By Dr. Andree Zahir,
Assistant Manager Research & Development,
ETAS GmbH & Co.KG

An integrated concept of handling preemptions and interrupts for automotive real-time operating systems

The main duty of a RTOS in embedded automotive systems is the scheduling of the applications tasks. Using a microcontroller providing several interrupt priorities, the scheduling of interrupt service requests is performed by the hardware directly. In cases a task wants to lock certain interrupt levels or a task has to be activated from an interrupt routine, interactions between both scheduling schemes are introduced. These interactions at the interface between hardware and software are very subtle and thus have to be treated in a profound manner. Deficiencies in the RTOS design concerning the interrupt handling may result in errors that only occur in rare event scenarios and are thus hard to detect. This paper describes a method to treat the priorities of tasks and interrupts in a homogeneous way. Common errors found in RTOS designs are avoided and the overall efficiency benefits from transfering parts of the scheduling effort to the hardware.

INTRODUCTION

Electronic control units (ECUs) are increasingly being used in modern cars to assume control for various functions. The more complex the embedded control software becomes, the more useful it is to give a real-time operating system (RTOS) control over the application. Reasons to introduce an RTOS in embedded automotive systems are:

- Encapsulating basic hardware functionality by providing a high level interface to the controller.
- Improving run-time efficiency by introducing dynamic scheduling.
- Enhancing the portability of application software by using standardized interfaces.
- Enhancing the test depth for some widely used components.
- Defining an design concept for embedded hard real-time systems.

Control units in cars cover a wide range of functionality from feature electronics improving comfort and convenience for the driver to core function electronics which control the basic components of a car. Multimedia or navigation systems as well as seat and climate controls are examples of feature electronics. Engine management, transmission control or other powertrain systems provide core functionality. Since the core functions are essential for the driveability and the safety of the car, the corresponding control units can be classified as hard real-time systems. Attributes like dependability, reliability, availability and safety should characterize these systems. In this paper, we will consider these hard real-time applications since a general RTOS concept has to meet the highest demands of automotive electronics.

In this article we present some essential concepts of handling preemptions and interrupts which are implemented in the ERCOS^{EK} operating system [1] for hard real-time automotive applications. ERCOS^{EK} adapts the OSEK operating system standard for automotive systems [2] but circumvents all of the problems related to the treatment of interrupts.

SCHEDULING AND DESIGN ASPECTS

Major portions of the application software in powertrain applications are time driven. One reason for this is that control algorithms typically require a numerical solution of a differential equation for discrete and equidistant steps in time. Furtheron, time triggered applications offer a higher level of dependability, predictability and safety than event-triggered systems.

Scheduling denotes the job of planning the execution order of all tasks in a computer system. It is required that all tasks meet their deadlines, even for a defined peak-load scenario. Predictability and efficiency typically lead to some kind of fixed priority scheduling where all tasks are assigned a static priority(1). This assignment is based on activation periods for cyclic tasks or response time and jitter requirements.

(1) Dynamic priority scheduling policies like e.g. the Earliest Deadline First (EDF) algorithm may offer benefits in terms of efficiency for some applications but reduce the possibilty for Off-Line analysis and optimisation considerably.

Copyright 1999 by Real-Time Magazine 99-3 (http://www.realtime-info.com)

Nevertheless, handling interrupts is still common and more efficient than polling if the required response time to the external event **tr** is considerably shorter than the minimum interarrival period (MINT). Modern high-end microcontrollers often provide a large number of on-chip peripherals and multiple interrupt levels. The arbitration and execution of interrupts is done by the interrupt control logic. These mechanisms implemented in hardware are comparable to preemptive scheduling of tasks. Once an interrupt of a certain priority is serviced, all other interrupt requests of the same or lower priority remain pending until the execution has finished. For preemptions and interrupts a context switch has to be implemented to resume the correct operation of the interrupted task. Additionally, the timing requirements for some interrupts may be in the same range as for tasks with high activation frequencies. It is therefore suitable to introduce an integrated concept of tasks and interrupts.

According to this concept the entire application software is structured into a set of tasks. For the remainder of this article, interrupts and the corresponding services are treated as a preemptive HW-tasks. Hardware (HW)-tasks are started by an interrupt source requesting the service. Scheduling of HW-tasks is performed by the interrupt logic of the controller and therefore denoted as HW-scheduling. The term software (SW)-task is used for tasks that are activated from the application itself, with a special timer, or event service from the operating system. Scheduling of SW-tasks is performed by the ERCOS^EK scheduler and therefore denoted as SW-scheduling.

For any scheduling analysis the worst case execution time (WCET) of a task is an essential attribute. Scheduling literature often distinguishes between periodic and sporadic tasks. The latter category is represented by HW-tasks that are triggered by irregular hardware events and SW-tasks that are directly invoked by the application software. For unification all tasks **T** (HW- or SW-activated) are characterized by their attributes **tp**, **tw** and **td** and written as **T(tp,tw,td)**.

tp defines the activation period for a periodic task or the MINT for a sporadic task respectively.

tw defines the WCET of the task.

td defines the deadline by which the task must finish.

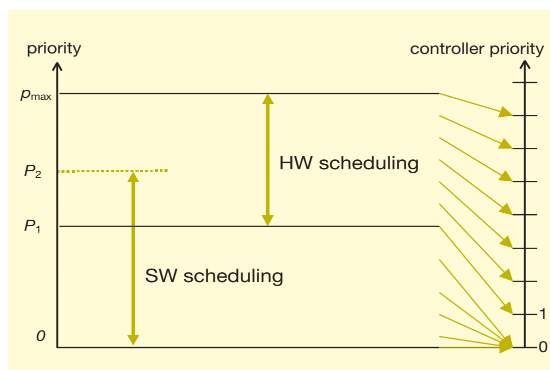

Figure 1. Configurable priority scheme of the operating system

After defining a unified way to characterize all tasks **T(tp,tw,td)** in a system it is obvious to offer as much flexibility as possible for assigning task priorities according to the overall timing requirements. Limitations to the priority configuration are usually determined by the controller hardware and should not be introduced by the RTOS design.

Figure 1 shows the configurable priority scheme supported by the RTOS. The configuration parameters p_1, p_2 and p_{max} are defined by the application regarding some limitations of the RTOS implementation and the underlying hardware. The scale on the right explains how the priorities of the RTOS are mapped onto the interrupt levels of the microcontroller. Beginning with the first level which is shared between SW- and HW-tasks, all RTOS priorities are mapped to a hardware level. In the remainder of this article the terms priority and level are used synonymously. The parameters are defined as follows:

- Priority level 0 is defined as the lowest priority level in the system.
- p_1 defines the first priority that may be shared by HW- and SW-tasks. p_1 is mapped on the first interrupt priority level of the microcontroller.
- p_2 defines the internal execution priority of the RTOS scheduler and the RTOS timer services. SW-tasks may be executed on priorities on level $0 <= p < p_2$.
- **Pmax** denotes the highest priority known to the RTOS. On some architectures it can be meaningful to allow interrupt service routines (ISR) which do not interact with the RTOS on priorities $p > p_{max}$.

The number of priority levels configured for HW-scheduling ($p_{max}-p_1$) cannot exceed the number of interrupt levels of the microcontroller. A major advantage of the priority scheme presented in Figure 1 is the possibility of overlapping priorities for HW-and SW-tasks.

PREVENTING INTERRUPTIONS AND PREEMPTIONS

In a preemptive multitasking environment, tasks (HW- and SW-activated) are in principal executed in parallel. Accessing shared memory or other hardware resources that are jointly used by more than one task has to be secured against unintended modifications by preempting tasks in order to achieve consistent read or write operations. Code sections containing such operations are denoted as critical and the duration of such a critical section is abbreviated with the symbol **tc**.

In a preemptive multitasking environment which does not support blocking or waiting states for tasks, we propose two different means for achieving consistent resource accesses:

- Accessing the resource by using a protocol guaranteeing a secured access, e.g. the priority ceiling protocol.
- Locking out interrupts to prevent unforeseen context switches.

Copyright 1999 by Real-Time Magazine 99-3 (http://www.realtime-info.com)

Considering that many embedded automotive systems rely on a large scale on reacting to external events by means of interrupts, locking them is potentially dangerous. The longest period where interrupts are locked `tl` defines the worst case response time (WCRT) for all HW-tasks. As a general design rule, locking of interrupts should only be performed when absolutely necessary and `tl` should be kept as short as possible.

Priority ceiling protocol

The priority ceiling protocol (PCP) guarantees secured access to a shared resource by raising the priority[(2)] `p` of the requesting task to the highest priority of all tasks that also access the resource. This dedicated priority level is an attribute of the resource and denoted as its ceiling priority `cp`. By dynamically altering the priority of the task, all tasks competing for the resource are prevented from being executed. Figure 2 shows task 1 accessing resource `R` with ceiling priority `cp`. While running through the critical section, task 2 is activated and has to be scheduled after releasing resource `R` before continuing execution of task 1.

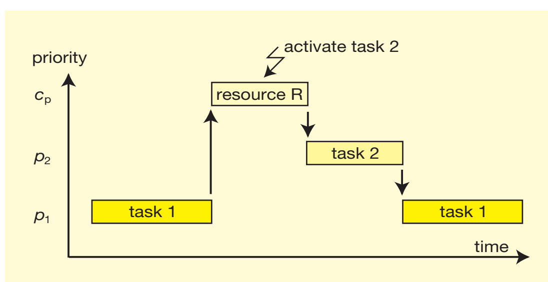

Figure 2. Resource access using the priority ceiling protocol

The main benefits of the priority ceiling protocol – compared to suspending interrupts – are:

- The security mechanism is limited to the necessary priority levels, i.e. tasks with priorities higher than the ceiling priority `cp` are not affected by the resource access.
- The PCP can be implemented in a transparent way in order to offer the functionality to SW- and HW-tasks.
- The ceiling priority can be treated as a configuration parameter which allows to place the task code accessing the resource in a library and define the ceiling priority later.

For distributed software development of embedded systems the latter point is of utmost importance. It is already common practice that the code for an automotive ECU is developed at different companies and integrated by exchanging object libraries. Delivering source codes does not protect the intellectual property of the project partners and is therefore often refused.

The major drawback of the PCP is efficiency. First, the protocol is invoked every time the resource is accessed -even in situations where the case of conflict cannot happen. Second, the protocol has to implement some rescheduling capabilities for cases during which the resource accesses a task with priority `p < x <= cp` was activated. Such a situation is shown in Figure 2. For very short resource accesses the protocol overhead could exceed the duration of the critical section `tc`.

Locking interrupts

Considering the issues mentioned in this paper so far, we are able to set up some rules for the treatment and the locking of interrupts. These rules are important because we have to deal with two different approaches in handling interrupts and we will show that only one is reasonable for hard real-time systems. The different approaches are:

- The RTOS directly deals with interrupt sources and provides services to enable and disable the capability of peripheral devices to generate interrupts.
- The RTOS only deals with priorities and always generally locks all interrupts up to the requested priority level.

Independent of the implemented concept (of handling interrupt sources or priority levels) the current situation is denoted as the interrupt state. The rules can be summarized as follows:

- Interrupts should only be locked when absolutely necessary.
- The duration of locking interrupts must be as short as possible.
- The interrupt locking mechanism must ensure that no interrupt request is lost[(3)].
- The process of distributed development has to be addressed.

Stacking interrupt states

When addressing the issue of distributed development, a major requirement is that the current interrupt state has to be stacked before it is changed. In a preemptive multitasking environment, nested calls of the corresponding services have to be allowed. Various RTOS implementations use the semantics of `DisableInterrupt` and `EnableInterrupt` as defined in the OSEK specification to lock out interrupts during critical sections. First we assume that these services lock out certain interrupt sources as requested by the parameter `mask`. Subsequently, we compare pseudo code sections to secure a `critical section 0` from preemptions (see Figure 3).

In this case of nested critical sections it can be easily observed that the called function `Foo1()` may globally enable some interrupts after leaving `critical section 1` that were locked out by `maskD0` to secure `critical section 0`. Unintentionally, the expected security of the `critical section 0` in function `Example0()` is broken. The scenario shown in Figure 3 is especially dangerous in cases of distributed development where the behavior of function `Foo1()` is not known in detail by the developer of function `Example0()`.

(2) The implementation has to guarantee that the priority of a task can never be lowered by accessing a resource.

(3) Individual interrupt requests may be lost if the locking period `tl` exceeds the minimum interarrival period (MINT) of the service request. We consider this constellation poor software design.

Copyright 1999 by Real-Time Magazine 99-3 (http://www.realtime-info.com)

```
Example0() {
   DisableInterrupt(maskD0);
   start critical section 0
   call library function Foo1 ------->    Foo1(){
                                             DisableInterrupt(maskD1)
                                             critical section 1
                                             EnableInterrupt(maskE1)
                                             return;
   <-----------------------------------   }
   end of section 0   //interrupts
   enabled here!!
   EnableInterrupt(maskE0);
}
```

Figure 3. Abolished security by nesting critical sections

Apart from calling library functions inside critical sections, the same problem might occur if function **Foo1()** is called from a HW- or SW-task still allowed to preempt *critical section 0*. The operating system could handle this situation by saving and restoring the interrupt state at context switches. Then the problem is shifted to interrupt service routines (ISR) not handled by the RTOS. Many of these ISRs may be present in applications for efficiency reasons especially if the functionality was initially realized without using an RTOS.

Regarding the arguments mentioned to date, the only practical way of handling these problems is to write the application code self-contained as shown in Figure 4. **Example0()** is reworked for this purpose.

If all critical sections of the application are secured as presented in Figure 4, nesting of critical sections is handled properly. Still, the presented solution is not the best for two reasons:

- The services **BeginAtomic** and **EndAtomic** are not standardized interfaces and dependent on the microcontroller in use[(4)]. Therefore, the pseudo code in Figure 4 is not portable.
- The runtime overhead for that solution is considerable and might in many cases exceed the duration of the critical section itself.

Efficiency will be improved and the application code portable and reusable if stacking the former interrupt state is managed by the service **SuspendInterrupts**. The corresponding routine service **ResumeInterrupts** does not take a parameter but restores the interrupt state before the last call of **SuspendInterrupts.** Again, we rework **Example0()** using the new services (see Figure 5).

```
Example0() {
   IntDescriptorType actIntState;
   BeginAtomic
   GetInterruptDescriptor(&actIntState);
   DisableInterrupt(maskD0);
   EndAtomic
   critical section 1
   EnableInterrupt(actIntState);
}
```

Figure 4. Self contained interrupt locking

Tracking interrupt states

Having solved the problem of modifying and restoring interrupt states, another complicated issue when treating interrupts by enabling and disabling the sources arises in tracking the interrupt state through task

(4) The services are needed to prevent any interrupts in order to keep the treatment of interrupt states consistent.

```
Example0() {
   SuspendInterrupts(maskD0);
   start critical section 0
   call library function Foo1 ------->    Foo1(){
                                             SuspendInterrupts(maskD1);
                                             critical section 1
                                             ResumeInterrupts;
   <-----------------------------------      return;
   end of section 0   //interrupts        }
   still disabled!!
   ResumeInterrupts;//interrupts now
   enabled!!
}
```

Figure 5. Maintained security by stack-based interrupt suspension

preemption scenarios. Consider the case that task **T0(tp_0,tw_0,td_0)** modifies the interrupt state by disabling several sources according to **maskD0**. Still it is possible for other interrupts (HW-tasks) to be executed and they are allowed to activate other tasks with higher priority, e.g. task **T3(tp_3,tw_3,td_3)**.

The important question is how the interrupt state is treated by task **T3** after performing the context switch. Many RTOS implementations start every task with all interrupts enabled. In this case the intended security of task **T0** would be lost. An alternative approach would be to keep the interrupt state unchanged when starting the new task **T3**. Security problems can be solved this way but the overall system behavior becomes less predictable because task **T3** will run with various initial interrupt states, depending on the task preempted.

Regarding these issues an interrupt concept based on interrupt sources is only practical in smaller projects where one engineer is capable of having an overall view about the application. Having large scale automotive applications like engine management systems in mind, where many developers – eventually also from other companies – are involved the job of tracking interrupt states becomes virtually impossible.

Pending interrupts

Since interrupts occur asynchronously to the running application, the case, that an interrupt request becomes active while servicing the interrupt is disabled or suspended, needs to be considered. The implemented mechanism of handling interrupts has to ensure that this interrupt request will not be lost. Looking at some microcontrollers commonly used in automotive electronics, such as Motorola's HC08, HC11 or HC12 devices, the hardware only latches the interrupt request if the interrupt source is enabled. This special behavior is not well defined by many semiconductor suppliers but essential for the operating system implementation. Further, this behavior depends on the type of interrupt source.

Interrupts are always latched if the capability to request an interrupt remains enabled at the source but is suppressed at the controller core. Dealing with interrupts at the controller core is achieved by means of globally disabling interrupt processing in a status register or masking out all priority levels up to the currently active processing level.

The fact that the hardware of a microcontroller supporting multiple interrupt levels schedules interrupt requests according to priorities makes it highly advisable to adapt this scheme for handling interrupts from the RTOS' side. In fact, the configurable priority scheme of the ERCOS^{EK} operating system as shown in Figure 1 presents a unified priority scheme for interrupts and tasks, or better HW- and SW-tasks. Typically, scheduling analysis approaches, which are able to guarantee scheduleability for a given set of tasks, consider preemptive task switches and priority assignments according to activation frequencies or deadline analysis. Such analysis is contravened by locking individual interrupt sources without regarding the interrupt priority. Hence, for hard real-time applications the concept of handling interrupts according to their assigned priorities is vital.

Furthermore, the possibility to lose individual interrupt requests on certain microcontrollers by disabling the interrupt source renders this concept useless. A concept for handling interrupts implemented in an RTOS that is intended to be ported to various platforms has to consider the general case.

Implementation in ERCOS^{EK}

For the purpose of locking out interrupts ERCOS^{EK} implements the concept of suppressing certain interrupt levels and stacking the former level. The selection of interrupt levels that might be suspended is not free for the application but derived from the priority scheme shown in Figure 6.

The reason why some well-defined priority levels are chosen for interrupt suspension is efficiency. Suspending interrupts up to and including priority p_2 or higher prevents activation of SW-tasks while the critical section is executed. Hence, **ResumeInterrupts** does not have to perform a rescheduling but only restore the former processing level.

To be more flexible in suppressing certain interrupt levels, it is suggested to use the priority ceiling protocol as mentioned in Chapter "Priority ceiling protocol" on page 28. The decision between locking interrupts and using the PCP will therefore always be a tradeoff between runtime efficiency and flexibility.

TREATING INTERRUPTS IN A RTOS

Treating interrupts in a RTOS is a complicated and not exactly straightforward task. First, one has to be deeply aware of the interrupt control logic of the microcontroller in use. Second, the interaction between interrupts and the scheduler have to be examined when

SERVICE	FUNCTIONALITY
SuspendLsInterrupt	Suspends all interrupts up to level p_2 and stores the former level.
SuspendHsInterrupts	Suspends all interrupts up to level p_{max} and stores the former level.
SuspendAllInterrupts	Suspends all interrupt levels of the microcontroller and stores the former value.
ResumeInterrupts	Resumes the interrupt level that was valid before the last call to a **SuspendXYInterrupts** service.

Figure 6. Table of services to suspend interrupts

Copyright 1999 by Real-Time Magazine 99-3 (http://www.realtime-info.com)

RTOS services like task activations are called from the ISR.

Many RTOS implement a service call like **`EnterISR`** to register the execution of an ISR for the RTOS. In case the ISR activates a task of higher priority than the interrupted task, the RTOS has to continue execution with the activated task rather than returning to the interrupted task. The scheduler performing this context switch is invoked by calling the service **`LeaveISR`**. Typically this discontinuity in program flow is realized by patching the return address of the ISR.

This mechanism is potentially dangerous (see [3]) because the nested interrupts might occur before servicing the **`EnterISR`** system call. If this happens and a task switch is requested, the first ISR will not be completed. Such errors are hard to detect since they only occur in rare event scenarios.

Another point of trouble (as shown in [3]) might occur when the treatment of interrupt priorities is not properly defined within tasks. Especially RTOS that implement interrupt source based handling are prone to priority inversion situations or removed interrupt locks if tasks are always started with all interrupts enabled. The concept realized in the ERCOS^EK operating system resolves these issues by introducing the integrated scheme of hardware and software priorities as shown in Figure 1. By offering the services for suspending interrupts as listed in Figure 6, the broken interrupt-lock error can be resolved.

The scheduling interrupt

The ERCOS^EK operating system implements a scheduling interrupt which is used to start all preemptive SW-tasks. A maskable interrupt source that can be triggered by software with configurable priority is required from the RTOS to achieve this functionality. In order to trigger a task switch to the activated SW-task of currently highest priority, a scheduling interrupt is generated on the corresponding controller priority level. The complete task code is executed inside this scheduling interrupt ISR which therefore has to be implemented to support reentrance.

The method of starting preemptive tasks by means of scheduling interrupts and implementing a straightforward concept of treating interrupts based on priorities instead of manipulating the sources, offers the possibility to transfer major parts of the preemptive scheduler to the interrupt logic of the controller. All interrupt service routines are then terminated by regularly returning to the interrupted context. Pending interrupts are treated as defined by the hardware.

As a result of handing over the responsibility for scheduling preemptive HW- and SW-tasks to the controller hardware there is no need to implement a **`EnterISR-LeaveISR`**-protocol to register ISR for the RTOS. In fact on various controllers that provide a interrupt vector table the application is free to configure ISR that don't call the RTOS on any interrupt priority. Here we stick to the principle: *Who does not interact with the RTOS does not have to be known by the RTOS*. The tradeoff for this solution is the additional interrupt source the RTOS occupies for scheduling preemptive SW-tasks. The experience from various projects we serve – using this approach – to date, shows its viability.

CONCLUSION

In this article we presented a concept how an RTOS can handle tasks and interrupts in a consistent manner. The concept is validated and proven in production ECUs and used in vehicles from various car manufacturers in powertrain applications. Talking about operating systems is always a major issue because this subject is not limited to the basic scheduling functionality and how it's implemented. The design and the concepts of the RTOS always guide the design of the application software. Hence, the application developer of a hard real-time system has to crosscheck his design principles with the concept and the features provided by the RTOS. Reaching a common agreement here is vital for implementing high quality embedded software.

One has to remeber that the portability issue is not just a matter of standardized interfaces to the RTOS. It is equally important to point out, that only by using the proper design principles one can expect that the application will be portable to various different microcontrollers.

LITERATURE

[1] ERCOS^EK V3.0 User's Guide, ETAS GmbH & Co.KG, 1999

[2] OSEK/VDX Operating System, Version 2.0 revision1, 15.10.1997, http://www.osek-vdx.org

[3] K.W. Tindell, "RTOS interrupt handling: common errors and how to avoid them", Embedded Systems Programming Europe, June 1999

Dr. Andree Zahir studied electrical engineering at Stuttgart university (1984 – 1989).
Worked 5 years in solar energy research, especially the simulation of renewable energy systems Ph.D. thesis at the institute for the theory of electrical engineering, stuttgart university, 1994 Joined ETAS GmbH & Co. KG in march 1995 to work on distributed real-time systems.
Member of the OSEK working group OS since 1996. Since 1998 responsible for the development of embedded automotive software at ETAS.

Copyright 1999 by Real-Time Magazine 99-3 (http://www.realtime-info.com)