Migrating a OSEK run-time environment to the OVERSEE platform

Andreas Platschek
OpenTech EDV Research GmbH
Augasse21, A-2193 Bullendorf
andreas.platschek@opentech.at

Georg Schiesser
OpenTech EDV Research GmbH
Augasse21, A-2193 Bullendorf
georg.schiesser@opentech.at

Abstract

As virtualization techniques are being used in the automotive industry, in order to save hardware, reduce power consumption and allow the reuse of legacy applications, as well as allow the fast development and integration of new applications, the need for a run-time environment that is suitable and in wide use in the automotive industry emerges. The requirements for such an run-time environment are defined in the most widely used specification in this industry - OSEK/VDX.

One key feature the OVERSEE project is taking advantage of, is that co-locating a OSEK run-time environment and a full-featured GPOS GNU/Linux eliminates many limitations of OSEK/VDX by the extension through virtualization and notably allowing to mitigate some of the serious shortcomings in the security area by resolving these issues at the architectural level rather than trying to patch up the limited OSEK OS. This may well constitute a general trend to specialize operating systems and operate powerful hardware as an assortment of specialized FLOSS systems collaborating to provide different services, including full backwards compatibility to legacy operating systems.

Currently, several FLOSS implementation of this specification are available under different FLOSS license models and with a different degree of compliance. This paper gives an overview of the available implementations, a rational for the chosen implementation as well as a description of the efforts for the migration to XtratuM.

1 Introduction

In the effort to reduce costs by saving hardware and reuse of legacy code, the automotive industry is relying on well specified and standardized operating systems. The OSEK specification (Open Systems and the Corresponding Interfaces for Automotive Electronics) has been around since 1993 and after merging with the VDX (Vehicle Distributed Executive) it has grown to the most important operating system specification in the automotive industry.

OSEK's main goals are to specify an operating system that is suitable for the automotive industry, and that allows to write highly portable applications which only depend on an OSEK compliant API. Furthermore the OSEK communication specification provides a well specified API for internal as well as external communication, turning OSEK compliant operating systems into highly portable, scalable operating systems that support re-usability of legacy OSEK compliant applications.

Although it's successor [7] is going to be the future of the industry, OSEK/VDX will be around for quite some time, since it is the basis for AUTOSAR:

"The OS shall provide an API that is backward compatible to the API of OSEK OS. New require-

ments shall be integrated as an extension of the functionality provided by OSEK OS." [*BSW097*] *Existing OS, AUTOSAR, Requirements on Operating System V2.1.0 R4.0 Rev 1*

All this explains, why support for an OSEK compliant run-time environments is an indispensable requirement for a software platform - like the one developed in the OVERSEE [1] project - that targets the automotive industry. A high level view of this software platform can be seen in figure 1.

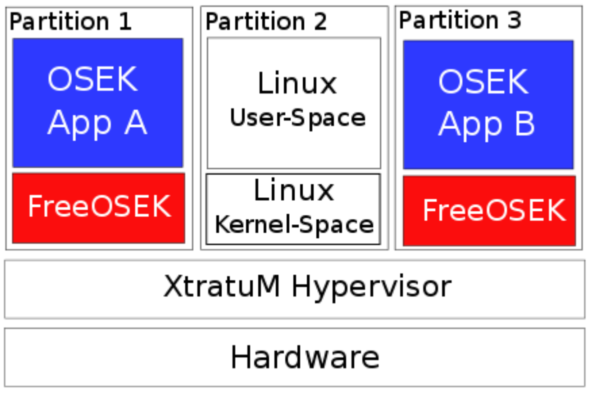

FIGURE 1: *High Level Architecture*

This paper will give an introduction to the OSEK/VDX operating system specification, and describe the efforts that were necessary to allow the execution of FreeOSEK [2] a FLOSS implementation of OSEK/VDX in a virtualized environment, namely the XtratuM hypervisor allowing to run several FreeOSEK run-time environments in parallel with other run-time environments like Linux partitions or LithOS [9] partitions, while guaranteeing the independence between those run-time environments.

2 OSEK/VDX

In the following, the open operating system specification OSEK/VDX [8] is summarized, looking at the highest conformance class ECC2 (extended conformance class). The lower conformance classes are subsets of ECC2, the relation between the conformance classes can be found in [4], Figure 3-3.

2.1 OSEK OS

The most important part of OSEK/VDX to understand the context of this paper is OSEK OS. It specifies a operating system, well suited for the needs of the automotive industry. The standardized API and well defined behavior of OSEK/VDX compliant operating systems, allow high portability of applications developed for such an operating system.

The following summarizes the essential points of OSEK/VDX, for more details, please refer to the homepage [4], where all parts can be downloaded free of charge, since it is an open standard.

Task Management OSEK/VDX distinguishes between two different types of tasks, basic tasks (BT) and extended tasks (ET). While a BT can only release the processor if it terminates, or if it is preempted by a higher priority task or an interrupt service routine (ISR), an ET can also go into a *waiting state*, allowing the scheduler to dispatch a lower priority task, without terminating the higher priority task. An example for this would be, if the ET is waiting for some kind of event to happen. Instead of just polling and wasting CPU time, it can go into the waiting state, in this state it is not scheduled, before the event is signaled (more on signals below).

OSEK/VDX provides a Task state Model ([4], section 4.2) that describes the states a task can be in, and the transitions between those tasks. The task state model for extended tasks is shown in figure 2. For basic tasks the task state model is essentially the same, but without the *waiting* state.

The states a task can be in are the following:

- **running** - a task in the running state is currently active and executed. At all times only one task can be in the running state. (OSEK/VDX is specified for single core CPUs only, multi-core solutions are covered by newer versions of AUTOSAR)
- **ready** - all schedulable tasks are in the ready state, waiting for their turn to transition into the running state.
- **suspended** - tasks in the suspended task are currently inactive and wait for their activation to become ready.
- **waiting** - extended tasks that are waiting for some event to happen can decide to go into the waiting state instead of wasting CPU time. A task in the waiting state will be *released* from the waiting state as soon as the desired event has happened.

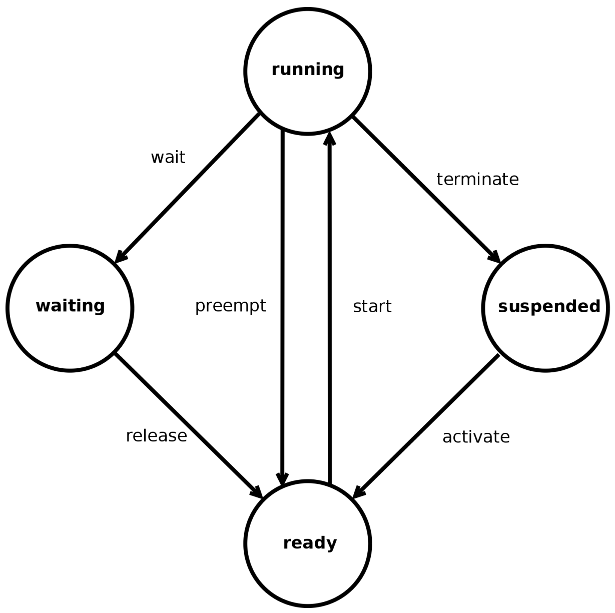

FIGURE 2: *OSEK Task State Model*

In the OSEK/VDX task management, the scheduling policy is assigned by the system integrator. A systems scheduling policy can be configured to be fully preemptive, non-preemptive or mixed (both preemptable as well as non-preemptable tasks are running at the same time). The scheduling decision itself is based on priority scheduling, with static priorities (where 0 is the lowest priority and bigger numbers denote higher priorities). Depending on the conformance class, one or more tasks of the same priority can exist at the same time. If the system is configured to allow preemptive tasks, a priority ceiling protocol is provided to prevent priority inversion.

If preemption is disabled, only voluntary preemption of tasks is possible, rescheduling happens only in the following cases:

- the *running* task terminates successfully
- explicit call of the scheduler by the *running* task
- the *running* task transitions into the *waiting* state

Interrupts - OSEK/VDX distinguishes between 2 types of interrupts:

- category1 ISRs do not use operating system services, and after they are finished, execution continues exactly at the point where it was before the ISR has been called (no influence on task management).
- category2 ISRs are allowed to use operating system services that are concerned with handling interrupts (enable, disable, etc.), these ISRs prepare the system for a RTE to run a dedicated user routine (comparable to Bottom Halves). After a category2 ISR has been executed, the execution does not return to the last point before the interrupt, instead the scheduler is invoked, in order to check if a dedicated user routine (bottom halve) has a higher priority than the current running task.

Depending on the scheduling policy, the point of rescheduling is either, when the event is *set* (fully preemptive) or at the next point of rescheduling in non-preemptive mode (listed above in Task Management).

Events are a means of synchronization. They are only available for extended tasks, since they are used to transition tasks into and out of the *waiting state.* Events are objects assigned to tasks, and uniquely identified through their name and the task they belong to. At task activation of an extended task, all the events are cleared automatically. Events can be set by any task (also basic tasks) as well as category2 ISRs, to change the task state of the events owner from the *waiting* to the *ready* state, but only the owner of the event is allowed to clear the event after-wards.

Depending on the scheduling policy, the point of rescheduling is either, when the event is *set* (fully preemptive) or at next point of rescheduling in non-preemptive mode (listed above in Task Management).

Resource Management In order to allow the concurrent task execution model described above, a resource management has to be provided, in order to assure

- mutually exclusive access to resources
- prevent priority inversion
- detect and prevent deadlocks
- and access to a resource must never lead to a transition into a *waiting* state

All these problems are high probable error sources, the goal of the OSEK resource management system is to do everything possible to prevent them from the operating system side. To reach these goals, the following mechanisms are specified by OSEK/VDX:

OSEK Priority Ceiling Protocol [4], section 8.5, introduces the OSEK Priority Ceiling Protocol, used to avoid priority inversion and deadlocks between tasks. This protocol provides a ceiling priority for each resource (this ceiling priority is statically assigned at system generation), which shall be set to priority of the highest-prior task using the resource.

If a task with a lower priority accesses the resource, it's own priority is risen to the resources

priority temporarily. After the task releases the resource, it's priority is set back to it's old priority. This way, it is not possible that the task is preempted by an higher prior task that competes for the same resource, while the lower prior task is holding the resource.

Section 8.6 of [4] introduces an optional extension of the OSEK Priority Ceiling Protocol, that includes ISRs.

Restrictions when using Resources OSEK/VDX defines restrictions on the system calls that may be used, while a task is holding a resource. The calls forbidden while holding a resource are *TerminateTask, ChainTask, Schedule* and *WaitEvent*. As can be inferred from the names, those calls that invoke the scheduler and might lead to the scheduling of another task are the ones prohibited while holding a resource.

This is a simple an effective way of assuring the mutual exclusivity of resources, furthermore it helps to prevent deadlocks between tasks.

Scheduler as a Resource If a task wants to prevent itself from being preempted, it can lock the scheduler. If a task chooses to do so, the scheduler is still invoked, but not allowed to schedule any other tasks. Interrupts are received and processed independently of the state of the scheduler.

Alarms are special (time-dependent) events, offered by the OSEK OS, to activate tasks after a counter has experienced. A counter in OSEK is represented a counter value measured in *ticks*, if the counter reaches a predefined value, the alarm expires and the alarm-event is set off. The predefined value can be specified either relative to the actual counter value (relative alarm) or as an absolute value (absolute alarm).

The counter value can be incremented by all kinds of sources, of course this could be a real-time clock, but it could also be any other interrupt source that increments the counter.

While any number of alarms can be assigned to the same counter, each alarm has exactly one counter and exactly one alarm-callback routine assigned at system generation time.

Error Handling OSEK/VDX defines *hook routines* which can be used for a variety of tasks.

Hook Routines are part of the operating system, although implemented by the applications developer. They can be seen as a possibility for the application developer to extend the functionality of the operating system. The *hook routines* are called by the OS at pre-configured events, which events depends on the implementation of the operating system itself. Since *hook routines* are part of the OS, they have higher priority than all tasks, and they can not be interrupted by category2 ISRs. While the interface for *hook routines* are standardized, functionality is not and is up to the application developer.

Error Handling OSEK/VDX distinguishes between two categories of errors - *application errors* and *fatal errors*. In case of a *fatal error*, the integrity of the operating systems internal data can no longer be guaranteed, and the operating systems shuts down. If an *application error* occurs, a system call could not be serviced properly, but the internal data of the operating system is still assumed to be correct. If a system service routine returns an error code, an *error hook routine* is called. This hook routine has to be provided by the user, who has the responsibility to bring his application back on track.

System Startup/Shutdown All low level (hardware) initialization is up to the application developer, the specifications of the OSEK/VDX concern only the platform independent parts and start with the call to *StartOS*.

Shutdowns are a little more complicated, since each task has to be informed of the shutdown, so it can bring potential actuators into a safe state. Therefore before the system can actually shutdown, a *shutdown hook* is called.

Debugging is done via a *PreTaskHook* and a *PostTaskHook*, which are called on task switches. These hooks can be used for debugging and measurement purposes.

Standardized API in [4], sections 12 and 13, the system services provided by the API of an OSEK/VDX compliant operating system are specified. This API must be the only way for the application to use the above described operating subsystems, like alarms, events, etc.

2.2 Other parts of OSEK/VDX

OSEK/VDX consists of multiple parts, OSEK OS described previously is the most important one for the porting efforts of FreeOSEK to XtratuM, while the other parts do not really play a role in this context (except for some sections of OSEK Com). But for completeness, here is a short list of all parts:

OSEK COM - Communication Layer specifies a message based communication for (inter processor) communication - it shows a stunning resemblance with ARINC653 interpartition communication, but describes a communication system for internal and external communication.

OSEK NM - Network Management provides a standardized way of configuring networks of OSEK/VDX nodes, initialization of networking peripherals, network start-up, network monitoring and a lot more, everything that is needed to start, maintain and diagnose a network of nodes running OSEK/VDX compliant nodes.

OSEK OIL - OSEK Interpretation Language specifies a standardized configuration mechanism for OSEK/VDX compliant nodes. The configuration files as defined by OSEK OIL are per node (single CPU nodes only), and do not include network configuration.

OSEK Time - Time-Triggered OS specifies a time-triggered variant of OSEK VDX, the differences are e.g. time-triggered scheduling and the like. It is also possible to run a mixed variant, were a standard OSEK OS is run in time slots of the time-triggered OS.

OSEK FTCom - Fault-Tolerant Communication provides a standardized time-triggered networking variant that in order to achieve better fault-tolerance than with the standard OSEK/VDX networking layer.

2.3 AUTOSAR

While OSEK/VDX is currently the most used operating system standard in the automotive industry, its successor AUTOSAR [7] is on it's way to take over. The main reason for this is definitely the fact that the newest release - AUTOSAR 4.0 - is the first operating systems standard taking multi-core CPU's into account. Since the days of single-core CPU's are counted, this a real important topic that will shake the safety-community over the next years.

For the OVERSEE project, AUTOSAR is investigated, but not strictly followed, the reason is simply it's size and the fact that AUTOSAR is based on OSEK/VDX, so every application written for an OSEK/VDX compliant operating system can also be executed on a AUTOSAR compliant operating system. Nevertheless OVERSEE's design decisions are loosely based on the AUTOSAR architecture.

3 XtratuM

XtratuM is a type II (bare metal) hypervisor targeting safety related composable systems. The main guidelines for design come from one of the key IMA standards, ARINC 653 [3]. XtratuM is an active FLOSS project being developed at Instituto de Informatica Industrial, Universidad Politecnica de Valencia. While the OVERSEE project is focused on security aspects the goal is to provide a platform that in principle can also satisfy safety requirements. There is a strong sharing of core demands on the lowest OS layer with respect to safety and security, and while safety and security have sometimes conflicting demands at higher levels these differences are not present at the lowest level of a hypervisor [10]. The key to unify the requirements at the lowest level of safety and security is to provide a sound:

- Temporal isolation
- Spatial isolation

allowing to build high-level services on top that only allows explicitly permitted sharing of resources as well as communication. XtratuM thus is intentionally reduced close to the bare minimum that is needed to allow high-level services to operate in there respective OS environments and still give strong guarantees with respect to independence.

3.1 XM Hypercall Interface

XtratuM offers a relatively narrow interface of Hypercalls to it's partitions. This simplified things a lot for our porting efforts. In this section we will only briefly outline hypercalls that were used in this porting effort, for a full list of available hypercalls we refer you to the XtratuM Reference Manual [11] The intention of this section is to show the interface size used in the XtratuM guest management for a actual example.

- Time services: XtratuM provides an independent virtual time to each domain on which the guest-OS then can implement high-level timing services. In this sense the low-level services can be seen as mimicking hardware timing services.
  - XM_get_time: Time entities in XtratuM are of microsecond granularity, and are maintained relative to the last system reset. There are two basic clocks in the system. Clocks in XtratuM are strictly monotonic. Clocks are maintained for the system (XM_HW_CLOCK) as well as for the partitions execution (XM_EXEC_CCLOCK)
  - XM_set_timer: Interval timer service (providing one-shot behavior by setting the interval to 0). The expire time is an absolute time with respect to either hardware clock or execution clock. To a partition the expired timer is signaled as a virtual timer interrupt (emulating a hardware timer).
- Interrupt services: Signaling to partitions is provided via virtual interrupts, it is up to the guest-OS to then assign suitable meaning and response to the events. Note the absence of a interrupt request hypercall - as all resources are allocated statically in XtratuM there is no need for a request_irq.
  - XM_enable_irqs: globally disable interrupt delivery to this partition
  - XM_disable_irqs: globally enable interrupt delivery
  - XM_set_irqmask: used for masking (blocking) and unmasking of interrupts
- Basic partition management functions: Much of the partition management is related to the initialization and shutdown phase of a partition. The essence of the interface is that it minimizes the state information that needs to be handled by the hypervisor - leaving more or less all state related work to the partition.
  - XM_suspend_partition: This is a basic function that is only used in supervisor mode to manage a partition. It is used to block a partition (waiting on a resource) or temporarily stop a partition if errors are detected.
  - XM_resume_partition: Simply the opposite to the above partition suspension.
  - XM_shutdown_partition: As the hypervisor does not have information about the internal state of a partition shutdown is provided as an asynchronous notification. Basically a partition is sent a request to shut down via a dedicated interrupt and after cleaning up any internal state will then terminate it self.
  - XM_reset_partition: Conversely to the XM_shutdown_partition, the XM_reset_partition is a forced shutdown of a partition whereby a warm and cold reset is differentiated, a warm reset preserves some of the partitions initialized resources (i.e. open ports and memory areas) while a cold reset clears this all and thus can have side-effects on other partitions via communication channels no longer being served.
  - XM_halt_partition: A halted partition is set into an inactive state but no reclamation of resources (spatial or temporal) are done (that is left to the partition reset) in this state the partition is simply no longer scheduled by the hypervisor. The XM_halt_partition called by non-supervisor partitions can only pass self as the target of the halt.
  - XM_idle_self: This allows a partition to suspend it self within its time slot. The partition will only be re-woken on its next time-slot or if a NMI is received within its current time slot. This can be used to implement donation schemes for system partitions.
- Basic system management functions: Note that these are not directly related to the guest-OS as these calls are related to privileged domains - they are listed here for completeness.

  - XM_halt_system: The halt partition call (also described above) is used by system partitions to manage the system as a whole as well as individual partitions. Only supervisor partitions can halt other partitions. This is used to prepare a partition reset as well as mode switching.
  - XM_reset_system: Brute force system halt of the entire board after this only a hardware reset can reboot the system. No precautions are taken to put any partition into a sane state thus this is only the last step in a system shutdown as well as in extreme emergency situations.

- Low level Communication related functions: In practical implementations one does not actually use the low level object class functions but uses the wrappers provided to the commonly used objects (sampling and queuing ports as specified in ARINC 653). These wrappers thus are the actual hypercalls that will be issued though they are rarely used in guest-OS code.
  - XM_read_object: read the object, verifying access permissions and other low-level properties. Usage in all reading functions like XM_receive_queuing_message, XM_read_sampling_message, etc.
  - XM_write_object: write the object. This is used i.e. in XM_write_sampling/queuing_message, XM_send_queuing_message.
  - XM_ctrl_object: is used to create and manage objects with specific properties as well as query these objects (i.e. retrieve the id of the object). This hypercall is used in object management functions like XM_create_sampling/queuing_port, XM_get_sampling/queuing_port_status, etc.

While the overall hypercall set is a bit more elaborate than listed here, the essential calls used to implement the OSEK guest-OS are listed showing how small such a guest-OS interface actually can be constructed if the abstraction level is pulled down far enough. A full description of the interface is out of scope for this paper though.

4 FreeOSEK

FreeOSEK[2] is a OSEK implementation started by Mariano Cerdeiro. It originally ran on ARM and on POSIX compliant platforms (this is just a simulation environment, running FreeOSEK as a user-space process intended to allow everyone to test it on a normal Linux desktop).

FreeOSEK is licensed under the GPLv3 with link exception. This means, that you can link your code into FreeOSEK and can still license your code under whatever license you want (free or proprietary).

According to the FreeOSEK homepage, they currently run about 80% of the OSEK OS conformance tests, and of those about 95% pass. In addition, FreeOSEK is tested, using the static code checking tool splint.

Fortunately big parts of FreeOSEK are generic C-code (e.g. the task scheduler) and only the parts that directly deal with hardware had to be adapted (see section 5 for details).

While OSEK OS is almost complete, OSEK Com is more or less non existent in FreeOSEK, but this is no big problem for us, as we will see later in section 5.4, since most of the functionality needed for OSEK Com compliant communication is already provided by XtratuM.

5 Porting Efforts

The following section describes the efforts that have to be taken to run FreeOSEK as an run-time environment in a XtratuM partition.

This includes also a description of which steps already have been achieved successfully, and gives insight into the parts that will need more work. To anticipate the most important thing first: As of this writing, FreeOSEK can be used as an XtratuM runtime environment, but more work will be needed to make a full compliant version possible, most notably in the task management and communication subsystem some (re)work will be necessary.

5.1 Adaptation of the Build System

The first step to running FreeOSEK inside of an XtratuM partition, was to adapt FreeOSEK's build system, so that the resulting binary would be accepted by XtratuM. The most important thing here is, that FreeOSEK must not be compiled as an executable binary, but instead it has to be compiled as an relocatable object, that can be linked into an XtratuM partition - if necessary even in multiple partitions - at a memory address that is specified at configuration time in the XtratuM configuration file.

After this stage it is already possible to boot into FreeOSEK, and to put some `xprintf`'s[1] into the init code. Since most of the initialization code is generic (e.g. load the data of the application's task) this is already done without any changes to the FreeOSEK code base. The next point that really needed attention, was the x86 specific code for the task switches.

5.2 Task Management

In order to assure a flawless scheduling of tasks, it has to be assured,that for each possible point of rescheduling, the transition from the old to the new task is done properly.

Which actions have to be performed during dispatching, depends on the the event that led to the rescheduling - that is on the point of rescheduling itself.

OSEK OS lists the following 4 points of rescheduling for non-preemptive scheduling:

- Task Termination
- explicit activation of successor task
- explicit call of the scheduler
- a transition into a waiting state takes place

Let's have a quick look at those four points of rescheduling. The first two can be handled really easily, for those two, the task context of the old task does not have to be saved, since it terminates, before the new task is scheduled. Therefore, all that was needed to get a basic version of FreeOSEK running on XtratuM, was to set the stack pointer to the stack of the new task, and jump into task itself. This way, simple examples that activate non-preemptive tasks, and chain non-preemptive tasks can already be run.

If preemptive scheduling is desired, the following extended list of points of rescheduling has to be considered:

- Task Termination
- explicit activation of successor task
- activation of a task at task level
- explicit call of the scheduler
- a transition into a waiting state takes place
- setting a task to a waiting state
- release of a resource at the task level
- return from interrupt level to task level

In order to allow preemption of tasks (either voluntarily by going int o waiting states or involuntarily by hitting one of the points of rescheduling from the above list,

the context has to be saved before and restored after rescheduling, this part of the task management is not clean yet and will need some rework so it can be considered done. For a proof of concept as necessary by the OVERSEE project, other parts of OSEK are more important and will therefore need to be handled before finishing up task management.

5.3 Counters and Alarms

As described above, one way a task can be activated is if an alarm has expired. Each alarm is triggered by exactly one counter.

Counters can be incremented by all kinds of events but one of the most common ones are timers, in order to allow timed activation of tasks. All that was to do, to allow alarms that wake up tasks, was to add an IRQ handler which is triggered by the virtualized XM timer interrupts. Inside of this IRQ handler a counter is incremented, using the OSEK defined `IncrementCounter()` call. The virtualized timer is configured in the initialization code of FreeOSEK. Now one or more alarm(s) can be associated with the counter in the OIL configuration file of the application, to make those alarms go off as soon as the counter has reached a limit.

An example for such configuration could look like this (only the part that deals with counters and alarms):

```
COUNTER HardwareCounter {
        MAXALLOWEDVALUE = 100000;
        TICKSPERBASE = 1000;
        MINCYCLE = 1;
        TYPE = HARDWARE;
        COUNTER = HWCOUNTER0;
};

COUNTER SoftwareCounter {
        MAXALLOWEDVALUE = 100000;
        TICKSPERBASE = 100;
        MINCYCLE = 1;
```

[1]xprintf is a library function of `libxm` wrapping a XM_write_console, giving the application programmer a way to use formatted printing.

```
        TYPE = SOFTWARE;
};

ALARM IncrementSWCounter {
        COUNTER = HardwareCounter;
        ACTION = INCREMENT {
                COUNTER = SoftwareCounter;
        };
        AUTOSTART = TRUE {
                APPMODE = AppMode1;
                ALARMTIME = 1;
                CYCLETIME = 1;
        };
};

ALARM ActivateTaskA {
        COUNTER = SoftwareCounter;
        ACTION = ACTIVATETASK {
                TASK = TaskA;
        }
        AUTOSTART = FALSE;
};
```

The first section describes a hardware counter, that is incremented by the ticks from the virtualized XtratuM timer interrupts. This counter is used to increment a software counter using an alarm *IncrementSWCounter*. If this software counter has an overflow, the *ActivateTaskA* alarm is triggered, and the OSEK task `TaskA` goes from state *suspended* to state *ready*.

This looks like a waste of resources, but if you want different Alarms triggered by the the same hardware timer, you have to configure multiple software counter, which then activate the various tasks.

5.4 Interpartition Communication

Communication in OSEK is defined in [5], which defines the main goals of this specification as follows:

"It is the aim of the OSEK COM specification to support the portability, re-usability and interoperability of application software. The API hides the differences between internal and external communication as well as different communication protocols, bus systems and networks." [*OSEK Communication Specification 3.0.3, 1.1 Requirements*]

From this paragraph we already can deduce, that connecting FreeOSEK to the message passing interpartition communication system that is provided by XtratuM is conforming to the specification. More importantly, the latter part stating that communication protocols as well as communication media should be transparent to the application implies, that it has to be possible, to run a legacy OSEK compliant application, that uses OSEK COM. Specifically it can be run in an FreeOSEK run-time environment communicating via the XtratuM interpartition communication system instead of let's say a CAN bus, without even knowing it, and without the need of changing a single line of application code.

One further thing we can take into account, is the ARINC653 compliant interpartition communication system provided by XtratuM, and the resemblance of the OSEK Com system and the ARINC 653 interpartition communication system. This similarity in communication mechanisms leads to a huge simplification in the external communication which can be done me wrapper functions for the XtratuM hypercalls, which confiugre the XtratuM (ARINC 653) ports to behave the way expected from FreeOSEK and allow a OSEK Com compliant interface. Things like FIFO buffers for queueing messages do not have to be implemented, since they already are implemented in the XtratuM core.

6 Conclusion

Even if the FreeOSEK run-time environment is far from being perfect, the implementation shows the feasibility of running an OSEK compliant operating system as a XtratuM run-time environment,fulfilling one of OVERSEES main missions - reuse of existing automotive applicatinos in a security enhanced environment with minimum effort.

Furthermore, the theoretical mapping between OSEK compliant communication and ARINC653 compliant communication could be proven valid and compatible to the point where a legacy OSEK compliant application can be moved into a XtratuM runtime environment with a virtualized communication system replacing a legacy physical communications system, without the need of adapting the application itself.

The next steps in the port of FreeOSEK to XtratuM will be the cleanup of the context switch, in order to allow fully preemptive task scheduling. This is also the pre-requiste for most of the MODISTARC [6] tests which are already implemented in FreeOSEK and help to show the compliance with the OSEK/VDX specifications and finally the integration of the FreeOSEKport into the overall OVERSEE architecture Proof-of-Concept framework.

7 Acknowledgments

This paper has been produced in the context of the OVERSEE project (FP7-ICT-2009-4, Project ID 248333).

References

[1] *OVERSEE*, 2010, HTTP://OVERSEE-PROJECT.COM

[2] *FreeOSEK*, HTTP://OPENSEK.SOURCEFORGE.NET

[3] *ARINC653 - Avionics Application Software Standard Interface*,Airlines Electronic Engineering Committee, October 2003

[4] *OSEK Operating System Specification 2.2.3*, 2005, OSEK/VDX CONSORTIUM

[5] *OSEK/VDX Communication Version 3.0.3*, 2004, OSEK/VDX CONSORTIUM

[6] *Methods and tools for the validation of OSEK/VDX based distributed architectures*, 1999, OSEK/VDX CONSORTIUM

[7] *AUTOSAR - Automotive Open System Architecture*, HTTP://AUTOSAR.ORG/

[8] OSEK/VDX consortium,OSEK/VDX Homepage, HTTP://OSEK-VDX.ORG/

[9] M. Masmano et. al.: *LithOS: a ARINC-653 guest operating for XtratuM*, 2010

[10] John Rushby: *Partitioning forequirements, Mechanisms, and Assurance*, 1999

[11] Miguel Masmano, Ismael Ripoll, Alfons Crespo, Patricia Balbastre: *XtratuM Hypervisor for INTEL x86 - Volume 4: Reference Manual*, June 2011