OSEK/VDX Operating System (1)

Part 1 ~
Part 3 : ~3.4 Task Management

Contents

- Structure of ISO 17356
  - Part 1 : General structure and terms, definitions and abbreviated terms
  - Part 2 : OSEK/VDX specifications for binding OS, COM and NM
  - Part 3 : OSEK/VDX Operating system **(OS)**
  - Part 4 : OSEK/VDX Communication **(COM)**
  - Part 5 : OSEK/VDX Network management **(NM)**
  - Part 6 : OSEK/VDX implementation language **(OIL)**

Part 1. General structure and terms, definitions and abbreviated terms

What is OSEK/VDX?

- OSEK/VDX
  - a joint project of the automotive industry
- Its aim
  - industry standard for an open-ended architecture for distributed control units in vehicles
- 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

What is OSEK/VDX?

- OSEK/VDX
  - a joint project of the automotive industry
- Its aim
  - industry standard for an open-ended architecture for distributed control units in vehicles
- Problem :
  - 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

What is OSEK/VDX?

- Goal :
  - Support of the portability and reusability of the application software
    - 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.

What is OSEK/VDX?

- Advantages :
  - Clear savings in costs and development time
  - Enhanced quality of the software of control units
  - Standardized interfacing feature 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 H/W
  - Provides independence with regards to individual implementation, as the specification does not prescribe implementation aspects

ISO 17356 model

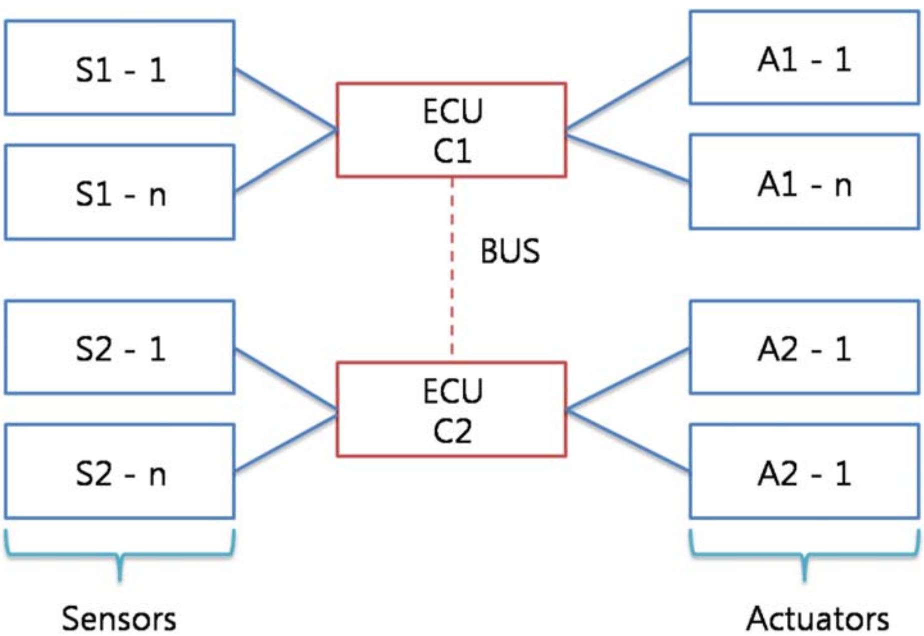


Fig.1 ISO 17356 model

Typical automotive architecture

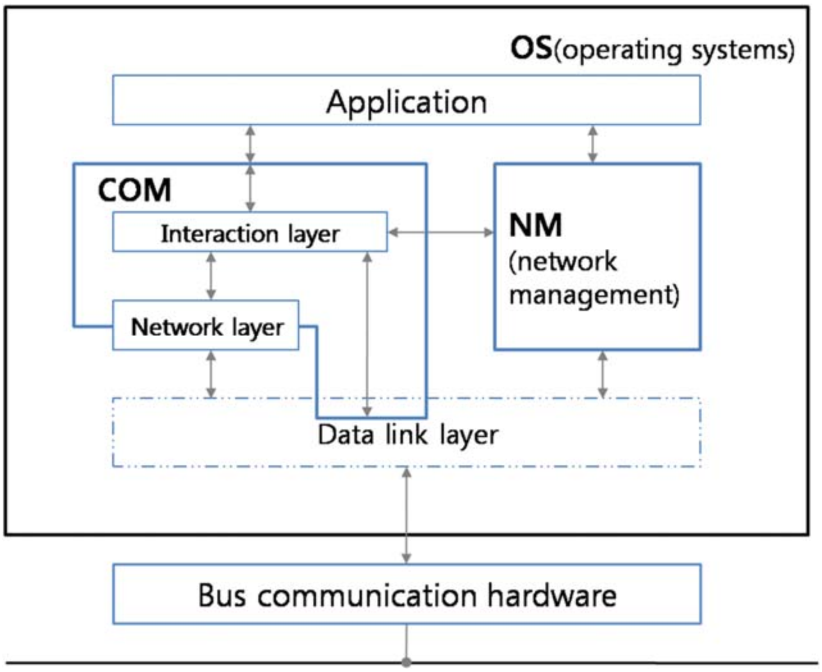


Fig 2. Typical automotive architecture

Part 2 : OSEK/VDK specifications for binding OS, COM and NM

Definition of StatusType

- The data type *StatusType is used within all parts of OSEK/VDX.*
- To be able to combine different parts of OSEK/VDX from different supplies, the definition of this type has to be handled with care to avoid conflicts.

```
#ifndef  STATUSTYPEDEFINED
#define STATUSTYPEDEFINED
typedef unsigned char StatusType;
#define E_OK 0
#endif
```

- These definitions have to be done in the header files supplied by the OSEK suppliers.

Definition of error codes

0	E_OK
1 to 31	OSEK OS error code
32 to 63	OSEK COM error code
64 to 95	OSEK NM error code
96 to 255	RESERVED

Part 3. OSEK/VDX Operating System(OS)

OSEK OS overview

- Features :
  - Standardized interfaces
    - The interface between the application software and the operating system is defined by system services.
  - Scalability
    - By Different conformance classes, various scheduling mechanisms and the configuration features
  - Error checking
    - The OSEK OS offers two levels of error checking, extended status for development phase and standard status for production phase.

OSEK OS overview

- Portability of application software
  - Portability - the ability to transfer an application software module from one ECU to another ECU without bigger changes inside the application.
  - Only the operating system interface to the application is considered.
- Special support for automotive requirement
  - OS is configured and scaled statically.
  - OS supports implementations capable of running on ROM.
  - OS supports portability of application task.
  - The specification of the OS provides a predictable and documented behavior, which meet automotive real-time requirements.
  - The specification of the OS allows the implementation of predictable performance parameters.

1. Architecture of the OS

1. Architecture of the OS

1. Processing Levels
2. Priority Rules
3. Conformance Classes

1.1 Processing Levels

- User Interfaces Entities
  - Interrupt service routines ISRs (managed by the OS)
  - Tasks (basic / extended tasks)
- 3 processing levels
  - Interrupt level
  - Logical level scheduler
  - Task level

Processing level of the OS

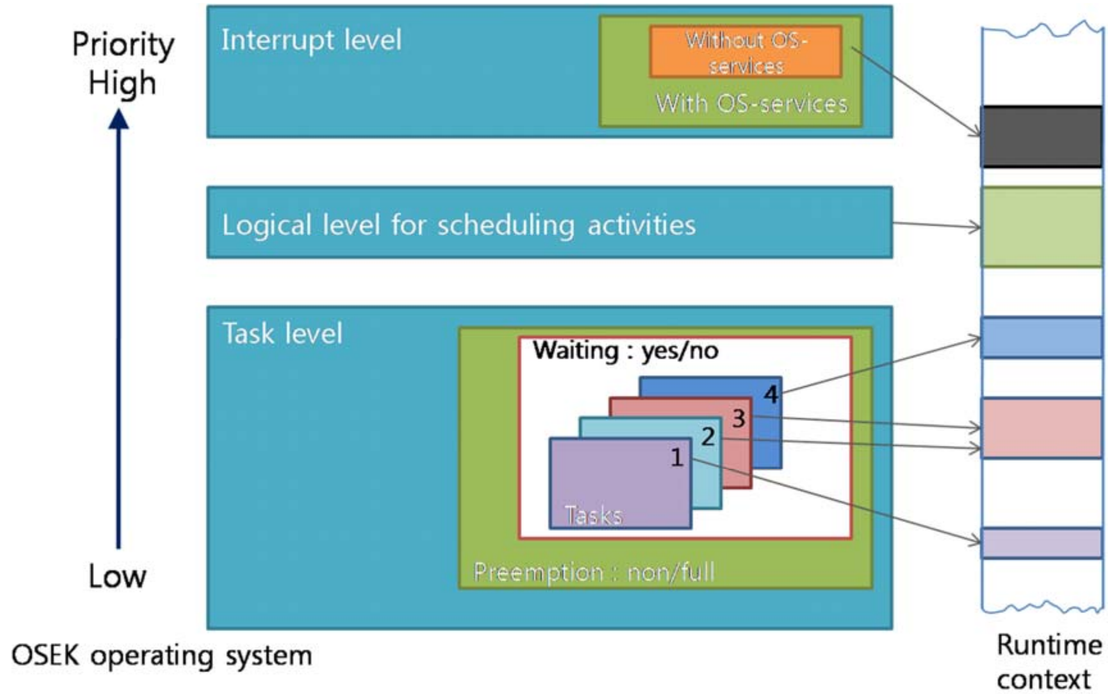


Fig3. Processing level of the OS

1.2 Priority Rules(in task level)

1. Interrupts have precedence over tasks
2. The interrupt processing level consists of one or more interrupt priority level
3. Interrupt service routines(ISR) have a statically assigned interrupt priority level.

1.2 Priority Rules(in task level) (cont)

4. Assignment of ISR to interrupt priority level is dependent on implementation and hardware architecture.
5. For task priorities and resource ceiling-priorities bigger numbers refer to higher priorities.
6. The task's priority is statically assigned by the user.

1.3 Conformance Classes

- Conformance class?
  - Can describe "Various required different features of the operating system".
- Purposes
  - To provide convenient groups of OS features
  - To allow partial implementations along predefined lines
  - To create an upgrade path from lesser functionality classes to higher functionality classes with no changes to the application using related features

1.3 Conformance Classes

- Conformance classes are determined by the following attributes
  - Multiple requesting of task activation
  - Task types
  - Number of tasks per priority

- Types of CC
  - BCC1(Basic CC)
  - BCC2
  - ECC1(Extended CC)
  - ECC2

1.3 Conformance Classes

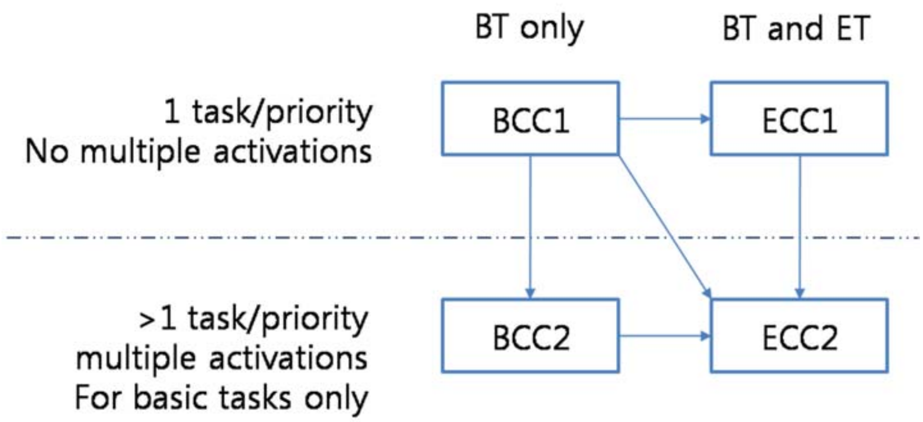


Fig 4. Restricted upward compatibility - upgrade path - for CC

1.3 Conformance Classes

	BCC1	BCC2	ECC1	ECC2
Multiple requesting of task activation	no	yes	BT[3]: no ET: no	BT: yes ET: no
Number of tasks which are not in the *suspended* state	8		16 (any combination of BT/ET)	
More than one task per priority	no	yes	no (both BT/ET)	yes (both BT/ET)
Number of events per task	—		8	
Number of task priorities	8		16	
Resources	RES_SCHEDULER	8 (including RES_SCHEDULER)		
Internal resources	2			
Alarm	1			
Application Mode	1			

2. Relationship between OSEK OS and OSEKtime OS

- OSEKtime OS is an operating system especially tailored to the needs of time triggered architectures.
- It allows OSEK OS to coexist with OSEKtime OS.
- OSEKtime assigns its idle time to be used by OSEK.
- OSEK OS interrupts and tasks have less importance (lower priority) than similar entities in OSEKtime OS.

- For more information, please refer to the specification of the OSEKtime OS.

2. Task management

2. Task management

1. Task Concept
2. Task State Model
3. Activating a task
4. Task priority
5. Scheduling policy
6. Termination of tasks

2.1 Task Concept

- Task
  - Is entity consisting of code and management information controlled by scheduler.
  - Provides the framework for the execution of the application.
  - Can be executed concurrently with other tasks.
  - Has a priority.
  - only release the processor if:
    - They terminate themselves.
    - The OS switches to a higher-priority task.
    - An interrupt occurs which causes the processor to switch to an ISR.

2.2 Task State Model

- Types
  - Basic Task(BT)
  - Extended Task(ET)
- Comparison of the task types
  - Basic tasks : No *waiting* state,
    - Thus only comprise synchronization points at the beginning and the end of the task.
    - Advantage - their moderate requirement regarding run time context (RAM).
  - Extended tasks : using *waiting* state
    - Advantage - can handle a coherent job in a single task, no matter which synchronization requests are active.
    - comprise more synchronization points than basic tasks.

2.2 Task State Model

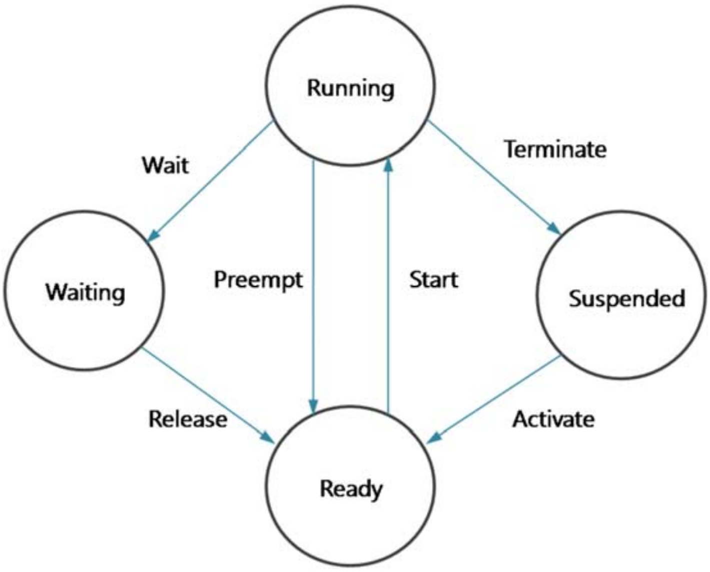


Fig 5. Extended Task state model

2.2 Task State Model

Transition	Former state	New state	Description
activate	*suspended*	*ready*	A new task is set into the *ready* state by a system service. The OSEK operating system ensures that the execution of the task will start with the first instruction.
start	*ready*	*running*	A *ready* task selected by the scheduler is executed.
wait	*running*	*waiting*	The transition into the waiting state is caused by a system service. To be able to continue operation, the *waiting* task requires an event.
release	*waiting*	*ready*	At least one event has occurred which a task has *waited* for.
preempt	*running*	*ready*	The scheduler decides to start another task. The *running* task is put into the *ready* state.
terminate	*running*	*suspended*	The *running* task causes its transition into the *suspended* state by a system service.

Table 2. States and status transitions for extended tasks

2.2 Task State Model

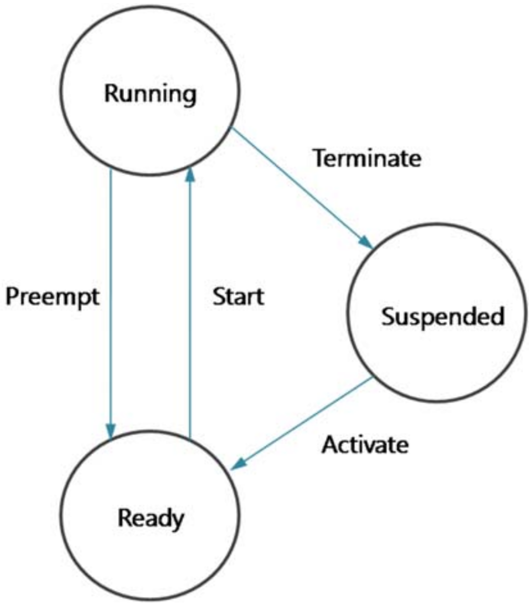


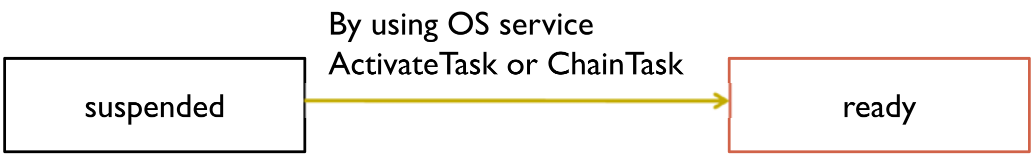
Fig 6. Basic tasks state model

2.2 Task State Model

Transition	Former state	New state	Description
activate	*suspended*	*ready*[4]	A new task is set into the *ready* state by a system service. The OSEK operating system ensures that the execution of the task will start with the first instruction.
start	*ready*	*running*	A *ready* task selected by the scheduler is executed.
preempt	*running*	*ready*	The scheduler decides to start another task. The *running* task is put into the *ready* state.
terminate	*running*	*suspended*	The *running* task causes its transition into the *suspended* state by a system service.

Table 3. States and status transitions for basic tasks

2.3 Activating a task



- After activation the task is ready to execute from the first statement.
- Basic task can be activated once or multiple times.
- Multiple requesting of task activation?
  - OS receives and records parallel activations of a basic task already activated.

2.4 Task priority

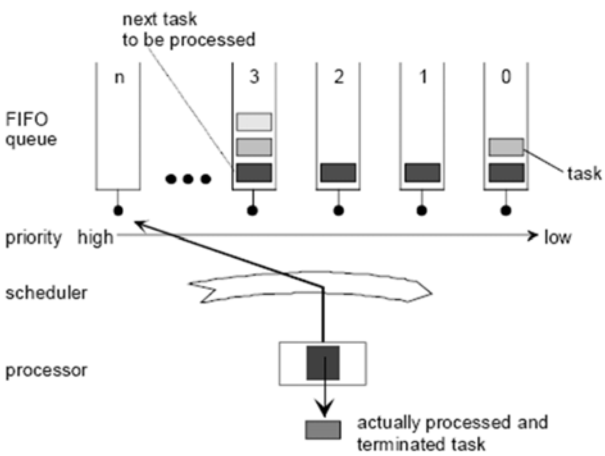


Fig 7. Operation of OSEK OS scheduler

- The fundamental steps to determine the next task to be processed
  1. The scheduler searches for all tasks in the *ready/running* state.
  2. From the set of tasks in the *ready/running* state, the scheduler determines the set of tasks with the highest priority.
  3. Within the set of tasks in the *ready/running* state and of highest priority, the scheduler finds the oldest task.

2.5 Scheduling policy

1) Full preemptive scheduling
2) Non-preemptive scheduling
3) Mixed preemptive scheduling

1) Full preemptive scheduling

- Full preemptive scheduling puts the *running* task into the *ready* state, as soon as a higher-priority task has gotten *ready.*

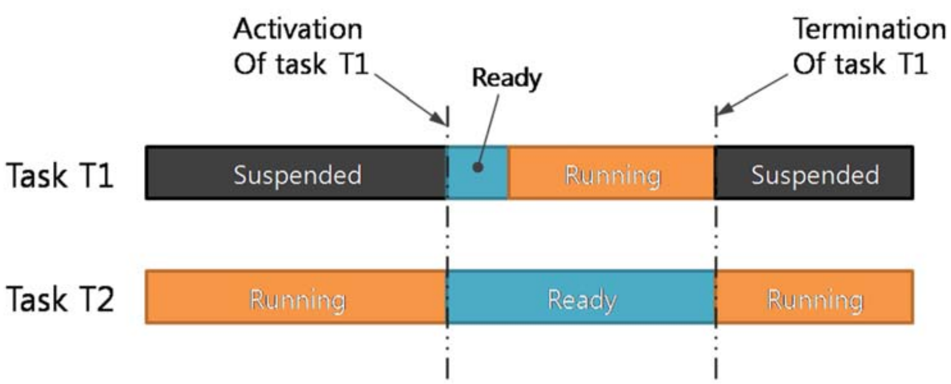


Fig 8. Full preemptive scheduling

- Rescheduling
  - Successful termination of a task
  - Successful termination of a task with explicit activating of a successor task
  - Activating a task at task level
  - Explicit wait call if a transition into the waiting state takes place
  - Setting an event to a waiting task at task level
  - Release of resource at task level
  - Return from interrupt level to task level
  - Cf: during ISR, no rescheduling.

2) Non-preemptive scheduling

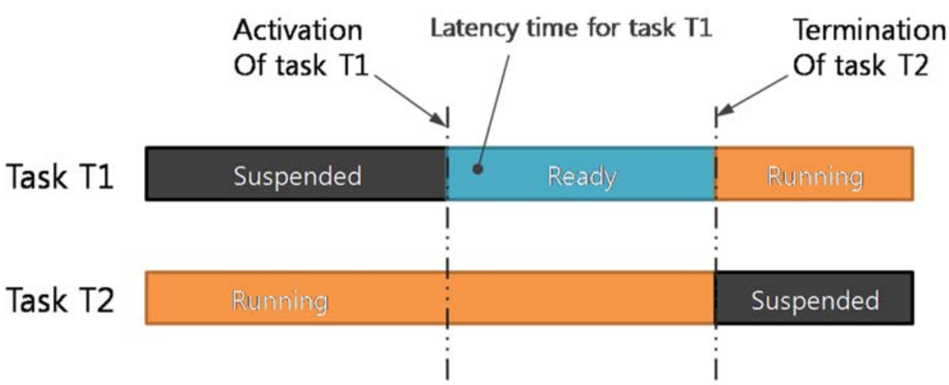


Fig 9. Non-preemptive scheduling

- Rescheduling
  - Successful termination of a task
  - Successful termination of a task with explicit activating of a successor task
  - Explicit call of scheduler
  - A transition into the waiting state takes place

3) Mixed preemptive scheduling

- When preemptable and non preemptable tasks are mixed on the same system
- Scheduling policy depends on the preemption properties of the **running task.**
- The definition of a non-preemptable task makes sense in a full preemptive OS:
  - If the execution time of the task is in the same magnitude of the time of a task switch
  - If RAM is to be used economically to provide space for saving the task context
  - If the task is not to be preempted.

2.6 Termination of tasks

- A task can only terminate itself ("self-termination").
- Each task shall terminate itself at the end of its code.