Dr. David Kalinsky,
Director of Customer Education,
Integrated Systems, Inc.

A Modern Standard for Super-Small Kernels

OSEK/VDX is the specification for a real-time operating system originally intended for automotive control processors, but much more generally applicable to deeply embedded processors in a wide spectrum of applications. The OSEK/VDX standard includes software interfaces to the operating system, a tasking model, and mechanisms for intertask communication and synchronization.

INTRODUCTION

This standard was originally developed as a joint project of European automotive industries. OSEK is a German acronym for "Offene Systeme und deren Schnittstellen fur die Elektronik im Kraftfahrzeug" (Open systems and the corresponding interfaces for automotive electronics). The French acronym VDX stands for "Vehicle Distributed Executive".

In this paper the focus will be on the kernel of OSEK/VDX, which is actually not at all specific to the automotive industry. (Some other parts of OSEK/VDX, such as interprocessor communication and network management, might be less applicable outside of automotive applications). The kernel is the heart of the operating system which provides basic services such as task scheduling and intertask communication which are needed in all sorts of embedded applications. In OSEK/VDX the kernel is quite small and tailorable in size to the minimal needs of each specific application.

THE OSEK/VDX TASKING MODEL

OSEK/VDX offers application developers two kinds of concurrent tasks, called 'Basic Tasks' and 'Extended Tasks'.

Basic tasks are much more rudimentary than the tasks to which we have become accustomed in top-of-the-line real-time operating systems (such as pSOS+, VxWorks, VRTX, etc.). A Basic task normally executes its code straight through from beginning to end. It can stop running only if terminated, preempted (by a higher priority task) or interrupted (by an interrupt service routine). Probably the most important fact about a Basic task is that it has no "Waiting" state (sometimes called a "Blocked" state). In other words, a Basic task can not be programmed to stop in mid-execution and wait for something to happen. Basic tasks impose very modest resource demands on an operating system.

Extended tasks in OSEK/VDX are more similar to tasks as we know them in other real-time operating systems. An Extended task does not have to execute its code straight through from beginning to end. It can stop and wait in mid-execution for something to happen. And when the happening occurs, the Extended task can resume execution from where it left off. Extended tasks have a "Waiting" state, and so their management by an operating system is more complex and their resource requirements are greater.

Even with Extended tasks, OSEK/VDX operating systems stay small and simple by avoiding some of the sophisticated features of other operating systems that might be "overkill" for many deeply embedded applications:

- Tasks can't be dynamically created or deleted in mid-application.
- Parameters can't be passed to a task when it starts to execute.
- Priorities of tasks can not be changed in mid-application.
- Preemptibility can not be turned on or off in mid-application.

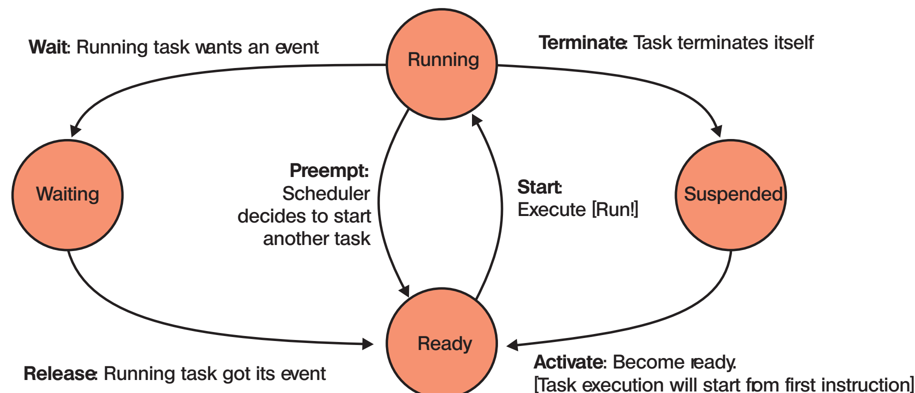

Figure 1. Extended Task State Model.

Copyright 1999 by Real-Time Magazine 99-4 (http://www.realtime-info.com)

- Round-robin time slicing is not available.
- Tasks can not terminate other tasks.

OSEK/VDX tasks can be run using either Preemptive scheduling, or Non-Preemptive scheduling. With Preemptive scheduling (called 'Full-Preemptive' in the OSEK/VDX specification), a task can be preempted (its execution stopped) at any instruction, if pre-set conditions occur. With Non-Preemptive scheduling, task execution can stop only if the task calls for an operating system service. A middle ground is a third option called 'Mixed-Preemptive' scheduling, in which some tasks are preemptible and others are not. As previously mentioned, the preemptibility status of a task can not be changed 'on the fly'.

CONFORMANCE CLASSES

The OSEK/VDX standard has introduced the idea of 'Conformance Classes' to allow an application developer to make the operating system's task scheduler simpler or more complex, depending on the needs of the application and the capacity constraints of the embedded system.

A developer needs to ask him/herself the following questions:

- Can I make do with only 1 task per priority?
- Can I make do with only Basic tasks?
- Or do I absolutely need some Extended tasks?
- Can I make do with only one Activation of each of my Basic tasks?
- Or do I absolutely need multiple parallel activations of some Basic tasks?

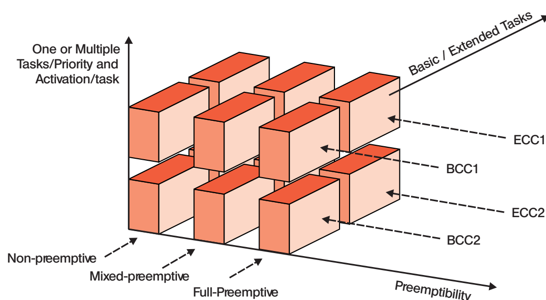

Figure 2. Task Scheduler Choices.

Depending on the answers to these questions, a Conformance Class can be chosen, from among 4 available conformance classes: If only basic tasks are needed, with only one task per priority and one activation per task, then Basic Conformance Class #1 (BCC1) should be chosen since its capacity requirements are minimal. If you also need some extended tasks, then you need to go up to Extended Conformance Class #1 (ECC1). If you don't need extended tasks but you do need multiple tasks at a given priority and/or multiple parallel activations of the same task, then you need Basic Conformance Class #2 (BCC2). And if you need everything available, then you need Extended Conformance Class #2 (ECC2). Of course, as you go up in conformance class, the ROM and RAM needs of the operating system kernel increase as do its capabilities.

For all conformance classes, tasks can be preemptively scheduled, Non-Preemptive, or Mixed-Preemptive. So in fact OSEK/VDX offers 12 different kinds of task scheduling and management, as shown below:

EVENTS

Events are the primary mechanism for synchronization between tasks. Events can also be used to synchronize between an Interrupt Service Routine (ISR) and a task. In OSEK/VDX, events are local to an individual task, not global. These events are said to be 'owned' by the task.

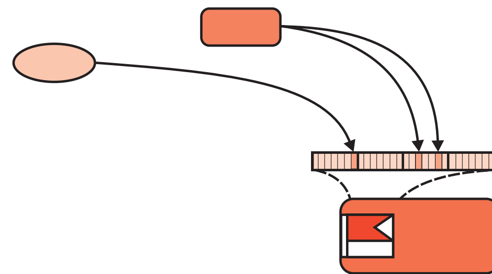

Figure 3. Working with Events.

Only Extended tasks can 'own' events. These tasks can wait for specific events, and the arrival of an event can take such a task from the Waiting state to the Ready state and (eventually?) into the Running state.

Any ISR or task, not necessarily the 'owner', can set an event (or events) for the 'owner' task. But these non-owners can not clear these events or wait for these events.

Figure 3 shows a non-owner task and an ISR in the upper portion of the diagram, setting events in the event flag group which is owned by the task at the bottom of the diagram.

RESOURCES

Resources in OSEK/VDX are a mechanism to regulate concurrent access to shared resources for which simultaneous access could cause problems. The Resource mechanism in OSEK/VDX is known as a "Priority Ceiling Mutex" in some other operating systems. It is a mechanism that replaces Binary Semaphores that would provide similar services in older operating systems.

OSEK/VDX Resources can be used to:

- Guard critical sections of software,
- Guard shared memory areas or data structures,
- Regulate access to shared hardware devices,
- Lock the OSEK/VDX task scheduler (to make the currently Running task non-preemptible on a temporary basis).

Figure 4 shows a typical usage.

Copyright 1999 by Real-Time Magazine 99-4 (http://www.realtime-info.com)

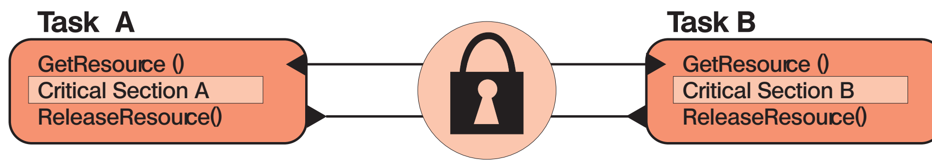

Figure 4. Working with a Resource.

Resources work by raising the priority of a task as soon as it asks to get a resource, and lowering the priority of the task back to its normal level when the task releases the resource. The temporary raised priority is known as the 'ceiling priority', and is higher than the priority of all tasks which can access the resource.

Resources or 'Priority Ceiling Mutexes' avoid many of the problems of traditional binary semaphores such as unbounded priority inversions and deadlocking.

ALARMS

Alarms in OSEK/VDX are a mechanism for handling repeating occurrences. I like to think of them as a generalization of the 'Timers' mechanism that is found in many other operating systems. The idea is that an Alarm can start a task running or set events, when it is triggered.

Alarms are a two-stage concept, as shown in Figure 5. In the first stage, occurrences are tallied by Counters. These Counters just count 'ticks'. If these ticks are coming from a hardware timer interrupt, the counter is just counting the time (in units of 'ticks'). If the ticks are coming from a medical ECG detector, the counter is counting heartbeats.

In the second stage, Alarms trigger tasks or set events for tasks.

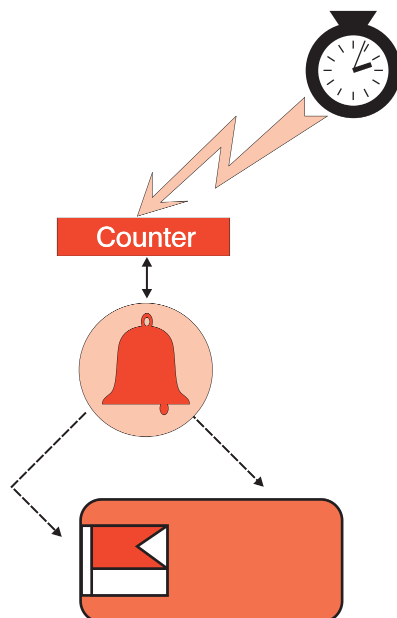

Figure 5. Working with an Alarm.

Alarms can be Relative Alarms (based on changes in the counter's value), or Absolute Alarms (based on the absolute count). Alarms can be either 'one-shot' or cyclic.

KEEPING IT SMALL AND SIMPLE

OSEK/VDX has intentionally avoided the 'feature bloat' which afflicts some other operating systems. And in this way it can achieve very high performance, deterministic timing performance, small executable code footprint, and small RAM footprint.

It has done this by not offering a long list of 'nice-to-have' features that are often unnecessary in tightly constrained embedded systems. For example:

- No malloc() , free()
- No heaps
- No memory regions
- No memory partitions
- No dynamic memory allocation facilities whatsoever
- No specific device I/O support
- No file system support
- No on-the-fly creation of operating system objects (tasks, events, resources, alarms,...)
- No on-the-fly deletion of operating system objects
- No on-the-fly programmed changes of task priority or preemptibility
- No round-robin time-slicing of tasks

What is left is a 'lean and mean' high-performance resource-modest kernel sufficient to power many tightly resource-constrained deeply embedded applications ■

Dr. DAVID KALINSKY is Director of Customer Education at Integrated Systems, Sunnyvale, CA.
In recent years, Dr. Kalinsky has been a popular lecturer and seminar leader on embedded computer and control systems software development technologies, in the U.S.A. as well as in Europe and the Far East. He has developed worldwide technical customer training programs at several Silicon Valley companies.
He has over 25 years of experience in software engineering technologies and in the development of real-time software for embedded systems. Dr. Kalinsky holds an A.B. in Physics from Brown University and a M.Phil. and Ph.D. in Nuclear Physics from Yale University.

Copyright 1999 by Real-Time Magazine 99-4 (http://www.realtime-info.com)