Real-Time Application Development with OSEK

A Review of the OSEK Standards

Peter H. Feiler

November 2003

Performance-Critical Systems Initiative

Technical Note
CMU/SEI-2003-TN-004

Unlimited distribution subject to the copyright.

The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.

Copyright 2003 by Carnegie Mellon University.

NO WARRANTY

THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

Use of any trademarks in this report is not intended in any way to infringe on the rights of the trademark holder.

Internal use. Permission to reproduce this document and to prepare derivative works from this document for internal use is granted, provided the copyright and “No Warranty” statements are included with all reproductions and derivative works.

External use. Requests for permission to reproduce this document or prepare derivative works of this document for external and commercial use should be addressed to the SEI Licensing Agent.

This effort was supported in part by the DARPA MoBIES program under contract F33615-00-C-1701 managed by the Wright Patterson Air Force Research Laboratories.

This work was created in the performance of Federal Government Contract Number F19628-00-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 252.227-7013.

For information about purchasing paper copies of SEI reports, please visit the publications portion of our Web site (http://www.sei.cmu.edu/publications/pubweb.html).

Contents

Abstract .... **v**

1 Introduction **1**

2 Background **2**

3 The Review Approach **4**

4 Conceptual Model of Real-Time Applications **5**

5 OSEK/VDX Operating System Version 2.2 .... **7**

5.1 Conformance Classes 7

5.1.1 Portability vs. Footprint Tradeoff 7

5.1.2 Support for Periodic Tasks 7

5.1.3 Frugal Use of Priorities and Tasks 8

5.1.4 Over-Constraints and Clarifications 8

5.2 Task Management 8

5.2.1 General Task Concept 9

5.2.2 Task Initialization and Dispatch 9

5.2.3 Task Instances and Multiple Dispatch Requests 10

5.2.4 Task Priorities and Scheduling 12

5.2.5 Non-Deterministic Non-FIFO Task Scheduling Behavior 15

5.2.6 Periodic Rates and Schedulability 15

5.2.7 Task Interaction Topology and Portability 16

5.3 Application Modes 16

5.3.1 System Initialization 17

5.4 Interrupt Processing 18

5.5 Event Mechanism 18

5.5.1 Events as Binary Task Flags 19

5.5.2 Event Processing 19

5.5.3 Conditional Event Processing 20

5.5.4 Event Broadcast 20

5.6 Resource Management 20

5.6.1 A Single Resource Mechanism 21

5.6.2 Non-Preemptable Regions 21

5.6.3 Nested Resource Acquisition 22
5.7 Alarm Mechanism 22
5.7.1 Alarm Expiration Occurrence 23
5.7.2 Time Unit Confusion 23
5.7.3 Support for Periodic Tasks 23
5.7.4 Variable Rate External Events 23
5.7.5 Alarms as Timeout Mechanism 24
5.8 Error Handling 24

6 Communication 25
6.1 Message Communication Concepts 25
6.1.1 Message Communication Model 25
6.1.2 Local and Network Communication 27
6.1.3 Message Communication Optimization 27
6.1.4 Message Communication Faults 27
6.2 Message Services: A Moving Target 28
6.2.1 Message Filtering 28
6.2.2 Message Communication Model Revisited 28
6.2.3 Multiple Senders and Receivers 29
6.2.4 Event Communication Through Zero Length Messages 31

7 System Modeling and OIL 33

8 Summary 35

References 39

List of Figures

Figure 1: Possible Task States 9

Figure 2: Dispatch Requests, Dispatching, and Priority-Based Scheduling 13

Figure 3: Task Priority Stack Concept 13

Figure 4: One-to-One and One-to-N Communication in OSEK COM Version 2.2.2 26

Figure 5: OSEK COM Version 3.0 Communication Model 30

Abstract

OSEK is an abbreviation for a German term that translates to "open systems and the corresponding interfaces for automotive electronics." OSEK OS is the operating system specification and OSEK COM is the communication specification. Both are application program interface (API) standards for automotive real-time application development. They are complemented by OSEK Implementation Language (OIL), a modeling language for describing the configuration of an OSEK application and operating system.

This paper covers the SEI evaluation of these standards from the perspective of real-time application development. The SEI identified shortcomings in the description and semantics of certain services offered by the OSEK API. These shortcomings introduce unnecessary complexity to application developers and limit application portability. The SEI also identified the potential of OIL as an architectural modeling language to support design-time analyses, such as schedulability analysis. OIL's potential as a basis for generating both real-time OS data tables and an application runtime executive was examined. Utilizing OIL in this way simplifies application component development. Correct use of OSEK API functionality is then relegated to a generation tool that operates on OIL. Such improvements would facilitate practitioners' adoption of OSEK by reducing its perceived complexity.

1 Introduction

OSEK is an abbreviation for the German term "Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug" (in English: "Open Systems and the Corresponding Interfaces for Automotive Electronics"). OSEK OS is OSEK Operating System Specification Version 2.2 [OS 01], and OSEK COM is OSEK Communication Specification Version 2.2.2 and Version 3.0 [COM 00, 03]. Both are application program interface (API) standards for automotive real-time application development. They are complemented with OSEK Implementation Language (OIL), a modeling language to describe the configuration of an OSEK application and operating system. Their intent is to facilitate application portability, minimal application footprint, and use of modern scheduling approaches to meet timing requirements while maintaining flexibility to configure and evolve embedded application systems.

We have evaluated these standards from the perspective of real-time application development. We have identified some shortcomings in the description and semantics of some of the services offered by the OSEK API. These shortcomings introduce unnecessary complexity to application developers and limit application portability. We also identified the potential of OIL as an architectural modeling language to support design-time analyses, such as schedulability analysis, and as a basis for not only generating real-time OS data tables, but also for generating an application runtime executive. This simplifies application component development as appropriate and optimal use of OSEK API functionality is relegated to a generation tool that operates on OIL. Such improvements would facilitate the adoption of OSEK by the practitioner community by reducing its perceived complexity.

2 Background

In May 1993 OSEK was founded as a joint project in the German automotive industry aiming at an industry standard for an open-ended architecture for distributed control units in vehicles. The OSEK Vehicle Distributed eXecutive (OSEK/VDX) website (http://www.osek-vdx.org/) sets forth the following.

The goal of OSEK is to support the development of automotive applications, which have stringent real-time requirements; that is,

to support the portability and reusability of the application software by

- *specification of interfaces which are abstract and as application-independent as possible*
- *specification of a user interface independent of hardware and network*
- *efficient design of architecture: The functionalities shall be configurable and scalable, to enable optimal adjustment of the architecture to the application in question.*
- *verification of functionality and implementation of prototypes in selected pilot projects*

The advantages of this support are as follows:

- *clear savings in costs and development time*
- *enhanced quality of the software of control units 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*
- *absolute independence with regards to individual implementation, as the specification does not prescribe implementation aspects*

In addition, the open architecture introduced by OSEK/VDX comprises the three areas:

1. *Operating System (Real-time executive for ECU software and basis for the other OSEK/VDX modules)*
2. *Communication (Data exchange within and between control units)*
3. *Network Management (Configuration determination and monitoring)*

For each of the three areas a standard document defines an API specification to a set of services. For Communication and Operating System, there are standard operating system/communication specifications and specifications covering globally synchronized fault-tolerant architectures (OSEK time specifications). In addition, a standard document defines OIL. Its role is to support a portable description of the task and communication architecture of an OSEK-based application system for a particular hardware separate from the application source code. OIL is the basis for system generation.

3 The Review Approach

The keys to successful adoption of any standard are

- conceptual simplicity and clarity
- semantic consistency among services implementing the concepts
- the ability to fulfill promises of portability, evolvability, and predictive analyzability with respect to key quality attributes such as timing, performance, reliability, and safety

OSEK offers a range of services to support task management, task communication, and task coordination of concurrent resource use. We will review these services for conceptual consistency. OSEK supports fixed-priority preemptive scheduling and use of the priority ceiling protocol (PCP)—technologies that are amenable to schedulability analysis of real-time applications. We will examine OSEK from the perspective of predictable analysis of timing requirements. OSEK supports application porting and evolution through

- flexibility in timeline management as a benefit of fixed-priority, preemptive scheduling and schedulability analysis
- modeling and generation of task and communication configurations through OIL

We will examine OSEK from the perspective of its ability to separate concerns of application component developers from those of application system integrators by minimizing system configuration information embedded by application developers in application code.

We proceed by summarizing the key concepts offered to the application developer and then reviewing the OSEK Operating System Specification Version 2.2 [OS 01], OSEK Communication Specification Version 2.2.2 and Version 3.0 [COM 00, 03], and OSEK Implementation Language (OIL) Specification Version 2.3 [OIL 01] in light of this conceptual model. The review of each specification examines specific services addressing issues of consistency of the conceptual model and issues of clarification.

4 Conceptual Model of Real-Time Applications

Statements that apply to real-time applications in general are below in plain text. The statements in italics apply specifically to OSEK.

The conceptual model of real-time applications represents time-critical applications that consist of multiple concurrent execution units executing on an execution platform that is possibly distributed.

In OSEK, tasks, interrupt service routines (ISRs), and callbacks represent execution threads.

If the set of execution units is statically known, schedulability can be determined through analysis at design time. An application may have different modes representing different statically known execution-unit and communication configurations. Such configurations represent subsets of dispatchable execution units.

In OSEK, the set of tasks and ISRs is statically specified in OIL. OSEK offers the concept of application modes.

The execution units are dispatched by external events, by events originating in other tasks, or by periodic timer.

In OSEK, tasks can be dispatched by explicit activation from another task and by setting events. In addition, they can be dispatched as actions resulting from a message communication and from periodic timer alarms and time-expiration alarms, as well as special counter condition alarms. ISRs are tasks that are dispatched by hardware interrupt events.

Execution units represent applications with application states that are to be initialized before their first dispatch, re-initialized on restart (e.g., as result of a failure condition), or re-initialized when the application switches modes.

In OSEK, extended tasks can execute initialization code before their first WaitEvent call. Message buffers can be initialized by a hook routine. Otherwise global application initialization code must be used. The concept of restart is not explicitly supported for either fault recovery or for switching between application modes.

Execution units may interact by exchanging information through communication and by shared access to information. Note that communicated or shared information may be data, control, or both.

In OSEK, the message service supports local and distributed communication of data and optional control (through message receipt action). The event mechanism locally communicates control and binary data. Explicit task activation locally communicates control. Data can be communicated through shared data access, which is coordinated through OSEK resources.

Each execution unit exists as a single dispatchable instance. When an execution unit is dispatched (i.e., passed to the scheduler for execution), additional dispatch requests do not result in immediate additional submissions to the scheduler. Additional dispatch requests may be queued first-in-first-out (FIFO) and serviced upon completion of the previous dispatch.

In OSEK, different dispatch mechanisms support different policies regarding the ability to queue dispatch requests, the policy used to discard dispatch requests, and the policy used to submit the dispatch to the scheduler.

Different execution units may be active at the same time; that is, they require the use of execution platform resources such as processors, memory, and network. Statically or dynamically assigned priorities determine the scheduling order. Execution units with the same priority are serviced FIFO.

In OSEK, a fixed-priority preemptive scheduling policy with support for non-preemptive tasks supports arbitration of processors. OSEK supports user-assignable task priorities and hardware-determined ISR priorities in a higher priority band.

Concurrent execution of execution units must be coordinated to assure integrity and consistency of shared data. This includes data explicitly shared between application components, data shared between applications and system services, and data shared between system services. Mutually exclusive access is guaranteed for any execution portion(s) of an execution unit.

OSEK offers three variants of a logical resource concept (resource, internal resource, linked resource) that assures mutual exclusion through the priority ceiling protocol. It can do so because the set of tasks requiring mutually exclusive access is known. In addition, application developers can disable dispatch of tasks and of two classes of ISRs. Finally, access to message buffers can be coordinated between applications and the system through resource locks. Coordination of shared access between system services is implicitly assumed.

5 OSEK/VDX Operating System Version 2.2

The OSEK/VDX Operating System Specification 2.2 specifies a set of services to be provided by a real-time operating system running on a single machine. These services focus on supporting coordinated concurrent execution. Tasks and ISRs are the units of concurrent execution that can operate under a particular application mode. Concurrent execution is coordinated through alarms and events that can trigger execution, and through logical resources that represent shared information across concurrent executions.

5.1 Conformance Classes

Conformance classes have been introduced to provide groups of features for easier understanding and for allowing partial implementations (BCC1, BCC2, ECC1, ECC2). Discriminators between conformance classes are as follows:

- support for more than one activation request per task
- support for more than one task per priority level
- support for waiting on events; number of available priority levels
- number of available resources

The specification introduces four conformance classes for tasks and minimum requirements for each conformance class. The specification states that portability of applications can only be assumed if the minimum requirements are not exceeded. This has a number of implications on the design of user applications in two ways. It places restrictions on the design of the task architecture. If these restrictions are not adhered to either the application is not considered portable, or the OS support for an application falls into a different conformance class with additional runtime overhead in terms of space and time.

5.1.1 Portability vs. Footprint Tradeoff

BCC1/BCC2 are limited to 8 priority levels, while ECC1/ECC2 have 16 priority levels. If an application requires 9 or 10 priorities an application developer has the choice of retaining a smaller OS footprint by staying with a non-portable application of category BCC1/2, or by switching to ECC1/ECC2 with a larger footprint but achieving portability.

5.1.2 Support for Periodic Tasks

The minimum requirement for all conformance classes is support for a single alarm. As discussed in a later section, alarm expiry actions are limited to a single task activation or setting of a single event. In order to retain application portability, application developers are

limited to a single periodic task. They may use this task to provide their own task dispatcher —defeating the purpose of OSEK's common task scheduler.

5.1.3 Frugal Use of Priorities and Tasks

Under minimum requirements the number of tasks and task priorities are limited to 8 under BCC and 16 under ECC. This requires application developers to carefully consider how they make use of tasks and task priorities.

For instance, precedence ordering of tasks may be established by using priorities. Tasks may be used to implement message queues supporting multiple senders, for example. In other words, in some cases developers are required to overcome shortcomings in the available set of API service calls by workarounds that involve additional tasks and/or priorities.

5.1.4 Over-Constraints and Clarifications

Under minimum requirements for resources, RES_SCHEDULER is explicitly called out as one of the internal resources. Its role is to achieve non-preemption of tasks. Under minimum requirements for internal resources, a limit of two is specified. It is assumed, but not clearly stated, that the internal resource associated with RES_SCHEDULER is counted against the limit, thus leaving one for other purposes.

BCC1 offers only the RES_SCHEDULER resource as a concurrent access coordination mechanism. This means that all user application tasks are blocked, not just other tasks accessing the same resource. To avoid excessive blocking an application developer must switch to BCC2 and accept the unnecessary additional cost of priority-level-ready queues.

BCC1 and ECC1 limit tasks to one per priority level. It is unclear whether this restriction is to be interpreted as one active task per priority level or one declared task per priority level. The latter interpretation is possible because BCC1/ECC1 support a minimum of 8/16 tasks not in suspended state and 8/16 priority levels.

5.2 Task Management

OSEK/VDX offers four concepts to represent units of concurrent execution: tasks, ISRs, task groups, and application modes. Mechanisms range from events, messages, and direct task activation to resources, non-preemptable tasks, and suppression of interrupt handling to manage concurrent access. As we will see, these different mechanisms are not always semantically consistent in how they offer the desired services. Their proliferation introduces unnecessary complexity for application developers.

This section focuses on tasks. We will show how these other mechanisms can be mapped into a simplified conceptual model that encompasses tasks, events, messages, resources, ISRs, and callbacks.

5.2.1 General Task Concept

A task is a single execution thread, whose execution is managed by a scheduler. Tasks have statically declared fixed priority and may execute with or without preemption. Preemptable and non-preemptable tasks may be combined on a single processor. Tasks can execute periodically (i.e., be dispatched at fixed time intervals), or aperiodically (i.e., be dispatched by some event).

A task may consist of an initialization sequence, code sequences executed upon (repeated) dispatch, and an optional termination sequence. Task initialization may be desirable during startup, when recovering from a fault state (restart), and when including a task in an application mode during application mode switching.

A task may be part of different task configurations known in OSEK as application modes. When part of an active application mode, a task is considered active and is dispatchable.

This set of possible states is illustrated in Figure 1.

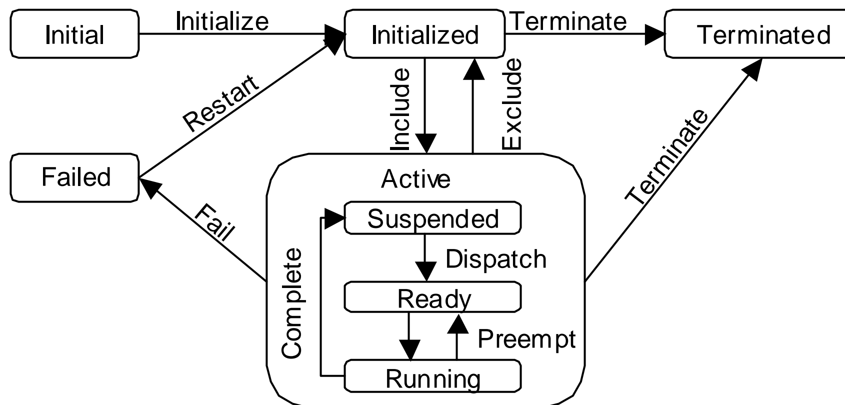

Figure 1: Possible Task States

5.2.2 Task Initialization and Dispatch

In this section we examine task initialization for basic and extended tasks.

OSEK basic tasks are dispatched by task activation. This is achieved by explicit ActivateTask or ChainTask calls or implicitly as action of message receipt or as alarm expiration action. Basic tasks complete their dispatch by explicit calls to TerminateTask or ChainTask. Note

that these are required calls, while for ISRs completion of a code sequence is interpreted as dispatch completion, and calls to TerminateTask or ChainTask are prohibited.

Unfortunately, there is no consistent support for task initialization and task restart initialization. OSEK basic tasks do not have an explicit initialization sequence, although tasks can be initialized as part of the global system startup initialization (see Section 5.3.1). Event flags are automatically cleared on task activation. Message buffers can be initialized at startup through a single initialization hook. Application data can be initialized through the global startup initialization hook or implemented as an application-level initialization task that activates other tasks. Similarly, termination sequences for tasks have to be included in a global system shutdown handler. Note that these global handlers require access to task state of all tasks.

OSEK extended tasks have their dispatch points represented by WaitEvent calls. A dispatch completes when the next WaitEvent call occurs. Activation of an OSEK extended task can either be interpreted as the initial dispatch (i.e., extended tasks have two dispatch mechanisms), or it can be interpreted as the initiation of an initialization sequence. Similarly, the code sequence between the last WaitEvent and the TerminateTask call can be used by convention as a termination sequence. OSEK extended tasks do not support multiple task activations (i.e., queuing of multiple activation requests)——an indication that they are intended to be activated only once.

The task model could be improved as follows. Task activation causes an initialization sequence to be executed. A task enters the suspended state by an AwaitDispatch call. This is interpreted as a WaitEvent on a predefined event flag called Dispatch. DispatchTask and ChainTask would result in a dispatch event. This dispatch event may be bound to the target task in the OIL specification. A TerminateTask call would cause the target task to enter its termination sequence. The WaitEvent call would provide a means of conditional wait on dispatch from specific origins; for more on this see the discussion in the Section 5.5. Restart and application mode switching initialization can be handled in a similar manner.

5.2.3 Task Instances and Multiple Dispatch Requests

The set of tasks is statically known. Each task exists as a single instance. This task instance may be part of an active application mode. When part of an active application mode, the task instance may be suspended, that is, ready to be dispatched for execution, or executing, that is, in ready or running state. A task may already be executing when a dispatch request is issued. In this case the dispatch request is queued with the task, if request queuing is specified for the task. If queuing is not specified or the request queue is full, the request is ignored, and an error status is not reported to the requestor. When a task completes its dispatch execution and the dispatch request queue is not empty, the task is immediately redispatched to the end of the respective priority-level-ready queue.

Several observations can be made with respect to differences in handling of multiple dispatch requests by different mechanisms. This can lead to unexpected behavior when choosing one mechanism over another.

OSEK basic tasks support queuing of multiple dispatch requests, while OSEK extended tasks effectively support queues of one length. The event mechanism, used to perform task dispatch of extended tasks, supports recording of one dispatch request in a binary flag while the task is executing.

OSEK basic tasks may be dispatched by explicit ActivateTask or ChainTask calls. In such a case an error status indicates to the requestor when a dispatch request is lost. OSEK extended tasks can be dispatched by a SetEvent call. In this case the requestor is also provided with an error status if dispatch requests are lost. Finally, tasks may also be dispatched by alarm expiration action or message receipt action. In that case the requestor is not informed about a possible loss of dispatch requests.

The description of multiple dispatch request support could benefit from clarification. The current description could lead to misunderstandings as to the order in which tasks of the same priority are executed.

A footnote in Section 4.2.2 indicates that activation is only recorded and performed later if a basic task has not terminated (is in ready or running state).

Section 4.3 states that "task activation is performed through the operating system services ActivateTask or ChainTask. After activation the task is ready to execute from the first statement." These statements seem to imply that these two operations always result in a task being placed into ready state. Instead, they perform a dispatch request, which may be queued (i.e., delayed until a previous dispatch completes), or result in an immediate task dispatch (i.e., enter the ready state).

Section 4.3 describes handling of multiple task activation requests. "Depending on the conformance class a basic task can be *activated* once or *multiple times*" and "Multiple requesting of task activation means that the OSEK operating system receives and records *parallel activations* of a basic task already activated." These statements suggest that a particular task can be dispatched more than once.

Section 4.3 also states "If the maximum number of multiple requests has not been reached, the request is queued. The requests of basic tasks are queued per priority in activation order." This suggests that a task dispatch request is placed on the priority ready queue at the time of the dispatch request, resulting in multiple task activation instances of the same task in a priority ready queue. Alternatively, the statements can be interpreted as activation requests being placed in a request queue per priority level (note that multiple tasks may exist for each priority level). Finally, the second statement can be interpreted as indicating that when a

dispatch request is removed from the task-specific request queue (i.e., the dispatch request is processed), the task is placed at the end of the priority level ready queue.

Section 13.2.3.1 describes the scheduling semantics of ActivateTask as transferring a task from the suspended state to the ready state with the following exception. "ActivateTask will not immediately change the state of the task in case of multiple activation requests. If the task is not suspended, the activation will only be recorded and performed later." Section 13.2.3.2 describes the scheduling semantics of TerminateTask as follows regarding multiple activation requests. "Terminating the current instance of the task automatically puts the next instance of the same task into the ready state." Note that the statement suggests a task may have multiple instances. It is ambiguous as to whether entry into the ready queue is determined by the dispatch request ("recorded") or by the time the dispatch is "performed" and the state "is changed."

A more appropriate statement would be "At any time a single instance of a task can be executing, that is, running, or ready. Additional task activation requests may be queued if the task is already executing. Upon completion of a task execution, queued dispatch requests result in resubmission of the task to the end of the priority ready queue."

5.2.4 Task Priorities and Scheduling

OSEK/VDX assumes a fixed-priority scheduler that supports mixed preemptable and non-preemptable tasks with multiple tasks at each priority level. In addition, it supports a priority ceiling protocol for mutually exclusive access to logical resources. This results in each task having a statically assigned priority and a current priority at which the task is actually executed.

Normally tasks are dispatched into a FIFO priority ready queue based on their static priority. The first task in the highest priority non-empty priority queue is selected for actual execution (running state). If a task is dispatched to a higher priority ready queue the running task is preempted. This means it is placed in the front of the ready queue of its current priority (initially the same as the static priority), and the higher priority task is selected.

Non-preemptable tasks have a static priority value and a separate current priority value. The latter is the highest application priority, i.e., the application priority ceiling (APC), also known as RES_SCHEDULER in OSEK. The task is dispatched at its static priority (i.e., placed in the respective ready queue). When selected for actual execution (running state) it assumes its current priority (APC). While running, this task cannot be preempted by any application task, but may be preempted by an ISR and placed in front of the ready queue at its current priority (APC). A non-preemptable task may also permit rescheduling with an explicit Schedule system call. In this case it is placed in the ready queue of its static priority. Figure 2 illustrates the dispatch and scheduling actions based on static and current priority.

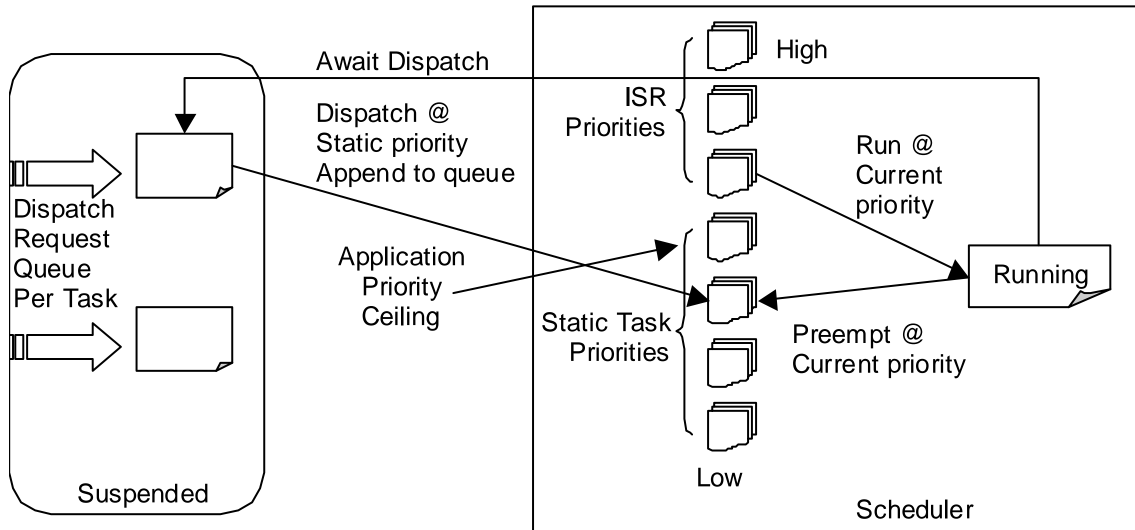

Figure 2: Dispatch Requests, Dispatching, and Priority-Based Scheduling

This view of static and current priority can easily be extended to support the priority ceiling protocol for resources. Conceptually each task has a priority stack. The first entry represents its statically assigned priority. The current priority refers to top of the stack, initially the first entry. In case of non-preemptable tasks the stack contains a second entry with APC as value. When a resource is acquired the larger of its priority ceiling or the current priority value is added to the stack. When a resource is released the top element of the stack is removed. This is illustrated in Figure 3. For further discussion of resources and priority ceiling protocol we refer to the section on OSEK resources.

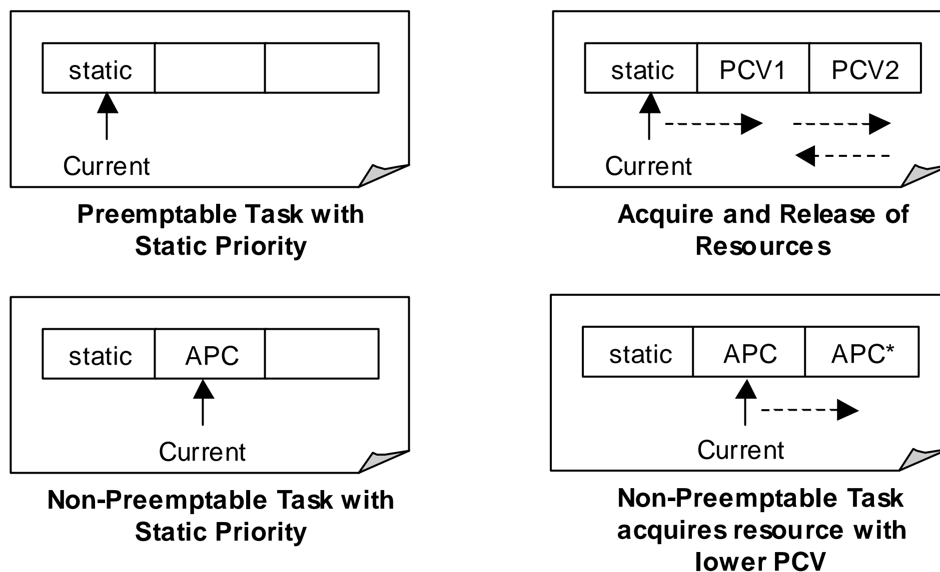

Figure 3: Task Priority Stack Concept

The above conceptual model describes the OSEK task dispatch and fixed-priority scheduling policy. OSEK tasks have statically assigned priorities—specified in OIL. Non-preemptable tasks are tagged as such when declared in OIL. This is interpreted as meaning that the OSEK internal resource of the priority RES_SCHEDULER is assigned to the task (OSEK OS Section 8.7). The effect of RES_SCHEDULER as an internal resource is illustrated in the above model.

The OSEK notion of rescheduling points (OSEK OS Sections 4.6.1 and 4.6.2) can be explained in a simple way. When a non-preemptable task is running, rescheduling occurs only if the task completes execution of its dispatch (TerminateTask, ChainTask, WaitEvent call), or explicitly requests rescheduling (Schedule call). A non-preemptable task may be preempted by an ISR, but on completion of the ISR execution, resumes execution as its current priority is APC.

Preemptable tasks have to assume that preemption (i.e., rescheduling) can occur at any time through dispatch of a higher priority task. Such dispatches can be initiated (or queued) by the task being preempted; by an ISR; by an alarm—through explicit task activation (ActivateTask, ChainTask); by implicit activation through message receipt or alarm expiration; or as result of setting a previous cleared OSEK event (SetEvent). In case of a dispatch request by an ISR, rescheduling to a higher priority application task occurs on completion of the ISR execution.

Unfortunately, the description of these capabilities is not always precise, leaving the reader with some uncertainty regarding the expected scheduling behavior. In the following, we highlight some of the text in [OS 01, v.2.2] that requires liberal reader interpretation:

FIFO behavior of ready queue: Section 4.5 describes handling of task priorities by the scheduler. It indicates that a FIFO ready queue is maintained for each priority level. A preempted task is considered to be the first task of its priority ready queue. "Tasks on the same priority level are started depending on the order of their activation, whereby tasks in waiting state do not block subsequent tasks at the same priority." As discussed under dispatch requests, the point in time of activation is not always clearly described. "A task being released from the waiting state is treated like the newest task in the ready queue of its priority." The explanation of Figure 4-5 of OSEK OS indicates that a task is "removed from the queue either due to termination or due to transition into the waiting state." This aligns with our view of treating continuation from WaitEvent as a task dispatch.

Semantics of self activation: Section 13.2.3.3 describes the scheduling semantics of ChainTask as "terminating the calling task" and then "activating the succeeding task." If the succeeding task is identical (i.e., the task activates itself) the semantic description can benefit from clarification. The section states that "This does not result in multiple requests," and "The task is not transferred to the suspended state." This can be interpreted as indicating that this is not an activation request and the task remains ready and possibly at the front of the

priority-level-ready queue. Section 4.7 clarifies that "chaining itself puts the task into the last element of the priority ready queue." This intends to state that the task completes and redispatches (i.e., it transitions back to the ready state and is added to the end of the priority level ready queue). If the activation request queue is non-empty the above description can be interpreted as the self-dispatch request taking priority over other requests in the activation queue. To avoid potential starvation, the semantics of self dispatch should require the oldest request in the activation queue to be processed and the self-dispatch request to be added to the request queue.

Dynamically changing task priority: Section 4.5 states that "to enhance efficiency dynamic priority management is not supported. Accordingly the priority of a task is defined statically, i.e., the user cannot change it at the time of execution. However, in particular cases the operating system can treat a task with a defined higher priority." In actuality, OSEK supports changing task priorities at runtime according to user specification, that is, by OIL specifications indicating that resources are sharable between tasks and by application code requesting such a priority change by acquiring a resource (GetResource call). Note that application developers can also emulate runtime changes to task priorities at the cost of additional tasks by breaking tasks into multiple tasks with different statically declared priorities and executed via ChainTask.

5.2.5 Non-Deterministic Non-FIFO Task Scheduling Behavior

OSEK OS Sections 4.6.1 and 4.6.2, including a footnote, indicate that if a task invokes the WaitEvent service and an event flag is set, the task is not considered transitioning to and from the waiting state and does not lead to rescheduling. In other words, this task retains its position in the front of its priority ready queue if the event was set while the task was executing. The task is placed at the end of the queue if the event is set after the task enters the waiting state. This results in non-deterministic behavior that is dependent on whether the event has been set at the time the task completes with a WaitEvent call.

Note that by remaining in front of the priority ready queue, whose policy is otherwise FIFO, extended tasks dispatched by events can introduce starvation. Effectively a task whose execution is triggered by an event has a higher priority than other tasks at the same priority level under high load conditions. This semantic difference may influence the design decision as to whether an alarm or message arrival should trigger task execution by activation or by event.

5.2.6 Periodic Rates and Schedulability

If priorities are assigned to periodic tasks according to the rate at which tasks execute, Generalized Rate-Monotonic Analysis (GRMA) can be used to determine whether a set of tasks is schedulable (i.e., guaranteed to meet their deadlines under worst-case execution time assumptions) [Klein 93]. Note also that the period of a task is determined by the increment

of a cyclic alarm. The periodicity of alarms can be specified in the OIL specification. Thus, scheduling analysis could be performed based on information in OIL specifications.

Unfortunately, the alarm increment can be changed at runtime through a SetRelAlarm or SetAbsAlarm call. This changes the period but would not result in a change to the statically assigned priority. As a result, the rate-monotonic priority assignment assumption could be violated, and the schedulability analysis result invalidated.

An approach to supporting controlled runtime changes to task periods is to represent task periods in OIL for different application modes—as is currently supported in the autostart specification in OIL. Schedulability can then be determined for each application mode in turn. Runtime switching between application modes results in task rate changes that adhere to RMA assumptions—as is done in MetaH [Vestal 93, Honeywell 98].

Schedulability analysis depends on a model of all task dispatching. Currently OIL models all tasks descriptions including priorities, as well as task activation by alarm or message. However, task dispatch by direct ActivateTask or ChainTask calls is not modeled. Modeling of task dispatching uniformly as dispatch events can alleviate this problem—as is done in MetaH with event ports and connections.

5.2.7 Task Interaction Topology and Portability

Explicit task dispatch by ActivateTask and ChainTask calls is a common way of establishing precedence ordering between tasks. Used as an alternative to assigning different task priorities, it reduces the need for priority levels. These mechanisms may also be used to chain together tasks of the same period in an attempt to maintain portability while being limited to a single alarm. Similarly, events can be used to sequence extended tasks in an attempt to avoid using additional alarms.

Unfortunately, the resulting application code has task topology information embedded, because the task to be dispatched has to be named by the requestor. This leads to reduced portability with respect to changes in the hardware architecture and to changes in mapping tasks to processors. Note that message-based task dispatching does not require the requestor of a dispatch to know the target. This information is recorded in the OIL specification. MetaH and the emerging Society of Automotive Engineers (SAE) standard for Avionics Architecture Description Language (AADL) demonstrates that an event concept similar to the zero length message in OSEK COM 3.0 can provide such additional portability [AADL 03].

5.3 Application Modes

Application modes represent different modes of operation. Each application mode represents a set of tasks and ISRs. Different application modes may contain the same tasks and ISRs. Only one application mode can be active at any one time. A particular application mode is

chosen at runtime and passed as parameter to StartOS, which then will automatically start the appropriate set of tasks and alarms.

Note that OSEK OS Section 5.3 indicates that runtime switching of application modes is not supported. However, there is no inherent limitation to this capability. In fact, MetaH and the SAE AADL provide such dynamic switching between statically known task and communication configurations (i.e., application modes). In OSEK this can be achieved by performing a ShutdownOS and a repeated StartOS with the appropriate mode value as parameter. See Lemieux for further information [Lemieux 01, p. 25].

Application modes are currently addressed in OSEK in two ways.

First, the system function StartOS takes an application mode as parameter to allow application developers to perform mode-specific initializations. StartOS itself calls on the StartupHook routine that is supplied by the application developer. Inside this routine the application developer can query for the application mode and perform initializations [OS 01, p. 42] that do not involve any system calls [OS 01, p. 45].

Second, the AutoStart service activates tasks and sets alarms for the current application mode after the execution of the startup hook. The OIL specification indicates which tasks should be activated and alarms set for a specific application mode at autostart time.

The OIL specification currently indicates which tasks should be started directly. Typically this is used to start initialization tasks or extended tasks that initialize themselves and then suspend themselves for dispatching by events. OIL currently does not specify the complete set of tasks that are part of a particular application mode (i.e., are part of a mode specific task configuration). Such task configuration information is essential for mode-specific schedulability analysis. Note that the revised view of tasks with initialization support would allow us to list all tasks that are part of an application mode.

5.3.1 System Initialization

A task has various data structures that contain state information. They include event flags, the content of unqueued messages, static data variables in the source code—both those local to a task, and those shared across tasks. In addition, an application system may contain counters.

Setting initial values of static data variables in source code is handled by the source code language. Many languages allow an initial value to be specified when a variable is declared.

OSEK specifies that event flags be cleared when a task is activated through ActivateTask [OS 01, p. 50] or through EliminateTask [OS 01, p 52].

Section 2.7.1 of OSEK COM 3.0 states the following for message initialization.

- For queued messages the message queue is initialized as empty.
- For unqueued messages the initial value can be specified in OIL, with an assumed default value of zero. Messages can also be initialized through a user supplied MessageInit routine, which is automatically called from StartCOM via a user supplied StartCOMExtension.
- "For internal transmit messages no initialization takes place." This can be interpreted as meaning that unqueued messages used for local communication are not initialized.
- "Once the kernel is started, an application calls StartCom." This implies that message initialization via StartCOMExtension occurs after tasks have been autostarted. Since autostart does not guarantee activation ordering within a priority level, an autostarted task with the highest priority must take the responsibility of calling StartCOM.

It is unclear whether initialization of counters (used in alarms) is the responsibility of application code, part of StartOS system initialization, or user-supplied initialization through the StartupHook. OS states that "Counters are – if possible – set to zero by the system initialization before alarms are autostarted." [OS 01, p. 43 footnote].

In summary, it is desirable to provide default initial values for all task state, user specified initial values (in OIL for OSEK objects), and explicit initialization and termination sequences as outlined earlier.

5.4 Interrupt Processing

ISRs can be viewed as tasks whose dispatch is requested by hardware. All ISRs are considered to have a priority higher than any of the regular application tasks. It is assumed that hardware will ensure that only one ISR dispatch request occurs at a time for each ISR (i.e., request queuing is not necessary).

Disabling of interrupts conceptually corresponds to acquiring a resource whose priority ceiling is that of the highest ISR priority. This interpretation is in line with OSEK's support for use of resources in ISRs, and between ISRs and application tasks. In the latter case, application tasks may have a current priority that requires ISR dispatching to be suppressed during the duration of the resource acquisition.

5.5 Event Mechanism

The event mechanism is advertised to be "a means of synchronization" that "initiates state transitions of tasks to and from the waiting state." "Events can be used to communicate binary information." "The meaning of events is defined by applications, e.g., signaling of an expiring timer, the availability of a resource, the reception of a message, etc." This promises more than the event mechanism delivers.

5.5.1 Events as Binary Task Flags

Events are flags local to individual processes, i.e., any synchronization state is distributed across tasks. Events can only be used to communicate within one processor. No support is provided to associate actions with the release of resources, i.e., it is the application programmer's responsibility to signal release of a resource through events.

Binary event communication is not reliable. Not every event results in task dispatch. The OSEK event mechanism is essentially a binary status flag that is associated with a task. It can be set by another task through the SetEvent operation. When the event flag is set no indication is given that it was set before. Thus, events can get lost without an error indication. Consequently OSEK events differ from a more common interpretation of events as separate entities, as found in many software development languages and in architecture description languages such as MetaH/AADL. As a matter of fact the concept of zero length message in OSEK COM 3.0 (see Section 6.2.4) attempts to support this more common interpretation.

In addition to events being mapped into a binary flag rather than represented as separate objects, processing of event flags that are set before a WaitEvent call is performed results in non-FIFO scheduling behavior with starvation potential (see Section 5.2.5). This can result in the following concurrency scenario.

A task may use SetEvent to pass control to another task before suspending itself. This may result in preemption of the task setting the event and unnecessary context switching if the second task is higher priority. This preemption increases the time the first task is in ready state; it thus increases the chances that it might receive another event before executing its WaitEvent call. This in turn increases the chances that the task remains at the head of its priority ready queue and potentially causes starvation to other tasks at the same priority level.

5.5.2 Event Processing

Events are not consumed by WaitEvent. It is the task's responsibility to clear the event flag through a ClearEvent operation. If it does not clear the event flag the flag remains set and is recognized as such at the next WaitEvent operation. The task remains in the ready state and at the head of its priority-level ready queue and immediately proceeds with its execution except for preemptions. Furthermore, potential race conditions may occur in that an event may be set after a WaitEvent call resumes, but before a ClearEvent call is performed. Such potential race conditions can be avoided by declaring the task to be non-preemptable (overkill for achieving atomic consumption of events) or by supporting non-preemptable regions that implicitly begin with task dispatch, as is the case for non-preemptable regions representing a complete task. See Section 5.6 regarding support for non-preemptable task regions.

5.5.3 Conditional Event Processing

The OSEK event mechanism provides great flexibility for waiting on multiple events. If used correctly, this mechanism can emulate communicating state machines. A task can have multiple WaitEvent calls, each with a different event subset mask. Each call represents a state machine state and each mask represents the subset of events triggering outgoing transitions. The particular transition to be followed can be determined through program logic by comparing the flag(s) matched by the WaitEvent call against a mask for each outgoing transition. Note that only OR conditions on transition events are supported directly. AND conditions on transitions can be modeled by multiple WaitEvent calls. However, all of the control logic must be designed by the application developer, and any mistake can lead to systems with starvation problems (i.e., runaway processes or deadlock situations).

5.5.4 Event Broadcast

In many cases it is desirable to treat an event as an occurrence that can be observed by multiple tasks. This can easily be done by zero length messages or, alternatively, through SetEvent calls. Using the OSEK event mechanism, an event must be broadcast through separate SetEvent calls, since each SetEvent call can only set event flags in one process. Note that a SetEvent call may preempt the broadcaster. This delays additional SetEvent calls from the broadcaster, while the preempting task may raise a new event. As a result of the delay a task may observe the latter event as a set flag before it observes the former. Broadcasting of events by multiple SetEvent calls can be made atomic through introduction of non-preemptable regions (see Section 5.6).

The most predictable and efficient solution is for OSEK either to support a SetEvent service to multiple tasks, or to merge the event concept with that of zero length messages. This results in application code that is unaffected by changes in hardware topology and in mapping of tasks to processors.

5.6 Resource Management

OSEK OS introduces an elegant solution to coordination of concurrent access through the priority ceiling protocol-based resource concept. Resources can be used to coordinate concurrent access between preemptable tasks, non-preemptable tasks, between tasks and ISRs, and between ISRs. It allows coordination to occur without maintaining explicit lock state by adjusting task priorities in a predictable manner that assures non-preemptable access with respect to other tasks with common access. Tasks are not queued on locked resources, but reside in their respective priority ready queue. OIL specifies which tasks have access to a resource; thus, the priority ceiling can be determined statically by examining an OIL specification.

When a set of tasks and/or ISRs acquires and releases the same resource, they as a group behave like non-preemptable tasks to tasks lower than the priority ceiling (i.e., the priority of the highest priority task in the group). This includes the task in the group itself—this is how mutual exclusion is achieved. At the same time the tasks as a group are preemptable with respect to tasks with priorities higher than the priority ceiling. This provides the semantics of OSEK task groups. This also provides the semantics of non-preemptable tasks by acquiring a resource with application priority ceiling. Finally, it conceptually describes disabling of interrupts as acquiring a resource at the interrupt priority ceiling.

Unfortunately the specification document uses several terms to describe the fact that resources can be acquired to enter a non-preemptable region and assure exclusive access, and released to leave non-preemptable regions. Some of the terms and descriptions suggest that some form of locking occurs. This results in unnecessary restrictions regarding the places in which resources can be acquired and released. These restrictions, in turn, have resulted in the introduction of three variants of resources (resource, internal resource, and linked resource).

5.6.1 A Single Resource Mechanism

By combining the best features of these resource variants into a single conceptual model we have reduced the complexity of OSEK that application developers are faced with. As suggested by OSEK, OS Section 8.7 states that besides not being able to explicitly require an internal resource "the behavior of internal resources is exactly the same as standard resources (priority ceiling protocol, etc.)." If a resource is acquired, but not released when a dispatch completes execution, we use the interpretation of internal resources (i.e., the resource is implicitly released, and upon dispatch implicitly reacquired). This can be accommodated by the scheduling mechanism discussed in Section 5.2.4. The current priority points to the priority ceiling value of the last acquired and not released resource. As a result, redispatch occurs based on the static priority, but execution resumes at the current priority.

Note that we have consistently introduced implicit acquisition of resources. OIL can be extended to support specification of implicitly acquired resources, thus permitting analysis based on OIL specifications.

5.6.2 Non-Preemptable Regions

Earlier we introduced the concept of non-preemptable regions. These non-preemptable regions can be defined for any priority level, either as a region that is common to a group of tasks, that is common to all application tasks, or that is common to all application tasks and ISRs. Non-preemptable regions can be of any length, begin with the dispatch, and end with the dispatch execution completion. Resource acquisition can be nested; that is, non-preemptable regions can be nested. Although explicit and implicit resource acquire and release must still be paired and the pairs nested, many other restrictions can be relaxed, such as

- non-preemptable tasks not being part of task group
- internal resources not being explicitly acquired
- tasks being required to release all resources before performing WaitEvent calls

5.6.3 Nested Resource Acquisition

Nested acquisition of resources is permitted, although OSEK prohibits nested acquisition of the same resource. The standard claims "Resources ensure that deadlocks do not occur by use of these resources." Furthermore, "In the rare cases when nested access (to the same resource) is needed, it is recommended to use a second resource with the same behavior as the first resource. The OIL language especially supports the definition of resources with identical behavior (so-called 'linked resources')." [OS 01, p. 29]

Note that the terms "same behavior" and "identical behavior" are not defined. For purposes of efficiency, linked resources can be interpreted as representing a reference to the original resource (i.e., have a common priority ceiling). In this case acquisition and release of linked resources can be ignored.

Note that the use of nested resources does not introduce deadlock. However, a system may encounter potential deadlock situations by task communication through events or messages (i.e., tasks waiting on each other to send messages or set events).

If memory footprint is of concern, OIL can be extended to provide a specification of a maximum resource nesting level for each task.

5.7 Alarm Mechanism

The alarm mechanism is represented by two concepts: counters and alarms. "A counter is represented by a counter value, measured in 'ticks,' and some counter-specific constants." OSEK offers at least one timer (clock)-based counter. There is no API to manipulate counters directly or OIL notation to associate them with external device events. Thus, there is no OSEK specified way to cause counters to be incremented to record "events"—although OS suggests that alarms, which are associated with counters, "may expire upon receipt of a number of timer interrupts, when reaching a specific angular position, or when receiving a message" [OS 01, p. 36]. There is also no capability to explicitly reset a counter, but counters wrap around when reaching a maximum value.

An alarm specifies an expiration condition in terms of a counter value. This condition can be relative to the value at the time the alarm is set or as an absolute value. Alarms can be single, expiring once after having been set, or cyclic, expiring repeatedly at a specified counter interval. An action can be associated with the expiration of an alarm—the action being activation of one specified task, setting of event flag(s) in one specified task, or a callback

routine. Callback routines are tasks that in the case of alarms are dispatched at the application priority ceiling (APC) [OS 01, p. 37] or at ISR2 priority ceiling [OS 01, p. 36].

5.7.1 Alarm Expiration Occurrence

Although the specification does not explicitly state it, alarm expiration can occur at any time with respect to execution of application tasks and with respect to execution of ISRs. It is unclear whether alarm expiration is disabled while all interrupts are disabled. The specification is silent as to how system counter advancement is managed.

5.7.2 Time Unit Confusion

Counters and alarms deal with units of ticks. Counters have a "maximum allowed value" property, specified in ticks. When that value is reached the counter wraps around to zero. In addition, they have a "tick per base" property, defined as "number of ticks required to reach a counter-specific (significant) unit." The use of this property is unclear and "the interpretation is implementation specific" [OIL 01, p.19]. For the system counter (clock-triggered timer counter) an additional constant defines the duration of a tick in terms of nanoseconds (OSTICKDURATION [OS 01, p. 66]). This seems to be the counter-specific unit for the system counter.

5.7.3 Support for Periodic Tasks

Alarms can be used to implement periodic tasks. As noted earlier, there is a restriction of one task to be activated per alarm and a single alarm as minimum requirement for portable applications. This limits portable tasks to a single periodic task triggered by an alarm. Management of additional periodic tasks has to be implemented in application code. This can be done either by implementing a full dispatcher, or by embedding the dispatch of tasks in other tasks through explicit OSEK calls. Neither of these are attractive solutions as it is the responsibility of the OSEK OS to provide task management to relieve the application programmer.

To support schedulability analysis it is desirable to expose task dispatch rates in the OIL specification. If it is necessary to explicitly change rates, this should be reflected in OIL as proposed in Section 5.2.6.

5.7.4 Variable Rate External Events

Alarms may be driven by external events, whose rate may have stochastic or sporadic characteristics. For example, they may reflect rotation of a camshaft. To facilitate predictive modeling these rate characteristics can be represented as modal worst-case rates for different ranges (e.g., low, medium, and high) for camshaft rotation, or as statistical distributions to support real-time queuing theory based analysis [Lehoczky 97].

Alarms can be used as filters of events. External events or software events are recorded in counters. An alarm should only expire if a certain count is reached within a given time period, in which case the counter is reset.

5.7.5 Alarms as Timeout Mechanisms

Alarms can also be used as time-out mechanisms. An obvious application is to add a time-out capability to the WaitEvent call. This could be done with a set alarm call just before the WaitEvent call, and by adding a time-out event flag to the WaitEvent mask. This event flag is set as alarm action. A CancelAlarm call is placed right after the WaitEvent call to cancel the timeout if an event flag was set within the time limit. This is followed by a GetEvent call to retrieve the event flags and appropriate flags are cleared with a ClearEvent call. This should be placed in a non-preemptable code section, since we are emulating a WaitEvent with time-out, which must be an atomic operation. If it is not performed as an atomic operation the following problems are encountered:

- If the alarm expires before the WaitEvent call occurs, the task executing the WaitEvent remains in ready state and executes CancelAlarm. CancelAlarm raises an error condition since the alarm is not in use (has expired) and was not reset cyclically.
- If the task waits at the WaitEvent call and the time-out alarm expires the same situation occurs.
- If the task encounters a set event flag other than the time-out flag, but gets preempted before canceling the alarm, the alarm flag is set although the event was set in time.

This example illustrates the complexity that application developers face when they are required to implement timeout capability themselves. Therefore, it is desirable to incorporate it into key OSEK services, as is being done for message communication. ReceiveMessage with appropriate timeout semantics can be used to provide task dispatching with timeout. AwaitDispatch can be viewed as a special case ReceiveMessage waiting for a dispatch event represented by a zero length message.

5.8 Error Handling

OSEK OS 2.2 Section 11 offers an ErrorHook for centralized error handling. The specification elaborates on how user-supplied ErrorHook code can get access to error status information. However, the specification is silent as to what acceptable actions are and regarding execution resumption semantics. If the ErrorHook code performs recovery, it is unclear how control can be returned to the application code whose system call resulted in an error.

6 Communication

OSEK provides several forms of communication. Some forms of communication between tasks are supported by OSEK OS. Those services include direct task activation (see Section 5.2), binary task-specific event flags—typically used for task dispatching (see Section 5.5), and data sharing, wherein concurrent access can be managed by logical resources (see Section 5.6). Unfortunately, those services are limited to task interaction within one processor. Furthermore, in case of explicit activation or event-based communication the originator must know the destination task (i.e., task interaction topology information is embedded as system call parameters in the application code).

OSEK also provides a message communication service specified in OSEK COM. We have reviewed both OSEK COM Version 2.2.2 and the recently released OSEK COM Version 3.0. OSEK COM provides both local and network communication through the same API. Communication is performed through message objects that represent the communication connection. The association of message connections with tasks is specified in OIL. Application code has no knowledge of task interaction topology. Message service calls only refer to named message objects.

We proceed by first examining the concepts as introduced by OSEK COM 2.2.2. We then proceed to discuss differences between the concepts in Version 2.2.2 and Version 3.0 as well as concepts new to Version 3.0. As the reader will see, OSEK COM is still a moving target with message communication concepts rapidly changing and communication semantics lacking precise specification.

6.1 Message Communication Concepts

OSEK message communication occurs through message connections. It can be used to communicate message instances (i.e., data) as well as control events. Communicated data can be of statically known length or of dynamic length. Message connections can be unqueued or queued.

6.1.1 Message Communication Model

Message communication consists of several steps:

- initiation of the communication with a SendMessage call, which results in moving application data from an application-specific buffer (called accessor in OSEK COM 2.2.2) to a system buffer

- transfer of the message data from the sending system buffer to the receiving system buffer
- message receipt notification action in form of a task dispatch (task activation or setting of event)
- receipt of the message data through a ReceiveMessage call in the application

The notification action, if specified, acts as a control event trigger (for more detail see Section 6.2.4). Message transfer takes two forms: immediate transfer and periodic transfer. In the former case transfer is initiated with the arrival of the message data in the send system buffer; in the latter case it is initiated periodically. There is also a mixed transmission mode, whose semantics changed considerably between Version 2.2.2 and Version 3.0.

The OSEK COM Version 2.2.2 restrictions on multiple senders and multiple receivers are best understood with the following graphic (see Figure 4) and the fact that the development of the OSEK specification was primarily driven by concerns of efficient implementation. Message queuing occurs on the receiving side. Receive Message Queues are intended to have a simple implementation (i.e., should not be concerned with concurrent queue access). This resulted in the restriction that each Receive Message Queue could only have one receiver task. In case of local communication this translates to a single receiver for queued messages, but not for unqueued messages. In case of network communication this leads to multiple receivers, but only one receiver per processor. Note that the model for multiple senders/receivers changes considerably in OSEK COM Version 3.0 (see Section 6.2.2).

Note that the implementation of Send System Buffers as well as Receive System Buffers and Queues still must address concurrency issues. In particular for network communication SendMessage and ReceiveMessage, operations must be coordinated with message transfer operations to ensure data consistency in accessing the system buffers and queues.

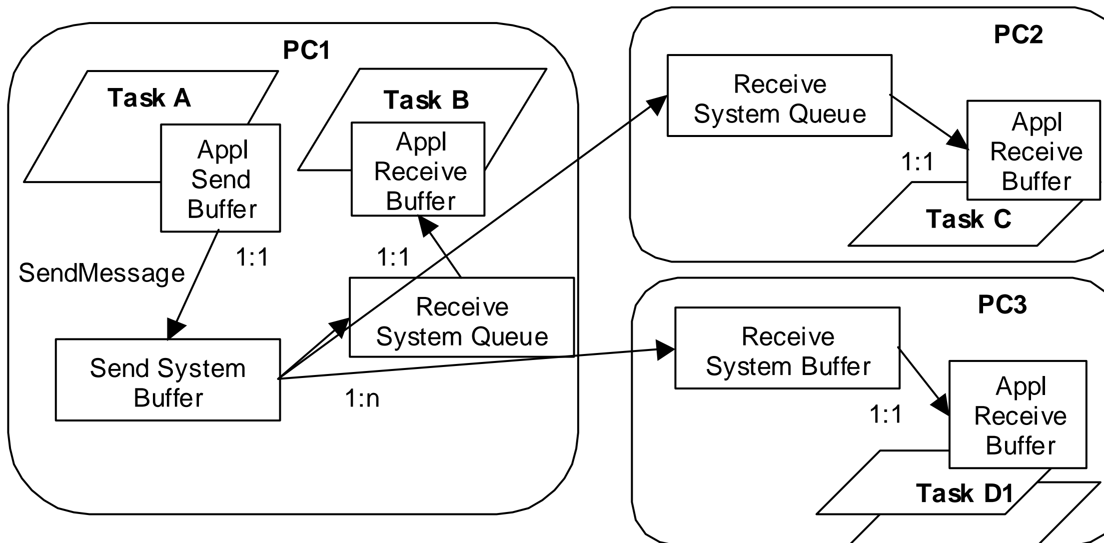

Figure 4: One-to-One and One-to-N Communication in OSEK COM Version 2.2.2

6.1.2 Local and Network Communication

Message communication semantics differ between local and network communication. For example, periodic transfer semantics are not available for local communication. Similarly, locally communicated queued messages can only have one receiver. OSEK COM Version 3.0 continues to have these semantic differences.

Such semantic differences drastically reduce application portability by limiting the ability to configure application components as tasks with different mapping to different hardware configurations. In order to achieve such portability, application developers would have to program to the least common denominator, thus severely limiting the service capability of message communication.

6.1.3 Message Communication Optimization

Efficiency concerns have led to variations of message communication concepts that increase the complexity of the API application developers have to deal with. In OSEK COM Version 2.2.2 it takes the form of specifying the option of operating with application message buffers (OSEK Accessors) that map directly into the system buffer (i.e., do not result in copying of message data). The WithoutCopy property can be associated with each OSEK accessor separately and is specified in OIL.

This feature is well intended. It is designed to allow application developers to express data interchange via messages while optimizing the implementation to a shared buffer solution. Unfortunately, use of this feature leaves the responsibility of coordinating concurrent access of this shared buffer to the application developer. If developers cannot assure mutually exclusive access statically, they are required to use a mutex lock service. In other words, the application code contains knowledge of this optimization although the WithoutCopy property is located in an OIL specification.

6.1.4 Message Communication Faults

OSEK COM provides several mechanisms for reporting error conditions that occur in message communication. Three major categories are 1) transmission errors, including transmission deadline errors, 2) message receipt errors, including message interarrival deadline errors, and 3) message queue overflow conditions.

The first two categories result in a task dispatch action, in the form of task activation, setting of event, or callback. It is up to the application developer to specify appropriate tasks to be dispatched and to communicate the fact that an error condition occurred.

Message queue overflow conditions are not considered transmission or receipt errors, nor do they result in an error notification on occurrence. Instead the receiving task is informed of the error condition when performing a ReceiveMessage call. Unfortunately, message queue

overflow is handled such that on one hand the last arriving message is dropped, on the other hand the error status is reported when the most recent message is retrieved (i.e., a message that is a queue length away from the dropped message). Thus, if the receiver application wanted to perform error handling it would have to track how many messages to receive before it encountered a gap in the message sequence due to the drop. This could be simplified by dropping the oldest message when message queue overflow occurs.

6.2 Message Services: A Moving Target

OSEK COM Version 3.0 has made progress and improvements in several areas. It has improved support for message initialization and has added message-filtering support. It has also made attempts to improve support for m:n message communication and in communication of control events.

With respect to message initialization, OSEK COM 3.0 specifies default initial values, and the ability to declare initial unqueued message values in OIL.

6.2.1 Message Filtering

In OSEK COM Version 2.2.2 message filtering capabilities were hidden in the mixed transmission mode of message transfer as well as in the message receipt notification mechanisms. In OSEK COM Version 3.0 message filtering has become its own mechanisms.

Application developers are able to specify conditions under which the message should not be transferred. These conditions can be in terms of message data content or in terms of message instance occurrence counts. Unfortunately, message filtering has not been extended to support the newly introduced zero length message concept (see Section 6.2.4).

6.2.2 Message Communication Model Revisited

OSEK COM Version 3.0 unnecessarily exposes the application developer to another layer of mechanisms. It does so by exposing message transmission containers, requiring specification of packing and transmission conditions in OIL (triggered and pending transfer), and by making the packing and transfer semantics dependent on the SendMessage call order of messages to be packed and transmitted together. In other words, application code affects the semantics of message transmission.

The suggested use of this feature is that a task can send data messages with pending transfer property to accumulate all the data to be communicated. To indicate a request for actual communication, the application issues a SendMessage, typically a zero length message, with a triggered transfer property to cause the actual transfer to occur [COM 00, v.3.0 p. 59]. Such a request must be made for every message transmission container. In other words, if the set of

messages spans more than one container, several messages with trigger transfer property must be issued.

The same effect can be achieved by requiring the application developer to package multiple logical messages into a single structure. However, this may cause the message to grow to a size that cannot be fit into a single transmission container—a requirement of OSEK COM 3.0.

Alternatively, OSEK could adopt an approach successfully used in MetaH and the emerging SAE AADL standard. In this approach, application components follow a simple input-compute-output model. Upon dispatch, all inputs are available in application message buffers (called ports). An application component performs its computations and makes its output available in application message buffers. Upon dispatch execution completion, communication of data is initiated, either through periodic or direct transfer. Note that packing of messages into message transmission containers is not exposed to application developers. Note also that initiating the communication of a set of messages does not require a separate system call. Finally, note that movement of data between application message buffers and system message buffers/queues occurs at well-defined points in time. This eliminates non-deterministic unqueued message send and receipt due to variation in actual task execution. This increased determinism simplifies data consistency management and provides opportunity for message communication optimization by reducing double buffering needs. Note that such optimization can be performed as part of analysis of OIL specifications when generating a runtime executive that contains the appropriate dispatch and communication code tailored to individual application tasks.

6.2.3 Multiple Senders and Receivers

The semantics of triggered and pending transfer properties gets more confusing for application developers. Note that multiple tasks on the same machine can send to a send message buffer, and can share message transmission containers. This model is sketched in Figure 5 for Tasks A1 and A2, showing the message container as a dashed box.

In general multiple sending tasks can submit their messages to system buffers via SendMessage calls. Messages in a message transmission container are not transferred until a message with a triggered message property is submitted. At that point the container (i.e., all messages contained in the container) is transferred. Messages are then transferred to multiple receiving system buffers or queues.

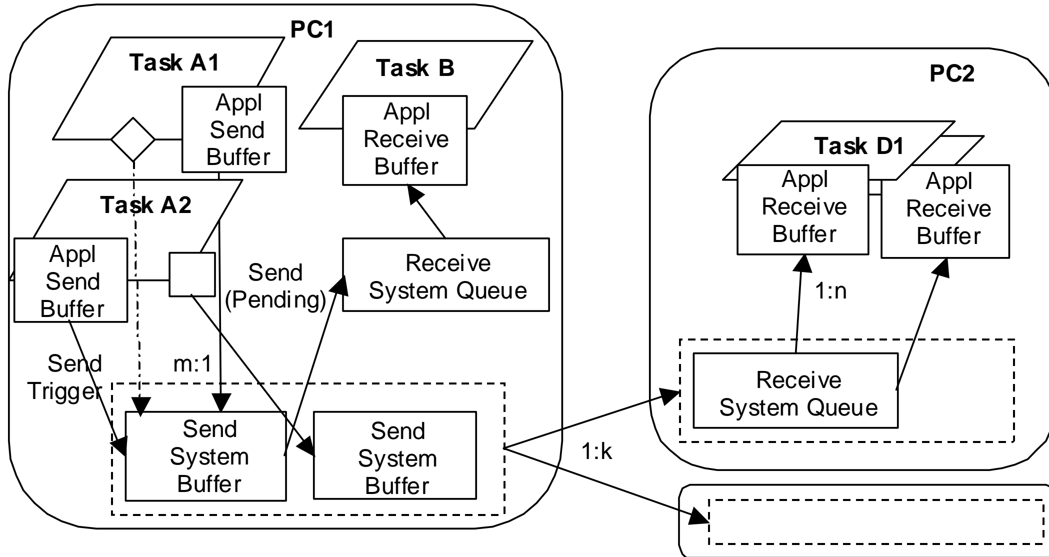

Figure 5: OSEK COM Version 3.0 Communication Model

Multiple tasks can receive from a Receive Message Buffer or Receive Message Queue via ReceiveMessage calls. In the case of Message Buffers, all receivers see the same message instance until the next message instance arrives. Note that the ReceiveMessage calls are performed as part of the application code at the application priority (i.e., the occurrence in time of message copying into the application message buffer is dependent on the preemption behavior of the task). In case of a Receive Message Queue only one task receives a particular message (i.e., multiple tasks can service a queue). OSEK COM Version 3.0 states that it is the responsibility of the application to ensure data consistency when accessing the queue (i.e., applications are expected to enclose such ReceiveMessage operations with a non-preemption region) [COM 03, p. 59].

While multiple tasks can perform ReceiveMessage calls on the same buffer or queue, only one task can be dispatched when a message enters the Receive Message Queue or Buffer, and the task has to be statically known. Thus, multiple tasks servicing a queue cannot be dispatched by the receipt notification mechanism.

Message receipt notification occurs when a message enters the Receive Message Queue or Buffer. If the notification action is a task dispatch request, this request is queued in the dispatch request queue—in addition to the message data being queued in the Receive Message Queue (i.e., two data structures must be maintained at runtime).

Message notification actions include setting of a flag. This flag seems to be a separate object that is intended to be associated with Send Message Buffers and Receive Message Buffers and Queues. This flag can be set every time a message instance is transmitted or received. Note that this is a binary flag that is associated with the message object, not the

communicated message instance. Thus, it is not possible for the receiving task to discern status information for specific messages. Furthermore, each notification class requires its own flag.

These flags are automatically reset when a SendMessage or ReceiveMessage call is performed. This results in a race condition. A task may check the flag value with a GetFlag call and then perform a SendMessage, ReceiveMessage, or ResetFlag call. Unfortunately, a new message may arrive after the GetFlag call, causing a flag to be set and immediately reset without the resetting task being aware of that fact. The OSEK OS specification is silent on the issue of preemption of tasks or ISRs by message arrival in the underlying network layer.

Note that the feature of message send and receive without copy has been removed in OSEK COM Version 3.0. This will lead application developers to stay with communication through global variables.

6.2.4 Event Communication Through Zero Length Messages

Messages can be used to communicate control events. In OSEK COM Version 2.2.2 this is done as follows.

Messages with a dummy content act as control events that can be used as a queued or unqueued message. If they are used as unqueued messages, the receipt notification action results in queuing of the dispatch request. Note that dispatch request queuing is bounded for direct activation and is limited to size 1 for dispatch via event. If used as queued message, the control events are queued in the message queue. In addition the dispatch requests are queued in the task dispatch request queue. Again limitations apply to queuing of dispatch requests. Once dispatched the application code removes the control event message through a ReceiveMessage.

OSEK COM Version 3.0 tries to address communication of control events directly by introducing the concept of a zero length message. This concept plays a dual role in terms of control flow: it can be used to initiate transfer of messages; it can also be used to represent a dispatch request.

A SendZeroMessage call always triggers transfer of the message container this message is contained in. No data is transferred for a zero length message. No message filtering can be applied, although one of the filtering conditions is based on message occurrence and is independent of message data content.

Zero length messages are not queued at the receiving end. However, receipt notification action results in a dispatch request in the form of activation of a single task or setting of a single event, which may result in dispatch request queuing. This results in the same dispatch request semantics and shortcomings as found in OSEK COM Version 2.2.2.

Use of zero length messages is further constrained by the fact that it is only supported by the most inclusive message communication conformance class (CCC1).

Again OSEK could adopt an approach chosen by MetaH and the emerging SAE AADL. Events are messages that represent control flow in the form of dispatch requests. These dispatch requests are communicated through event ports. The event port connection specified in an OIL-like notation indicates the flow of these dispatch requests (i.e., the task interaction topology is localized in OIL). Tasks wanting to issue a task dispatch can place such a dispatch request in their event port. The dispatch request is initiated at the dispatch execution completion of the requestor—as is any data communication. An incoming event port of a task may be queued, i.e., represent the dispatch request queue. It may also have additional properties indicating whether the currently executing dispatch should be aborted or completed before the next dispatch request is serviced. A task may have multiple incoming event ports indicating that it may be dispatched by any of the events and may have associated different code sequences for different event ports. Such an approach retains the simplicity of an input-compute-output model for application code. Furthermore, application code can be viewed as a plug-in that fits into the runtime executive generated from an OIL description (see Section 7).

7 System Modeling and OIL

The OSEK Implementation Language, OIL, is a notation to support the specification of components of the task and communication architecture of automotive applications. Its primary purpose has been to specify the configuration of OSEK-based applications and runtime support and to allow for appropriate configuration tables to be generated. As a result the focus of OIL has been to support the specification of OSEK components that make up an OSEK-based application, such as tasks and ISRs, message objects (message connections), alarms, events, and resources.

OIL, as it currently stands (OSEK OIL Version 2.3), satisfies that need, but has a number of shortcomings as a task and communication architecture specification notation. It has not been successful in extracting all task and communication information into a specification. For example, OIL does not provide a complete specification of task interaction. While task interaction via events, resources, and messages is recorded in OIL, interaction via explicit task activation is not. Furthermore, information about the task interaction topology is recorded in OIL and in application code in the form of explicit task activation, and setting of events requires the destination task to be specified as a system call parameter.

OIL is limited to specifying the collection of OSEK objects that make up an application system residing on a single processor. Distributed application systems that span multiple processors are described by multiple OIL specifications. Any changes in the allocation of application system tasks to processors require deletion of the task object in one OIL specification and addition in another. In other words, the OIL specification does not support an explicit way of indicating the binding of software components (tasks) to different hardware, the specification of the hardware architecture, or the communication between tasks on different processors. Task binding can be changed by moving task specifications between OIL specification files, each of which represents a single processor.

As embedded real-time systems increase in complexity and as they get integrated into large-scale distributed systems, developers must increasingly deal with system-level issues, such as system timeliness and responsiveness, system throughput, system reliability and availability, and system safety. These system-level properties are characterized by the appropriate component-level properties and are driven by the particular choices in the task and communication architecture of the software system and its binding to a particular execution platform (hardware and infrastructure software).

OIL currently does not support schedulability analysis for several reasons. Its description of task interaction is incomplete, that is, not all control flow in the form of task dispatches is described. The set of relevant task properties is incomplete. For some tasks the dispatch

period can be inferred from OSEK alarm properties in OIL, but worst-case execution time and deadline information is not provided. Furthermore, some of these properties can be changed by application code through system calls (task periods by changing the OSEK alarm settings). These shortcomings can be corrected to grow OIL into an architecture description language (ADL). As such it could provide system architecture specifications that are formally analyzable with respect to key system properties such as schedulability and performance.

OSEK OIL could benefit from the experience with other ADLs, in particular MetaH and the emerging SAE AADL standard. The focus of these ADLs has been on modeling and analysis of software and system architectures. They support schedulability, performance, safety, and reliability analysis, as well as modeling and analysis of component interaction behavior, for example, in the form of interaction protocols. In the latter case model checking techniques can be applied to validate such protocols against expected system behavior.

Such an ADL can also be the basis for system generation and build. Such a generation capability can not only generate the runtime system configuration components, but also reflect the system architecture in a generated runtime executive that is tailored to the application system at hand. Runtime efficiency is gained in two ways. First, the generated runtime executive code is tailored to the specific statically known task interaction topology. Second, the generator can optimize the runtime executive code by choosing the most efficient system services to accomplish the specific task interaction on a case-by-case basis. The feasibility of this approach has been demonstrated by MetaH.

8 Summary

In this paper we have examined the OSEK OS, OSEK COM, and OSEK OIL specifications from the perspective of a real-time application developer. The OSEK OS specification has contributed to the portability of embedded automotive applications across different real-time operating system implementations, but has limitations. Note that we have focused our examination on the OSEK standard itself. Several commercial products based on the OSEK standard exist, but have not been considered in this document.

OSEK OS and COM present a complex set of system services through its API. The application developer is exposed to a wide range of mechanisms with small but critical variations in semantics (e.g., non-preemption and task dispatching/scheduling).

This complexity and the need for a small runtime footprint lead to application development with the most restrictive conformance class (BCC1). This conformance class supports task dispatch by direct activation and no dispatch request queuing. Concurrency management support is limited to a single explicit resource with non-preemption of all tasks and two (internal) resources with implicit acquisition and release. Similarly, communication can be limited to the most restrictive conformance class (CCCA) for local communication and its distributed counterpart (CCC0). Message communication can be used to avoid embedding of task interaction topology in application code. Unqueued message communication can also be used as an alternative to task communication via shared data whose concurrent access is coordinated by OSEK resource acquisition.

We have identified shortcomings in the OSEK alarm and event mechanisms. A single alarm expiration action and a single alarm minimum requirement leads application developers to code their own task dispatching, either as

- an application task that implements a dispatcher, or
- by embedding dispatch ordering of tasks in the same rate group in ChainTask calls – trading a small real-time OS footprint for increased application size and complexity

Shortcomings in the event mechanism bring potential for starvation in heavy-load conditions due to a) different dispatch semantics when events are pending, b) the queuing of events, and c) the non-consumable nature of events. The shortcomings can be corrected by changing the WaitEvent semantics to always place the task at the end of the priority-level ready queue, to support event queuing (in the form of a counter), and to make events consumable items, in essence moving toward the notion of an event message (zero length message).

OSEK COM is a moving target with major changes in its communication model between successive releases of the standard specification. Task initialization has been simplified, message filtering introduced as a separate concept, an attempt made towards improved control flow communication (zero length messages), and message packaging for transmission introduced.

Zero length messages get mapped into task activation and event setting actions with varying semantics for task dispatch. A simpler solution is to introduce the notion of a dispatch event (request) message. This message is communicated through an event message object, can be queued at the receiving event message object, and results in dispatch of the task by consuming an event message as appropriate. This model corresponds to an event model found in MetaH and the emerging SAE AADL standard.

The intent of message packaging is to provide more efficient transfer of multiple messages between tasks. Unfortunately, in its current instantiation of OSEK COM 3.0, OSEK message call sequences affect the packaging semantics (i.e., when specifying packaging properties in OIL these call sequences must be known). The message packaging and transfer semantics could be conceptually simplified by using a dispatch/input-compute-output/complete model of computation for application components. This model is similar to the OSEK basic task concept and allows for message communication to be associated with task dispatch and completion, which are well-defined scheduling points. MetaH has demonstrated how this model can be used to automatically generate a tailored runtime executive with service calls that perform message packaging for transfer and avoid unnecessary context switches. Task interaction topology information from the OIL model is used to choose efficient local and non-local dispatch and communication services. Application component code is free of system calls and task interaction information. It acts as a plug-in to the generated executive through a simple application message buffer and awaits dispatch API.

In addition, OIL can be the basis for system analysis with respect to a number of system properties relevant to the automotive domain, including timing, performance, safety, and reliability. With extensions, OSEK OIL has the potential of becoming an ADL that supports modeling, analysis, and generation of embedded application systems.

The time is right for going beyond OS portability, as embedded real-time applications are becoming integrated components of larger systems that operate on a common distributed execution platform. With the integration of embedded real-time system components into larger distributed systems, development focus shifts to a system architecture view of concurrent tasks, their interaction, and their binding to execution platforms. This view is necessary to address performance-critical aspects of the application system such as timeliness and fault tolerance. Key to the development and maintenance of such application systems is separation of concerns and conceptual simplicity. Separation of concerns means separating the application code focus of an automotive application engineer from the task and communication architecture focus of a software system engineer. Conceptual simplicity is key to

system understanding and to the ability to analyze a system early in its design with respect to key system properties such as timing, performance, safety, or reliability.

A number of community efforts are underway to address these issues. ISO/IEEE POSIX is an example of interface standards whose API specification addresses real-time concerns. Real-time Specification of Java is an effort to enhance an implementation language with predictable and analyzable real-time capabilities [RTJ 01]. The Object Management Group (OMG) recently addressed real-time support in the UML design notation [OMG 02]. Standard task and communication architectures have emerged in different embedded system domains; for example, Time-Triggered Architecture (TTA) in the automotive domain, and time partitioning (ARINC653) in integrated modular avionics systems. Modeling notations to describe software tasks and communication architectures and their mapping to a hardware configuration that supports predictive analysis and automatic generation are emerging as standards; for example, OSEK OIL, and Society of Automotive Engineers (SAE) Avionics Systems Division Architecture Description Language standard (AS-2C Subcommittee) [AADL 03]. OSEK has the opportunity of leveraging these efforts to improve its capabilities and services.

References

URLs are accurate as of the publication date of this paper.

[AADL 03] Society of Automotive Engineers (SAE) Avionics Systems Division (ASD) AS-2C Subcommittee. "Avionics Architecture Description Language Standard." Draft v0.8. 2003. Email: bruce.lewis@sed.redstone.army.mil

[COM 00] OSEK Communication Specification Version 2.2.2 <http://www.osek-vdx.org/mirror/com2-2-2.pdf> (2000).

[COM 03] OSEK Communication Specification Version 3.0 Release candidate 2 <http://www.osek-vdx.org/mirror/com301.pdf> (2003).

[Honeywell 98] Honeywell Technology Center *MetaH User's Manual*, Version 1.27. Minneapolis, MN < http://www.htc.honeywell.com/metah/index.html> (1998).

[Klein 93] Klein, M.; Ralya, T.; Pollak, B.; Obenza, R.; & Gonz, M. *A Practitioner's Handbook for Real-time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*. Boston, MA: Kluwer, 1993.

[Lehoczky 97] Lehoczky, J. P. "Using Real-Time Queueing Theory to Control Lateness in Real-Time Systems." *Performance Evaluation Review 25*, 1(1997): 158-168.

[Lemieux 01] Lemieux, J. *Programming in the OSEK/VDX Environment.* Gilroy, CA: CMP Books, 2001.

[OIL 01] OSEK Implementation Language Specification Version 2.3 <http://www.osek-vdx.org/mirror/oil23.pdf> (2001).

[OMG 02] Object Management Group "UML Profile for Schedulability, Performance, and Time Specification."<http://www.omg.org/docs/ptc/02-03-02.pdf> (2002).

[OS 01] OSEK Operating System Specification Version 2.2 <http://www.osek-vdx.org/osekvdx_OS.html> (2001).

[RTJ 01] Real-Time for Java™ Expert Group "Real-time Specification for Java." <http://www.rtj.org/> (2001).

[Vestal 93] Vestal, S. & Binns, P. "Scheduling and Communication in MetaH," 194-200. *Proceedings on Real-Time Systems Symposium*. Raleigh-Durham, NC, Dec. 1993. New York, NY: IEEE, 1993.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. **AGENCY USE ONLY**
   (Leave Blank)
2. **REPORT DATE**
   November 2003
3. **REPORT TYPE AND DATES COVERED**
   Final
4. **TITLE AND SUBTITLE**
   Real-Time Application Development with OSEK
   A Review of the OSEK Standards
5. **FUNDING NUMBERS**
   F19628-00-C-0003
6. **AUTHOR(S)**
   Peter H. Feiler
7. **PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**
   Software Engineering Institute
   Carnegie Mellon University
   Pittsburgh, PA 15213
8. **PERFORMING ORGANIZATION REPORT NUMBER**
   CMU/SEI-2003-TN-004
9. **SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)**
   HQ ESC/XPK
   5 Eglin Street
   Hanscom AFB, MA 01731-2116
10. **SPONSORING/MONITORING AGENCY REPORT NUMBER**
11. **SUPPLEMENTARY NOTES**

12A **DISTRIBUTION/AVAILABILITY STATEMENT**
Unclassified/Unlimited, DTIC, NTIS

12B **DISTRIBUTION CODE**

13. **ABSTRACT (MAXIMUM 200 WORDS)**

OSEK is an abbreviation for a German term that translates to "open systems and the corresponding interfaces for automotive electronics." OSEK OS is the operating system specification and OSEK COM is the communication specification. Both are application program interface (API) standards for automotive real-time application development. They are complemented by OSEK Implementation Language (OIL), a modeling language for describing the configuration of an OSEK application and operating system.

This paper covers the SEI evaluation of these standards from the perspective of real-time application development. The SEI identified shortcomings in the description and semantics of certain services offered by the OSEK API. These shortcomings introduce unnecessary complexity to application developers and limit application portability. The SEI also identified the potential of OIL as an architectural modeling language to support design-time analyses, such as schedulability analysis. OIL's potential as a basis for generating both real-time OS data tables and an application runtime executive was examined. Utilizing OIL in this way simplifies application component development. Correct use of OSEK API functionality is then relegated to a generation tool that operates on OIL. Such improvements would facilitate practitioners' adoption of OSEK by reducing its perceived complexity.

14. **SUBJECT TERMS**
    OSEK, OIL, API, standards, real-time, design-time analysis, schedulability analysis
15. **NUMBER OF PAGES**
    45
16. **PRICE CODE**

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL