What is OSEK/VDX and where did it come from?

A consortium of mostly European automotive companies created the OSEK standard in 1993. The consortium’s goal was to address the high recurring costs in developing and redeveloping Electronic Control Unit (ECU) software and to improve the compatibility of those applications. In 1994, OSEK merged with VDX, a similar initiative in the French automotive industry. The two standards bodies presented their harmonized OSEK/VDX standard in October 1995. Since then, OSEK/VDX has gained broader recognition in the industry, and a steering committee that includes Adam Opel, BMW, DaimlerChrysler, PSA, Renault, Robert Bosch, Siemens, and Volkswagen currently manages it.

It is important to note that OSEK/VDX is a standard, not a technology. For purposes of simplicity, OSEK/VDX will sometimes appear as OSEK throughout this article. OSEK/VDX includes the following specifications:

- **OSEK Operating System (OS)** defines a standard interface for a single-processor operating system and offers the necessary functionality to support event-driven control systems. The OSEK OS is the primary focus of this article.
- **OSEK Communication (COM)** defines a protocol for inter-task and inter-module communications among deeply embedded systems.
- **OSEK Network Manager (NM)** defines protocols for managing networks during runtime. The NM provides standardized features, which ensure the functionality of inter-networking by standardized interfaces.
- **OSEK Run-Time Interface (ORTI)** enhances interoperability and portability by defining a common interface for any microcontroller platform and any OSEK vendor. ORTI allows different development tool vendors to debug various OSEK implementations. In addition, an ASCII interface for the ORTI file makes extensions easy and manageable.
- **OSEK Implementation Language (OIL)** is the configuration language that allows embedded designers to describe the complete OSEK system for system configuration and generation. OIL creates readable, archivable ASCII text files and is the key to enabling interoperability between different OSEK OS vendors.
- **Time Triggered OS (OSEKtime)** is an emerging standard that extends the OSEK OS specification to allow systems designers to sequence applications and communications (via FTCom) on local and remote controllers as if they were running on the same controller.
- **Fault Tolerant Communication (FTCom)** usually represents part of an OSEKtime implementation and defines a standard interface for fault tolerant communication.

All these specifications tend to move forward independently of one another. The consortium and participants in the working groups...
are often common among the technical committees. However, the timing of the releases of each different specification does not necessarily synchronize with the other specifications. To sustain a high-level of interoperability between these separate and continually changing parts, the standards body introduced the concept of a binding level. These binding levels correlate the various independent releases of the specifications into a group, which uses a common base. In this way, a developer can review the specifications of the different OSEK technologies and select the single specification to match their needs. If multiple technologies are required, they can peruse the various documents and determine the binding level that most closely matches the requirements of their end system.

Furthermore, any vendor who uses the OSEK/VDX trademark must meet OSEK certification requirements set by the standards body. The certification process ensures that different implementations conform to the specification. This is helpful to those purchasing OSEK/VDX technology because it allows the buyer to select from only those vendors whose products are portable and interoperable with other OSEK tools.

Why is OSEK/VDX important to the embedded market?
Portability and reliability are perhaps the biggest advantages the OSEK standard offers. With OSEK, designers write applications to a standard application protocol interface (API) and not to a unique operating system, so it’s easier to switch from one processor to another. In addition, a static kernel is the basis of the OSEK operating system, rather than a dynamic one, which means an application contains only the specified capabilities and requirements. This configuration greatly reduces the amount of testing needed to ensure the optimum performance of the application.

“To sustain a high-level of interoperability between these separate and continually changing parts, the standards body introduced the concept of a binding level.”

Integration is another key benefit of the OSEK/VDX specification. OSEK enables tools from a variety of industries to interoperate, in part, through the OIL specification. OIL creates a system template of which any tool capable of processing an OIL file can read. For instance, an application development tool can read an OIL file generated by a software-modeling tool. The application development tool then produces an OSEK implementation designed for the targeted microprocessor. In this case, OSEK enables a high-level tool to interact with a low-level capability, which demonstrates how integration between disparate tools increases the value of both technologies. The result is a greater choice of products for customers, cost savings for vendors, and accelerated time to market.

A range of industries can leverage the OSEK/VDX OS. For example, the storage application and home appliance markets often have strict requirements for high performance and low memory footprint systems. The OSEK specification is ideal for developing applications that require operating systems with a footprint below 5 Kbytes. In some OSEK OS implementations, it is possible to fit an OS into less than 700 bytes of code. Due to OSEK’s wide application and strong worldwide adoption, it has attracted the interest of technology leaders throughout the embedded systems community. More than 50 of these companies, including Metrowerks, a Motorola company, have joined the OSEK technical committee to ensure OSEK/VDX continues to evolve in a positive way.

However, since the automotive industry created OSEK/VDX for its own market, it has perhaps made the most use of the specification. Today, for example, designers commonly use it for controls such as body application, air-bag deployment, chassis integration, and power-train systems. In fact, many automotive OEMs require an OSEK OS as a mandatory component for controller software that they receive from a supplier. In the future, as deeply embedded applications become ubiquitous, the specification might evolve to take advantage of Memory Management Units (MMUs). As higher end microcontrollers decrease in price and become more widely used, the OSEK standards committee or a group of vendors should consider how to apply the specification to the memory management market.
As a widely supported standard, OSEK has significantly reduced the time and costs associated with integrating ECU applications from different Tier 1 suppliers. Now manufacturers can choose from several vendors writing software to the same standard. The results have been higher quality vehicle controls, reduced costs due to integrated and reusable software, and increased innovation throughout the industry. In addition, as environmental regulations increase the cost and complexity of power-train systems, OSEK/VDX technology helps extend the life of existing systems and reduces the risk of migrating to new technology. Above all, improved vehicle reliability and reduced development costs – two critical needs for auto manufacturers – have been the biggest business and technical advantages for the industry.

**What's under the hood?**

The OS is the first component of the OSEK/VDX standard and defines an operating system API and a set of services. The OS is statically defined, which is especially important to industries where safety and reliability are critical. A static implementation means that a designer can include only the features and capabilities they want in order to contain the implementation. This reduces complexity and unknown factors and increases the ability to adequately test the system. By contrast, a dynamic specification is more open and flexible, and therefore subject to more risks.

Overall, the OSEK OS kernel specification enables high-performance, portable, and scalable applications that use minimal resources and can adhere to stringent real-time requirements. Figure 1 illustrates the relationships between the objects that make up the operating system.

---

**Tasks**

There are two categories that describe tasks: basic and extended. The main differences between the two types of tasks are the number of states they support in their state machines and their ability to support event services. In the OS specification, there are four defined states in the state machine:

- Ready
- Running
- Suspended
- Waiting

Basic tasks do not support a transition to the waiting state. In addition, they do not support event services for transitioning to this state. The developer, therefore, chooses between basic and extended tasks depending on the necessity of the waiting state.

The four “conformance classes” defined by the specification allow scaling of the OS to the application demands while still enabling code reuse. These conformance classes are basic, BCC1 and BCC2, and extended, ECC1 and ECC2. The primary difference between classes 1 and 2 is the number of tasks they can activate at one time. BCC1 allows only basic tasks and is limited to one activation request per task and one task per priority. BCC2 allows multiple requests for task activation and more than one task per priority. The same is true for ECC1 and ECC2, but they include extended tasks, as described above. Although basic tasks may sound less sophisticated, many embedded application developers find that they can produce complex applications with tighter code using only the BCC1 conformance class.

Each conformance class includes either a standard status or extended status attribute for error checking. In standard status, the OS performs only a few system tests. Whereas, in extended status, the OS has more comprehensive reporting requirements. Therefore, it is better able to reveal development errors. Due to the overhead in the operating system and increased memory usage when using extended status, the OSEK/VDX standards group recommends using it only in the design and debug phase of development rather than in the final product because it puts less demand on RAM and processor resources. It also results in a more finely tuned system. By finding and correcting application errors early in the development process and then transitioning to standard status checking, developers can reduce the total product development costs and limit the memory and performance overhead in the final product.

Tasks can also be preemptive or non-preemptive. The OS allows an application to comprise a number of independently scheduled tasks, and it can schedule them through fixed priority preemptive scheduling. This means the OS will interrupt one task in favor of a higher priority task. The OS will resume the original tasks once the processor has completed all the high-priority tasks. The non-preemptive option allows a task to run its course without interruption from other tasks. This option helps protect access to shared data. The system can also be a combination of preemptive and non-preemptive tasks, which permits a mixed-mode scheduler. The developer can assign this combination anywhere in the system.

“Although basic tasks may sound less sophisticated, many embedded application developers find that they can produce complex applications with tighter code using only the BCC1 conformance class.”

---

Reprinted from *Embedded Computing Design* / Winter 2003
Interrupts
The OS defines two categories of interrupt service routines (ISRs). There is one main difference between them: category 1 interrupts do not include OS system calls, and category 2 does include them. The hardware schedules the interrupts and it can interrupt both preemtable and non-preemtable tasks. If an interrupt routine activates a task, then the OS schedules the task after the end of all active interrupt routines. The latest release of the OSEK OS specification has simplified ISR processing, but there are challenges regarding rescheduling of tasks after termination of a nested set of ISRs. Designers should investigate these issues early in the requirement process to ensure they meet product-timing requirements.

Events
Events are owned by and linked to extended tasks. Events are used to synchronize different tasks and for signaling between tasks. Any task, including basic tasks, can set an event. However, only the owner task can either clear the event or wait for the event.

Resource management
Resource management controls access to shared resources such as memory and hardware. Within this specification, OSEK enables a priority ceiling protocol to prevent two tasks or ISRs from occupying the same resource at the same time. The priority ceiling protocol also addresses the problems of priority inversion and deadlock. Priority inversion occurs when a lower-priority task delays the execution of a higher-priority task. When the system is deadlocked, it means the task cannot execute because it is waiting interminably for mutually locked resources.

“If an interrupt routine activates a task, then the OS schedules the task after the end of all active interrupt routines.”

Alarms and counters
The OSEK OS also defines alarms and counters, which synchronize task activation with recurring events. This feature supports portability and code reuse by allowing separation between the timing and functional behavior of a task.

The OS can define alarms as either single alarms or cyclic alarms. It also provides services to cancel alarms and to get the current state of an alarm. The OS statically assigns an alarm to one counter, one task, and one action. The action could be to either activate a task or set an event. Counters are measured in ticks and can represent time or number of pulses received, for example. Each implementation provides at least one timer counter and can schedule periodic events as well. An API specific manipulates other counters to each implementation of the OSEK OS. The standards group included the counter API in the 1.0 version of the specification, but they removed it from the 2.2.1 version by request, due to the complexity of timer interfaces among various microcontrollers.

Error handling and hook routines
OSEK/VDX provides minimal runtime error handling because in a static system, runtime errors are rare. OSEK provides enough error handling via extended return functionality to debug applications in the initial development stages. Then, it can turn off error handling to eliminate the extra code after release of the product.

Hook routines are part of the operating system and the OS can use them as diagnostics during system startup or system shutdown, as in the case of a severe error. The OS uses the hooks to build testing algorithms for application-dependent debugging purposes and, therefore, they have a higher priority than all tasks. The OS will call one hook routine on an error occurrence. However, the specification does not allow hook routines to call most operating system services. This restriction is necessary to reduce system complexity.
What should a designer look for in OSEK/VDX technology?

How does a designer choose the right tools? When the time comes to select an OSEK/VDX vendor, the vendor’s product should be OSEK-certified. This requirement ensures that the product has met the rigorous requirements of the OSEK standards body. In addition, the designer will be able to fully leverage the key advantages of OSEK/VDX: high performance, small memory footprint, and reliability.

Next, look for a vendor that works across several platforms, e.g., 8, 16, and 32-bit microcontrollers. The vendor not only should support integration between industry-leading tools and technologies, but also should have a Capability Maturity Model (CMM) rating of Level 3 or above. This rating indicates that the vendor employs a manageable and repeatable software development process resulting in end products with few defects. This is important for those working in an industry where safety and reliability are critical.

Also, look for a flexible business model to meet the needs for low, medium, and high-volume applications. When comparing vendors, it’s important to have elasticity in this space if your embedded applications require a small footprint and high reliability. Finally, the vendor’s technology should support the ORTI specification and integrate with the compiler for easier system debugging.

Metrowerks, for example, offers a robust, OSEK-certified OS that they developed in accordance with the Software Engineering Institute’s (SEI) highest CMM rating. In addition, Metrowerks offers deterministic scheduling for its OSEK/VDX implementation. Deterministic scheduling employs repeatable, mathematical techniques to evaluate and determine the worst-case system load, which dramatically improves the end product. Designers use these proven mathematical tools in a user-friendly graphical format to accelerate design and test time using worst-case and what-if analysis. When a developer designs and tests systems using deterministic scheduling, they can tune the processor and memory configurations to provide maximum capabilities with minimized costs. Figure 2 provides a checklist of features to look for in development tools.

Where to go from here

To learn more about the OSEK specification and how to fully leverage its capabilities, refer to the following resources:

- **www.osek-VDX.org**
  This is the primary Web site for the OSEK/VDX specification and provides details on all the specifications.

- **www.CodeWarriorU.com**
  To learn how to develop embedded applications, CodeWarriorU provides Web-based training on OSEK/VDX application development.

- **Programming in the OSEK/VDX Environment, by Joseph Lemieux, 2001.**
  This book explains how to develop OSEK/VDX applications and addresses the three main components of the standard: the OS, COM, and NM. Readers can learn about the standard in its entirety or refer only to those sections that cover the specifications they’re implementing. A fourth section details input/output programming, of which the specification does not explicitly cover.

Michael O’Donnell currently serves as director of transportation and standards product marketing at Metrowerks. With more than 10 years of experience in the embedded market, Michael has developed embedded software for a variety of automotive applications, including transmission controllers, while with General Motors’ Powertrain Division. Michael earned a bachelor’s degree in electrical engineering from The University of Texas at Austin.

Metrowerks Corporation creates CodeWarrior software and hardware products and services for developers with a particular focus on automotive, consumer electronics, wireless, networking, and communications industries. As a leading OSEK/VDX technology provider, Metrowerks supports 8, 16, and 32-bit microcontrollers and offers an OSEK-certified operating system that’s been developed in accordance with the Software Engineering Institute’s highest CMM rating, level 5.

Founded in 1985, Metrowerks is an independently operating subsidiary of Motorola, Inc. For more information, contact Michael or the company directly.