks. At most, one internal resource shall be assigned to a task during system generation. If an internal resource is assigned to a task, the internal resource shall be managed as follows:

— The resource shall be automatically taken when the task enters the *running* state (but not when it is activated), except when it has already taken the resource. As a result, the priority of the task is automatically changed to the ceiling priority of the resource.

— At the points of rescheduling as defined in 4.6.2, the resource shall be automatically released. The implementation may optimize, e.g. only free/take the resource within the system service *Schedule* if there is a need for rescheduling.

NOTE Internal resources are not released when a task is preempted.

The resulting behaviour for tasks which have the same internal resource assigned is described in 4.6.4. Non-preemptable tasks are a special group with an internal resource of the same priority as RES_SCHEDULER assigned (see 4.6.2). Internal resources can be used in all cases when it is necessary to avoid unwanted rescheduling within a group of tasks. More than one group (more than one internal resource) can be defined in a system. A typical example is presented in 14.3.6.

The general restriction that some system calls shall be called with resources occupied (see 8.3) does not apply to internal resources, as internal resources are handled within those calls. However, all standard resources shall be released before the internal resource is released (see 8.3).

The tasks which have the same internal resource assigned cover a certain range of priorities. It is possible to have tasks which do not use this internal resource in the same priority range. The application shall decide if this makes sense.

9 Alarms

9.1 General

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

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

9.2 Counters

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

The OS does not provide a standardized API to manipulate counters directly.

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

The OS offers at least one counter that is derived from a (hardware or software) timer.

9.3 Alarm management

The OS provides services to activate tasks, set events or call an alarm-callback routine when an alarm expires. An alarm-callback routine is a short function provided by the application.

An alarm shall 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). For example, alarms may expire upon receipt of a number of timer interrupts, when reaching a specific angular position, or when receiving a message.

Alarms can be defined to be either single alarms or cyclic alarms. In addition, the OS provides services to cancel alarms and 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; and

— one task or one alarm-callback routine.

Depending on the configuration, this alarm-callback routine shall be called, or this task shall be activated, or an event shall be set for this task when the alarm expires. Alarm-callback routines run with category 2 interrupts disabled. System services allowed in alarm-callback routines are listed in Table 4. Task activation and event setting when an alarm expires have the same properties as normal task activation and event setting.

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

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

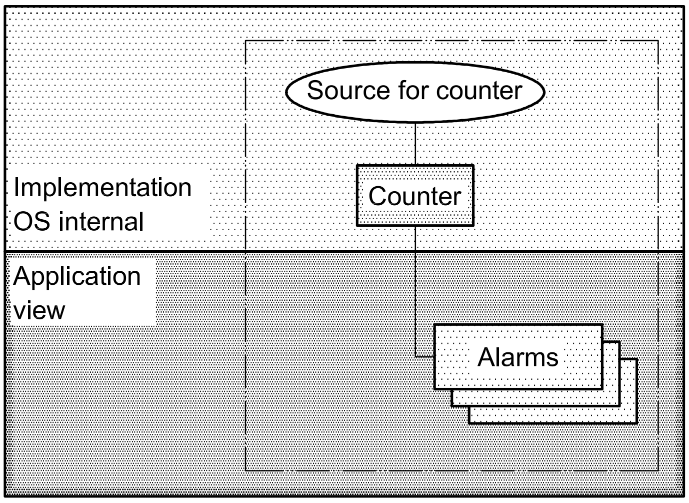

Figure 17 — Layered model of alarm management

9.4 Alarm-callback routines

Alarm-callback routines can have neither parameter nor return value.

The following format of the alarm-callback prototype shall apply:

```
ALARMCALLBACK(AlarmCallbackRoutineName);
```

Example for an alarm-callback routine:

```
ALARMCALLBACK(BrakePedalStroke)
{
     /* do application processing */
}
```

The processing level of alarm-callback routines is the one used by the scheduler, or ISR, depending on implementations.

10 Messages

For an implementation to be compliant, message handling for intra-processor communication shall be offered. The minimum functionality required is CCCA as described in ISO 17356-4. CCCB is the only other acceptable class as it is a superset of CCCA.

If an implementation offers even more functionality, which is specified in other conformance classes described in ISO 17356-4, the implementation shall stick to syntax and semantic of the respective ISO 17356-4 functionality.

Please note that for messages the rules stated in ISO 17356-4 are valid. For example, ISO 17356-4 system interfaces do not call *ErrorHook*. However, if the ISO 17356-4 functionality internally calls an OS system service like *ActivateTask*, *ErrorHook* shall be called if necessary from *ActivateTask*. For more details, refer to ISO 17356-4.

11 Error handling, tracing and debugging

11.1 Hook routines

The OS provides system specific hook routines to allow user-defined actions within the OS internal processing. Those hook routines are:

- called by the OS, in a special context depending on the implementation of the OS;
- higher priority than all tasks;
- not interrupted by category 2 interrupt routines;
- part of the OS;
- implemented by the user with user-defined functionality;
- standardized in interface, but not standardized in functionality (environment and behaviour of the hook routine itself), thus usually making hook routines non-portable;
- only allowed to use a subset of API functions (see Table 4); and
- mandatory, but configurable via ISO 17356-6.

In the OS, hook routines may be used for:

- System start-up (see 11.3): The corresponding hook routine (*StartupHook*) is called after the OS start-up and before the scheduler is running.
- System shutdown (see 11.4): The corresponding hook routine (*ShutdownHook*) is called when a system shutdown is requested by the application or by the OS in case of a severe error.
- Tracing or application dependent debugging purposes, as well as user-defined extensions of the context switch (see 11.5).
- Error handling.

EachOS implementation shall describe the conventions for the hook routines.

If the application calls a non-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.

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

11.2 Error handling

11.2.1 General remarks

An error service is provided to handle temporarily and permanently occurring errors within the OS. Its basic framework is predefined and shall be completed by the user. This gives the user a choice of efficient centralized or decentralized error handling.

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, centralized error treatment is called. Additionally, the OS returns the error by the status information for decentralized error treatment. It is up to the user to decide what to do depending on which error has occurred.

— Fatal errors: The OS can no longer assume correctness of its internal data. In this case, the OS calls the centralized system shutdown.

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

The OS shall offer two levels of error checking: standard status and extended status. If a task is activated in the version with standard status, “E_OK” or “Too many task activations” can be returned. Moreover, in a version with extended status, the additional return values “Task is invalid” or “Task still occupies resources”, etc. can be returned. These extended return values need 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.

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

11.2.2 Error hook routine

The error 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 shall never occur). Any possibly occurring error by calling system services from the *ErrorHook* can only be detected by evaluating the return value.

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

11.2.3 Error management

To allow for an effective error management in *ErrorHook*, the user can access additional information. Figure 18 summarizes the logical architecture for error management.

The macro OSErrorGetServiceId() provides the service identifier where the error has been raised. The service identifier is of type OSServiceIdType. Possible values are OSServiceId_xxxx, where xxxx is the name of the system service. Implementation of OSErrorGetServiceId is mandatory. If parameters of the system service which called *ErrorHook* are supplied, the following access macro name building scheme shall be used: OSError_Name1_Name2, whereby:

- Name1: is the name of the system service

- Name2: is the official name of the parameter within ISO 17356-3

For example, the macros to access the parameters of *SetRelAlarm* are:

- OSError_SetRelAlarm_AlarmID()
- OSError_SetRelAlarm_increment()
- OSError_SetRelAlarm_cycle()

The macro to access the first parameter of a system service is mandatory if the parameter is an object identifier. For optimization purposes, the macro access can be switched off within ISO 17356-6.

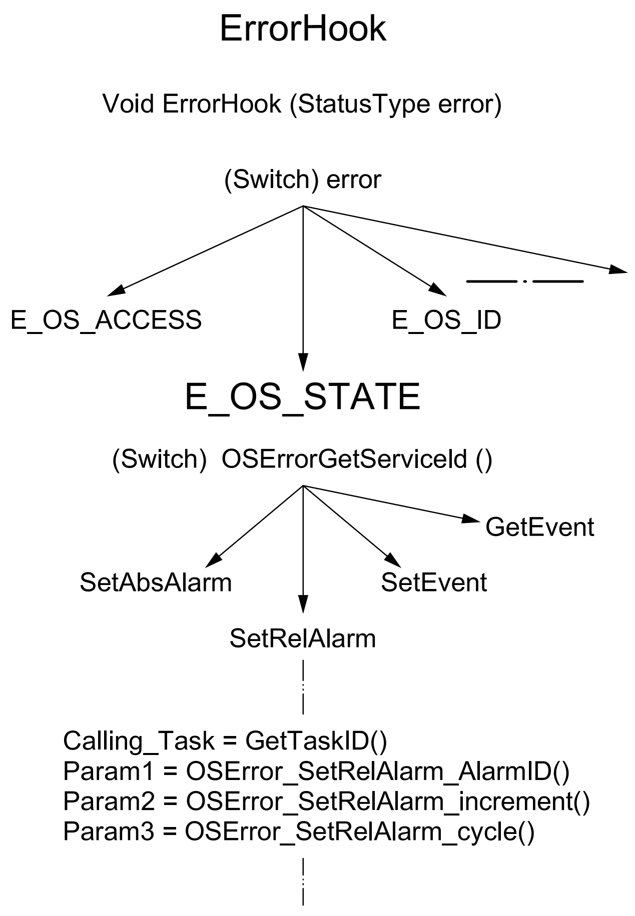
ErrorHook
Void ErrorHook (StatusType error)
(Switch) error
E_OS_ACCESS
E_OS_ID
E_OS_STATE
(Switch) OSErrorGetServiceId ()
GetEvent
SetAbsAlarm
SetEvent
SetRelAlarm
Calling_Task = GetTaskID()
Param1 = OSError_SetRelAlarm_AlarmID()
Param2 = OSError_SetRelAlarm_increment()
Param3 = OSError_SetRelAlarm_cycle()

Figure 18 — Example of centralized error handling (extended status)

11.3 System start-up

Initialization after a processor reset is up to the implementation, but OS offers support for a standardized way of initialization.

Interfaces for initialization of hardware, OS and application shall be clearly defined by the implementation.

This part of ISO 17356 does not force the application to define special tasks which shall be started after the OS initialization, but it allows the user to specify autostart tasks and autostart alarms during system generation.

After a reset of the CPU, hardware-specific application software is executed (no OS 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 OS and OSEKtime OS coexist (not reflected in Figure 19), the OSEKtime initialization shall always run first, and the remaining parts of the ISO 17356 initialization shall be performed after OSEKtime enters the idle loop, which causes 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 OS, i.e. *StartOS* with the application mode as a parameter. After the OS is initialized (scheduler is not running), *StartOS* calls the hook routine *StartupHook*, where the user can place the initialization code for all his OS-dependent initialization. In order to structure the initialization code in *StartupHook* according to the started application mode, the service *GetActiveApplicationMode* is provided. After returning from that hook routine, the OS enables the interrupts and starts the scheduler. After that, the system is running and executing user tasks.

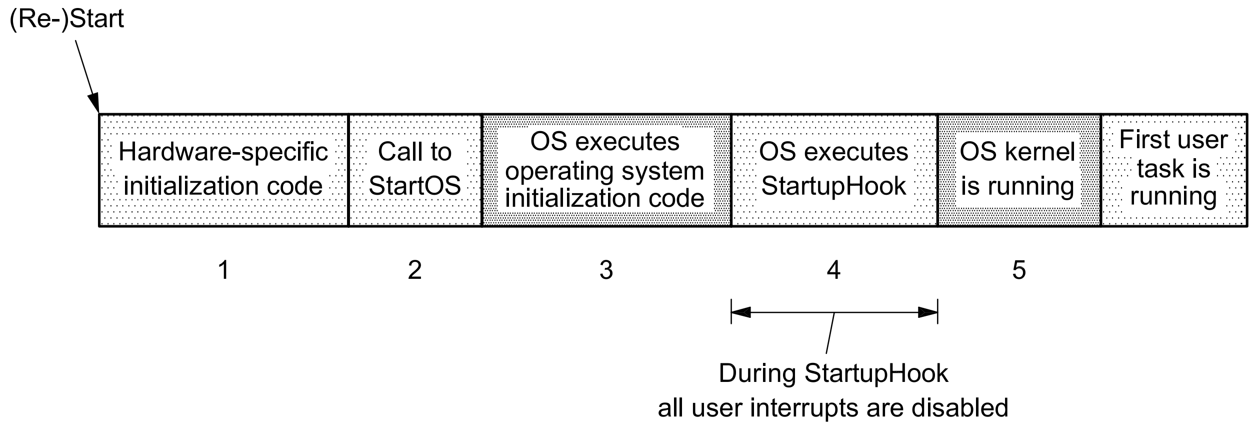
(Re-)Start
Hardware-specific initialization code
Call to StartOS
OS executes operating system initialization code
OS executes StartupHook
OS kernel is running
First user task is running
1
2
3
4
5
During StartupHook all user interrupts are disabled

Figure 19 — System start-up

System start-up consists of the following steps (see Figure 19):

a) After a reset, the user is free to execute (non-portable) hardware-specific code. Interrupts of category 2 or OS system services are not allowed to run until the phase 5. The non-portable section ends by detection of the application mode.

b) Call *StartOS* with the application mode as a parameter. This call starts the OS (if OSEKtime is present, this is done automatically).

c) The OS performs internal start-up functions.

d) The OS calls the hook routine *StartupHook*, where the user may place initialization procedures. During this hook routine, all user interrupts are disabled.

e) The OS enables user interrupts and starts the scheduling activity. The operating system starts the autostart tasks and alarms declared for the current application mode. The activation order of autostarted tasks of equal priority is not defined. Autostart of tasks is performed before autostart of alarms.

NOTE Counters are, if possible, set to zero by the system initialization before alarms are autostarted. Exceptions are calendar timers, etc. For autostarted alarms, all values are relative values.

11.4 System shutdown

ISO 17356-3 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 OS shall 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 13.8.2.3.) However, in case of a system where the OS coexists with OSEKtime OS, there are restrictions with respect to functionality which may be performed in *ShutdownHook*. It is possible that only the OS is shut down, whereas OSEKtime OS remains intact. Consequently, I/O devices which are handled within OSEKtime shall not be reset in *ShutdownHook*, and *ShutdownHook* shall return.

11.5 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 each time directly before the old task; *PreTaskHook* is called each time directly after a new task enters the *running* state. Because the task is still/already in the running state, *GetTaskId* will not return INVALID_TASK.

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.*

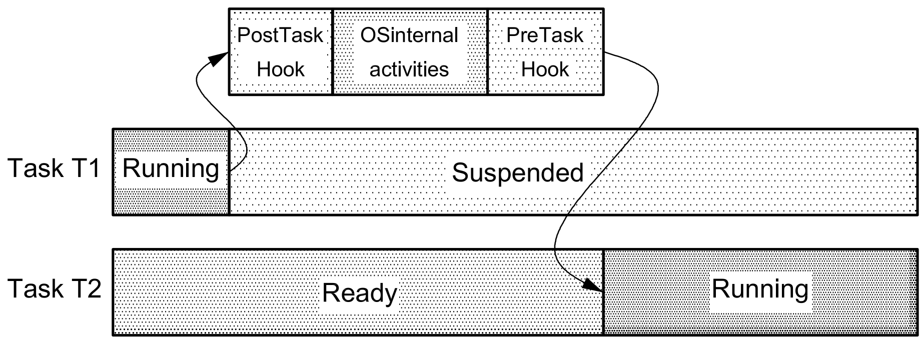

Figure 20 — PreTaskHook and PostTaskHook

12 Description of system services

12.1 Definition of system objects

Within the OS, all system objects shall be determined statically by the user. The OS supplier shall provide the definition of the OS objects. 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 OS objects. There are no system services available to dynamically create system objects. Declarations provide information that a system object is being used which has been created at another location. The names are used as identifiers within the system services.

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

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 shall not assume a specific representation.

The creation of system objects may require additional tools that 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.

12.2 Conventions

12.2.1 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, by macros of the C pre-processor, for example. A specific type of implementation shall not be assumed.

12.2.2 Legitimacy of calls

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

Table 4 lists all system services and shows in which situation they are allowed to be called (✓).

Table 4 — API service restrictions

Service	Task	ISR category 1	ISR category 2	ErrorHook[a]	PreTaskHook[a]	PostTaskHook[a]	StartupHook[a]	ShutdownHook[a]	alarm-callback[a]
ActivateTask	✓		✓						
TerminateTask	✓								
ChainTask	✓								
Schedule	✓								
GetTaskID	✓		✓[b]	✓[b]	✓	✓			
GetTaskState	✓		✓	✓	✓	✓			
DisableAllInterrupts	✓	✓	✓						
EnableAllInterrupts	✓	✓	✓						
SuspendAllInterrupts	✓	✓	✓	✓	✓	✓			✓
ResumeAllInterrupts	✓	✓	✓	✓	✓	✓			✓
SuspendOSInterrupts	✓	✓	✓						
ResumeOSInterrupts	✓	✓	✓						
GetResource	✓		✓						
ReleaseResource	✓		✓						
SetEvent	✓		✓						
ClearEvent	✓								
GetEvent	✓		✓	✓	✓	✓			
WaitEvent	✓								
GetAlarmBase	✓		✓	✓	✓	✓			
GetAlarm	✓		✓	✓	✓	✓			
SetRelAlarm	✓		✓						
SetAbsAlarm	✓		✓						
CancelAlarm	✓		✓						
GetActiveApplicationMode	✓		✓	✓	✓	✓	✓	✓	
StartOS									
ShutdownOS	✓		✓	✓			✓		

[a] Behaviour of system services is only defined for the services marked in the table.

[b] It may happen that currently no task is *running*. In this case, the service returns the task ID *INVALID_TASK* (see 11.2.3.5, GetTaskID).

12.2.3 Error characteristics

To keep the system efficient and fast, the OS does not test all errors. If the application uses OS 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 recognizes 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 "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. With respect to the description above, a status other than E_OK which is returned in standard mode is a “warning”. 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 OS is not specified. Whenever multiple errors occur, it is implementation-dependent which status is returned to the application.

In case of application errors, the OS shall call the hook routine *ErrorHook* if defined. The purpose of *ErrorHook* is to treat status information as centralized.

The *ErrorHook* routine is only called if a return value other than E_OK is generated.

The *ErrorHook* routine is configured within the ISO 17356-6 file.

For *ErrorHook* routine management, the standard mode corresponding to standard status management is distinguished from the extended mode corresponding to extended status management.

The system passes additional information to the *ErrorHook* routine. For performance reasons and stack consumption, a global structure including complementary information about the last error is used. This global structure is filled at execution time depending on given services and given implementation constraints. To allow efficient and adaptive implementation, the format of this error management structure is not prescribed. However, in order to achieve source code portability in the *ErrorHook* routine, standardized macros to access to the different parameters are defined.

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 OS 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 OS 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.

13 Specification of OS services

13.1 Basics

13.1.1 Structure of the description

OS 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.

13.1.2 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 utilization;
- Conformance: Specifies the conformance classes where the constructional element is provided.

13.1.3 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 OS service;
- Particularities: Explanation of restrictions relating to the utilization of the OS service;
- Status: List of possible return values:
  - Standard: List of return values provided in the OS's standard version. Special case: Service does not return;
  - Extended: List of additional return values in the OS's extended version;
- Conformance: Specifies the conformance classes where the operating system service is provided.

The specification of OS 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).

13.2 Common data types

13.2.1 StatusType

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

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

The following error values are defined.

13.2.2 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.

13.2.3 Internal errors of the OS

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 shall not overlap.

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

Examples:

- E_OS_SYS_STACK,
- E_OS_SYS_PARITY,
- ... and other implementation-specific errors, which shall be described in the vendor-specific documentation.

The names and range of numbers of the internal errors of the OS do not overlap the names and range of numbers of other services (i.e. communication and network management) or the range of numbers of the API error values. For details please refer to the ISO 17356-2.

13.3 Task management

13.3.1 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.

13.3.2 Constructional elements

13.3.2.1 DeclareTask

— Syntax: DeclareTask (<TaskIdentifier>);

— Parameter (In):

  — TaskIdentifier: 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.

13.3.3 System services

13.3.3.1 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. (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.) The OS ensures that the task code is being executed from the first statement.

— Particularities: The service may be called from interrupt level and from task level (see Table 4). Rescheduling after the call to *ActivateTask* depends on the place it is called from (ISR, non-preemptable task, preemptable 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;

    — Too many task activations of <TaskID>, E_OS_LIMIT;

  — Extended: Task <TaskID> is invalid, E_OS_ID;

— Conformance: BCC1, BCC2, ECC1, ECC2.

13.3.3.2 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. (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.)
- Particularities: An internal resource assigned to the calling task is automatically released. Other resources occupied by the task shall be released before the call to *TerminateTask*. If a 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 cannot 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. A task function shall not be ended without a call to *TerminateTask* or *ChainTask,* as this 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.

13.3.3.3 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 transferred to the suspended state. An internal resource assigned to the calling task is automatically released, even if the succeeding task is identical with the current task. Other resources occupied by the calling task shall be released before *ChainTask* is called. If a resource is still occupied in standard status the behaviour is undefined. If called successfully, *ChainTask* does not return to the call level and the status cannot be evaluated. In case of error, the service returns 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;
    - Too many task activations of <TaskID>, E_OS_LIMIT;
  - Extended:
    - Task <TaskID> is invalid, E_OS_ID;
    - Calling task still occupies resources, E_OS_RESOURCE;
    - Call at interrupt level, E_OS_CALLEVEL;
- Conformance: BCC1, BCC2, ECC1, ECC2.

13.3.3.4 Schedule

- Syntax: StatusType Schedule (void);
- Parameter (In): none;
- Parameter (Out): none;
- Description: If a higher-priority task is *ready*, the internal resource of the task is released, 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: Rescheduling can only take place if an internal resource is assigned to the calling task during system generation. For these tasks, *Schedule* enables a processor assignment to other tasks with lower or equal priority than the ceiling priority of the internal resource and higher priority than the priority of the calling task in application-specific locations. When returning from *Schedule*, the internal resource has been taken again. This service has no influence on tasks with no internal resource assigned (preemptable tasks).
- Status:
  - Standard: No error, E_OK;
  - Extended:
    - Call at interrupt level, E_OS_CALLEVEL;
    - Calling task occupies resources, E_OS_RESOURCE;
- Conformance: BCC1, BCC2, ECC1, ECC2.

13.3.3.5 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 Table 4). 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.

13.3.3.6 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 Table 4). When a call is made from a task in a full preemptive 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.

13.3.4 Constants

The following are 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 an undefined task.

13.3.5 Naming convention

The OS shall 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 OS 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 shall be generated from the `TaskName` during system generation time.

NOTE 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`.

13.4 Interrupt handling

13.4.1 Data types

No special data types are defined for the interrupt handling functionality.

13.4.2 System services

13.4.2.1 EnableAllInterrupts

— Syntax: void EnableAllInterrupts (void);

— Parameter (In): none;

— Parameter (Out): none.

— Description: This service restores the state saved by *DisableAllInterrupts*.

— Particularities: The service may be called from an ISR category 1 and category 2 and from the task level, but not from hook routines. This service is a counterpart of *DisableAllInterrupts* service, which shall be called before, and its aim is the completion of the critical section of code. No API service calls shall be made 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.

13.4.2.2 DisableAllInterrupts

- Syntax: void DisableAllInterrupts (void);
- Parameter (In): none;
- Parameter (Out): none;
- Description: This service disables all interrupts for which the hardware supports disabling. The state before is saved for the *EnableAllInterrupts* call.
- Particularities: The service may be called from an ISR category 1 and category 2 and from the task level, but not from hook routines. This service is intended to start a critical section of the code. This section shall be finished by calling the *EnableAllInterrupts* service. No API service calls shall be made 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*/*ResumeOSInterrupts* or *SuspendAllInterrupt*/*ResumeAllInterrupts* should be used.
- Status:
  - Standard: none;
  - Extended: none;
- Conformance: BCC1, BCC2, ECC1, ECC2.

13.4.2.3 ResumeAllInterrupts

- Syntax: void ResumeAllInterrupts (void);
- Parameter (In): none;
- Parameter (Out): none;
- Description: This service restores the recognition status of all interrupts saved by the *SuspendAllInterrupts* service.
- Particularities: The service may be called from an ISR category 1 and category 2, from alarm-callbacks and from the task level, but not from some hook routines. This service is the counterpart of *SuspendAllInterrupts* service, which shall be called before, and its aim is the completion of the critical section of code. No API service calls besides *SuspendAllInterrupts*/*ResumeAllInterrupts* pairs and *SuspendOSInterrupts*/*ResumeOSInterrupts* pairs shall be made within this critical section. The implementation should adapt this service to the target hardware providing a minimum overhead. *SuspendAllInterrupts*/*ResumeAllInterrupts* can be nested. In case of nesting pairs of the calls *SuspendAllInterrupts* and *ResumeAllInterrupts,* the interrupt recognition status saved by the first call of *SuspendAllInterrupts* is restored by the last call of the *ResumeAllInterrupts* service.
- Status:
  - Standard: none;
  - Extended: none;
- Conformance: BCC1, BCC2, ECC1, ECC2.

13.4.2.4 SuspendAllInterrupts

- Syntax: void SuspendAllInterrupts (void);
- Parameter (In): none;
- Parameter (Out): none;
- Description: This service saves the recognition status of all interrupts and disables all interrupts for which the hardware supports disabling.
- Particularities: The service may be called from an ISR category 1 and category 2, from alarm-callbacks and from the task level, but not from some hook routines. This service is intended to protect a critical section of code from interruptions of any kind. This section shall be finished by calling the *ResumeAllInterrupts* service. No API service calls beside *SuspendAllInterrupts*/*ResumeAllInterrupts* pairs and *SuspendOSInterrupts*/*ResumeOSInterrupts* pairs shall be made within this critical section. The implementation should adapt this service to the target hardware providing a minimum overhead.
- Status:
  - Standard: none;
  - Extended: none;
- Conformance: BCC1, BCC2, ECC1, ECC2.

13.4.2.5 ResumeOSInterrupts

- Syntax: void ResumeOSInterrupts (void);
- Parameter (In): 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 category 1 and category 2 and from the task level, but not from hook routines. This service is the counterpart of *SuspendOSInterrupts* service, which shall be called before, and its aim is the completion of the critical section of code. No API service calls beside *SuspendAllInterrupts*/*ResumeAllInterrupts* pairs and *SuspendOSInterrupts*/*ResumeOSInterrupts* pairs shall be made within this critical section. The implementation should adapt this service to the target hardware providing a minimum overhead. *SuspendOSInterrupts*/*ResumeOSInterrupts* can be nested. 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.

13.4.2.6 SuspendOSInterrupts

— Syntax: void SuspendOSInterrupts (void);

— Parameter (In): none;

— Parameter (Out): none;

— Description: This service saves the recognition status of interrupts of category 2 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 shall be finished by calling the *ResumeOSInterrupts* service. No API service calls beside *SuspendAllInterrupts*/*ResumeAllInterrupts* pairs and *SuspendOSInterrupts*/*ResumeOSInterrupts* pairs shall be made 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. 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.

13.4.3 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 interrupt service routines, no naming convention is prescribed; their definition is implementation-specific.

13.5 Resource management

13.5.1 Data types

ResourceType: Data type for a resource.

13.5.2 Constructional elements

13.5.2.1 DeclareResource

— Syntax: DeclareResource (<ResourceIdentifier>);

— Parameter (In):

  — ResourceIdentifier: 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.

- Conformance: BCC1, BCC2, ECC1, ECC2.

13.5.3 System services

13.5.3.1 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 shall always be left using *ReleaseResource*.

- Particularities: The priority ceiling protocol for resource management is described in 8.6. Nested resource occupation is only allowed if the inner critical sections are completely executed within the surrounding critical section (strictly stacked, see 8.3). There shall be no nested occupation of one and the same resource. Corresponding calls to *GetResource* and *ReleaseResource* should appear within the same function on the same function level. Services which are points of rescheduling for non-preemptable tasks (*TerminateTask*, *ChainTask*, *Schedule* and *WaitEvent*, see 4.6.5) shall not be used in critical sections. Additionally, critical sections shall be left before completion of an interrupt service routine. Generally speaking, critical sections should be short. The service may be called from an ISR and from task level (see Table 4).

- 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.

13.5.3.2 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 Table 4).
- 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 shall 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.

13.5.4 Constants

`RES_SCHEDULER`: Constant of data type ResourceType (see Clause 8).

13.6 Event control

13.6.1 Data types

Data types include:

- EventMaskType: Data type of the event mask; and
- EventMaskRefType: Reference to an event mask.

13.6.2 Constructional elements

13.6.2.1 DeclareEvent

- Syntax: DeclareEvent (<EventIdentifier>);
- Parameter (In):
  - EventIdentifier: 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.

13.6.3 System services

13.6.3.1 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 shall not be set as the referenced task is in the *suspended* state, E_OS_STATE.
- Conformance: ECC1, ECC2.

13.6.3.2 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.

13.6.3.3 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 the task is *waiting* for. The service may be called from interrupt service routines, task level and some hook routines (see Table 4). The current status of the event mask of task <TaskID> is copied to <Event>.

— Particularities: The referenced task shall 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.

13.6.3.4 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 rescheduling, if the wait condition occurs. If rescheduling takes place, the internal resource of the task is released while the task is in the *waiting* state. This service shall only 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.

13.7 Alarms

13.7.1 Data types

Data types include:

- TickType: This data type represents count values in ticks;

NOTE All elements of the structure are of data type TickType.

- 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).
- **AlarmBaseRefType**: This data type points to the data type AlarmBaseType;
- **AlarmType:** This data type represents an alarm object.

13.7.2 Constructional elements

13.7.2.1 DeclareAlarm

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

13.7.3 System services

13.7.3.1 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 Table 4).
- Status:
  - Standard: No error, E_OK;
  - Extended: Alarm <AlarmID> is invalid, E_OS_ID;
- Conformance: BCC1, BCC2, ECC1, ECC2.

13.7.3.2 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. It is allowed on task level, ISR, and in several hook routines (see Table 4).
- 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.

13.7.3.3 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 shall 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 or the alarm-callback routine is called.
- 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* or the alarm-callback may be called 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> should not already be in use. To change values of alarms already in use, the alarm shall be cancelled first. If the alarm is already in use, this call shall be ignored and the error E_OS_STATE shall be returned. This is 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.

13.7.3.4 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 shall 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 or the alarm-callback routine is called.

— Particularities: If the absolute value <start> is very close to the current counter value, the alarm may expire, and the task may become *ready* or the alarm-callback may be called before the system service returns to the user. If the absolute value <start> already was reached before the system call, the alarm shall only expire when the absolute value <start> is 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> should not already be in use. To change values of alarms already in use, the alarm shall be cancelled first. If the alarm is already in use, this call shall be ignored and the error E_OS_STATE shall be returned. This is 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.

13.7.3.5 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.

13.7.4 Constants

For all counters, the return values of *GetAlarmbase* are also available as constants:

— **`OSMAXALLOWEDVALUE_x`**: Maximum possible allowed value of counter x in ticks;

— **`OSTICKSPERBASE_x`**: Number of ticks required to reach a specific unit of counter x;

— **`OSMINCYCLE_x`**: Minimum allowed number of ticks for a cyclic alarm of counter x.

Thus, if the counter name is known, it is not necessary to call *GetAlarmBase.*

There always exists at least one counter which is a time counter (system counter). The constants of this counter are additionally accessible via the following constants:

— **`OSMAXALLOWEDVALUE`**: Maximum possible allowed value of the system counter in ticks;

— **`OSTICKSPERBASE`**: Number of ticks required to reach a specific unit of the system counter;

— **`OSMINCYCLE`**: Minimum allowed number of ticks for a cyclic alarm of the system counter.

Additionally the following constant is supplied:

— **`OSTICKDURATION`**: Duration of a tick of the system counter in nanoseconds.

13.7.5 Naming convention

Within the application, an alarm-callback is defined according to the following pattern:

```
ALARMCALLBACK (AlarmCallBackName)
{
}
```

13.8 OS execution control

13.8.1 Data types

Data types include:

— AppModeType: This data type represents the application mode.

13.8.2 System services

13.8.2.1 GetActiveApplicationMode

— Syntax: AppModeType GetActiveApplicationMode (void);

— Description: This service returns the current application mode. It may be used to write mode dependent code.

— Particularities: See Clause 5 for a general description of application modes. It is allowed for task, ISR and all hook routines.

— Conformance: BCC1, BCC2, ECC1, ECC2.

13.8.2.2 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 Clause 5.
- Particularities: Only allowed outside of the OS; therefore implementation-specific restrictions may apply. See also 11.3, especially with respect to systems where ISO 17356-3 and OSEKtime coexist. This call does not need a return.
- Conformance: BCC1, BCC2, ECC1, ECC2.

13.8.2.3 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 OS 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 OS. If *ShutdownHook* returns, further behaviour of ShutdownOS is implementation-specific. In case of a system where OS and OSEKtime OS coexist, *ShutdownHook* shall return. <Error> shall be a valid error code supported by the OS. In case of a system where 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 is shut down after OS shutdown.
- Particularities: After this service, the OS is shut down. It is allowed at task level, ISR level, in *ErrorHook* and *StartupHook*, and also called internally by the operating system. If the OS calls *ShutdownOS,* it never uses E_OK as the passed parameter value.
- Conformance: BCC1, BCC2, ECC1, ECC2.

13.8.3 Constants

Constants include:

- **`OSDEFAULTAPPMODE`**: Default application mode, always a valid parameter to *StartOS*.

13.9 Hook routines

13.9.1 Data types

Data types include:

- OSServiceIdType: This data type represents the identification of system services.

13.9.2 System services

13.9.2.1 ErrorHook

- Syntax: void ErrorHook (StatusType <Error>);
- Parameter (In):
  - Error error occurred
- Parameter (Out): none;
- Description: This hook routine is called by the OS 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 11.1 for general description of hook routines.
- Conformance: BCC1, BCC2, ECC1, ECC2.

13.9.2.2 PreTaskHook

- Syntax: void PreTaskHook (void);
- Parameter (In): none;
- Parameter (Out): none;
- Description: This hook routine is called by the OS 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 11.1 for general description of hook routines.
- Conformance: BCC1, BCC2, ECC1, ECC2

13.9.2.3 PostTaskHook

- Syntax: void PostTaskHook (void);
- Parameter (In): none;
- Parameter (Out): none;
- Description: This hook routine is called by the OS after executing the current task, but before leaving the task's *running* state (to allow evaluation of the TaskID by *GetTaskID*).
- Particularities: See 11.1 for general description of hook routines.
- Conformance: BCC1, BCC2, ECC1, ECC2.

13.9.2.4 StartupHook

- Syntax: void StartupHook (void);
- Parameter (In): none;
- Parameter (Out): none;
- Description: This hook routine is called by the OS at the end of the OS initialization and before the scheduler is running. At this time, the application can initialize device drivers, etc.
- Particularities: See 11.1 for general description of hook routines.
- Conformance: BCC1, BCC2, ECC1, ECC2.

13.9.2.5 ShutdownHook

- Syntax: void ShutdownHook (StatusType <Error>);
- Parameter (In):
  - Error: error occurred;
- Parameter (Out): none;
- Description: This hook routine is called by the OS when the OS service *ShutdownOS* has been called. This routine is called during the OS shut down.
- Particularities: *ShutdownHook* is a hook routine for user defined shutdown functionality (see 11.4).
- Conformance: BCC1, BCC2, ECC1, ECC2.

13.9.3 Constants

Constants include:

- OSServiceId_xx: Unique identifier of system service *xx*.

EXAMPLE OSServiceId_ActivateTask. OSServiceId_xx is of type OSServiceIdType.

13.9.4 Macros

Macros include:

- OSErrorGetServiceId: Provides the service identifier where the error has been raised. The service identifier is of type OsServiceIdType. Possible values are OSServiceId_xx, where xx is the name of the system service.
- OSError_x1_x2: Names of macros to access (within *ErrorHook*) parameters of the system service which is called *ErrorHook*, where *x1* is the name of the system service and *x2* is the parameter name.

14 Implementation- and application-specific topics

14.1 General

This clause is neither normative nor mandatory. It provides information for implementers and application programmers.

14.2 Implementation hints

ISO 17356 specifies an OS interface and its functionality. Implementation aspects are not prescribed. There is no restriction on the implementation of the OS, as long as the implementation corresponds to any of the defined conformance classes.

14.2.1 Aspects of implementation

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

As a result:

- The OS can be implemented with various degrees of efficiency.
- The linker needs only to link those objects and services of the OS which are actually used.
- The OS used in a product (e.g. in a control unit's EPROM) cannot be described as an operating system, but as an operating system which conforms to an operating system conformance class.
- The tool environment of the OS configuration and initialization is not part of this part of ISO 17356, and therefore implementation-specific.
- Commercial systems which provide the user with all OS-specific services and their functionality via an adaptation layer are also 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 application software should be 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.

14.2.2 Parameters of implementation

The OS vendor should provide 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 should be mentioned, e.g. functionality, target CPU, clock speed, bus configuration, wait states, etc.

14.2.2.1 Functionality

Functionality includes:

- Maximum number of tasks;
- Maximum number of non-suspended tasks;
- Maximum number of priorities;

- Number of tasks per priority (for BCC2 and ECC2);
- Upper limit for number of task activation (shall 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.

14.2.2.2 Hardware resources

Hardware resources include:

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

14.2.2.3 Performance

Performance includes:

- Total execution time for each service;

NOTE 1 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 shall be extracted from a common benchmark procedure.

- OS start-up time (beginning of *StartOS* until execution of first task in standard mode) without invoking hook routines;
- Interrupt latency for ISRs of category 1 and 2;

NOTE 2 Interrupt latency is the 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 specifies the OS overhead.

- Task switching times for all types of switching;

NOTE 3 This 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 activation from ISRs of type 2.

- Base load of system without applications running.

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

14.2.2.4 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 preemptability) and depending on the scheduling mechanism (non-, mixed- or full preemptive) the run time context may have different sizes. Tasks which can never preempt each other may be executed in the same run time context in order to achieve an efficient utilization of the available RAM space.

The OS 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 optimize the design of his application regarding RAM requirements versus run time efficiency.

14.3 Application design hints

14.3.1 General

The purpose of this clause is to provide additional information about possible problems which might arise when designing applications for the OS. Not all of the consequences for the system design can be mentioned in this part of ISO 17356. Other design hints result from the experience of current ECU applications.

14.3.2 Resource management

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

14.3.2.1 Occupation in LIFO order

Each access to a resource should be encapsulated with calls to the services *GetResource* and *ReleaseResource*. Resources shall 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 are allowed. The occupation of resources shall be performed in strict LIFO order (stack principle). If the code accessing the resource as shown above is preempted 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.

14.3.2.2 Call level of API services

The 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.

14.3.2.3 Resources still occupied at task termination or interrupt completion

The access to a resource should be encapsulated directly by the calls of *GetResource* and *ReleaseResource*. Otherwise the release of the resource might be missed 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 OS a task terminates, or in standard or extended status an interrupt completes without releasing all of the occupied resources, the resulting behaviour is not defined by this part of ISO 17356. Depending on the implementation of the OS, the resource may be locked forever since further accesses are rejected by the operating system.

14.3.3 Placement of API calls

For the same reasons as mentioned above, the placement of API services *TerminateTask* and *ChainTask* is crucial for the OS. Both services are used to terminate the *running* task. Calling these services from a subroutine level of the task, the OS 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.

14.3.4 Interrupt service routines

The user shall be aware of some possible error cases when using ISRs of category 1 and 2 as described in Clause 6.

14.3.4.1 Nested interrupts of different categories

Since all interrupts are of higher priority than the task levels, the processing of interrupts shall 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 OS is not invoked after termination of ISR1 in order to perform a rescheduling.

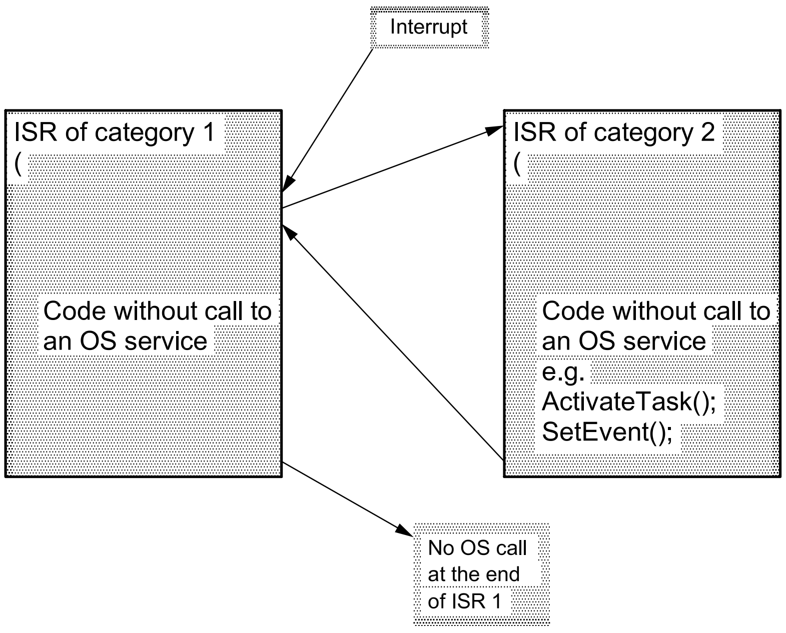
Interrupt
ISR of category 1
(
Code without call to an OS service
ISR of category 2
(
Code without call to an OS service e.g. ActivateTask(); SetEvent();
No OS call at the end of ISR 1

Figure 21 — Nested interrupts

Because ISRs of category 1 do not run under control of the OS, the OS has no possibility to perform a rescheduling when the ISR terminates. Thus, any activities corresponding to the calls of the OS in the interrupting ISR2 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 cannot occur (e.g. in non-preemptive systems). The rules shall 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 shall have a higher or equal hardware priority compared with interrupts of category 2.

14.3.4.2 Direct manipulation of interrupt levels

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

14.3.5 Priority and preemption

Tasks are scheduled by the OS according to their priority. A task is declared as being preemptable/non-preemptable (see 4.6.2). The application shall treat these two task attributes in a consistent manner to avoid conflicts in the run-time behaviour of the system. Care shall be taken because non-preemptable tasks of lower priority delay tasks of higher priority.

Typically, the preemption 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 shall be coordinated precisely. To achieve a well-defined run-time behaviour of the system, this coordination is crucial.

14.3.6 Examples of usage of internal resources

Besides for non-preemptable tasks, internal resources can be used in a number of situations.

In general, they protect a group of tasks against being preempted by another task of the same group, except if the running task within the group explicitly allows rescheduling by calling *Schedule*, *WaitEvent* or *TerminateTask/ChainTask*. If, for example, all tasks of a group call those functions only on first procedure level, a stack of those tasks can be highly optimized.

An example, besides non-preemptable tasks, is a concept sometimes referred to as “cooperative tasks”, where the lowest priority tasks share the same internal resource and can freely be preempted by higher priority tasks, but not among themselves.

This example can be extended by excluding the lowest priority of the system for usage as a background task. This task would again be preemptable by all tasks.

The concept of non-preemptable tasks and cooperative tasks can be easily combined within one system by using two different internal resources within one configuration.

Tasks which do not have an internal resource assigned are preemptable and act as described in 4.6.1.

14.3.7 Parameter to pass to ShutdownOS

The parameter passed to *ShutdownOS* is also passed to the *ShutdownHook*. If the OS calls *ShutdownHook*, the passed parameter is an implementation-dependent error value. If the user calls *ShutdownOS* he/she shall use one of the existing OS error numbers. If OSEKtime and ISO 17356-3 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 OS calls of *ShutdownOS* and application calls.

14.3.8 Error handling

Errors in the application software are typically caused by:

— Errors on handling the OS, i.e. incorrect configuration/initialization/dimensioning of the OS 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 organization.

Test of implementation

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

As an example, users can set time stamps enabling them to trace the program execution at the following locations before calling OS 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; and

— when occupying and releasing resources or at critical locations.

Time monitoring

The OS need 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 OS status.

Constructional elements

Constructional elements (e.g. DeclareTask) were introduced in the 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 in ISO 17356-6, most implementations do not need them any more. However, they are still kept for compatibility.

14.3.9 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:

— *ActivateTask*, *ChainTask:* E_OS_LIMIT (standard);

— *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 shall be taken into account.

14.4 Implementation-specific tools

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

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

— Version A, with all declarations related to task properties etc. within the code; and

— Version B, using a separate generation tool for these task properties, etc.

For definitions which should be supplied with portable code, please consult ISO 17356-6.

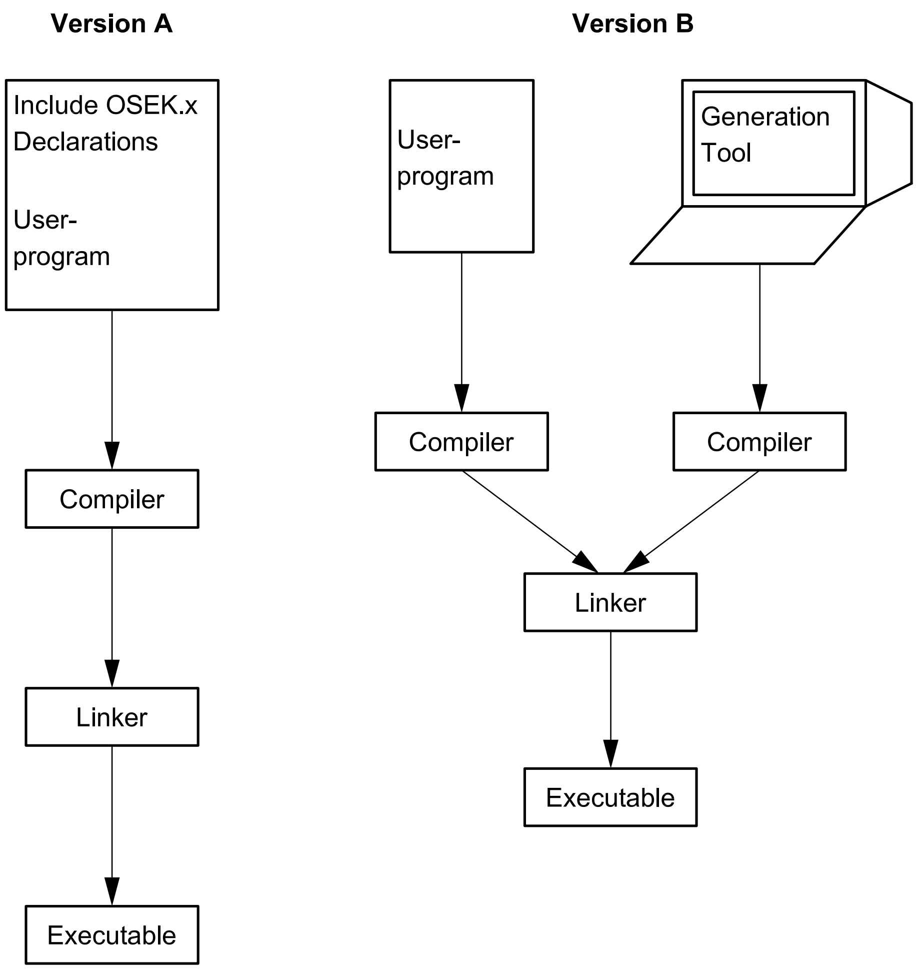

Figure 22 — Implementation-specific tools

ICS 43.040.15
Price based on 61 pages