OSEK/VDX

OS Test Procedure

Version 1.0

April, 24th, 1998

What is OSEK/VDX?

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.
A real-time operating system, software interfaces and functions for communication and network management tasks are thus jointly specified.
The term OSEK means ”Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug” (Open systems and the corresponding interfaces for automotive electronics).
The term VDX means ”Vehicle Distributed eXecutive”. The functionality of OSEK operating system was harmonized with VDX. For simplicity OSEK will be used instead of OSEK/VDX in this document.

OSEK partners:

Adam Opel AG, BMW AG, Daimler-Benz AG, IIIT University of Karlsruhe, Mercedes-Benz AG, Robert Bosch GmbH, Siemens AG, Volkswagen AG., GIE.RE. PSA-Renault.

Motivation:

- High, recurring expenses in the development and variant management of non-application related aspects of control unit software.
- Incompatibility of control units made by different manufacturers due to different interfaces and protocols.

Goal:

Support of the portability and reusability of the application software by:

- Specification of interfaces which are abstract and as application-independent as possible, in the following areas: real-time operating system, communication and network management.
- Specification of a user interface independent of hardware and network.
- Efficient design of architecture: The functionality shall be configurable and scaleable, to enable optimal adjustment of the architecture to the application in question.
- Verification of functionality and implementation of prototypes in selected pilot projects.

Advantages:

- Clear savings in costs and development time.
- Enhanced quality of the control units software of various companies.
- Standardized interfacing features for control units with different architectural designs.
- Sequenced utilization of the intelligence (existing resources) distributed in the vehicle, to enhance the performance of the overall system without requiring additional hardware.
- Provides absolute independence with regards to individual implementation, as the specification does not prescribe implementational aspects.

OSEK conformance testing

OSEK conformance testing aims at checking conformance of products to OSEK specifications. Test suites are thus specified for implementations of OSEK operating system, communication and network management.

Work around OSEK conformance testing is supported by the MODISTARC project sponsored by the Commission of European Communities. The term MODISTARC means ”Methods and tools for the validation of OSEK/VDX based DISTributed ARChitectures”.

This document has been drafted by the MODISTARC members of the OS-Workgroup:

Bernd Büchs	Adam Opel AG
Wolfgang Kremer	BMW AG
Stefan Schmerler	FZI
Franz Adis	FZI
Yves Sorel	INRIA
Robert France	Motorola
Barbara Ziker	Motorola
Jean-Emmanuel Hanne	Peugeot Citroën S.A.
Eric Brodin	Sagem SA
Gerhard Goeser	Siemens Automotive SA
Patrick Palmieri	Siemens Automotive SA

Table of Contents

1 Introduction

This document describes the test procedure for the conformance test of the operating system. The test procedure contains the definition of test cases and test sequences.

The test cases determine what will be tested. They are developed on the basis of the test assertions described in document [2] supported by the classification tree method. The classification trees are described in chapter 2and the corresponding test cases in chapter 2.

The test sequences determine how the test cases will be tested. This contains the sequence of actions that must be taken by the test program, and their expected reactions. The test sequences are described in chapter 3.

2 Test cases

This chapter contains the test cases which will be used to test an implementation of an operating system to be OSEK conform. Thus, they are developed on the basis of the OSEK OS specification.

2.1 Classification Tree Method

2.1.1 Introduction

The Classification Tree Method supports in a systematic and methodical way the determination of test cases. It helps to realize the test object and its mostly unclear input data range, in order to get structured test cases.

The input data range of a test object is classified by the Classification-Tree Method into test relevant aspects. These classifications divide the data range disjunctively and completely into a finite number of classes.

Using the Classification-Tree Method it is possible to identify exactly the input parameters relevant for testing by combining classes of different classifications. In doing so, exactly one class from each classification must be considered. For complex systems, it is necessary to check the combinations for logical compatibility.

If the concept of classification is used recursive on classes, then these classes are further refined.

2.1.2 Test case Trees for OSEK OS

The aim of classifying the OSEK OS in the classification trees was to describe every possible system state and its reactions to a call of an API service or an internal event like expiring of an alarm or occurring of an interrupt. This ensures that every situation that may occur during execution of an application is covered by the conformance tests.

The OSEK OS was divided into eight test groups which are handled separately. These groups are

- Task Management,
- Interrupt processing,
- Event mechanism,
- Resource management,
- Alarms, and
- Error handling, hook routines and OS execution control.

A test case is defined by a call to a OS service within a special system state and the reactions and answers performed by the system. The test trees ensure that each possible state is taken into account.

To keep the test trees simple the following conventions have been reached.

- The test trees don't contain the static properties of the OS (conformance class, scheduling policy, return status). This information is redundant and can be recovered from the test cases itself and is attached to the textual description of the test cases.

- Only the system environment (runtime properties) that influences the performed OS call is modelled in the test trees (execution level, running task's type, etc.).
- The reaction (answer) of the executed is not contained in the test trees (except for the return status). This can again be recovered from the test case itself and is attached to the textual description.

The test cases are chosen in that way that the OS service are called that often that each situation which is described in the specification is provoked at least once.

Each test case is defined by one line of a classification tree and the corresponding textual description which is printed below the classification tree. The textual description is presented in a table of the following structure:

Test case No.	Sched. policy Conf. class Status	Action	Expected Result
1	n, m, f B1, B2, E1, E2 e	Call `ActivateTask()` from task-level with invalid task ID (task does not exist)	Service returns E_OS_ID

Scheduling policy of OS
n: non-preemptive
m: mixed-preemptive
f: full preemptive

Actions that must be executed for this test case

Expected result of this test case

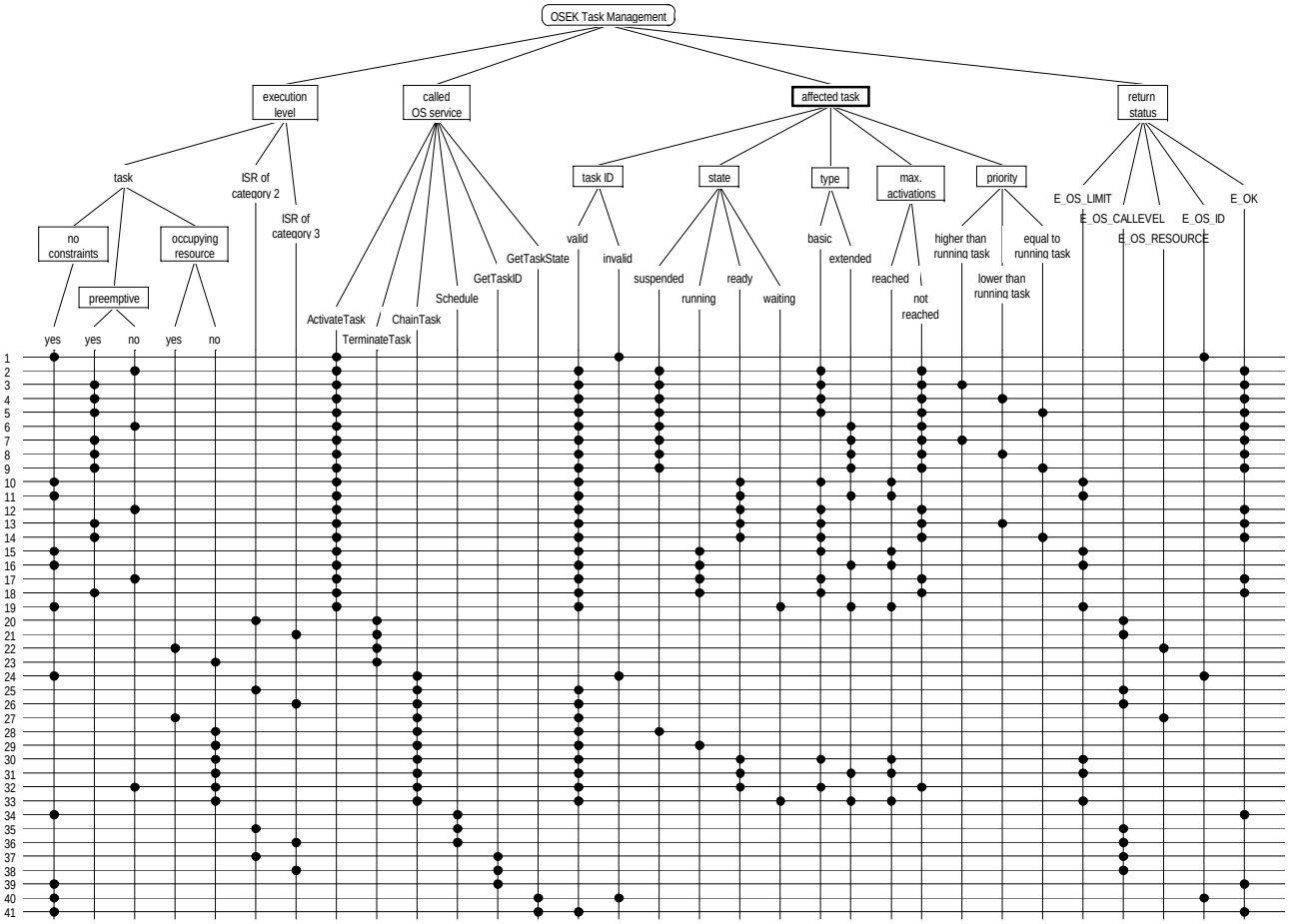
Conformance class of OS
B1: BCC1
B2: BCC2
E1: ECC1
E2: ECC2

OS status of OS services
s: standard
e: extended

The specification of OSEK OS in its current version (2.0 rev 1) is at some points ambiguous. This leads to wholes, which allow ambiguous interpretation of the specification. In order to do conformance tests this wholes had to be filled. Thus, some assumption had to be made, what is the correct interpretation in the "spirit" of OSEK. In the introduction to each of the following tables those assumption are expressed.

A general assumption that had to be taken is about the minimum number of task supported by the OS for applications. The specification doesn't provide this number. Therefore it is assumed that there are at least 8 tasks available in BCC1/BCC2 and at least 16 tasks in ECC1/ECC2. This numbers conform to fig. 12-1 of the specification.

2.2 Task management



Test case No.	Sched. policy Conf. class Status	Action	Expected Result
1	n, m, f B1, B2, E1, E2 e	Call `ActivateTask()` from task-level with invalid task ID (task does not exist)	Service returns E_OS_ID
2	n, m B1, B2, E1, E2 s, e	Call `ActivateTask()` from non-preemptive task on *suspended* basic task	No preemption of *running* task. Activated task becomes *ready*. Service returns E_OK
3	m, f B1, B2, E1, E2 s, e	Call `ActivateTask()` from preemptive task on *suspended* basic task which has higher priority than running task.	*Running* task is preempted. Activated task becomes *running*. Service returns E_OK
4	m, f B1, B2, E1, E2 s, e	Call `ActivateTask()` from preemptive task on *suspended* basic task which has lower priority than running task.	No preemption of *running* task. Activated task becomes *ready*. Service returns E_OK
5	m, f B2, E2 s, e	Call `ActivateTask()` from preemptive task on *suspended* basic task which has equal priority as running task.	No preemption of *running* task. Activated task becomes *ready*. Service returns E_OK

Test case No.	Sched. policy Conf. class Status	Action	Expected Result
6	n, m E1, E2 s, e	Call `ActivateTask()` from non-preemptive task on *suspended* extended task	No preemption of *running* task. Activated task becomes *ready* and its events are cleared. Service returns E_OK
7	m, f E1, E2 s, e	Call `ActivateTask()` from preemptive task on *suspended* extended task which has higher priority than running task.	*Running* task is preempted. Activated task becomes *running* and its events are cleared. Service returns E_OK
8	m, f E1, E2 s, e	Call `ActivateTask()` from preemptive task on *suspended* extended task which has lower priority than running task.	No preemption of *running* task. Activated task becomes *ready* and its events are cleared. Service returns E_OK
9	m, f E2 s, e	Call `ActivateTask()` from preemptive task on *suspended* extended task which has equal priority as running task.	No preemption of *running* task. Activated task becomes *ready* and its events are cleared. Service returns E_OK
10	n, m, f B1, B2, E1, E2 e	Call `ActivateTask()` on *ready* basic task which has reached max. number of activations	Service returns E_OS_LIMIT
11	n, m, f E1, E2 e	Call `ActivateTask()` on *ready* extended task	Service returns E_OS_LIMIT
12	n, m B2, E2 s, e	Call `ActivateTask()` from non-preemptive task on *ready* basic task which has not reached max. number of activations	No preemption of *running* task. Activation request is queued in ready list. Service returns E_OK
13	m, f B2, E2 s, e	Call `ActivateTask()` from preemptive task on *ready* basic task which has not reached max. number of activations and has lower than running task[1]	No preemption of *running* task. Activation request is queued in ready list. Service returns E_OK
14	m, f B2, E2 s, e	Call `ActivateTask()` from preemptive task on *ready* basic task which has not reached max. number of activations and has equal priority as running task	No preemption of *running* task. Activation request is queued in ready list. Service returns E_OK
15	n, m, f B1, B2, E1, E2 e	Call `ActivateTask()` on *running* basic task which has reached max. number of activations	Service returns E_OS_LIMIT
16	n, m, f E1, E2 e	Call `ActivateTask()` on *running* extended task	Service returns E_OS_LIMIT

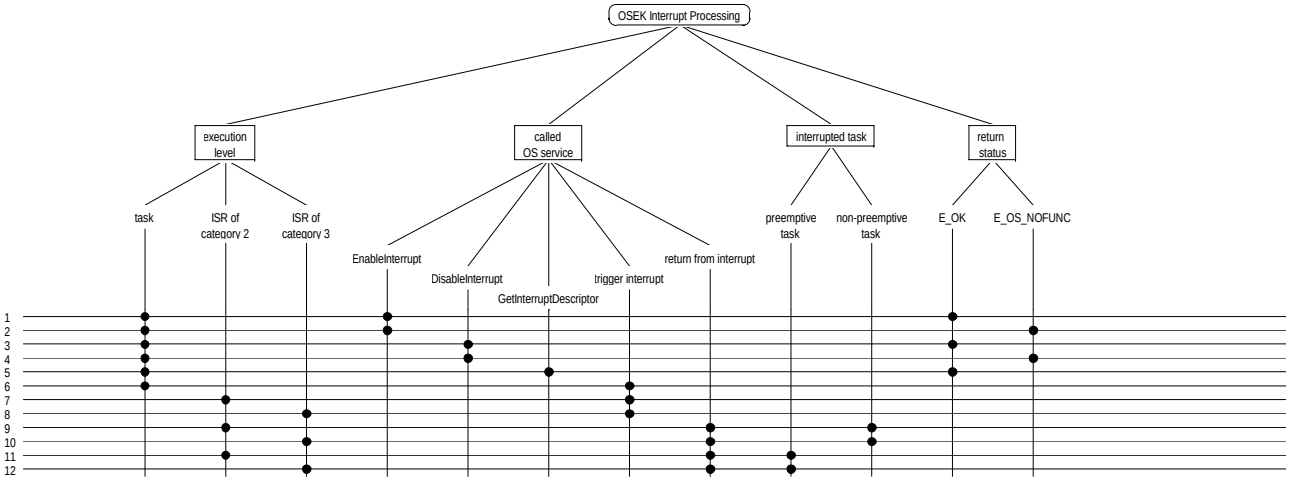
[1] Activating a higher priority task which is already ready from a preemptive task is not possible as the higher priority task would be running.

Test case No.	Sched. policy Conf. class Status	Action	Expected Result
17	n, m B2, E2 s, e	Call `ActivateTask()` from non-preemptive task on *running* basic task which has not reached max. number of activations	No preemption of *running* task. Activation request is queued in ready list. Service returns E_OK
18	m, f B2, E2 s, e	Call `ActivateTask()` from preemptive task on *running* basic task which has not reached max. number of activations	No preemption of *running* task. Activation request is queued in ready list. Service returns E_OK
19	n, m, f E1, E2 e	Call `ActivateTask()` on *waiting* extended task	Service returns E_OS_LIMIT
20	n, m, f B1, B2, E1, E2 e	Call `TerminateTask()` from ISR category 2	Service returns E_OS_CALLEVEL
21	n, m, f B1, B2, E1, E2 e	Call `TerminateTask()` from ISR category 3	Service returns E_OS_CALLEVEL
22	n, m, f B1, B2, E1, E2 e	Call `TerminateTask()` while still occupying a resource	*Running* task is not terminated. Service returns E_OS_RESOURCE
23	n, m, f B1, B2, E1, E2 s, e	Call `TerminateTask()`	*Running* task is terminated and *ready* task with highest priority is executed
24	n, m, f B1, B2, E1, E2 e	Call `ChainTask()` from task-level. Task-ID is invalid (does not exist).	Service returns E_OS_ID
25	n, m, f B1, B2, E1, E2 e	Call `ChainTask()` from ISR category 2	Service returns E_OS_CALLEVEL
26	n, m, f B1, B2, E1, E2 e	Call `ChainTask()` from ISR category 3	Service returns E_OS_CALLEVEL
27	n, m, f B1, B2, E1, E2 e	Call `ChainTask()` while still occupying a resource	*Running* task is not terminated. Service returns E_OS_RESOURCE
28	n, m, f B1, B2, E1, E2 s, e	Call `ChainTask()` on *suspended* task	*Running* task is terminated, chained task becomes *ready* and *ready* task with highest priority is executed
29	n, m, f B1, B2, E1, E2 s, e	Call `ChainTask()` on *running* task	*Running* task is terminated, chained task becomes *ready* and *ready* task with highest priority is executed
30	n, m, f B1, B2, E1, E2 e	Call `ChainTask()` on *ready* basic task which has reached max. number of activations	*Running* task is not terminated. Service returns E_OS_LIMIT

Test case No.	Sched. policy Conf. class Status	Action	Expected Result
31	n, m, f E1, E2 e	Call `ChainTask()` on *ready* extended task	*Running* task is not terminated. Service returns E_OS_LIMIT
32	n, m B2, E2 s, e	Call `ChainTask()` from non-preemptive task on *ready* basic task which has not reached max. number of activations	*Running* task is terminated, activation request is queued in ready list and *ready* task with highest priority is executed
33	n, m, f E1, E2 e	Call `ChainTask()` on *waiting* extended task	Service returns E_OS_LIMIT
34	n, m, f B1, B2, E1, E2 s, e	Call `Schedule()` from task.	*Ready* task with highest priority is executed. Service returns E_OK
35	n, m, f B1, B2, E1, E2 e	Call `Schedule()` from ISR category 2	Service returns E_OS_CALLEVEL
36	n, m, f B1, B2, E1, E2 e	Call `Schedule()` from ISR category 3	Service returns E_OS_CALLEVEL
37	n, m, f B1, B2, E1, E2 e	Call `GetTaskID()` from ISR category 2	Service returns E_OS_CALLEVEL
38	n, m, f B1, B2, E1, E2 e	Call `GetTaskID()` from ISR category 3	Service returns E_OS_CALLEVEL
39	n, m, f B1, B2, E1, E2 s, e	Call `GetTaskID()` from task	Return task ID of currently *running* task. Service returns E_OK
40	n, m, f B1, B2, E1, E2 e	Call `GetTaskState()` with invalid task ID (task does not exist)	Service returns E_OS_ID
41	n, m, f B1, B2, E1, E2 s, e	Call `GetTaskState()`	Return state of queried task. Service returns E_OK

2.3 Interrupt processing

No conformance tests will be made for interrupt service routines (ISR) of category 1 because they do not run under the control of the OS. Thus, there is no possibility to check if an ISR1 is active or not. The same holds true for ISRs of category 3 outside the ISR frame build by the calls to `Enter/LeaveISR()`.

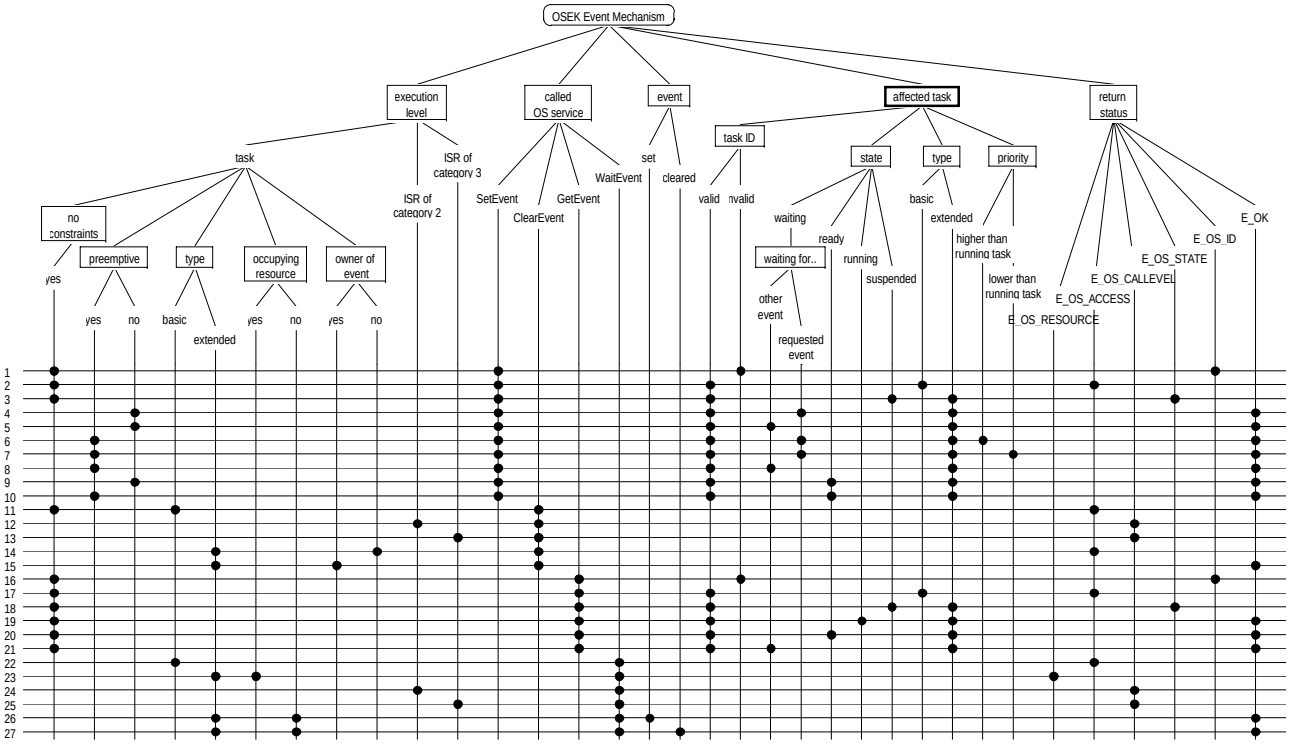


Test case No.	Sched. policy Conf. class Status	Action	Expected Result
1	n, m, f B1, B2, E1, E2 s, e	Call `EnableInterrupt()`. All requested interrupts are disabled	Enable interrupts. Service returns E_OK
2	n, m, f B1, B2, E1, E2 e	Call `EnableInterrupt()`. At least one of the requested interrupts is already enabled	Enable interrupts. Service returns E_OS_NOFUNC
3	n, m, f B1, B2, E1, E2 s, e	Call `DisableInterrupt()`. All requested interrupts are enabled	Disable interrupts. Service returns E_OK
4	n, m, f B1, B2, E1, E2 e	Call `DisableInterrupt()`. At least one of the requested interrupts is already disabled	Disable interrupts. Service returns E_OS_NOFUNC
5	n, m, f B1, B2, E1, E2 s, e	Call `GetInterruptDescriptor()`	Returns current interrupt descriptor. Service returns E_OK
6	n, m, f B1, B2, E1, E2 s, e	Interruption of *running* task	Interrupt is executed
7	n, m, f B1, B2, E1, E2 s, e	Interruption of ISR2	Interrupt is executed
8	n, m, f B1, B2, E1, E2 s, e	Interruption of ISR3	Interrupt is executed
9	n, m B1, B2, E1, E2 s, e	Return from ISR2. Interrupted task is non-preemptive	Execution of interrupted task is continued
10	n, m B1, B2, E1, E2 s, e	Return from ISR3. Interrupted task is non-preemptive	Execution of interrupted task is continued

Test case No.	Sched. policy Conf. class Status	Action	Expected Result
11	m, f B1, B2, E1, E2 s, e	Return from ISR2. Interrupted task is preemptive	*Ready* task with highest priority is executed (Rescheduling)
12	m, f B1, B2, E1, E2 s, e	Return from ISR3. Interrupted task is preemptive	*Ready* task with highest priority is executed (Rescheduling)

2.4 Event mechanism

Events are not queued. I.e. if an event is set twice before it could be cleared, then the task owning this event is notified only once. Therefore one event gets lost. This behaviour is not clearly expressed by the specification and is therefore not object of conformance testing.

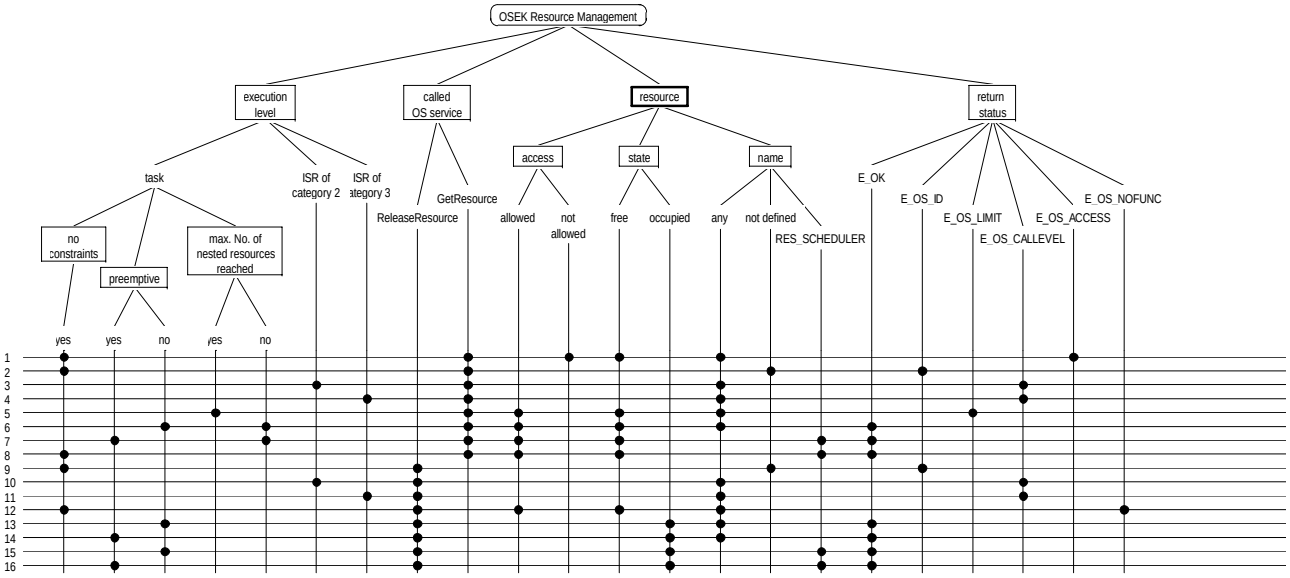


Test case No.	Sched. policy Conf. class Status	Action	Expected Result
1	n, m, f E1, E2 e	Call `SetEvent()` with invalid Task ID	Service returns E_OS_ID
2	n, m, f E1, E2 e	Call `SetEvent()` for basic task	Service returns E_OS_ACCESS
3	n, m, f E1, E2 e	Call `SetEvent()` for *suspended* extended task	Service returns E_OS_STATE

Test case No.	Sched. policy Conf. class Status	Action	Expected Result
4	n, m E1, E2 s, e	Call `SetEvent()` from non-preemptive task on *waiting* extended task which is waiting for at least one of the requested events	Requested events are set. *Running* task is not preempted. *Waiting* task becomes *ready.* Service returns E_OK
5	n, m E1, E2 s, e	Call `SetEvent()` from non-preemptive task on *waiting* extended task which is not waiting for any of the requested events	Requested events are set. *Running* task is not preempted. *Waiting* task doesn't become *ready*. Service returns E_OK
6	m, f E1, E2 s, e	Call `SetEvent()` from preemptive task on *waiting* extended task which is waiting for at least one of the requested events and has higher priority than *running* task	Requested events are set. *Running* task becomes *ready* (is preempted). *Waiting* task becomes *running*. Service returns E_OK
7	m, f E1, E2 s, e	Call `SetEvent()` from preemptive task on *waiting* extended task which is waiting for at least one of the requested events and has equal or lower priority than *running* task	Requested events are set. *Running* task is not preempted. *Waiting* task becomes *ready*. Service returns E_OK
8	m, f E1, E2 s, e	Call `SetEvent()` from preemptive task on *waiting* extended task which is not waiting for any of the requested events	Requested events are set. *Running* task is not preempted. *Waiting* task doesn't become *ready*. Service returns E_OK
9	n, m E1, E2 s, e	Call `SetEvent()` from non-preemptive task on *ready* extended task	Requested events are set. *Running* task is not preempted. Service returns E_OK
10	m, f E1, E2 s, e	Call `SetEvent()` from preemptive task on *ready* extended task	Requested events are set. *Running* task is not preempted. Service returns E_OK
11	n, m, f E1, E2 e	Call `ClearEvent()` from basic task	Service returns E_OS_ACCESS
12	n, m, f E1, E2 e	Call `ClearEvent()` from ISR2	Service returns E_OS_CALLEVEL
13	n, m, f E1, E2 e	Call `ClearEvent()` from ISR3	Service returns E_OS_CALLEVEL
14	n, m, f E1, E2 s, e	Call `ClearEvent()` from extended task	Requested events are cleared. Service returns E_OK
15	n, m, f E1, E2 e	Call `GetEvent()` with invalid Task ID	Service returns E_OS_ID
16	n, m, f E1, E2 e	Call `GetEvent()` for basic task	Service returns E_OS_ACCESS

Test case No.	Sched. policy Conf. class Status	Action	Expected Result
17	n, m, f E1, E2 e	Call `GetEvent()` for *suspended* extended task	Service returns E_OS_STATE
18	n, m, f E1, E2 s, e	Call `GetEvent()` for *running* extended task	Return current state of all event bits. Service returns E_OK
19	n, m, f E1, E2 s, e	Call `GetEvent()` for *ready* extended task	Return current state of all event bits. Service returns E_OK
20	n, m, f E1, E2 s, e	Call `GetEvent()` for *waiting* extended task	Return current state of all event bits. Service returns E_OK
21	n, m, f E1, E2 e	Call `WaitEvent()` from basic task	Service returns E_OS_ACCESS
22	n, m, f E1, E2 e	Call `WaitEvent()` from extended task which occupies a resource	Service returns E_OS_RESOURCE
23	n, m, f E1, E2 e	Call `WaitEvent()` from ISR2	Service returns E_OS_CALLEVEL
24	n, m, f E1, E2 e	Call `WaitEvent()` from ISR3	Service returns E_OS_CALLEVEL
25	n, m, f E1, E2 s, e	Call `WaitEvent()` from extended task. None of the events waited for is set	*Running* task becomes *waiting* and *ready* task with highest priority is executed. Service returns E_OK
26	n, m, f E1, E2 s, e	Call `WaitEvent()` from extended task. At least one event waited for is already set	No preemption of *running* task. Service returns E_OK

2.5 Resource management



Test case No.	Sched. policy Conf. class Status	Action	Expected Result
1	n, m, f B1, B2, E1, E2 e	Call `GetResource()` from task which has no access to this resource	Service returns E_OS_ACCESS
2	n, m, f B1, B2, E1, E2 e	Call `GetResource()` from task with invalid resource ID	Service returns E_OS_ID
3	n, m, f B1, B2, E1, E2 e	Call `GetResource()` from ISR2	Service returns E_OS_CALLEVEL
4	n, m, f B1, B2, E1, E2 e	Call `GetResource()` from ISR3	Service returns E_OS_CALLEVEL
5	n, m, f B1, B2, E1, E2 e	Call `GetResource()` from task with too many resources occupied in parallel	Service returns E_OS_LIMIT
6	n, m B1, B2, E1, E2 s, e	Test Priority Ceiling Protocol: Call `GetResource()` from non-preemptive task, activate task with priority higher than running task but lower than ceiling priority, and force rescheduling	Resource is occupied and *running* task's priority is set to resource's ceiling priority. Service returns E_OK. No preemption occurs after activating the task with higher priority and rescheduling
7	m, f B1, B2, E1, E2 s, e	Test Priority Ceiling Protocol: Call `GetResource()`from preemptive task, and activate task with priority higher than running task but lower than ceiling priority	Resource is occupied and *running* task's priority is set to resource's ceiling priority. Service returns E_OK. No preemption occurs after activating the task with higher priority

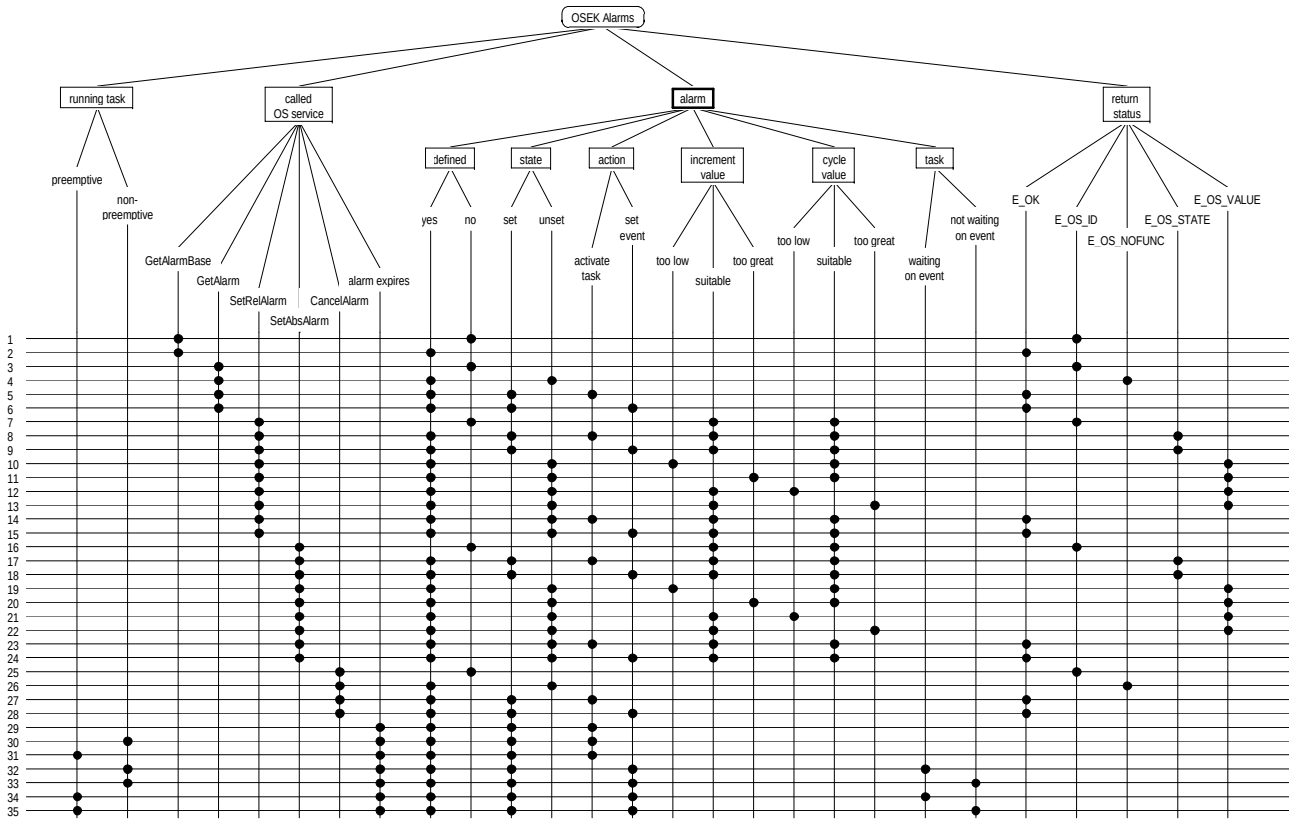
Test case No.	Sched. policy Conf. class Status	Action	Expected Result
8	n, m, f B1, B2, E1, E2 s, e	Call `GetResource()` for resource RES_SCHEDULER	Resource is occupied and *running* task's priority is set to resource's ceiling priority. Service returns E_OK
9	n, m, f B1, B2, E1, E2 e	Call `ReleaseResource()` from task with invalid resource ID	Service returns E_OS_ID
10	n, m, f B1, B2, E1, E2 e	Call `ReleaseResource()` from ISR2	Service returns E_OS_CALLEVEL
11	n, m, f B1, B2, E1, E2 e	Call `ReleaseResource()` from ISR3	Service returns E_OS_CALLEVEL
12	n, m, f B1, B2, E1, E2 e	Call `ReleaseResource()` from task with resource which is not occupied	Service returns E_OS_NOFUNC
13	n, m B1, B2, E1, E2 s, e	Call `ReleaseResource()` from non-preemptive task	Resource is released and *running* task's priority is reset. No preemption of *running* task. Service returns E_OK
14	m, f B1, B2, E1, E2 s, e	Call `ReleaseResource()` from preemptive task	Resource is released and *running* task's priority is reset. Ready task with highest priority is executed (Rescheduling). Service returns E_OK
15	n, m B1, B2, E1, E2 s, e	Call `ReleaseResource()`from non-preemptive task for resource RES_SCHEDULER	Resource is released and *running* task's priority is reset. No preemption of *running* task. Service returns E_OK
16	m, f B1, B2, E1, E2 s, e	Call `ReleaseResource()`from preemptive task for resource RES_SCHEDULER	Resource is released and *running* task's priority is reset. Ready task with highest priority is executed (Rescheduling). Service returns E_OK

2.6 Alarms

The behaviour of the OS is not defined by the specification if the action assigned to the expiration of an alarm can not be performed, because

- it would lead to multiple task activation, which is not allowed in the used conformance class or the max. number of activated tasks is already reached, or
- it would set an event for a task which is currently suspended.

The expected behaviour is, that at least the error hook is called. But as this situation is not covered by the specification, it is not part of conformance testing.



Test case No.	Sched. policy Conf. class Status	Action	Expected Result
1	n, m, f B1, B2, E1, E2 e	Call `GetAlarmBase()` with invalid alarm ID	Service returns E_OS_ID
2	n, m, f B1, B2, E1, E2 s, e	Call `GetAlarmBase()`	Return alarm base characteristics. Service returns E_OK
3	n, m, f B1, B2, E1, E2 e	Call `GetAlarm()` with invalid alarm ID	Service returns E_OS_ID
4	n, m, f B1, B2, E1, E2 s, e	Call `GetAlarm()` for alarm which is currently not in use	Service returns E_OS_NOFUNC
5	n, m, f B1, B2, E1, E2 s, e	Call `GetAlarm()` for alarm which will activate a task on expiration	Returns number of ticks until expiration. Service returns E_OK
6	n, m, f E1, E2 s, e	Call `GetAlarm()` for alarm which will set an event on expiration	Returns number of ticks until expiration. Service returns E_OK
7	n, m, f B1, B2, E1, E2 e	Call `SetRelAlarm()` with invalid alarm ID	Service returns E_OS_ID

Test case No.	Sched. policy Conf. class Status	Action	Expected Result
8	n, m, f B1, B2, E1, E2 s, e	Call `SetRelAlarm()` for already activated alarm which will activate a task on expiration	Service returns E_OS_STATE
9	n, m, f E1, E2 s, e	Call `SetRelAlarm()` for already activated alarm which will set an event on expiration	Service returns E_OS_STATE
10	n, m, f B1, B2, E1, E2 e	Call `SetRelAlarm()` with increment value lower than zero	Service returns E_OS_VALUE
11	n, m, f B1, B2, E1, E2 e	Call `SetRelAlarm()` with increment value greater than `maxallowedvalue`	Service returns E_OS_VALUE
12	n, m, f B1, B2, E1, E2 e	Call `SetRelAlarm()` with cycle value lower than `mincycle`	Service returns E_OS_VALUE
13	n, m, f B1, B2, E1, E2 e	Call `SetRelAlarm()` with cycle value greater than `maxallowedvalue`	Service returns E_OS_VALUE
14	n, m, f B1, B2, E1, E2 s, e	Call `SetRelAlarm()` for alarm which will activate a task on expiration	Alarm is activated. Service returns E_OK
15	n, m, f E1, E2 s, e	Call `SetRelAlarm()` for alarm which will set an event on expiration	Alarm is activated. Service returns E_OK
16	n, m, f B1, B2, E1, E2 e	Call `SetAbsAlarm()` with invalid alarm ID	Service returns E_OS_ID
17	n, m, f B1, B2, E1, E2 s, e	Call `SetAbsAlarm()` for already activated alarm which will activate a task on expiration	Service returns E_OS_STATE
18	n, m, f E1, E2 s, e	Call `SetAbsAlarm()` for already activated alarm which will set an event on expiration	Service returns E_OS_STATE
19	n, m, f B1, B2, E1, E2 e	Call `SetAbsAlarm()` with increment value lower than zero	Service returns E_OS_VALUE
20	n, m, f B1, B2, E1, E2 e	Call `SetAbsAlarm()` with increment value greater than `maxallowedvalue`	Service returns E_OS_VALUE
21	n, m, f B1, B2, E1, E2 e	Call `SetAbsAlarm()` with cycle value lower than `mincycle`	Service returns E_OS_VALUE
22	n, m, f B1, B2, E1, E2 e	Call `SetAbsAlarm()` with cycle value greater than `maxallowedvalue`	Service returns E_OS_VALUE

Test case No.	Sched. policy Conf. class Status	Action	Expected Result
23	n, m, f B1, B2, E1, E2 s, e	Call `SetAbsAlarm()` for alarm which will activate a task on expiration	Alarm is activated. Service returns E_OK
24	n, m, f E1, E2 s, e	Call `SetAbsAlarm()` for alarm which will set an event on expiration	Alarm is activated. Service returns E_OK
25	n, m, f B1, B2, E1, E2 e	Call `CancelAlarm()` with invalid alarm ID	Service returns E_OS_ID
26	n, m, f B1, B2, E1, E2 s, e	Call `CancelAlarm()` for alarm which is currently not in use	Service returns E_OS_NOFUNC
27	n, m, f B1, B2, E1, E2 s, e	Call `CancelAlarm()` for already activated alarm which will activate a task on expiration	Alarm is cancelled. Service returns E_OK
28	n, m, f E1, E2 s, e	Call `CancelAlarm()` for already activated alarm which will set an event on expiration	Alarm is cancelled. Service returns E_OK
29	n, m, f B1, B2, E1, E2 s, e	Expiration of alarm which activates a task while no tasks are currently *running*	Task is activated
30	n, m B1, B2, E1, E2 s, e	Expiration of alarm which activates a task while *running* task is non-preemptive	Task is activated. No preemption of *running* task
31	m, f B1, B2, E1, E2 s, e	Expiration of alarm which activates a task with higher priority than *running* task while *running* task is preemptive	Task is activated. Task with highest priority is executed
32	m, f B1, B2, E1, E2 s, e	Expiration of alarm which activates a task with lower priority than *running* task while *running* task is preemptive	Task is activated. No preemption of *running* task.
33	n, m E1, E2 s, e	Expiration of alarm which sets an event while *running* task is non-preemptive. Task which owns the event is not *waiting* for this event and not *suspended*	Event is set
34	n, m E1, E2 s, e	Expiration of alarm which sets an event while *running* task is non-preemptive. Task which owns the event is *waiting* for this event	Event is set. Task which is owner of the event becomes *ready.* No preemption of *running* task
35	m, f E1, E2 s, e	Expiration of alarm which sets an event while *running* task is preemptive. Task which owns the event is not *waiting* for this event and not *suspended*	Event is set

Test case No.	Sched. policy Conf. class Status	Action	Expected Result
36	m, f E1, E2 s, e	Expiration of alarm which sets an event while *running* task is preemptive. Task which owns the event is *waiting* for this event	Event is set. Task which is owner of the event becomes *ready.* Task with highest priority is executed (Rescheduling)

2.7 Error handling, hook routines and OS execution control

The specification doesn't provide an error status when calling an OS service which is not allowed on hook level from inside a hook routine. It is assumed that the correct behaviour would be to return E_OS_CALLEVEL. As this is not prescribed by the specification, this will not be used as a criteria for the conformance of the implementation. Anyway, the conformance tests will check that restricted OS services return a value not equal E_OK.

Test case No.	Sched. policy Conf. class Status	Action	Expected Result
1	n, m, f B1, B2, E1, E2 s, e	Call `GetActiveApplicationMode()`	Return current application mode
2	n, m, f B1, B2, E1, E2 s, e	Call `StartOS()`	Start operating system
3	n, m, f B1, B2, E1, E2 s, e	Call `ShutdownOS()`	Shutdown operating system
4	n, m, f B1, B2, E1, E2 s, e	Check PreTaskHook/PostTaskHook: Force rescheduling	PreTaskHook is called before executing the new task, but after the transition to *running* state. PostTaskHook is called after exiting the current task but before leaving the task's *running* state
5	n, m, f B1, B2, E1, E2 s, e	Check ErrorHook: Force error	ErrorHook is called at the end of a system service which has a return value not equal E_OK
6	n, m, f B1, B2, E1, E2 s, e	Check StartupHook: Start OS	StartupHook is called after initialisation of OS
7	n, m, f B1, B2, E1, E2 s, e	Check ShutdownHook: Shutdown OS	ShutdownHook is called after the OS shut down
8	n, m, f B1, B2, E1, E2 e	Check availability of OS services inside hook routines according to fig. 9-1 of OS spec.	OS services which must not be called from hook routines return status not equal E_OK

3 Test sequences

This chapter contains the specification of the test sequences that will be run during the conformance tests. The test sequences define the sequence of actions that will be done during the execution of the test program, i. e. the sequence of instructions executed by each task. Each test sequence fulfils the test for one ore more of the test cases defined in the previous chapter.

In order to check during the execution of the conformance tests if the sequences are passed correctly, it is necessary to make the observable system state traceable. This requires that the system state must be coded in a logable format. where it can by bit patterns. Each bit of a pattern represents the state of an OS element (task, event, ...). Thus, the system state can be traced by logging this patterns, which can be done by writing them into a special part of the RAM where it can be read out later, or by writing them to some pins of the test platform where it can be observed by a logic analyzer.

The logging of the patterns requires an additionally library which contains functions to write out the patterns. This library must be provided by the vendor of the OS implementation and the manufacturer of the test platform respectively. The specification of the API of this library will be done later.

Conformance testing contains the following steps:

1. Transfer the test sequences into a executable test program.
2. Execution of the test program on the test platform. Thereby, the patterns are generated.
3. Comparison of the generated pattern sequence with the expected sequence. If the pattern sequences match the test is passed, otherwise it failed.

3.1 Task management

Test Sequence 1:

Test Cases:	1, 10, 15, 20, 21, 22, 24, 25, 26, 27, 30, 35, 36, 37, 38, 40
Scheduling policy:	non-, mixed-, full-preemptive
Conformance class:	BCC1, BCC2, ECC1, ECC2
Return status:	extended
Parameters:	N = max. number of multiple activations (1 for BCC1/ECC1)

Tasks:	`Task1`
	type = basic
	priority = 1
	activation = 1
	autostart = true
	resource = R1
	`Task2`
	type = basic
	priority = 2
	activation = 1
	autostart = false
	`Task3`
	type = basic
	priority = 3

activation = 1
autostart = false

ISR: ISR2
category = 2
ISR3
category = 3

Resources: R1

Running task	Called OS service	Return status
Task1	`ActivateTask(Task5)`	E_OS_ID
Task1	`GetTaskState(Task5)`	E_OS_ID
Task1	`ChainTask(Task5)`	E_OS_ID
Task1	`ActivateTask(Task2)`	E_OK
Task1	`Schedule()`	E_OK
Task2	`ActivateTask(Task1)`	E_OS_LIMIT
Task2	`ActivateTask(Task2)`	E_OS_LIMIT
Task2	`TerminateTask()`	
Task1	`GetResource(R1)`	E_OK
Task1	`Terminate()`	E_OS_RESOURCE
Task1	`ChainTask(Task3)`	E_OS_RESOURCE
Task1	`ReleaseResource(R1)`	E_OK
Task1	`ActivateTask(Task3)`	E_OK
Task1	`Schedule()`	E_OK
Task3	`ChainTask(Task1)`	E_OS_LIMIT
Task3	`TerminateTask()`	
Task1	`TriggerInterrupt(ISR2)`	
ISR2	`TerminateTask()`	E_OS_CALLEVEL
ISR2	`ChainTaskTask(Task3)`	E_OS_CALLEVEL
ISR2	`Schedule()`	E_OS_CALLEVEL
ISR2	`GetTaskID()`	E_OS_CALLEVEL
ISR2	`ReturnFromInterrupt()`	
Task1	`TriggerInterrupt(ISR3)`	
ISR3	`EnterISR()`	
ISR3	`TerminateTask()`	E_OS_CALLEVEL
ISR3	`ChainTaskTask(Task3)`	E_OS_CALLEVEL
ISR3	`Schedule()`	E_OS_CALLEVEL
ISR3	`GetTaskID()`	E_OS_CALLEVEL
ISR3	`LeaveISR()`	
ISR3	`ReturnFromInterrupt()`	
Task1	`TerminateTask()`	

Test Sequence 2:

Test Cases: 2, 34
Scheduling policy: non-, mixed-preemptive
Conformance class: BCC1, BCC2, ECC1, ECC2
Return status: standard, extended

Tasks: Task1
type = basic
schedule = non
priority = 1
autostart = true
Task2
type = basic
schedule = non
priority = 2
autostart = false
Task3
type = basic
schedule = non
priority = 3
autostart = false

Running task	Called OS service	Return status
Task1	`ActivateTask(Task2)`	E_OK
Task1	`ActivateTask(Task3)`	E_OK
Task1	`Schedule()`	E_OK
Task3	`TerminateTask()`	
Task2	`TerminateTask()`	
Task1	`TerminateTask()`	

Test Sequence 3:

Test Cases: 3, 4
Scheduling policy: mixed-, full-preemptive
Conformance class: BCC1, BCC2, ECC1, ECC2
Return status: standard, extended
Tasks: Task1
type = basic
schedule = full
priority = 1
autostart = true
Task2
type = basic
schedule = full
priority = 2
autostart = false
Task3
type = basic
schedule = full
priority = 3
autostart = false

Running task	Called OS service	Return status
Task1	`ActivateTask(Task3)`	E_OK

Running task	Called OS service	Return status
Task3	`ActivateTask(Task2)`	E_OK
Task3	`TerminateTask()`	
Task2	`TerminateTask()`	
Task1	`TerminateTask()`	

Test Sequence 4:

Test Cases: 6
Scheduling policy: non-, mixed-preemptive
Conformance class: ECC1, ECC2
Return status: standard, extended
Tasks: Task1
type = extended
schedule = non
priority = 1
autostart = true
Task2
type = extended
schedule = non
priority = 2
autostart = false

Running task	Called OS service	Return status
Task1	`ActivateTask(Task2)`	E_OK
Task1	`GetEvent(Task1)`	E_OK, all events must be cleared
Task1	`GetEvent(Task2)`	E_OK, all events must be cleared
Task1	`Schedule()`	E_OK
Task2	`TerminateTask()`	
Task1	`TerminateTask()`	

Test Sequence 5:

Test Cases: 7, 8
Scheduling policy: mixed-, full-preemptive
Conformance class: ECC1, ECC2
Return status: standard, extended
Tasks: Task1
type = basic
schedule = full
priority = 1
autostart = true
Task2
type = extended
schedule = full
priority = 2
autostart = false
Task3
type = extended

schedule = full
priority = 3
autostart = false

Running task	Called OS service	Return status
Task1	`ActivateTask(Task3)`	E_OK
Task3	`GetEvent(Task3)`	E_OK, all events must be cleared
Task3	`ActivateTask(Task2)`	E_OK
Task3	`TerminateTask()`	
Task2	`GetEvent(Task2)`	E_OK, all events must be cleared
Task2	`TerminateTask()`	
Task1	`TerminateTask()`	

Test Sequence 6:

Test Cases: 11, 16, 19, 31, 33, 41
Scheduling policy: non-, mixed-, full-preemptive
Conformance class: ECC1, ECC2
Return status: extended
Tasks: Task1
type = extended
schedule = full
priority = 1
autostart = true
Task2
type = extended
schedule = full
priority = 2
autostart = false
event = E1
Events: E1

Running task	Called OS service	Return status
Task1	`ActivateTask(Task2)`	E_OK
Task1	`Schedule()`	E_OK
Task2	`ActivateTask(Task1)`	E_OS_LIMIT
Task2	`ActivateTask(Task2)`	E_OS_LIMIT
Task2	`WaitEvent(E1)`	E_OK
Task1	`GetTaskState(Task2)`	E_OK, *waiting*
Task1	`ActivateTask(Task2)`	E_OS_LIMIT
Task1	`ChainTask(Task2)`	E_OS_LIMIT
Task1	`SetEvent(Task2, E1)`	E_OK
Task1	`Schedule()`	E_OK
Task2	`ChainTask(Task1)`	E_OS_LIMIT
Task1	`TerminateTask()`	

Test Sequence 7:

Test Cases: 12, 17, 32

Scheduling policy: non-, mixed-preemptive
Conformance class: BCC2, ECC2
Return status: standard, extended
Tasks: Task1
type = basic
schedule = non
priority = 1
activation = 2
autostart = true
Task2
type = basic
schedule = non
priority = 2
activation = 2
autostart = false
Task3
type = basic
schedule = non
priority = 3
activation = 2
autostart = false

Running task	Called OS service	Return status
Task1	`ActivateTask(Task2)`	E_OK
Task1	`ActivateTask(Task2)`	E_OK
Task1	`Schedule()`	E_OK
Task2	`TerminateTask()`	
Task2	`TerminateTask()`	
Task1	`ActivateTask(Task3)`	E_OK
Task1	`ChainTask(Task3)`	
Task3	`TerminateTask()`	
Task3	`TerminateTask()`	
Task1	`ActivateTask(Task1)`	E_OK
Task1	`TerminateTask()`	

Test Sequence 8:
Test Cases: 5, 13, 14, 18
Scheduling policy: mixed-, full-preemptive
Conformance class: BCC2, ECC2
Return status: extended
Parameters: max. number of multiple activations (1 for BCC1 and ECC1)
Tasks: Task1
type = basic
schedule = full
priority = 1
activation = 2
autostart = true
Task2

type = basic
schedule = full
priority = 2
activation = 2
autostart = false

Task3

type = basic
schedule = full
priority = 2
activation = 1
autostart = false

Running task	Called OS service	Return status
Task1	`ActivateTask(Task2)`	E_OK
Task2	`ActivateTask(Task1)`	E_OK
Task2	`ActivateTask(Task3)`	E_OK
Task2	`TerminateTask()`	
Task3	`TerminateTask()`	
Task1	`ActivateTask(Task1)`	E_OK
Task1	`TerminateTask()`	

Test Sequence 9:

Test Cases: 20, 25, 26, 36, 38
Scheduling policy: non-, mixed-, full-preemptive
Conformance class: BCC1, BCC2, ECC1, ECC2
Return status: standard, extended
Tasks: Task1

type = basic
schedule = non
priority = 1
activation = 2
autostart = true

Task2

type = basic
schedule = non
priority = 2
activation = 2
autostart = false

Task3

type = basic
schedule = non
priority = 2
activation = 2
autostart = false

Running task	Called OS service	Return status
Task1	`GetTaskID()`	E_OK, Task1

Running task	Called OS service	Return status
Task1	`GetTaskState(Task1)`	E_OK, *running*
Task1	`GetTaskState(Task2)`	E_OK, *suspended*
Task1	`ActivateTask(Task2)`	E_OK
Task1	`Schedule()`	E_OK
Task2	`GetTaskState(Task1)`	E_OK, *ready*
Task2	`TerminateTask()`	
Task1	`ChainTask(Task3)`	
Task3	`ChainTask(Task3)`	
Task3	`TerminateTask()`	

Test Sequence 10:

Test Cases: 9
Scheduling policy: mixed-, full-preemptive
Conformance class: ECC2
Return status: standard, extended
Tasks: Task1
type = basic
schedule = full
priority = 1
autostart = true
Task2
type = extended
schedule = full
priority = 2
autostart = false
Task3
type = extended
schedule = full
priority = 2
autostart = false

Running task	Called OS service	Return status
Task1	`ActivateTask(Task2)`	E_OK
Task2	`GetEvent(Task2)`	E_OK, all events must be cleared
Task2	`ActivateTask(Task3)`	E_OK
Task2	`TerminateTask()`	
Task3	`GetEvent(Task2)`	E_OK, all events must be cleared
Task3	`TerminateTask()`	
Task1	`TerminateTask()`	

Test Sequence 11:

Extended Task returns from waiting-state to ready-list, where ready task with same priority waits.
Extended Task is treated as oldest task in its list of priority.
Scheduling policy: non-, mixed-, full-preemptive
Conformance class: ECC2
Return status: standard, extended

Tasks: Task1
type = basic
priority = 1
activation = 1
autostart = false
Task2
type = extended
priority = 2
autostart = true
event = Event2
Task3
type = basic
priority = 2
activation = 1
autostart = false
Task4
type = basic
priority = 3
activation = 1
autostart = false

Running task	Called OS service	Return status
Task2	`ActivateTask(Task1)`	E_OK
Task2	`WaitEvent(Event2)`	E_OK
Task1	`ActivateTask(Task3)`	E_OK
Task1	`Schedule()`	E_OK
Task3	`ActivateTask(Task4)`	E_OK
Task3	`Schedule()`	E_OK
Task4	`SetEvent(Task2, Event2)`	E_OK
Task4	`TerminateTask()`	
Task2	`TerminateTask()`	
Task3	`TerminateTask()`	
Task1	`TerminateTask()`	

3.2 Interrupt processing

The test cases 7 and 8 can not be tested, because more than one ISR is necessary. This leads to priority issues which are not covered by the OSEK OS specification.
The test cases 9, 14 and 15 can not be tested, because it is not possible to trigger an interrupt while no task is running.

Test Sequence 1:
Test Cases: 1, 3, 5, 6, 7, 8
Scheduling policy: non-, mixed-, full-preemptive
Conformance class: BCC1, BCC2, ECC1, ECC2
Return status: standard, extended

Tasks: Task1
type = basic
priority = 1
activation = 1
autostart = true
ISR: ISR1
category = 1
ISR2
category = 2
ISR3
category = 3
IntMask-Interrupts are disabled, sets ISR1, ISR2, ISR3.

Running task	Called OS service	Return status
Task1	`EnableInterrupt(IntMask)`	E_OK
Task1	`GetInterruptDescriptor(IntRef)`	E_OK, IntRef=IntMask
Task1	`TriggerInterrupt(ISR2)`	
ISR2	`TriggerInterrupt(ISR3)`	
ISR3	`TriggerInterrupt(ISR1)`	
ISR1	`ReturnFromInterrupt()`	
ISR3	`ReturnFromInterrupt()`	
ISR2	`ReturnFromInterrupt()`	
Task1	`DisableInterrupt(IntMask)`	E_OK
Task1	`TriggerInterrupt(ISR2)`	
Task1	`TerminateTask()`	

Test Sequence 2:
Test Cases: 2, 4
Scheduling policy: non-, mixed-, full-preemptive
Conformance class: BCC1, BCC2, ECC1, ECC2
Return status: extended
Tasks: Task1
type = basic
priority = 1
activation = 1
autostart = true
ISR: ISR2
category = 2
IntMask-Interrupts are enabled.

Running task	Called OS service	Return status
Task1	`EnableInterrupt(IntMask)`	E_OS_NOFUNC
Task1	`DisableInterrupt(IntMask)`	E_OK
Task1	`DisableInterrupt(IntMask)`	E_OS_NOFUNC
Task1	`TerminateTask()`	

Test Sequence 3:

Test Cases: 9, 10
Scheduling policy: non-, mixed-preemptive
Conformance class: BCC1, BCC2, ECC1, ECC2
Return status: standard, extended
Tasks: Task1
type = basic
priority = 1
schedule = non
activation = 1
autostart = true
Task2
type = basic
priority = 2
activation = 1
autostart = false

Task3
type = basic
priority = 3
activation = 1
autostart = false
ISR: ISR2
category = 2
ISR3
category = 3

Running task	Called OS service	Return status
Task1	`TriggerInterrupt(ISR2)`	
ISR2	`ActivateTask(Task2)`	E_OK
ISR2	`ReturnFromInterrupt()`	
Task1	`TerminateTask()`	
Task2	`TriggerInterrupt(ISR3)`	
ISR3	`EnterISR()`	E_OK
ISR3	`ActivateTask(Task2)`	E_OK
ISR3	`LeaveISR()`	
ISR3	`ReturnFromInterrupt()`	
Task2	`TerminateTask()`	
Task3	`TerminateTask()`	

Test Sequence 4:

Test Cases 11, 12
Scheduling policy: mixed-, full-preemptive
Conformance class: BCC1, BCC2, ECC1, ECC2
Return status: standard, extended
Tasks: Task1
type = basic

priority = 1
schedule = full
activation = 1
autostart = true

Task2

type = basic
priority = 2
activation = 1
autostart = false

ISR: ISR2

category = 2

ISR3

category = 3

Running task	Called OS service	Return status
Task1	`TriggerInterrupt(ISR2)`	
ISR2	`ActivateTask(Task2)`	E_OK
ISR2	`ReturnFromInterrupt()`	
Task2	`TerminateTask()`	
Task1	`TriggerInterrupt(ISR2)`	
ISR3	`EnterISR()`	E_OK
ISR3	`ActivateTask(Task2)`	E_OK
ISR3	`LeaveISR()`	
ISR3	`ReturnFromInterrupt()`	
Task2	`TerminateTask()`	
T1	`TerminateTask()`	

3.3 Event mechanism

Test Sequence 1:

Test Case: 1, 2, 3, 11, 12, 13, 15, 16, 17, 21, 22, 23, 24:
Scheduling policy: non-, mixed-, full-preemptive
Conformance class: ECC1, ECC2
Return status: extended
Tasks: Task1

type = basic
priority = 1
activation = 1
autostart = true

Task2

type = basic
priority = 2
activation = 1
autostart = false

Task3

type = extended
priority = 3
activation = 1
autostart = false

resource = Res1
event = Event1

ISR: ISR2
category = 2
ISR3
category = 3

Running task	Called OS service	Return status
Task1	`SetEvent(NoTask)`	E_OS_ID
Task1	`SetEvent(Task2,Event1)`	E_OS_ACCESS
Task1	`SetEvent(Task2, Event1)`	E_OS_STATE
Task1	`ClearEvent(Event1)`	E_OS_ACCESS
Task1	`TriggerInterrupt(ISR2)`	
ISR2	`ClearEvent(Event1)`	E_OS_CALLEVEL
ISR2	`WaitEvent(Event1)`	E_OS_CALLEVEL
ISR2	`ReturnFromInterrupt()`	
Task1	`TriggerInterrupt(ISR3)`	
ISR3	`EnterISR()`	E_OK
ISR3	`ClearEvent(Event1)`	E_OS_CALLEVEL
ISR3	`WaitEvent(Event1)`	E_OS_CALLEVEL
ISR3	`LeaveISR()`	
ISR3	`ReturnFromInterrupt()`	
Task1	`GetEvent(NoName, EventRef)`	E_OS_ID
Task1	`GetEvent(Task2, EventRef)`	E_OS_ACCESS
Task1	`GetEvent(Task2, EventRef)`	E_OS_STATE
Task1	`WaitEvent(Event1)`	E_OS_ACCESS
Task1	`ChainTask(Task3)`	
Task3	`GetResource(Res1)`	E_OK
Task3	`WaitEvent(Event1)`	E_OS_RESOURCE
Task3	`ReleaseResource(Res1)`	E_OK
Task3	`TerminateTask()`	

Testsequence 2:

Test Case 14, 18, 19, 20, 25, 26
Scheduling policy: non-, mixed-, full-preemptive
Conformance class: ECC1, ECC2
Return status: standard, extended
Tasks: Task1
type = extended
priority = 2
autostart = true
event = Event1
Task2
type = extended
priority = 1
autostart = false
event = Event2

Running task	Called OS service	Return status
Task1	`ActivateTask(Task2)`	E_OK
Task1	`WaitEvent(Event1)`	E_OK
Task2	`GetTaskState(Task1, StateRef)`	E_OK, StateRef = waiting
Task2	`GetEvent(Task1, EventRef)`	E_OK, EventRef = 0x0
Task2	`SetEvent(Task2, Event2)`	E_OK
Task2	`GetEvent(Task2, EventRef)`	E_OK, EventRef = Event2
Task2	`WaitEvent(Event2)`	E_OK
Task2	`SetEvent(Task1, Event1)`	E_OK
Task2	`Schedule()`	E_OK
Task1	`GetTaskState(Task2, StateRef)`	E_OK, StateRef = ready
Task1	`GetEvent(Task2, EventRef)`	E_OK, EventRef = Event1
Task1	`TerminateTask()`	

Testsequence 3:

Test Case 4, 5, 9
Scheduling policy: non-, mixed-preemptive
Conformance class: ECC1, ECC2
Return status: standard, extended
Tasks: Task1
type = basic
priority = 1
schedule = non
activation = 1
autostart = false
event =
Task2
type = extended
priority = 2
autostart = true
event = Event1, Event2, Event3

Running task	Called OS service	Return status
Task2	`WaitEvent(Event1)`	E_OK
Task1	`SetEvent(Task1, Event2)`	E_OK
Task1	`GetTaskState(Task2, StateRef)`	E_OK, StateRef = waiting
Task1	`SetEvent(Task1, Event1)`	E_OK, StateRef = ready
Task1	`SetEvent(Task1, Event3)`	E_OK
Task1	`GetEvent(Task1, EventRef)`	E_OK, EventRef = Event1 | Event2 | Event3
Task1	`TerminateTask()`	
Task2	`TerminateTask()`	

Testsequence 4:

Test Case 6, 7, 8, 10
Scheduling policy: mixed-, full-preemptive
Conformance class: ECC1, ECC2
Return status: standard, extended
Tasks: Task1
type = basic
priority = 2
schedule = full
activation = 1
autostart = false
Task2
type = extended
priority = 3
schedule = full
autostart = true
event = Event1, Event2
Task3
type = extended
priority = 1
schedule = full
autostart = false
event = Event3
Task4
type = basic
priority = 4
schedule = full
autostart = false

Running task	Called OS service	Return status
Task2	`ActivateTask(Task1)`	E_OK
Task2	`WaitEvent(Event1)`	E_OK
Task1	`SetEvent(Task2, Event2)`	E_OK
Task1	`GetTaskState(Task2, StateRef)`	E_OK, StateRef = waiting
Task1	`GetEvent(Task2, EventRef)`	E_OK, EventRef = Event2
Task1	`ActivateTask(Task3)`	E_OK
Task1	`GetTaskState(Task3, StateRef)`	E_OK, StateRef = ready
Task1	`SetEvent(Task3, Event3)`	E_OK
Task1	`GetEvent(Task3, EventRef)`	E_OK, EventRef = Event3
Task1	`SetEvent(Task2, Event1)`	E_OK
Task2	`ClearEvent(Event1)`	E_OK
Task2	`WaitEvent(Event1)`	E_OK
Task1	`ActivateTask(Task4)`	E_OK
Task4	`SetEvent(Task2, Event1)`	E_OK
Task4	`GetTaskState(Task2, StateRef)`	E_OK, StateRef = ready
Task4	`TerminateTask()`	

Running task	Called OS service	Return status
Task2	`TerminateTask()`	
Task1	`TerminateTask()`	
Task3	`TerminateTask()`	

3.4 Resource management

Testsequence 1:

Test Case: 1, 2, 3, 4, 5, 9, 10, 11, 12
Scheduling policy: non-, mixed-, full-preemptive
Conformance class: BCC1, BCC2, ECC1, ECC2
Parameters: Number of max. occupied resources in parallel = N
Return status: extended
Tasks: Task1
type = basic
priority = 1
activation = 1
autostart = true
resource = Res0, Res1, Res2, ... , ResN
`Task2`
type = basic
priority = 2
activation = 1
autostart = false
resource = ResA
ISR: ISR2
category = 2
ISR3
category = 3

Running task	Called OS service	Return status
Task1	`GetResource(ResA)`	E_OS_ACCESS
Task1	`GetResource(NoResource)`	E_OS_ID
Task1	`GetResource(Res0)`	E_OK
Task1	`...`	
Task1	`GetResource(Res[N-1])`	E_OK
Task1	`GetResource(Res[N])`	E_OS_LIMIT
Task1	`ReleaseResource(Res[N-1])`	E_OK
Task1	`...`	
Task1	`ReleaseResource(Res0)`	E_OK
Task1	`TriggerInterrupt(ISR2)`	
ISR2	`GetResource(Res0)`	E_OS_CALLEVEL
ISR2	`ReleaseResource(Res0)`	E_OS_CALLEVEL
ISR2	`Return`	
Task1	`TriggerInterrupt(ISR3)`	
ISR3	`EnterISR()`	E_OK

Running task	Called OS service	Return status
ISR3	`GetResource(Res0)`	E_OS_CALLEVEL
ISR3	`ReleaseResource(Res0)`	E_OS_CALLEVEL
ISR3	`LeaveISR()`	
ISR3	`ReturnFromInterrupt()`	
Task1	`ReleaseResource(Res0)`	E_OS_NOFUNC
Task1	`ReleaseResource(NoRes)`	E_OS_ID
Task1	`ChainTask(Task2)`	
Task2	`TerminateTask()`	

Testsequence 2:

Test Case: 6, 8, 13, 15
Scheduling policy: non-, mixed-preemptive
Conformance class: BCC1, BCC2, ECC1, ECC2
Return status: standard, extended
Tasks: Task1
type = basic
priority = 1
schedule = non
activation = 1
autostart = true
resource = RES_SCHEDULER, Res0
Task2
type = basic
priority = 2
activation = 1
autostart = false
resource = RES_SCHEDULER, Res0
Task3
type = basic
priority = 3
activation = 1
autostart = false
resource = RES_SCHEDULER

Running task	Called OS service	Return status
Task1	`GetResource(RES_SCHEDULER)`	E_OK
Task1	`ActivateTask(Task2)`	E_OK
Task1	`Schedule`	E_OK
Task1	`ReleaseResource(RES_SCHEDULER)`	E_OK
Task1	`Schedule`	E_OK
Task2	`TerminateTask()`	E_OK
Task1	`GetResource(Res0)`	E_OK
Task1	`ActivateTask(Task2)`	E_OK
Task1	`Schedule()`	E_OK

Running task	Called OS service	Return status
Task1	`ActivateTask(Task3)`	E_OK
Task1	`Schedule()`	E_OK
Task3	`TerminateTask()`	
Task1	`ReleaseResource(Res0)`	E_OK
Task1	`Schedule()`	E_OK
Task2	`TerminateTask()`	
Task1	`TerminateTask()`	

Testsequence 3:

Test Case: 7, 8, 14, 16
Scheduling policy: mixed-, full-preemptive
Conformance class: BCC1, BCC2, ECC1, ECC2
Return status: standard, extended
Tasks: Task1
type = basic
priority = 1
schedule = full
activation = 1
autostart = true
resource = RES_SCHEDULER, Res0
Task2
type = basic
priority = 2
activation = 1
autostart = false
resource = RES_SCHEDULER, Res0
Task3
type = basic
priority = 3
activation = 1
autostart = false
resource = RES_SCHEDULER

Running task	Called OS service	Return status
Task1	GetResource(RES_SCHEDULER)	E_OK
Task1	ActivateTask(Task2)	E_OK
Task1	ReleaseResource(RES_SCHEDULER)	E_OK
Task2	`TerminateTask()`	E_OK
Task1	GetResource(Res0)	E_OK
Task1	ActivateTask(Task2)	E_OK
Task1	GetTaskState(Task2, StateRef)	E_OK, StateRef = ready
Task1	ActivateTask(Task3)	E_OK
Task3	`TerminateTask()`	
Task1	ReleaseResource(Res0)	E_OK
Task2	`TerminateTask()`	

Running task	Called OS service	Return status
Task1	`TerminateTask()`	

3.5 Alarms

Test Sequence 1

Test Case: 1, 3, 7, 10, 11, 12, 13, 16, 19, 20, 21, 22, 25
Scheduling policy: non-, mixed-, full-preemptive
Conformance class: BCC1, BCC2, ECC1, ECC2
Return status: extended
Tasks: Task1
type = basic
priority = 1
activation = 1
autostart = true
Alarms: Alarm1
counter = timer
action = activatetask
task = Task1

Running task	Called OS service	Return status
Task1	`GetAlarmBase(NoAlarm)`	E_OS_ID
Task1	`GetAlarm(NoAlarm)`	E_OS_ID
Task1	`GetAlarmBase(Alarm1, AlarmBaseRef)`	E_OK
Task1	`SetRelAlarm(NoAlarm, AlarmBaseRef.mincycle, 0)`	E_OS_ID
Task1	`SetRelAlarm(Alarm1, -1, 0)`	E_OS_VALUE
Task1	`SetRelAlarm(Alarm1, AlarmBaseRef.maxallowedvalue +1, 0)`	E_OS_VALUE
Task1	`SetRelAlarm(Alarm1, AlarmBaseRef.mincycle, AlarmBaseRef.mincycle-1)`	E_OS_VALUE
Task1	`SetRelAlarm(Alarm1, AlarmBaseRef.maxallowedvalue , AlarmBaseRef.maxallowedvalue +1)`	E_OS_VALUE
Task1	`SetAbsAlarm(NoAlarm, AlarmBaseRef.mincycle, 0)`	E_OS_ID
Task1	`SetAbsAlarm(Alarm1, -1, 0)`	E_OS_VALUE

Running task	Called OS service	Return status
Task1	`SetAbsAlarm(Alarm1, AlarmBaseRef.maxallowedvalue +1, 0)`	E_OS_VALUE
Task1	`SetAbsAlarm(Alarm1, AlarmBaseRef.mincycle, AlarmBaseRef.mincycle-1)`	E_OS_VALUE
Task1	`SetAbsAlarm(Alarm1, AlarmBaseRef.mincycle AlarmBaseRef.maxallowedvalue +1)`	E_OS_VALUE
Task1	`CancelAlarm(NoAlarm)`	E_OS_ID
Task1	`TerminateTask()`	

Test Sequence 2:

Test Cases: 2, 4, 5, 8, 14, 17, 23, 26, 27, 29
Scheduling policy: non-, mixed-, full-preemptive
Conformance class: BCC1, BCC2, ECC1, ECC2
Return status: standard, extended
Tasks: Task1
type = basic
priority = 3
activation = 1
autostart = true
Task2
type = basic
priority = 2
activation = 1
autostart = false
Task3
type = basic
priority = 1
activation = 1
autostart = false

Alarms: Alarm1
counter = timer
action = activatetask
task = Task2

Running task	Called OS service	Return status
Task1	`GetAlarmBase(Alarm1, AlarmBaseRef)`	E_OK, Value in AlarmBaseRef
Task1	`CancelAlarm(Alarm1)`	E_OS_NOFUNC
Task1	`SetRelAlarm(Alarm1, AlarmBaseRef.maxallowedvalue, 0)`	E_OK (maxallowed vielleicht zu groß)

Running task	Called OS service	Return status
Task1	`SetRelAlarm(Alarm1, AlarmBaseRef.maxallowedvalue, 0)`	E_OS_STATE
Task1	`GetAlarm(Alarm, AlarmRef)`	E_OK, AlarmRef < AlarmBaseRef.maxallowedvalue
Task1	`repeat`	
Task1	`until GetAlarm(Alarm1) = E_OS_NOFUNC`	
Task1	`GetTaskState(Task2, StateRef)`	E_OK, StateRef=ready
Task1	`SetRelAlarm(Alarm1, AlarmBaseRef.maxallowedvalue, 0)`	E_OK
Task1	`CancelAlarm(Alarm1)`	E_OK
Task1	`GetAlarm(Alarm1, AlarmRef)`	E_OS_NOFUNC
Task1	`ChainTask(Task3)`	
Task2	`TerminateTask()`	
Task3	`SetAbsAlarm(Alarm1, 0, 0)`	E_OK
Task3	`TerminateTask()`	
	`Scheduler`	
Task2	`TerminateTask()`	

Test Sequence 3

Test Cases: 6, 9, 15, 18, 24, 28
Scheduling policy: non-, mixed-, full-preemptive
Conformance class: ECC1, ECC2
Return status: standard, extended
Tasks: Task1
type = basic
priority = 2
activation = 1
autostart = true
Task2
type = extended
priority = 1
autostart = false
event = Event2

Alarm: Alarm1
counter = timer
action = setevent
task = Task2
event = Event2

Running task	Called OS service	Return status
Task1	`GetAlarmBase(Alarm1, AlarmBaseRef)`	E_OK

Running task	Called OS service	Return status
Task1	`ActivateTask(Task2)`	E_OK
Task1	`SetRelAlarm(Alarm1, AlarmBaseRef.maxallowed, 0)`	E_OK (maxallowed vielleicht zu groß)
Task1	`SetRelAlarm(Alarm1, AlarmBaseRef.maxallowed, 0)`	E_OS_STATE
Task1	`GetAlarm(Alarm1, AlarmRef)`	E_OK, AlarmRef < AlarmBaseRef.maxallowedvalue
Task1	`repeat`	
Task1	`until GetAlarm(Alarm1) = E_OS_NOFUNC`	
Task1	`GetEvent(Task2, EventRef)`	E_OK, EventRef = Event2
Task1	`SetAbsAlarm(Alarm1, 0, 0)`	E_OK
Task1	`GetAlarm(Alarm1, AlarmRef)`	E_OK, AlarmRef > 0
Task1	`CancelAlarm(Alarm1)`	E_OK
Task1	`TerminateTask()`	
Task2	`TerminateTask()`	

Test Sequence 4:

Test Cases: 30
Scheduling policy: non-, mixed-preemptive
Conformance class: BCC1, BCC2, ECC1, ECC2
Return status: standard, extended
Tasks: Task1
type = basic
priority = 1
schedule = non
activation = 1
autostart = true
Task2
type = basic
priority = 2
activation = 1
autostart = false
Alarms: Alarm1
counter = timer
action = activatetask
task = Task2

Running task	Called OS service	Return status
Task1	`SetRelAlarm(Alarm1, 1000, 0)`	E_OK
Task1	`repeat`	

Running task	Called OS service	Return status
Task1	`until GetAlarm(Alarm1) = E_OS_NOFUNC`	
Task1	`GetTaskState(Task2, StateRef)`	E_OK, StateRef=ready
Task1	`TerminateTask()`	
Task2	`TerminateTask()`	

Test Sequence 5:

Test Case 31, 32
Scheduling policy: mixed-, full-preemptive
Conformance class: BCC1, BCC2, ECC1, ECC2
Return status: standard, extended
Tasks: Task1
type = basic
priority = 1
schedule = full
activation = 1
autostart = true
Task2
type = basic
priority = 2
activation = 1
autostart = false
Task3
type = basic
priority = 3
schedule = full
activation = 1
autostart = false
Alarms: Alarm1
counter = timer
action = activatetask
task = Task2

Running task	Called OS service	Return status
Task1	`GetAlarmBase(Alarm1, AlarmBaseRef)`	E_OK
Task1	`SetRelAlarm(Alarm1, AlarmBaseRef.mincycle, 0)`	E_OK
Task1	`until GetAlarm(Alarm1) = E_OS_NOFUNC`	
Task2	`TerminateTask()`	
Task1	`ChainTask(Task3)`	
Task3	`GetAlarmBase(Alarm1, AlarmBaseRef)`	E_OK

Running task	Called OS service	Return status
Task3	`SetRelAlarm(Alarm1, AlarmBaseRef.mincycle, 0)`	E_OK
Task3	`repeat`	
Task3	`until GetAlarm(Alarm1) = E_OS_NOFUNC`	
Task3	`GetTaskState(Task2, StateRef)`	E_OK, StateRef = ready
Task3	`TerminateTask()`	
Task2	`TerminateTask()`	

Test Sequence 6:

Test Cases: 33, 34
Scheduling policy: non-, mixed-preemptive
Conformance class: ECC1, ECC2
Return status: standard, extended
Tasks: Task1
type = basic
priority = 2
schedule = non
activation = 1
autostart = false
Task2
type = extended
priority = 1
Schedule = non
autostart = true
event = Event2
Task3
type = basic
priority = 3
schedule = non
activation = 1
autostart = false

Alarms: Alarm1
counter = timer
action = setevent
task = Task2
event = Event2

Running task	Called OS service	Return status
Task2	`ActivateTask(Task1)`	E_OK
Task2	`Schedule`	E_OK
Task1	`GetAlarmBase(Alarm1, AlarmBaseRef)`	E_OK

Running task	Called OS service	Return status
Task1	`SetRelAlarm(Alarm1, AlarmBaseRef.mincycle, 0)`	E_OK
Task1	`repeat`	
Task1	`until GetAlarm(Alarm1) = E_OS_NOFUNC`	
Task1	`GetEvent(Task2, EventRef)`	E_OK, EventRef = Event2
Task1	`TerminateTask()`	
Task2	`ActivateTask(Task3)`	E_OK
Task2	`ClearEvent(Event2)`	E_OK
Task2	`WaitEvent(Event2)`	E_OK
Task3	`GetAlarmBase(Alarm1, AlarmBaseRef)`	E_OK
Task3	`SetRelAlarm(Alarm1, AlarmBaseRef.mincycle 0)`	E_OK
Task3	`repeat`	
Task3	`until GetAlarm(Alarm1) = E_OS_NOFUNC`	
Task3	`GetTaskState(Task2, StateRef)`	E_OK, StateRef = ready
Task3	`TerminateTask()`	
Task2	`TerminateTask()`	

Test Sequence 7:

Test Cases: 35, 36
Scheduling policy: mixed-, full-preemptive
Conformance class: ECC1, ECC2
Return status: standard, extended
Tasks: Task1
type = basic
priority = 1
schedule = full
activation = 1
autostart = false
Task2
type = extended
priority = 2
Schedule = full
autostart = true
event = Event2
Task3
type = basic
priority = 3
schedule = full
activation = 1
autostart = false
Task4

type = basic
priority = 4
schedule = full
activation = 1
autostart = false

Alarms: Alarm1

counter = timer
action = setevent
task = Task2
event = Event2

Running task	Called OS service	Return status
Task2	`ActivateTask(Task1)`	E_OK
Task2	`WaitEvent(Event2)`	E_OK
Task1	`ActivateTask(Task3)`	E_OK
Task3	`GetAlarmBase(Alarm1, AlarmBaseRef)`	E_OK
Task3	`SetRelAlarm(Alarm1, AlarmBaseRef.mincycle 0)`	E_OK
Task3	`repeat`	
Task3	`until GetAlarm(Alarm1) = E_OS_NOFUNC`	
Task3	`GetTaskState(Task2, StateRef)`	E_OK, StateRef = ready
Task3	`TerminateTask()`	E_OK
Task2	`ClearEvent(Event2)`	E_OK
Task2	`ActivateTask(Task4)`	E_OK
Task4	`GetAlarmBase(Alarm1, AlarmBaseRef)`	E_OK
Task4	`SetRelAlarm(Alarm1, AlarmBaseRef.mincycle, 0)`	E_OK
Task4	`repeat`	
Task4	`until GetAlarm(Alarm1) = E_OS_NOFUNC`	
Task4	`GetEvent(Task2, EventRef)`	E_OK, EventRef = Event2
Task4	`TerminateTask()`	
Task2	`TerminateTask()`	
Task1	`TerminateTask()`	

3.6 Error handling, hook routines and OS execution control

Test Sequence 1:

Test Cases: 1, 2, 3, 4, 5, 6, 7, 8
Scheduling policy: non-, mixed-, full-preemptive
Conformance class: BCC1, BCC2, ECC1, ECC2
Return status: standard, extended

Hook routines: StartupHook = true
ErrorHook = false
ShutdownHook = true
PreTaskHook = true
PostTaskHook = true

Tasks: Task1
type = basic
schedule = non
priority = 1
activation = 1
autostart = false

Task2
type = basic
schedule = non
priority = 2
activation = 1
autostart = false

Running task	Called OS service	Return status
StartupHook	`ActivateTask(Task1)`	E_OK
PreTaskHook	`GetTaskID()`	E_OK, Task1
PreTaskHook	`GetTaskState(Task1)`	E_OK, *running*
PreTaskHook	`GetTaskState(Task2)`	E_OK, *suspended*
Task1	`ActivateTask(Task2)`	E_OK
Task1	`Schedule()`	E_OK
PostTaskHook	`GetTaskID()`	E_OK, Task1
PostTaskHook	`GetTaskState(Task1)`	E_OK, *running*
PostTaskHook	`GetTaskState(Task2)`	E_OK, *suspended*
PreTaskHook	`GetTaskID()`	E_OK, Task2
PreTaskHook	`GetTaskState(Task1)`	E_OK, *ready*
PreTaskHook	`GetTaskState(Task2)`	E_OK, *running*
Task2	`TerminateTask()`	E_OK
PostTaskHook	`GetTaskID()`	E_OK, Task2
PostTaskHook	`GetTaskState(Task1)`	E_OK, *ready*
PostTaskHook	`GetTaskState(Task2)`	E_OK, *running*
PreTaskHook	`GetTaskID()`	E_OK, Task1
PreTaskHook	`GetTaskState(Task1)`	E_OK, *running*
PreTaskHook	`GetTaskState(Task2)`	E_OK, *suspended*
T1	`ShutdownOS()`	E_OK
ShutdownHook		

Test Sequence 2:

Test Cases: 1, 2, 3, 4, 5, 6, 7
Scheduling policy: non-, mixed-, full-preemptive
Conformance class: ECC1, ECC2
Return status: standard, extended
Hook routines: StartupHook = true
ErrorHook = false
ShutdownHook = true

PreTaskHook = true
PostTaskHook = true

Tasks: Task1

type = basic
schedule = non
priority = 1
activation = 1
autostart = false

Task2

type = basic
schedule = non
priority = 2
activation = 1
autostart = false

Running task	Called OS service	Return status
StartupHook	`ActivateTask(Task1)`	E_OK
PreTaskHook	`GetTaskID()`	E_OK, Task1
PreTaskHook	`GetTaskState(Task1)`	E_OK, *running*
PreTaskHook	`GetTaskState(Task2)`	E_OK, *suspended*
Task1	`ActivateTask(Task2)`	E_OK
Task1	`Schedule()`	E_OK
PostTaskHook	`GetTaskID()`	E_OK, Task1
PostTaskHook	`GetTaskState(Task1)`	E_OK, *running*
PostTaskHook	`GetTaskState(Task2)`	E_OK, *suspended*
PreTaskHook	`GetTaskID()`	E_OK, Task2
PreTaskHook	`GetTaskState(Task1)`	E_OK, *ready*
PreTaskHook	`GetTaskState(Task2)`	E_OK, *running*
Task2	`TerminateTask()`	E_OK
PostTaskHook	`GetTaskID()`	E_OK, Task2
PostTaskHook	`GetTaskState(Task1)`	E_OK, *ready*
PostTaskHook	`GetTaskState(Task2)`	E_OK, *running*
PreTaskHook	`GetTaskID()`	E_OK, Task1
PreTaskHook	`GetTaskState(Task1)`	E_OK, *running*
PreTaskHook	`GetTaskState(Task2)`	E_OK, *suspended*
Task1	`ShutdownOS()`	E_OK
ShutdownHook		

Test Sequence 2:

Test Cases: 1, 2, 3, 4, 5, 6, 7
Scheduling policy: non-, mixed-, full-preemptive
Conformance class: BCC1, BCC2, ECC1, ECC2
Return status: extended
Hook routines: StartupHook = true
ErrorHook = false
ShutdownHook = true
PreTaskHook = true
PostTaskHook = true

Tasks: Task1

type = basic
schedule = non
priority = 1
activation = 1
autostart = false

Task2

type = basic
schedule = non
priority = 2
activation = 1
autostart = false

Running task	Called OS service	Return status
StartupHook	`ActivateTask(Task1)`	
PreTaskHook	`GetTaskID()`	E_OK, Task1
Task1	`ChainTask(Task2)`	E_OK
PostTaskHook	`GetTaskID()`	E_OK, Task1
PreTaskHook	`GetTaskID()`	E_OK, Task2
Task2	`ShutdownOS()`	E_OK
PostTaskHook	`GetTaskID()`	E_OK, Task2
ShutdownHook		

4 Abbreviations

API	Application Programming Interface
COM	Communication
DLL	Data Link Layer
ECU	Electronic Control Unit
ISO	International Standard Organization
ISR	Interrupt Service Routine
IUT	Implementation Under Test
LT	Lower Tester
NM	Network Management
OPDU	OSEK Protocol Data Unit
OS	Operating System
PDU	Protocol Data Unit
PCO	Point of Control and Observation
SDL	Specification and Description Language
TMP	Test Management Protocol
TM_PDU	Test Management - Protocol Data Unit
TTCN	Tree and Tabular Combined Notation
UT	Upper Tester

5 References

[1] OSEK/VDX Conformance Testing Methodology - Version 1.0 - 19th of December 1997

[2] OSEK/VDX OS Test Plan - Version 1.0 - 4th of March 1998

[3] OSEK/VDX Certification Procedure - F. Kaag, J. Minuth, K.J. Neumann, H. Kuder - Proceedings of the 1st International Workshop on Open Systems in Automotive Networks - October 1995.

[4] OSEK/VDX Operating System - Version 2.0 revision 1- 15th of October1997

[5] ISO/IEC 9646-1 - Information technology, Open Systems Interconnection, Conformance testing methodology and framework, part 1 : General Concepts, 1992.

[6] ISO/IEC 9646-3 - Information technology, Open Systems Interconnection, Conformance testing, methodology and framework, part 3 : The Tree and Tabular Combined Notation (TTCN), 1992.

[7] Benutzerdokumentation "Classification-Tree Editor - CTE für MS-Windows", Version 1.2 - ATS Automated Testing Solutions GmbH, Daimler-Benz AG, 1st of February 1998.