ESC-320: Software Development for Transportation Systems

Colin Walls, Mentor Graphics

Part 2 – An Introduction to OSEK/VDX

Introduction

OSEK/VDX is a standard – or rather a set of standards – which emerged from the European automotive industry. First there was OSEK [an acronym in German which translates to *Open Systems and the Corresponding Interfaces for Automotive Electronics*], developed by the German companies. Then there was VDX [*Vehicle Distributed eXecutive*], produced by their French counterparts. These two standards were combined in the mid-1990s to form OSEK/VDX, which is gaining very wide acceptance around the world. It is normally referred to as just *OSEK*.

OSEK consists of three main standards, which cover the operating system [*OS*], inter-process and inter-processor communication [*COM*] and network management [*NM*]. These will all be outlined in this paper, but a detailed study of the APIs is not included.

OSEK applications

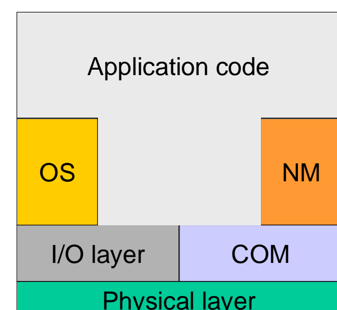
Although automotive companies developed OSEK for automotive applications, there is nothing in the standard that limits its usefulness to this context. Almost any application that requires a compact, predictable RTOS could utilize an OSEK system. Obviously other transportation applications may be considered, but the needs of other safety critical systems are readily addressed. The key characteristics of a system, for which OSEK may be useful, are:

- real time
- deeply embedded
- safety critical
- distributed
- no requirement for dynamic object creation

A good example would be medical electronics, where all of these characteristics are common.

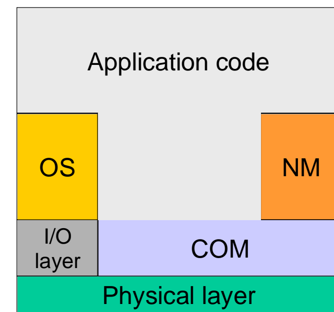
OSEK configuration

The OSEK standards may be applied in very flexible ways. All the three core components are options – even the OS. This results in a variety of possible configurations. Since there is no standard for the input/output API, this leads to some interesting possibilities.



Access to I/O may be direct, which is fast, but somewhat non-portable.

Alternatively, the I/O system may be treated as a task and be communicated with via COM. This will have lower performance, but be more portable and flexible.



OSEK OS

The OS is seen as the core component of OSEK. In this paper, the focus is upon the features that make OSEK OS different from other RTOS products and standards.

OSEK tasks

Tasks, like other OSEK objects, are static – they are created at build time. There is no facility for the creation and destruction of tasks during execution. This is an attractive characteristic for safety critical applications, where dynamic operations are often troublesome, as resource control may be impossible.

There are two types of OSEK tasks:

> **Basic** tasks always run to completion and cannot "pause" to await an event. All the basic tasks in a system may share a common stack.
>
> **Extended** tasks can wait on an event, but each must have its own stack space.

Tasks may be preemptive [i.e. they can be preempted by another task of higher priority] or they can be non-preemptive.

Conformance classes

The configuration of an OSEK system is determined by a set of four *Conformance Classes*. These provide bounds to the characteristics of tasks, which, in turn, constrain the complexity of the system. Each class is a superset of the "lower" ones:

> **BCC1** – Only Basic tasks, with just one task per priority level. Each task may only be "activated" once – i.e. only a single instantiation of each task is permitted.
>
> **BCC2** – Only Basic tasks, but with multiple tasks per priority level. Multiple activations [instantiations] are permitted.
>
> **ECC1** – Extended tasks supported, with just one task per priority level. Each task may only be activated once.
>
> **ECC2** – Extended tasks, with multiple tasks per priority level and multiple activations.

System configuration

Since an OSEK system is configured at build time, most vendors provide source code, which lends itself most readily to such configuration. However, in order to protect their intellectual property, some

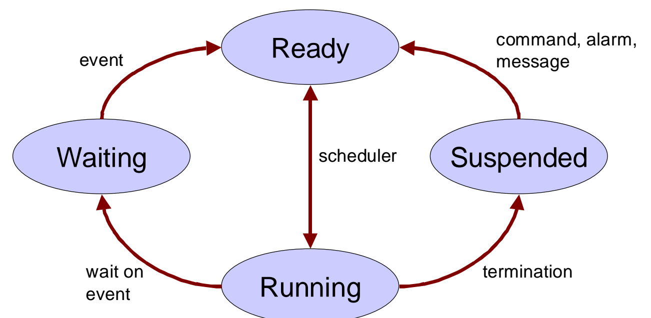
vendors “scramble” the source code [i.e. intentionally render it unreadable]. This limits its use by the developer to just system configuration.

Task priority

As with most real time operating systems, OSEK tasks have a priority. According to the OSEK specification, the higher the number, the higher the priority. The lowest priority is 0; the highest is implementation dependent [but 255 is common]. This is the reverse of some RTOSes, where 0 is highest. A task’s priority is defined at build time and cannot be adjusted at run time.

Task states

An OSEK task may be in one of four states:



> **Running** – actually executing at the moment
>
> **Ready** – available to run, when the scheduler determines that it is the best candidate
>
> **Waiting** – waiting on some event to occur
>
> **Suspended** – terminated, not available to run

The state transition diagram for OSEK is, compared with some RTOSes, simple and symmetric.

The terminology requires some care. The states are clearly defined, but their names have different meanings in some other RTOSes. For example, the equivalent of OSEK’s *Waiting* is often called *Suspended*.

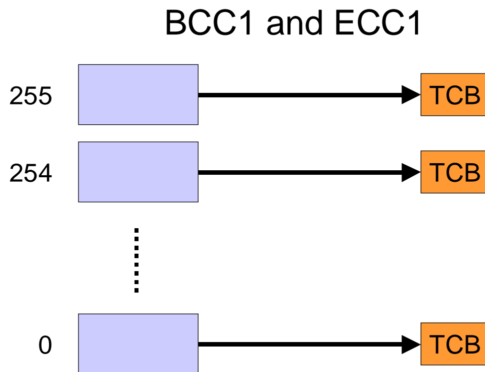
Task scheduling

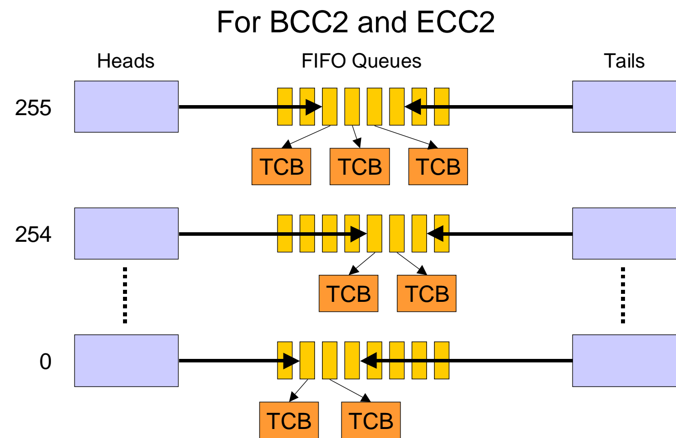
The scheduler is invoked as a result of various events, depending upon whether a preemptive or non-preemptive task is running:

> **Non-preemptive** – calls to: `ChainTask()`, `TerminateTask()`, `WaitEvent()`, `Schedule()`
>
> **Preemptive** – calls to: `ChainTask()`, `TerminateTask()`, `WaitEvent()`, `ActivateTask()`, `SetEvent()`; message arrives; alarm expires

The scheduler simply reviews which task in the Ready queue has the highest priority and moves it to the running state.

The data structures [and hence code] of the scheduler for a system conforming to BCC1 or ECC1 is considerably simpler than a system conforming to BCC2 or ECC2. This is the result of the number of possible tasks at each priority level [just one for BCC1 or ECC1].





Interrupts

An OSEK system supports 3 types of interrupt:

> **Category 1** – These are quite independent of the OS. Hence they are fast, but cannot make use of OS services.
>
> **Category 2** – These are integrated into the OS and can make full use of OS facilities, but have somewhat limited performance.
>
> **Category 3** – These are a hybrid. Most of the code has no access to OS services, but a section may be defined where such facilities are available. This is enclosed by calls to **`EnterISR()`** and **`LeaveISR()`**.

Category 3 interrupts are interesting for many applications. Here is an example:

```
interrupt void inputchar()
{
	static char buffer[100];
	static int index=0;
	char c;

	c = getcharacter();
	if (c == EOM)
	{
		EnterISR();
		SetEvent(InputProcess, MSG);
		LeaveISR();
	}
	else
	{
		buffer[index++] = c;
	}
}
```

In this case, the ISR is called for each character, as it arrives. Normally, they are just buffered [and in real code there would be overflow checking!], which is very fast. When the message is complete, an event is asserted to advise the appropriate task to process the data.

Synchronization – events

Events are a key feature of an OSEK system for the synchronization of tasks with one another and external activity. An event is owned by a task [specifically an *extended* task]. An event may be set [asserted] by any task, but only the owner can wait on or clear an event.

Synchronization – counters and alarms

Counters and alarms are designed to synchronize recurring events. They can avoid the unnecessary use of specific tasks for such purposes.

A counter is driven by a *tick*, which can represent a time interval or any other occurrence. An OSEK system, by definition, always has at least a timer counter. The counter API is implementation specific, as particular hardware may be available.

An alarm is an aggregate object, consisting of a counter and an action [which is, in effect, association with an event/task]. On completion, an alarm may activate a task or set an event. Alarms may be cyclic, to enable a periodic activity to be scheduled, or single shot, to implement timeouts etc.

Priority inversion

A common problem, with all real time systems, is the situation where a task locks a resource and effectively blocks a task of higher priority from running. This is termed *priority inversion*. There are various strategies available to address this difficulty. The OSEK specification defines a *priority ceiling protocol*. The task locking the resource has its priority temporarily raised to the level of the highest other task that is in contention for the resource. Thus, blocking of the higher priority task(s) is minimized.

Errors and debug

Since the focus of an OSEK system is speed and compactness, the specification makes minimal demands with respect to runtime error handling. This may be provided with a specific implementation. A common approach is to use macros to answer the OSEK API. When a debug flag is set, the macros include extra code for error checking; when the flag is clear, "production" code, with no error checking, is generated.

An OSEK system incorporates a number of "hooks". These are opportunities for the OS to call a user-defined function, under specific circumstances. If no function is defined, no call takes place. There are six hook routines:

> **`StartupHook()`** – This function is called before the scheduler is started. It is a useful place to set up alarms, activate tasks and send messages to other networked devices. In general, the application may be placed into a known state.
>
> **`PreTaskHook()`** and **`PostTaskHook()`** – These functions are called just before a task is scheduled and just before it is "descheduled", respectively. They offer the opportunity for tracing and debug. They are really of more use to OS developers and vendors than end users.

`ErrorHook()` – This function is called as a result of an application error, which can be detected by the OS; e.g. attempt to activate a non-existent task. Such errors may be recoverable – i.e. there is no damage to the integrity of the OS – but that recovery is application dependent.

`ShutDownHook()` – An OSEK system will attempt to call this function if a fatal error occurs. Typically, this will be the result of a lack of integrity being detected in the OS's internal data structures. The only likely action is a system reset, but there is the opportunity for a "graceful" shutdown of external systems.

OSEK COM

COM provides a means for application tasks to communicate via messages. This communication may be "inter-process" [the tasks are running on the same processor] or "inter-processor" [communication is between networked devices].

No assumption about the network topology or technology is assumed in the design of COM and the applications programmer does not need to know. Typically it would be a CAN or J1850 bus, but could be Ethernet or any other networking medium.

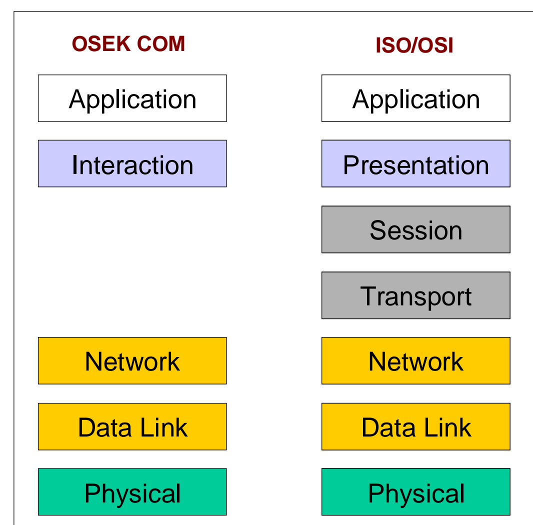
COM layers

Like most communication protocols, COM may be modeled as a series of layers. There are five layers, which map quite well onto the ISO/OSI seven-layer model.

Application layer – This layer is the application code itself.

Interaction layer – This layer provides the API, which includes functions for send, receive, status check and resource lock/release. This layer handles all on-chip [inter-process] communication. All inter-process communication is passed on to the Network layer.

Network layer – This layer receives only inter-processor traffic. It handles any necessary segmentation of messages – division of a large message in to two or more smaller ones. Messages are passed on to the appropriate Data Link layer for transmission.



Data Link layer – This layer handles the network protocol – e.g. CAN. There may be several Data Link layers in a complex system, each handling a different protocol.

COM operation

COM functions asynchronously in parallel with the application code. Success or failure may be indicated in a number of ways: task activation, event setting, and alarm expiry.

Messages are defined as specific objects in an OSEK system. They are quite separate from tasks, but have a defined association [see Sending/Receiving below]. The design is intended to reduce the requirement to define utility tasks to process messages.

COM is actually independent of the underlying OS. In principle, at least, the OS need not even be OSEK-compliant.

Message characteristics

COM messages can have a wide range of characteristics. A system may be constrained by one of the four conformance classes:

	CCC0	CCC1	CCC2	CCC3
Direct transmission	X	X	X	X
Periodic transmission		X	X	X
Mixed direct/periodic		X	X	X
Message deadline monitoring		X	X	X
Unqueued messages	X	X	X	X
Queued messages				X
Unsegmented protocol	X	X	X	X
Segmented protocol			X	X
Notification by task activation		X	X	X
Notification by event setting		X	X	X
Static address/size	X	X	X	X
Dynamic address/size			X	X

Transmission mode – There are three available modes: Direct, Periodic and Mixed. Direct Mode is when a message transmission is simply initiated by a task. Periodic Mode is when a message is sent automatically at a predetermined frequency. Mixed Mode is when a message is sent periodically, but can also be sent on an “update” [when a variable changes and the change meets specified criteria]; the periodic timer is not reset on an update transmission.

Message deadline monitoring – This is an optional facility. An alarm is started on transmission and cancelled on confirmation. Periodic messages may also be monitored.

Queued/unqueued messages – If messages are unqueued, there is a single object in COM for each message. Queued messages are held in a FIFO buffer.

Segmented/unsegmented protocol – Messages may or may not be divided into multiple frames, as required by the underlying communications link. This is handled by the Network layer and is only relevant to inter-processor communications.

Static/dynamic address/size – Message size may be fixed [in which case it may still be segmented] or variable. Similarly the address may be determined at build or run time.

Sending/receiving

Since each message is an independent entity, they are defined individually. The form of this definition is not covered by the OSEK standard, but typical parameters include:

- message size
- queued/unqueued
- network/local
- transmission mode
- static/dynamic size/address
- monitoring alarm

In addition, the message usage for each task must be defined:

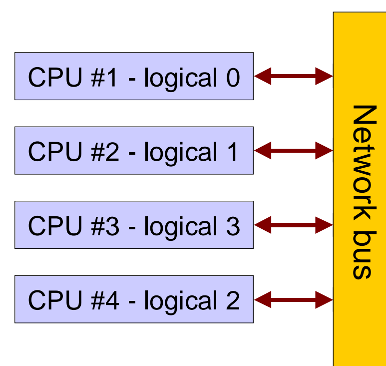
- **send/receive** – There can by only one sending task, but many receivers
- **copy or not** – A copy or zero-copy mechanism may be selected
- **task action** – For each task there is a need to select: task activation, event setting, alarm set/clear

OSEK NM

Since it is very likely that OSEK systems will be deployed in networked environments [like a car], it is useful that some network management facilities are included in the standard. The facilities are basic - primarily aimed at giving each node of a system a "picture" of the network as a whole.

There are two distinct methods available: Direct and Indirect Network Management.

Direct network management



Each node is assigned a logical number. These numbers need not be sequential. There is a logical ring structure, where the next node is the next higher value, going back to the start after the highest value.

The configuration state of other nodes in the network is determined by a series of "Ring" and "Alive" messages. A node's own state is determined by its ability to send messages. There are two possible states: *present* or *absent*.

The structure of messages is quite simple - just three fields:

Source – The logical address of the sending node

Destination – The logical address of the receiving node

Op-code – There are 8 possibilities: 2 Alive, 4 Ring, and 2 Limp Home. The encoding is not specified, as it would depend upon many factors [like the underlying network]. For CAN, this may be a 29-bit header of 11-bit header with a data field.

The normal operation is for Ring messages to be sent around the network from one logical node to the next.

When a node comes online, it will send an Alive message around the network in the same way, thus registering its existence. Thereafter it will participate in the Ring message circulation.

The concept of *Bus Sleep* is defined in order to shut a system down to conserve power. Any node can issue a sleep request, via a Ring message. All nodes must acknowledge for the request to be successful.

Indirect network management

This is a somewhat simpler scheme, which is adequate for many applications.

There are no logical node numbers, just a fixed physical address for each node.

The normal operation utilizes three monitoring mechanisms:

Transmission – expecting acknowledgement

Reception – using COM timeout

Status – use Data Link layer to check bus

When nodes come online, they initially assume that all other nodes on the network are also present. In due course, they determine which nodes, if any, are absent. When new nodes come online or go offline, the other nodes in the system notice their arrival/departure.

There is no specific Bus Sleep mechanism defined. Each node may enter sleep mode. A master/slave set-up is possible, where one node simply commands the others to enter sleep mode.

Other OSEK Standards

The OSEK standard also covers the system implementation/configuration language [OIL – OSEK Implementation Language] and debug interface [ORTI – OSEK Run Time Interface]. A detailed discussion of these is beyond the scope of this paper.

Conclusions

OSEK is a good, complete standard for a real time operating system and its surrounding components. It may be applicable in a wide range of contexts, not limited to automotive or other transportation applications. The availability of a clear standard gives users a high degree of vendor independence and inter-operability of products.

References

www.osek-vdx.org

Lemieux, Joseph – *Programming in the OSEK/VDX Environment*