Improving commercial RTOS performance using a custom interrupt management scheduling policy

Eugen Dodiu, Vasile Gheorghita Gaitan, Adrian Graur
Stefan cel Mare University
Universitatii no. 13
ROMANIA
edodiu@usv.ro

Abstract: Hard real time systems should be built using a predictive approach since high importance tasks or human health is involved. The system must provide a real-time response to the changes of the environmental conditions and actions produced by humans or other hardware equipment. It is mandatory that task computations are both correct and finished within a specified deadline. This is often not possible with commercial RTOS because the high importance tasks can be interrupted by interrupts belonging to low priority tasks, thus delaying the task deadline. The paper presents a custom interrupt management policy that is meant to eliminate this inconvenience.

Key-Words: Hard real-time, Fixed priority scheduling, Preemption, Interrupt latency

1 Introduction
Most of the commercial RTOS do not respect the most rigorous conditions that apply for deterministic task execution. It is the case for those that use two priority domains for tasks and interrupts. In such a system high priority tasks usually get interrupted by low priority interrupts that target low priority tasks. Consider a noisy radio environment whose associated interrupts disturb the execution flow of an important task in the system. If these interrupts occur very often there is a big chance that high urgency tasks will miss their deadlines.

Real-time systems are a little bit different from the classical ones because they have to offer a certain response within a specified time period. With real-time systems, correct execution of tasks will depend not only on the correctness of the results but also on the time when they are provided. Unlike soft real-time systems where deadline miss is not a major problem, missing a time constraint in a hard-real time system can cause severe material damage or even human health injury [1],[2],[3],[4],[16]. A system that is similar to the one described above is not considered hard-real time since deadlines for tasks cannot be guaranteed. In this paper we propose a method to eliminate this drawback.

The main resource that needs strict planning is the processor. In this paper, we will refer only to situations where single core platforms are involved. The analysis of real time systems is related to two main concepts: scheduling and feasibility. For a task set to be schedulable, we must always determine the necessary and sufficient conditions that need to be satisfied. A task set is schedulable under a scheduling algorithm if all the tasks instances meet their deadline. Periodic task scheduling was first studied by Liu and Layland [1] back in 1973. They proposed a scheduling algorithm for fixed priority tasks which also provides an equation that allows verification of the scheduling condition for a task set in the worst case scenario. This happens for a critical instant when all tasks are launched in execution at the same time. Using a less pessimistic approach, Lehoczky, Sha and Ding [2] make a detailed analysis of the rate monotonic algorithm finding a necessary and sufficient scheduling condition for any situation related to task startup times. Even though the aforementioned papers are leading references when it comes to real-time scheduling they do not say anything about the overhead related to interrupts. In the second chapter we add some mathematical background that gives a better image on real-time scheduling algorithms.

The interrupt scheduling policy that we implemented was proposed for Micrium’s µC/OS-II kernel on a commercial hardware architecture built around Cirrus Logic EP9302 processor. The scheduler modifications and their behavior will be presented in the third chapter.

2 Task set schedulability analysis
Tasks are the main entities that are executed in a real-time operating system. They can be periodic or aperiodic, and they may have time restraints. The task main parameters are:
- r-release time (periodic or event triggered)
- C-worst case execution time
- D-task relative deadline
- T-task period (specific only to periodic tasks)
- u=C/T is the processor utilization factor and must always be equal to or less than 1
- h=C/D is the processor load factor and must always be equal to or less than 1.
- s is the start time of task execution
- e is the finish time of task execution
- D(t)=d-t is the residual relative deadline at time t: 0≤D(t)≤D
- C(t) is the pending execution time: 0≤C(t)≤C
- L=D-C is the nominal laxity of a task and represents the maximum lag for the start time s when it has sole use of the processor.

A task is well formed if the expression 0<C≤D≤T is true [4].

According to Lehockzy, Sha and Ding [2], for a task set t₁…tᵣ the processor utilization factor is expressed as:

\[ Wᵢ(t) = \sum_{j=1}^{n} \left( \frac{t_j}{T_j} \right) \]  \hspace{1cm} (1)

Using the task set t₁…tᵣ, a single task tᵢ can be scheduled by using the rate monotonic algorithm only if the following condition is met:

\[ Lᵢ ≤ 1 \] \hspace{1cm} (2)

Where \( Lᵢ = \min_{0≤t≤Cᵢ} Lᵢ(t) \) and \( Lᵢ = \frac{Wᵢ(t)}{t} \)

By using the task set t₁…tᵣ, the whole task set can be scheduled if the following condition holds:

\[ L ≤ 1 \] \hspace{1cm} (3)

The model described by Lehockzy, Sha and Ding [2] provides a processor utilization factor of almost 88% for n ranging between 50 and 100. In real life situations, however, this shall not exceed 70% regardless of the tasks number [3]. An alternative for the RMA is the Deadline driven scheduling algorithm that was first described by Liu and Layland in [1]. According to this, the tasks that have the earliest deadline will be scheduled first. The necessary scheduling condition for a task set with m tasks is:

\[ \left( \frac{C₁}{T₁} \right) + \left( \frac{C₂}{T₂} \right) + … + \left( \frac{Cₘ}{Tₘ} \right) = 1 \] \hspace{1cm} (4)

For the condition described by (4), it is considered that the processor has no idle times; therefore it has a higher utilization bound (up to 100%) in comparison to the RMA. The conditions described in [1] and [2] are pure theoretical and they don’t include any references to interrupt and context switching overhead. Real life applications should take into consideration the overhead for [5]:

- Interrupts (interrupts with high occurrence rate may lead to additional overhead)
- Peripheral devices cycle stealing (e.g. DMA)
- Task deadline missing
- Keeping interrupts disabled for longer periods then necessary
- Unpredictable errors

The following can also be considered:

- Cache memory effect
- MMU
- Unconditional hardware errors that can lead to interrupt flooding or even CPU stalling
- Unprecise interrupt treatment

The model proposed by Jeffay and Stone [6] assumes that both the application tasks and the interrupt handlers are sequential program sections that are executed as reply to external events. They make an assumption that events are periodic but this is not the case for real life situations. Since interrupts are aperiodic, a classical approach will treat them by using servers.

The polling server described by Sha, Rajkumar and Lehoczky [7] is a periodic task with fixed priority and capacity that can execute the code for all aperiodic events that were triggered between two successive executions of the server. The deferrable server algorithm [8] has a similar functionality but it has the ability to preserve its execution time in the absence of aperiodic requests. The sporadic server algorithm [9] is similar to the previous mentioned, the only thing that differs is the way it conserves and replenishes its capacity.

A modern approach is proposed by Leyva-del-Foyo and Mejia-Alvarez [10] using a unified interrupt and task priority space. The model implements a Low Level Interrupt Handler in kernel for handling all the interrupts in the system. The abstraction layer that is built inside informs the kernel about all the events in the system. Within the “Interrupts as Threads” scheduling, interrupts are actually executed inside Interrupt Service Tasks instead of ISR.

In case of schedulers with fixed priorities all the rules that govern the correct execution of the tasks must be known a priori. Both Liu and Layland [1] and Lehoczky, Sha and Ding [2] proposed some mathematical models that do not take in consideration the parameters specific to real life situations. They made the following assumptions:

- tasks are periodic and ready for execution each time at the beginning of their period
-each task must be finished until the next activation.
-tasks are independent, do not synchronize, do not suspend themselves, and do not block each other.
-the overhead due to interrupts, ISRs, scheduler and context switching is considered 0.

Katcher, Arakawa and Strosnider [11] will include for the first time in the already known scheduling formulas the overhead and blocking (or priority inversion). In order to describe the complete mathematical model, the following notations must be made:

-\( C_{\text{int}} \) is the necessary time spent for minimal processing of an interrupt (context saving and scheduler call).
-\( C_{\text{sch}} \) necessary execution time for scheduler code that is responsible with the selection of the next task.
-\( C_{\text{res}} \) context restoration time.
-\( C_{\text{st}} \) time needed for saving the task current state into the TCB.
-\( C_{\text{ld}} \) time needed for starting a task that is situated in the run queue.
-\( C_{\text{trap}} \) old task TCB saving time and selection of the new active task.

The overhead for the integrated event-driven scheduling is described by:

\[
C_{\text{preempt}} = C_{\text{int}} + C_{\text{sch}} + C_{\text{res}} + C_{\text{ld}}
\]
\[
C_{\text{exit}} = C_{\text{trap}} + C_{\text{ld}}
\]  
(5)

\( C_{\text{preempt}} \) and \( C_{\text{exit}} \) from (5) represent the greatest overhead value specific to this type of scheduling. A task set composed of \( n \) tasks is schedulable (6) if the following equation holds:

\[
\forall i, 1 \leq i \leq n, \min_{0 < T \leq D_j} C_j + C_{\text{preempt}} + C_{\text{exit}} \left[ \frac{t}{T_j} \right] \leq 1
\]  
(6)

Other scheduling method where interrupts are used for triggering tasks and where external interrupts are still allowed to interrupt the execution flow of the program will insert additional overhead: \( C_{\text{nonpreempt}} \). A task set that is scheduled under the nonintegrated interrupt event-driven algorithm is schedulable (7) if the following equation holds:

\[
\forall i, 1 \leq i \leq n, \min_{0 < T \leq D_j} \left[ \sum_{j=1}^{\bar{n}} C_j + C_{\text{preempt}} + C_{\text{exit}} \left[ \frac{t}{T_j} \right] \right] + \sum_{j=n+1}^{\bar{n}} C_{\text{preempt}} \leq 1
\]  
(7)

For the situation where the scheduler runs on timer interrupts additional overhead may appear:

\[
C_{\text{timer}} = C_{\text{int}} + C_{\text{sch}} + C_{\text{res}} + C_{\text{ld}}
\]
\[
C_{\text{preempt}} = C_{\text{res}} + C_{\text{ld}}
\]
\[
C_{\text{exit}} = C_{\text{trap}} + C_{\text{ld}}
\]  
(8)

For schedulers that run on timer interrupts a task set is schedulable if the following equation holds:

\[
\forall i, 1 \leq i \leq n, \min_{0 < T \leq D_j} C_j + C_{\text{preempt}} + C_{\text{exit}} \left[ \frac{t}{T_j} \right] \leq \frac{t}{T_{\text{ic}}} + \left[ \frac{t}{T_{\text{ic}}} \right] \leq 1
\]  
(9)

Stewart and Arora [12] propose a fixed priority scheduling method that takes into consideration the computation time of the interrupts. Considering \( \Delta_{\text{thr}} \) the task switching overhead and \( \Delta_{\text{int}} \) the interrupt overhead the following equations are true:

\[
\forall i, 1 \leq i \leq n, \min_{0 < T \leq D_j} \left( \sum_{j=1}^{\bar{n}} C_j + 2\Delta_{\text{thr}} \left[ \frac{t}{T_j} \right] \right) \leq 1
\]  
(10)

\[
\forall i, 1 \leq i \leq n, \min_{0 < T \leq D_j} \left( \sum_{j=1}^{\bar{n}} C_j + 2\Delta_{\text{int}} \left[ \frac{t}{T_j} \right] \right) \leq 1
\]  
(11)

If the static scheduling is used together with an aperiodic server, from (10) and (11) we find that:

\[
\forall i, 1 \leq i \leq \left( n_{\text{int}} + n_{\text{dr}} \right), \min_{0 < T \leq D_j} \left( \sum_{j=n_{\text{int}}+1}^{\bar{n}} C_j + 2\Delta_{\text{int}} \left[ \frac{t}{T_j} \right] \right) \leq 1
\]  
(12)

Leyva-del-Foyo and Mejia-Alvarez [10] proposed a scheduling policy that reunites user tasks and interrupt handlers. The major drawback is that it adds an additional overhead for the Hardware activated tasks. This also increases the CPU utilization bound.

\[
C_j^H = c_j^H + \delta^j
\]  
(13)

\[
U_i^j = \frac{C_j}{T_i} + \sum_{j \in \{p(i)-H(i)\}} \frac{C_j}{T_j} + \sum_{j \in H(i)} \frac{c_j^H + \delta^j}{T_j^H}
\]  
(14)

\( H(i) \) is the set of activities \( t^H_i \) with execution time \( C_j^H \) between successive activations \( T_j^H \). \( \delta^j \) is the time CPU spends for saving registers and for entering and
 exiting the ISR. $C^H_j$ is the handler execution time. The total execution time of a ISR can be calculated using (13) and (14). Another hardware scheduling policy mentioned by Strosnider and Katcher [11] is not used because only Intel 80960 supports it. Unified interrupts and tasks space have also been proposed by Hofer [13] by using the “Tasks as interrupts” model.

3 Implementation of the custom interrupt management scheduling for µC/OS-II

This kernel has the following properties:

- preemptive execution
- fixed priority tasks
- fixed priority interrupts
- deterministic execution for scheduler body parts
- centralized offline scheduling
- multitasking up to 64 tasks
- selective stack size before compilation
- services like: mail boxes, messages, time management, semaphores

The integration tests were performed on a TS7300 boards from Technologic Systems that uses an ARM920T processor rated at 200MHz [14]. As for the software environment, we used IAR Embedded Workbench compiler version 5.4.

In µC/OS-II, the ISR are written in assembler. Fig. 1 presents the way interrupts are handled. If the ISR routine makes ready a high importance task than the new task is executed. If there is no new task activated then the CPU is passed back to the interrupted task. The kernel must be notified of the interrupt occurrence by using the OSIntEnter() function or simply incrementing the OSIntNesting counter. Since the counter is a 32 bit address value and the processor does not use atomic instructions to increment it, it should be used together with OS_ENTER_CRITICAL() and OS_EXIT_CRITICAL() macros. The OSIntExit() function calls OSIntCtxSw() and that is the reason high priority activated tasks can start right away as soon as the ISR is over.

With Real-Time Systems it is possible that a high priority task to be interrupted by a low importance ISR. This may lead to deadline misses and fatal errors in the systems. Since the kernel does not handle the interrupt priority levels it is susceptible to this kind of errors.

In order to solve this problem we propose a simple and efficient method that uses a unified priority space similar to the one described by Layva-del-Foyo and Alvarez [10]. This method is implemented for ARM7 and ARM9 processors that use the ARM PrimeCell PL192 and PrimeCell PL190 VIC controllers but it should work fine for any other VIC controller that has the following characteristics:

- VIC cell chaining
- Individual masking and activation of interrupts
- Programmable priorities with vector addresses
- FIQ and IRQ interrupts

We make the following assumptions:

- a task can be activated by multiple interrupts
- an interrupt can activate only a single task

Fig. 2. Task and interrupt priority organization in one of the demo application used with the proposed scheduling algorithm.

For the example in Fig. 2, the Worst Case Execution Times of Task T1 (Sensor acquisition task) appear if the device is placed in a noisy radio environment. There will be a large number of useless interrupts arriving unconditionally in the system. Any interrupt is allowed to interrupt any task at any time. Usually µC/OS-II cannot handle this situation.

We next arrange the interrupts and tasks in a unified interrupt space as follows:

<table>
<thead>
<tr>
<th>I Timers</th>
<th>I GPION</th>
<th>I ADC</th>
<th>I CAN</th>
<th>I 86 MHz</th>
<th>I SPI</th>
<th>I Timer 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scheduler</td>
<td>Sensor acquisition task</td>
<td>Network task</td>
<td>Console task</td>
<td>Start task</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For the example in Fig. 2, the Worst Case Execution Times of Task T1 (Sensor acquisition task) appear if the device is placed in a noisy radio environment. There will be a large number of useless interrupts arriving unconditionally in the system. Any interrupt is allowed to interrupt any task at any time. Usually µC/OS-II cannot handle this situation.

We next arrange the interrupts and tasks in a unified interrupt space as follows:

<table>
<thead>
<tr>
<th>T0</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>T1</td>
<td>12</td>
<td>T14</td>
<td>T2</td>
</tr>
</tbody>
</table>
We consider that I0 is the highest priority in the system and I6 the lowest. The same rules apply for the tasks: T1 has the highest priority, T4 the lowest.

<table>
<thead>
<tr>
<th>Tasks</th>
<th>Allowed interrupts for that task</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>ITMR1, I0, I1</td>
</tr>
<tr>
<td>T2</td>
<td>ITMR1, I0, I1, I2, I13, I14</td>
</tr>
<tr>
<td>T3</td>
<td>ITMR1, I0, I1, I2, I13, I14, I15</td>
</tr>
<tr>
<td>T4</td>
<td>ITMR1, I0, I1, I2, I13, I14, I15, I16</td>
</tr>
</tbody>
</table>

Table 1 Rules for interrupts acceptance

We consider that each task can only be interrupted by higher priority interrupts. We generate a couple of interrupt masks for each task because the VIC has two daisy chained controllers with 64 interrupt sources.

### 3.1 Software modifications

For implementing the previous scheme the user must assign the interrupt masks to each task after being created. The recommended method is to load the masks in the TCB of each task. To this end, we use the `OSTCBExtPtr` that can be modified to point to a mask structure. TCB mask loading must be done the first time a task runs before it reaches its main loop.

![Fig. 3. Previous unused pointer is now used for mask storage.](image)

The code sequence described by the flow chart in Fig. 4 is a critical section. `OSCtxSw` first saves the context of the interrupted task and also the stack pointer to in the task TCB. Second, it will give the user the chance to run some custom code inside the `OSTaskSwHook` branch, then it will restore the context of the new task to be executed from its TCB. The masks are loaded in the VIC before the new context is loaded.

![Fig. 5. Scheduler mask reconfiguration](image)

### 3.2 Case study

The following situations may appear:
- a task can explicitly pass control to another task
- an interrupt can interrupt a task
- an interrupt can interrupt another less important interrupt if interrupt chaining is enabled by the programmer.

In the first case once a `OSMboPost()`, `OSQPost()`, `OSSemPost()`, `OSTimeDly()` is called, the function `OS_TASK_SW()` is also called from `OS_Sched()` and mask reloading will be made.

For the second case the functionality is described by Fig. 1. Here, the call for `OSIntExit()` will automatically reconfigure the interrupt filtering masks if the ISR activated a higher priority task.

Context saving is done on the IRQ stack or on the current task’s stack. Interrupts can also be executed outside the kernel to prevent additional overhead. This can be done if the `OSIntNesting` is not incremented and if the interrupts are disabled.

In the third case if a task switch is involved, `OSIntExit()` is also called so masks are changed. The context saving is done directly on the IRQ stack declared in the processor initialization sequence.

When the critical section described in Fig. 4 is finished, the new task to be executed will run using its own interrupt mask. The VIC hardware allows preservation of untreated interrupts for a short time until the critical section is over (at the end of `OSCtxSw`). Those interrupts may be executed if the new filtering mask allows it.

### 4 Conclusions

The proposed method is simple and easy to implement for ARM processors that have similar
VIC controllers with the ones described above. This method uses a tiny portion of flash memory needed only for interrupt mask storage (two 32 bit works for each task). Additionally the users must perform the following actions:

- Define interrupt filtering masks for each task
- Load the mask structure in the TCB at first task cycle.

**Strong points**

- Masks can be modified at runtime
- Facile adaptability to various theoretical models for testing and analysis
- Can be used in various hardware VIC architectures
- Interrupt priority configuration according to user requests.
- Allows execution of simple interrupt outside the kernel to minimize overhead and to speed up the ISR code that is not addressed to the kernel or tasks.
- ISR latency is not increased
- Additional overhead is insignificant

**Weak points**

- Monitor type kernel that needs disabling of interrupts.

With the proposed method, the major disadvantage presented in the first chapter is eliminated, and by doing so, the overall real-time feature of the kernel is improved. As a future work we intend to study the behavior of the proposed method when the kernel does not have the highest priority in the system. We will also build a mathematical model after the ones described in chapter 2 that will also take into account the overhead related to task context switch and ISR. We will make a benchmark to measure the improvements added by this method.

**References:**


