Open Systems and the Corresponding Interfaces

for Automotive Electronics

OSEK/VDX

OS Test Plan

Version 1.0

March 11th, 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 5
2 Test suite structure 6
2.1 Description 6
2.2 Test purposes structure 6
3 Test purposes 8
3.1 Implementation specific parameters 8
3.2 Task management 9
3.3 Interrupt processing 10
3.4 Event mechanism 11
3.5 Resource management 11
3.6 Alarms 12
3.7 Error handling, hook routines and OS execution control 12
4 Appendix I 14
5 Abbreviations 15
6 References 16

1 Introduction

This document contains the test plan for the conformance test of the operating system. It is therefore the basis for the definition of the test cases, which are used to certify conformance of an OS implementation.

According to the Conformance Testing Methodology [1], definition of the conformance test is a two-stage process. In the first stage, the OS specification is analysed and the test purposes are extracted from it. The assembly of the test purposes makes up the test plan. In the second stage test cases are defined, which specify the sequence of the interactions between the test application and the implementation to verify one or more test purposes. The assembly of the test cases makes up the test suite. Together with all information needed to implement and execute the conformance tests make up the test procedure.

According to the different functionalities of the operating system (task management, resource management, ...) it is reasonable to structure and group the test purposes. This structure is explained in chapter 2.

2 Test suite structure

2.1 Description

As agreed in the Conformance Testing Methodology [1] the implementation to be tested is seen as a black box whose external interfaces – the OS API – are accessible only. Implementation-specific details will not be taken into account and only that parts of the specification which are accessible and observable by the operating system's service routines, can be tested for conformance. Therefore, executing the conformance test means that a test application is generated and executed together with the implementation to be tested. The actions and verifications this application has to perform are defined by a test suite. The definition of the conformance test suite is done in two steps:

- definition of the test purposes,
- definition of the test cases.

The test purposes are developed by analysing the specification and extracting checkable assertions. The assertions determine what can and what must be tested. Testable assertions are, on the one hand observable actions (task switches, interrupts, etc.) performed by the operating system, on the other hand the correctness of the return status of OS services. Thus, during the conformance test each OS service routine has to be called at least once for each specified return status.

In order to define the test cases it is necessary to further refine the assertions developed before. Refinement means that it is necessary to analyse the assertions and detect all situations and states of the system which may have an influence on the behaviour of a special assertion. This task will be done by means of the classification-tree method which provides a systematic way for generating test cases. A classification tree describes a complete decomposition of all possible situations and states of the system. On this basis, test sequences have to be evolved which execute and verify these test cases.

To apply the conformance tests on an OS implementation several adjustments will be necessary. Parameters for the conformance test are, among others, the maximum number of tasks, the maximum number of priorities, etc. Furthermore, special routines are required for system dependant functions like logging of variable values, trigger interrupts by software, etc. This extensions may be made available by the vendor of the operating system in form of a library, or they have to be created during the test integration.

This document describes the test purposes and assertions which are derived from the specification of the operating system. First, the structure of the assertions will be shown. This includes the grouping of assertions according to the OS's service groups as well as determining to which variants of the operating system they rely on. In the second part the assertions themselves will be presented.

2.2 Test purposes structure

It is reasonable to group the assertions derived from the specification according to the service groups and functionalities of the operating system. They will be classified according to the following service groups:

- Task management,
- Interrupt processing,

- Event mechanism,
- Resource management,
- Alarms,
- Error handling, hook routines and OS execution control (including start-up/shutdown of OS).

To deal with various requirements of the application software for the system and various capabilities of a specific system (e.g. processor, memory) the OSEK OS offers the possibility to generate several variants of a system. The variants apply to the following categories:

- Conformance class:
    - BCC1 (only basic tasks, limited to one request per task and one task per priority, while all tasks have different priorities)
    - BCC2 (like BCC1, plus more than one task per priority possible and multiple requesting of task activation allowed)
    - ECC1 (like BCC1, plus extended tasks)
    - ECC2 (like BCC2, plus extended tasks without multiple requesting admissible)
- Scheduling policy:
    - non-preemptive
    - mixed-preemptive
    - full-preemptive
- Return status:
    - standard (return values of system services provided in the standard version)
    - extended (return values of system services provided in the extended version for debugging purposes)

For each assertion has to be checked for which variants it is relevant, because some assertions are not checkable under certain circumstances. E.g. the assertions about the event mechanism are not relevant for the conformance classes BCC1 and BCC2, as they don't support events.

3 Test purposes

This chapter describes the test purposes relevant to the functionality and behaviour of the operating system. They were established by reading the specification and extracting checkable assertions. The assertions were analysed to remove redundancies. These assertions build the basis on which the test cases and the test suite are developed. Therefore, it is necessary to further refine these assertions. According to the Conformance Testing Methodology [1] this refinement will be done by means of the classification-tree method. This method was developed at Daimler-Benz AG and is supported by the commercial tool CTE by ATS Automated Testing Solution GmbH [6]. The resulting test cases and the sequences used to evaluate them will be described in the test procedure.

As mentioned in the previous chapter, the assertions are grouped according to several aspects of the operating system. Each of the following chapters represents one group of test purposes. The test purposes are listed in a table which contains for each assertion:

- a sequence number used as a reference for test suite traceability,
- the description of the test purpose extracted from the specification,
- the variants of the specification to which the purpose applies,
- a reference to the paragraph in the specification allowing traceability to be provided against the specification.

3.1 Implementation specific parameters

In accordance with the specification 2.0 of the OSEK operating system, the vendor has to provide a list of parameters specifying the implementation. This list gives detailed information concerning the functionality, performance and memory demand, as well as the basic conditions to reproduce the measurement of those parameters.

In order to test the conformance of a specific implementation to the OSEK OS specification, one has to ensure that the list with implementation-specific parameters provided by the vendor exists, and contains all prescribed parameters. It is important to point out that the conformance test neither includes a test for the correctness of these parameters, nor does it specify any limit for hardware requirements or performance figures that must be kept. To achieve conformance it is sufficient for the operating system vendor to provide a list of parameters specifying the implementation's behaviour. To allow their verifications, this list must include a sufficient description of the methods used to collect the presented informations.

This chapter refers to those parameters which describe basic functionalities of the OS implementation. Therefore, they are needed in order to build and execute the test applications. It is reasonable to provide additional parameters, like required hardware resources and performance issues. They are listed in appendix I which may be changed in the future. Indeed it is not obvious, from today's point, which parameters are relevant for customers to evaluate an OS implementation.

The following table lists each parameter which must be contained in the list of parameters as one assertion.

No.	Assertion	Page	Paragraph in spec.	Affected variants
1	Maximum number of tasks	63	12.2.1	All
2	Maximum number of active tasks (*running*/ *ready*/ *waiting*) (≥8 for BCC1/BCC2, ≥16 for ECC1/ECC2)	63	12.2.1	All
3	Maximum number of priorities (≥8)	63	12.2.1	All
4	Number of tasks per priority (>1)	63	12.2.1	BCC2, ECC2
5	Upper limit for number of task activations	63	12.2.1	BCC2, ECC2
6	Maximum number of events per task (≥8)	63	12.2.1	ECC1, ECC2
7	Limits for the number of alarm objects (per system/ per task)	63	12.2.1	All
8	Limits for the number of nested resources (per system/ per task)	63	12.2.1	All
9	Lowest priority level used internally by the OS	63	12.2.1	All
10	Timer units reserved for the OS	63	12.2.2	All
11	Interrupts, traps and other hardware resources occupied by the OS	63	12.2.2	All

3.2 Task management

Task management concerns the activation and scheduling of tasks. The behaviour of the scheduler depends on the conformance class and the scheduling policy.

Several attributes are assigned to each task:

- task type: basic, extended
- priority
- scheduling type: full-, non-preemptive

No.	Assertion	Page	Paragraph in spec.	Affected variants
1	Interrupts and OS have higher priority than tasks.	14	3.1	All
2	OS has to provide at least 8 levels of task priorities.	63	12.2	All
4	States for EXTENDED tasks are: *running*, *ready*, *suspended*, *waiting*. EXTENDED tasks release the processor, if • they terminate • they are preemptive and OS is executing a higher priority task • an Interrupt is executed • they go to *waiting* state • a transition from *running* to *waiting* state occurs, if the task waits for an event.	17	4.2.1	ECC1, ECC2
5	Tasks in *ready* state wait for allocation of the processor. When no task with higher priority is in *ready* or *running* state, this task is put to *running* state, if no interrupt is processed.	17	4.2.1	All

No.	Assertion	Page	Paragraph in spec.	Affected variants
6	A task in *suspended* state is not active. Task activation puts it to *ready* state.	17	4.2.1	All
7	A task in *waiting* state waits at least for one event. With the occurrence the task is set to *ready* state.	17	4.2.1	ECC1, ECC2
8	Pre-empted task is treated as the first task in the *ready* list of its priority.	17	4.2.1	All
9	States for BASIC tasks are: *running*, *ready*, *suspended.* BASIC tasks release the processor, if • they terminate • they are preemtive and OS is executing a higher priority task • an Interrupt is executed	18	4.2.2	All
10	The OS ensures that after a task has been activated its execution will start with the task's first instruction.	20	4.3	All
11	Multiple activation is supported in BCC2/ECC2 for basic tasks, a task attribute limits the number of multiple activation.	20	4.3	BCC2, ECC2
12	Multiple task activations are stored in a FIFO structure in order to preserve activation order	20	4.3	BCC2, ECC2
13	Bigger Numbers refer to higher priorities. (0 is lowest)	20	4.5	All
14	In BCC2 and ECC2 tasks with same priority are possible. Processing of the tasks with same priority depends on their order of activation.	20	4.5	BCC2, ECC2
15	A task being released from *waiting* state is treated like the newest task in the *ready* queue of its priority.	21	4.5	ECC2
16	Points of rescheduling (possible task switch) with non-preemptive scheduling: • A task terminates itself via *TerminateTask* or *ChainTask* • An explicit call of the scheduler (*Schedule*) • The task waits for an event	21	4.6.1	Non-preemptive
17	Within full-preemptive scheduling a task switch occurs, whenever a task with higher priority is set to *ready* state.	22	4.6.2	Full-preemptive
18	Scheduling policies can be mixed. A task can be defined non-preemptive in a mixed-preemptive OS, i.e. no preemption can occur as long as this non-preemptive task is running.	23	4.6.3	Mixed-preemptive

3.3 Interrupt processing

The OSEK OS provides several services to handle interrupts. They can be used to enable and disable interrupts and to allow the use of OS services within an interrupt service routine. But the handling of interrupts is very hardware specific.

This concerns in particular interrupts of category 1, as no ISR-frame is prepared for the operating system. Therefore, it is not allowed to call any OS service, which prevents observation of the behaviour of the interrupt service routine.

No.	Assertion	Page	Paragraph in spec.	Affected variants
1	Interrupts of category 2: Calls to OS services are restricted. Calling a forbidden OS service produces the error E_OS_CALLEVEL.	26	5	Extended error status
2	Interrupts of category 3: Calls to OS services are restricted. They are allowed if enclosed within a *Enter/LeaveISR* frame. Within this frame calling a forbidden OS service produces the error E_OS_CALLEVEL, outside this frame the behaviour is not defined.	26	5	Extended error status

3.4 Event mechanism

The event mechanism is a means of synchronisation. It is provided for extended tasks only. Events are objects managed by the operating system. Each event is assigned to an extended task. Various system services are provided to manipulate events.

Events are supported in the extended conformance classes (ECC1, ECC2) only.

No.	Assertion	Page	Paragraph in spec.	Affected variants
1	An event is assigned to an extended task.	28	6	ECC1, ECC2
2	One task can own at least 8 events. This is the minimum value for the parameter „Number of events per task“	63	12.2	ECC1, ECC2
3	When at least one event a task is waiting for occurs, this task is set to *ready* state.	28	6	ECC1, ECC2
4	An event can only be cleared by its owner by calling *ClearEvent*.	28	6	ECC1, ECC2
5	When activating an extended task by calling *ActivateTask*, its events are cleared by the OS.	28	6	ECC1, ECC2
6	Any task can set events.	28	6	ECC1, ECC2
7	If an extended task tries to wait for an event, which has already occurred at least once, it remains in *running* state.	28	6	ECC1, ECC2

3.5 Resource management

The resource management is used to co-ordinate concurrent accesses of several tasks to shared resources. It has to ensure that two tasks cannot occupy the same resource at the same time and that priority inversion or deadlocks cannot occur. The specification implies to use the priority ceiling protocol even when it is not mandatory. However, the behaviour of the system must be identical to the priority ceiling protocol whether the implementation uses it or not.

No.	Assertion	Page	Paragraph in spec.	Affected variants
1	A task cannot terminate or switch to *waiting* state, while it occupies a resource. This can only be checked if OS supports extended error states, otherwise the behaviour is undefined.	30	7.2	Extended error status
2	The scheduler is treated like a resource which is accessible to all tasks. A standard resource with a defined name (constant RES_SCHEDULER) is generated. It can be occupied to prevent interruptions by other tasks.	30	7.3	All
3	OS ensures (e.g. by priority ceiling protocol) that tasks are only transferred from the *ready* state to the *running* state, if all resources, the task might need, are released.	30	7.1	All
4	After a task has occupied a resource any other task which might occupy the same resource does not enter the *running* state, even if its priority is higher than the priority of the task occupying this resource. This behaviour is equivalent to the priority ceiling protocol.	32	7.5	All

3.6 Alarms

Expiration of alarms is determined on the basis of counters. As there exists no API for counters their functionality cannot be tested. The same holds true for non-variant alarms.

No.	Assertion	Page	Paragraph in spec.	Affected variants
1	Alarm will expire when a predefined counter value is reached	33	8.2	All
2	Alarms are statically assigned to • One counter • One task • A notation, if that task is to be activated or an event is to be set (only in ECC1, ECC2)	33	8.2	All ECC1, ECC2
3	Alarms can be manipulated by the user.	34	8.2	All
4	Absolute and relative alarms are supported, both may be set to cyclic or single alarms.	34	8.2	All
5	The OS provides at least one counter which is derived from a timer. User can assume existence of this counter.	34	8.2	All

3.7 Error handling, hook routines and OS execution control

The OSEK operating systems provides hook routines which allow user-defined actions within the OS internal processing, e.g. at task switches. The interface of hook routines is implementation dependant except the first parameter which is fixed.

Error handling of the OSEK operating system is limited to a status information returned by the system services. If fatal errors occur a centralised system shutdown is called. But, as the conditions for this shutdown are implementation dependant, this is not testable.

No.	Assertion	Page	Paragraph in spec.	Affected variants
1	The first parameter of hook routines is fixed, additional parameters are optional and implementation specific.	35	9.1	All
2	Hook routines are a part of the OS, but user-defined. Therefore they are higher prior than all tasks and thus can’t be preempted.	35	9.1	All
3	Hook routines are only allowed to use a subset of OS services. It can not be checked if a OS service is called which is not part of this subset.	35	9.1	All
4	Every OS call returns the status code. If the OS could not execute the requested service correctly, status code is not equal E_OK.	37	9.2.1	All
5	The operating system starts with a call to *StartOS* with the application mode as a parameter.	38	9.3	All
6	After the OS is initialised (scheduler not running), it calls the *StartupHook* before starting the first user task.	38	9.3	All
7	During execution of StartupHook, all user interrupts are disabled.	38	9.3	All
8	After *StartupHook*, the interrupt mask is set according to INITIAL_INTERRUPT_DESCRIPTOR.	38	9.3	All
9	When *ShutdownOS* is called with a defined error code, the OS will shutdown and call the hook routine *ShutdownHook*.	60	11.7	All
10	*PostTaskHook* is called after executing the current task, but before leaving the task's *running* state.	60	11.7	All
11	*PreTaskHook* is called before executing the new task, but after the transition of the task to the *running* state.	60	11.7	All
12	*ErrorHook* is called if a system call returns a value not equal to E_OK.	60	11.7	All
13	Naming convention for status information: • all errors of API services start with E_ • errors of OS start with E_OS_ • internal errors of OS (implementation specific) start with E_OS_SYS_	43	11.1	All
14	Values of the status information API services offer: • E_OK = 0 • 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	43	11.1	All
15	The application mode that is passed to the StartOS function can be detected by the *GetActiveApplicationMode* function.	59	11.6	All

4 Appendix I

This appendix list implementation specific parameters which are proposed by the specification to be provided by the vendor. Anyway, as they are too dependant on the environment and the applications running on the system, to be useful to customers, it doesn't seem to be reasonable to determine them. Thus, the MODISTARC OS group decided not to use them as criteria for compliance and therefore put them into this appendix.

No.	Assertion	Page	Paragraph in spec.	Affected variants
1	RAM and ROM requirement for each of the OS components	63	12.2.2	All
2	Size for each linkable module	63	12.2.2	All
3	Application dependant RAM and ROM requirements for OS data (e.g. bytes RAM per task, RAM required per alarm, ...)	63	12.2.2	All
4	Execution context of the OS (e.g. size of OS internal tables)	63	12.2.2	All
5	Total execution time for each service	63	12.2.3	All
6	OS start-up time without invoking hook routines	63	12.2.3	All
7	Interrupt latency for ISR of category 1, 2 and 3	63	12.2.3	All
8	Task switching times for all types of switching	63	12.2.3	All
9	Idle CPU overhead	63	12.2.3	All

5 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

6 References

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

[2] 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.

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

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

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

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