Real-Time Scheduling of Interrupt Requests over Conventional PC Hardware

Luis E. Leyva-del-Foyo\textsuperscript{a}, Pedro Mejia-Alvarez\textsuperscript{b}, Dionisio de Niz \textsuperscript{c}
\textsuperscript{a}Dpto. Computación, Universidad de Oriente, 90500, Santiago de Cuba, Cuba
\textsuperscript{b}Sección de la Computación, CINVESTAV-IPN, Av. I.P.N. 2508, México, D.F., 07300
\textsuperscript{c}DESI, ITESO, Periférico Sur 8585, Tlaquepaque, Jal. México,
E-mail: luisleyva@acm.org, pmejia@cs.cinvestav.mx, dionisio@iteso.mx

Abstract

In this paper we analyze the traditional model of interrupt management and its incapacity to incorporate reliability and temporal predictability demanded on real-time systems. As a result of this analysis, we propose a model that integrates interrupts and tasks handling. We make a schedulability analysis to evaluate and distinguish the circumstances under which this integrated model improves the traditional model. The design of a flexible and portable kernel interrupt subsystem for this integrated model is presented. In addition, we present the rationale for the implementation of our design over conventional PC hardware and the analysis of its overhead. Finally, experimental results are conducted to show the deterministic behavior of our integrated model.

1. Introduction

Most embedded systems use the interrupt mechanisms to provide an interface with different peripheral devices. These devices communicate the system with its external environment. Many of these embedded systems perform control activities demanding strict timing restrictions and hence predictability. Two fundamental forms of asynchronous activities are found in those systems: tasks and Interrupt Service Routines (ISRs). Each of them has its own independent scheduling and synchronization policies and mechanisms that are semantically and syntactically different.

The interrupts mechanism synchronizes the occurrence of the external asynchronous events and the ISRs. The synchronization mechanism offered by the operating system synchronizes an internal event with the execution of a task. In order to obtain high efficiency and low latency in the response to interrupts, general purpose (and also real-time) operating systems offer a set of mechanisms to handle interrupts totally independent of those used for task management. Although this scheme is adequate for systems with high processing demands, as those found in database and networking operating systems, in existing real-time systems the differences in the scheduling and synchronization between ISRs and tasks can compromise the temporal predictability of the system.

The tasks are an abstraction of the concurrency model supported by the kernel and the responsibility of their management lies completely on the kernel itself, which provides services for the creation, elimination, communication and synchronization among tasks. On the other hand, interrupts are an abstraction of the computer’s hardware and the responsibility of their management lies in the hardware logic of a specialized circuit. This hardware allows the allocation of ISRs to different interrupt requests, the CPU context switch, the enabling and disabling of specific interrupt requests, and the scheduling of interrupts following a hardware priorities scheme. The operating system provides a set of services that allow the execution of these operations.

The interrupt-handling hardware is responsible for the ISRs scheduling according to its hardware priorities, whereas the tasks are scheduled by the kernel according to their software priorities. Hardware priorities have precedence over software priorities (Figure 1). In general purpose systems, tasks do not have strict timing requirements, so the only activities with “timing” requirements are the ISRs. Consequently, this arrangement makes sense, because it provides low latency to interrupts, avoiding data losses while other tasks are executing. Nevertheless, in real-time systems this scheme introduces unpredictable execution times that compromises the temporal guarantees needed by these tasks.

The synchronization among tasks is achieved using any of the mechanisms provided by the operating system for the synchronization between concurrent processes (i.e., semaphores, messages, mailboxes, etc.). The synchronization among ISRs is reduced to the mutual exclusion achieved with the help of its own scheme of priorities.

In most-common designs, a priority is assigned to each interrupt request, allowing the arrival of higher priority requests during the execution of an ISR. In this scheme, known as nested interrupts, each ISR is executed as a critical section with respect to itself, to lower priority ISRs, and with respect to the tasks. Although the ISRs are automatic critical sections with respect to the tasks, the opposite is not true. The mechanisms used to guarantee exclusive access to critical sections among tasks, do not guarantee exclusive access of the tasks against the ISRs. The mutual exclusion between ISRs and tasks is only obtained by disabling the interrupts. In order not to affect the system’s response time to urgent interrupts, interrupts are disabled by priorities. That is, a disable threshold (a.k.a. interrupt or IRQ level) is set so that if an interrupt of a priority lower or equal to the threshold occurs it is ignored, otherwise is allowed.

\* This research work has been supported in part by NSF-CONACyT Project 42449-Y, Mexico.
Since the traditional model of interrupt handling is strongly supported by hardware, it yields a fast response to external events and a low overhead. Consequently, it has been the method used in most of the operating systems for embedded and real-time systems. However, its use in these systems causes serious difficulties, which we expose next.

2.1. Problems related to two priority spaces

In the traditional model, the assumption that the timing execution requirements of an ISR have greater importance than those of a task is not valid in real-time systems. The response-time requirement of a real-time task may be even shorter than that of some ISRs. In such a case it may be necessary for the tasks to be assigned a higher priority than those ISRs. For example, the tasks with high priorities may be under the disturbance of hardware events necessary only for low priority tasks. On the other hand, low priority tasks associated to interrupts might not be executed due to temporal overloads, even though their associated ISRs are being executed. This arrangement affects the capacity of meeting the real-time requirements of the system causing priority inversion and decreasing its utilization bound.

2.2. Problems related to interrupt latency

Perhaps the most significant argument against the traditional model can be found in its fundamental objective: reducing interrupt latency to the minimum possible. In order to reduce this latency, the kernel disables the interrupts only for brief periods of time. Nevertheless, this approach cannot prevent the applications from disabling interrupts, because this is the only possible way of synchronization between tasks and ISRs. In fact, the system’s response time to the interrupts cannot be smaller than the maximum time in which the interrupts are disabled anywhere in the system. Since the application is capable of disabling the interrupts for more time than the kernel, the worst-case interrupt latency will be the sum of the latency introduced by the CPU plus the worst-case time on which the interrupts are disabled by the application. In conclusion, even though the kernel can establish a lower bound in the interrupt latency it cannot guarantee its worst-case. Evidences of these facts are discussed in [2].

2.3. Weaken Mutual Exclusion Synchronization

While a low priority task elevates the interruption level to a medium level, to enter to a critical section that it shares with an ISR of medium level, an interruption of high level can be occurring to activate a high priority task, preempting the low priority task. This will decrease the CPU interrupt level, destroying the interruption lock of the low priority task. In order to avoid this situation, the kernel could maintain the state of the interruptions without changes when executing a context switching. However, this approach affects the predictability of the system because the tasks will be executed with several states of interruption, depending on which task has been preempted. The alternative is to force the tasks to always set the interrupt level to the highest possible, to avoid context switches. Nevertheless, this alternative increases the context switch latency.
2.4. Sequencing restrictions

Commonly an ISR will make at least one call to the kernel to indicate the occurrence of some event. This call usually makes ready a task of higher priority than the current task. If a context switch is executed, before the ISR completes, the rest of the ISR will not be executed until the interrupted task is executed; leaving the system in an unstable state. Consequently, if these services are invoked within an ISR, the kernel will have to postpone any context switch until the ISR completes. All the existing solutions to solve this problem, which guarantee the logical correctness of the system, introduce an excessive priority inversion effect because of the context switching or exhibit a temporal behavior very difficult to model and hence to predict [18].

2.5. Complex synchronization constructs

The existing differences among the synchronization mechanisms, used according to the type of asynchronous activity, generate a great variety of situations for the cooperation among them; where only a limited number of situations should occur. This produces an increase in the complexity of the solution for the interactions among them. This situation makes the occurrence of design errors be more probable, affecting negatively the reliability of the system.

3. Integrated mechanism for tasks and interrupts handling

In this section, we present a solution to the problems discussed before, that consists on integrating both types of asynchronous activities (tasks and ISRs) through an unified mechanism of synchronization and scheduling.

The integration of the synchronization mechanism is obtained by handling all IRQs by a universal Low Level Interrupt Handler (LLIH) at the lowest level of the kernel. This LLIH convert all interrupts into synchronization events using the existing abstractions of communication and synchronization among tasks. With this model, the ISRs become Interrupt Service Tasks (ISTs) and will remain idle until an interrupt occurs. In this integrated model, the ISTs are blocked for example by executing wait() on a semaphore or a condition variable associated to the interrupt (for schemes based on communication using shared memory), or by executing receive() to accept messages (for schemes that allow message passing). When an interrupt occurs, the LLIH, will do everything necessary to make the IST executable.

This approach provides an abstraction that assigns the low level details of the interrupt treatment to the kernel, and eliminates the differences between the ISTs and the tasks. The real service of the interrupt lies within the IST, providing total flexibility and making unnecessary for the kernel to handle the specific details of the treatment of different interrupts.

The existence of a unique type of asynchronous activity and a uniform synchronization and communication mechanisms between tasks and ISRs offer a solution to the problems discussed in Section 2 and provide the following advantages:

- Eliminates completely the need of the application to disable interrupts, allowing the kernel to guarantee the worst-case response-time to external events (subsection 2.2).
- ISTs are executed in an environment where they can invoke, without restrictions, any service of the kernel or use any library. This lack of restrictions prevents all the difficulties associated to the mutual exclusion and sequencing restrictions between task and Interrupt handlers, discussed in subsections 2.3 and 2.4.
- Makes the development and maintenance of the system easier (subsection 2.5), because now there is only one mechanism for synchronization and communication among cooperating activities.

The unification of the synchronization mechanism is a necessary but not a sufficient step. The integrated mechanism, illustrated in Figure 2, also includes a unified and flexible space of dynamic priorities for all the activities. With this model we have Software Activated Tasks (SAT) and Hardware Activated Task (HAT). HATs are used to handle interrupts instead of the traditional ISRs (and IST). Both kinds of tasks are the same type of asynchronous activities. This scheme allows the assignment of priorities to all the activities of the real-time system in correspondence with their timing requirements. With this approach, the following advantages are obtained:

- Priority inversion associated to the independent priority space is avoided (subsection 2.1).
- The implementation of an enter/leave protocol to disable ISRs in the kernel is avoided, preventing from potential errors (subsection 2.3).
- The error of the broken interrupt lock (resulting from the task switching) is eliminated (subsection 2.3).
- Interrupts overload situations can be handled using some scheduling techniques, such as the sporadic server [11].

This completely integrated design eliminates the necessity to use the busy wait during the I/O operations, without sacrificing the temporal determinism of the system. In addition, the decrease on the complexity of this integrated design favors the development of reliable systems. Overall, this scheme allows the development of robust, predictable and hence verifiable systems. Consequently, in real-time system kernels, where the timely response to events and the reliability are determining factors, no justification exists to maintain both activities (ISRs and tasks) as separated abstractions.

![Figure 2 - Priorities in the integrated model.](image-url)
4. Schedulability analysis for both models

In this section, we develop a schedulability analysis to evaluate the integrated model. The decrease on the utilization bound is computed as well as the response time obtained from the independent priority space of the traditional model. Also, we analyze the decrease in the utilization bound from context switching in the integrated model. The analysis allows us to evaluate the conditions under which one model is more appropriate than the other.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure3.png}
\caption{Disturbances due to the separate space of tasks and interrupts.}
\end{figure}

4.1. Decrease in the utilization bound

According to the real-time scheduling theory, a task \( t_i \) is schedulable if the following holds:

\[ U_{lab} \geq U_j \]  

where \( U_{lab} \) is the least upper utilization bound, which is \( i(2^{n-1}) \) for Rate Monotonic Scheduling, or 1 if Earliest Deadline First is used. It is assumed that \( U_j \) is the CPU utilization due to task \( t_j \), plus the utilization from the interference of higher priority tasks. This can be computed as follows:

\[ U_j = \frac{C_i}{T_i} + \sum_{j \in P(i)} \frac{C_j}{T_j} \]  

where \( P(i) \) denotes the tasks with priorities higher than \( t_i \).

The timing disturbance of the ISRs on the scheduling of task \( t_i \) can be described using the Generalized Rate-Monotonic Scheduling Theory [6]. As shown in Figure 3, there are two types of such disturbances:

- The disturbance associated to interrupts, with minimum inter-arrival times inferior to those of task \( t_i \) and associated to soft real-time tasks. We call this \textit{disturbance due to soft real-time tasks}. Let us denote \( S(i) \) to be the set of ISRs \( t^S_i \) with these characteristics, each one with computation time \( C_i^S \) and periods \( T^S_i < T_i \). The utilization of an ISR \( t^S_i \) in \( S(i) \) is \( C_i^S/T^S_i \).

- The disturbance associated to ISRs with hard timing requirements, but with minimum inter-arrival times greater than those of task \( t_i \). This disturbance is known as \textit{rate monotonic priority inversion}. Let us denote \( L(i) \) as the set of ISRs \( t^L_i \) with these characteristics and \( C^L_i \) to its computation time. Since the inter-arrival times of these interrupts \( T^L_i \), are greater than \( T_i \), they can preempt only once to \( t_i \). In consequence, the worst-case utilization due to an ISR in \( L(i) \), is given by \( C^L_i/T_i \).

The equation for the utilization bound considering these two disturbances is as follows:

\[ U_i = \left( \frac{C_i}{T_i} + \sum_{j \in P(i)} \frac{C_j}{T_j} \right) + \left( \sum_{k \in S(i)} \frac{C_k^S}{T_k^S} + \frac{1}{T_i} \sum_{k \in L(i)} C^L_k \right) \]  

The first two terms of the equation are identical to those of Equation (2). Therefore, the third and fourth terms are the decrease on the least upper utilization bound produced by the use of an independent space of interrupt priorities. Let us call this utilization decrease as \( U_{IS} \), then Equation (1) can be rewritten as follows:

\[ U_{net} = U_{lab} - U_{Irr} > U_j \]  

where the utilization loss \( U_{Irr} = U_{IS} \) is:

\[ U_{IS} = \sum_{k \in S(i)} \frac{C_k^S}{T_k^S} + \frac{1}{T_i} \sum_{k \in L(i)} C^L_k \]  

In order to minimize \( U_{IS} \), the code of the ISRs \( C^S_i, C^L_i \) must be maintained to a minimum. In this way, an ISR will perform the processing necessary to avoid data losses and to activate a task. Once activated, this task will execute, as other, under the control of the kernel scheduler, assigning a priority to the task according to its timings requirements.

Although this arrangement minimizes the disturbance produced by the ISRs, it does not solve the predictability problem. This problem originates from the incapacity to predict the frequency of the interruptions from all devices in the system. Too many interrupts occurring during a short time interval cause the system unpredictable and may cause some tasks to miss their deadlines. In order to address this problem some systems introduce additional mechanisms to limit the number of the interruptions during certain time intervals [12]. However, it is clear that these mechanisms introduce an additional overhead.

4.2. Increment in the response time

In the traditional scheme, the response time of an event is equal to the worst-case response time of the task that communicates with the ISR. The existence of two spaces of priorities reflects on an increase on the response time of the tasks. The response time \( R_i \) of task \( t_i \), with execution time \( C_i \) and minimum inter-arrival time \( T_i \), can be computed by the following recurrence equation [1]:

\[ R_i^n = C_i + B_i + \sum_{j \in P(i)} \left[ \frac{R_{i}^{n-1}}{T_j} \right] C_j \]
where $R^*_i$ denotes the $n$-th iterative value ($R^*_i = C_i$), $B_i$ is the blocking time of task $t_i$ and $P(i)$ is the set of tasks with higher priority than that of $t_i$. The third term on Equation (6) denotes the total interference suffered by $t_i$ from tasks in the $P(i)$ set. This iterative process ends successfully when $R^*_O = R^*_P$, or unsuccessfully when $R^*_O > D_i$. Where $D_i$ denotes the deadline of task $t_i$. In order to consider the effect of the two spaces of independent priorities in the response time of task $t_i$, we must add to Equation (6) the interference of the ISR sets $S(i)$ and $L(i)$, to the response time of task $t_i$. Adding this interference to Equation (6) we have the following:

$$R^*_O = \left( C_i + B_i + \sum_{j \in P(i)} \left( \frac{R^*_P}{T_j} + C_j \right) \right) \sum_{j \in S(i)} \left( \frac{R^*_P}{T_j} + C_j \right) + \sum_{j \in L(i)} \left( \frac{R^*_P}{T_j} + C_j \right) \left( 7 \right)$$

The first section of Equation (7) includes three terms identical to those of Equation (6). The remaining terms (second section) denote the disturbance of using an independent space of priorities on the response time $R_O$. However, since Equation (7) is a recurrence equation we cannot quantify the terms of both sections separately, as in the utilization case (subsection 4.1). It is important to note that a small increase in the second section of the equation can produce a big increase in the response time of the task.

### 4.3. Overhead in the integrated model

The disadvantage of the proposed integrated model is the overhead introduced by the context switching of the HATs (that were treated before as ISRs). This overhead causes a decrease in the utilization bound.

Let $H(i)$ be the set of all activities $t^H_i$ with execution time $C^H_i$ and minimum inter-arrival time $T^H_i$ (lower than period $T_i$ of task $t_i$), which is handled by an ISR in the traditional model. Let $\delta^H$ be the total CPU time for the code of the enter and leave of the ISR (e.g., prologue and epilogue), needed to save and restore the state of the CPU and keep track of the nesting of the ISRs. Let $c^H_i$ be the execution time from the interrupt handler itself. Then, the total execution time of an ISR in the $H(i)$ set can be computed by $C^H_i = c^H_i + \delta^H_i$. Therefore, Equation (2), including $\delta^H_i$ can be re-written as follows:

$$U^H_i = \frac{C_i}{T_i} + \sum_{j \in P(i)-H(i)} \frac{C_j}{T_j} + \sum_{j \in H(i)} \frac{C^H_j}{T^H_j} + \delta^H_i \left( 8 \right)$$

On the other hand, since in the integrated model all activities in the $H(i)$ set are treated as HATs, their context switch time must be considered. Let $\delta^P$ be the context switch time. Then, the execution time $C^H_i$ of a HAT in the $H(i)$ set can be denoted by $C^H_i = c^H_i + 2\delta^P_i$. In consequence, Equation (2), including $\delta^P$ can be re-written as follows:

$$U^P_i = \frac{C_i}{T_i} + \sum_{j \in P(i)-H(i)} \frac{C_j}{T_j} + \sum_{j \in H(i)} \frac{C^P_j}{T^P_j} + \delta^P_i \left( 9 \right)$$

Therefore, the decrease in utilization $U_{ini} = U^H_i$ due to the overhead produced by the activities in the $H(i)$ set as HATs, is given by:

$$U^H_i = \sum_{j \in H(i)} \frac{C^H_j}{T^H_j} + \delta^H_i - \sum_{j \in H(i)} \frac{C^H_j}{T^H_j}$$

The overhead of the integrated model will be smaller than that of the priority inversion effect of the traditional model if the following condition holds:

$$U^H_i < U_{ini} \left( 10 \right)$$

Following Equation 11, if we compare the decrease in the utilization bound of the traditional interrupt model $U_{ini}$ (Equation 5), against the decrease introduced by the integrated interrupt model $U^H_i$ (Equation 8) due to the additional overhead in the context switch, it is possible to observe that in most of the cases the savings obtained using the traditional model are far smaller than those of the integrated model, because of the priority inversion effect introduced.

In any case, it could be possible to design a hybrid model with a configuration in which some activities are treated as ISRs and others as HATs to satisfy the condition stated in Equation (11). For instance, since the timer interrupt will always have the highest priority in the system and will never be handled by the application, it could be considered as an ISR. This reduce the $H(i)$ set, therefore reducing $U^H_i$.

---

**5. Design of the low-level interrupt management system**

In this section we describe the design of the integrated interrupt management system described in section 3. The UML diagram of Figure 4, shows the relationships among the components involved in the interrupt management system. All kernel components that communicate with the interrupt management system use the iKRNLINT interface.

The interrupt management system is divided into two components. The first is the KRNLINT (Kernel Interrupt Management) component, which contains the hardware-independent management code. The second is the INTHAL (Interrupt Hardware Abstraction Layer) component, which contains the hardware-dependent management code. All communication between KRNLINT and INTHAL occurs through the iINTHAL interface.

---

**Figure 4 – INTHAL component Interfaces**
5.1. Kernel interrupt management component

The responsibility of the KRNLINT component is to supply the low-level mechanisms which allow the rest of the system (specifically the scheduling modules and the synchronization and communication modules) to treat the interruptions using the same scheduling and synchronization policies than those used for the real-time tasks. The responsibilities of this component are:

- To allow the association among synchronization objects (i.e., semaphores, mailboxes, etc), all of them identified by a single synchronization identifier of type synctl for each one of the hardware interrupt request lines (addSync() in Figure 4).
- To generate a signal over the synchronization objects each time an interrupt request arrives.
- To supply mechanisms for allowing the management of the interrupt.

The KRNLINT component creates the kernel interrupt abstractions, which are identified by a pre-defined interrupt identifier of type irqId. Each kernel interrupt is associated with a priority within the unified space.

5.2. Interrupt hardware abstraction layer

The INTHAL component provides interrupt management at the lowest level. It is in charge of those aspects dependent of the interrupt hardware so that the system becomes as independent as possible from the computer architecture. The responsibilities of this abstraction are the following:

- Provide a set of interrupt request lines independent of the hardware architecture, denoted as IRQ (Interrupt Request), that go from IRQ0 to IRQn (only n depends on the hardware architecture).
- Provide the mechanisms to set the priorities for each of theIRQx lines, independently of the interrupt hardware priorities.
- Provide the capacity for setting an interrupt level under which interrupts are disabled.

The priorities for each interrupt can be set between 0 and 255 (the highest priority is 255). The value “0” is reserved and indicates that the corresponding interrupt request is disabled.

When the system is started (see Figure 5), all IRQs are in an ignored state. An IRQ changes to a captured state when the kernel requests attention to the IRQ explicitly by invoking an enableIrq() service. A captured IRQ can be in an enabled or disabled state. It is enabled when their IRQ level is above the current IRQ level. The activation of the captured and enabled IRQs produce the invocation of the IRQHandler() kernel routine. An IRQ is disabled when its level is below or equal to the current IRQ level (in this case IRQHandler() is not invoked).

Once an IRQ is captured, its priority can be modified at any moment using the setIrqPriority(irq, priority) service. The current system interrupt level can be set at any moment using the setIrqLevel(priority) service. All IRQs with a priority below the system interrupt level are disabled. After an IRQ has been captured and each time it is triggered, if its priority is greater than the current system interrupt level then control is transferred to the IRQHandler(irq) service (passing the corresponding IRQ as a parameter).

![Statechart of an interrupt request line](image)

6. Implementation over conventional PC interrupt hardware

The design described in Section 5 allows the kernel to be completely independent of the interrupt hardware. Several alternative modules of iINTHAL can be implemented, each one for different interrupt hardware architectures.

A possible implementation of this hardware was proposed in [9] using an FPGA to implement a Custom Programmable Interrupt Controller that cooperates with the kernel scheduler to schedule jointly tasks and interrupts. This implementation has the advantage of introducing a minimum overhead, but its implementation may not be possible on many systems with conventional interrupt hardware.

![Virtual custom programmable interrupt controller](image)
In order to provide an alternative to a customized implementation, we have made an implementation for the standard PC interrupt architecture. In this case, neither the 8259A traditional programmable interrupt controller (PIC) nor the modern advanced PIC (APIC) included in the recent PCs, provide the necessary flexibility to implement the integrated interrupt and task model proposed in this paper.

To address this problem, now the INTHAL has the duty of establishing an interrupt priority scheme incompatible with the built-in hardware interrupt architecture. In order to achieve this goal, first the INTHAL has to cancel the built-in hardware interrupt priorities. This in-habilitation is possible by manipulating appropriately the Interrupt Service Register (ISR) and the Interrupt Mask Register (IMR) of the 8259A PIC. This manipulation is achieved in two stages:

1. **Cancellation of the PICs automatic priority handling**: Essentially, the INTHAL must assure that the ISR registers of both 8259A be set to allow all IRQ enabled explicitly by the IMRs. This is possible by capturing all ISRs and:
   - **EOI Mode**: sending the end of interrupt command (EOI) to the 8259A controller.
   - **AEOI Mode**: using the 8259A automatic end of interrupt operation mode.

2. **Software Priority Management**: Once each IRQ occurs, the INTHAL must set explicitly the IMR registers of each 8259A with a mask to disable all IRQs with smaller or equal priority (included the current IRQ) and enable all IRQs with higher priorities.

To achieve this purpose the INTHAL is in charge of maintaining the state of a Virtual Custom Programmable Interrupt Controller (VCPIC) capable of supporting the integrated model (Figure 6). The VCPIC keeps a table with the current priority for each IRQ and the current system priority level. Any time there is a change in the status of the VCPIC the INTHAL calculates and sets the appropriate mask for the IMR of the two 8259A interrupt controllers.

### 6.1. Analysis of the software implementation

The implementation of the VCPIC introduces an additional overhead in the context switch due to the need of calculating (and setting) the current interrupt level in the system interrupt hardware. Let \( \delta^M \) be this overhead time. Then Equation (9) including \( \delta^M \) can be rewritten as follows:

\[
U_i^P = \frac{C_i + 2\delta^M}{T_i} \sum_{j \in \{(P(i))-H(i))\}} \frac{C_j + 2\delta^M}{T_j} + \sum_{j \in B(i)} \frac{C_j^H + 2\delta^H + 2\delta^M}{T_j}
\]

Now the decrease in utilization \( U_{lost} - U_i^{P*} \) due to the overhead of the integrate model will become:

\[
U_i^{P*} = \frac{2\delta^M}{T_i} \sum_{j \in \{(P(i))-H(i))\}} + \frac{2\delta^M}{T_j} \sum_{j \in B(i)} + 2\delta^H + 2\delta^M - \delta^i
\]  

As may be noted, designing and implementing the system using the VCPIC, imposes a penalty in the performance of the system. It is worth nothing that, even when \( U_i^{P*} \) will not be smaller than \( U_{lost} \), this scheme has the advantages discussed in Sections 3. Specially in those systems that the temporal determinism is so important, this implementation may be an attractive alternative to the interrupts avoidance [7]. Many applications of this kind, can pay in overhead for obtain determinism without sacrificing the benefits of the treatment by interrupt

### 6.2. Using virtual interrupt masking

The penalty to the performance analyzed in the previous section can be diminished greatly by the implementation of optimized priority management techniques in the kernel, such as an adaptation for the integrated model of the optimistic interrupt protection introduced in [17].

With this technique, that we call virtual masking, when the system interrupt level is raised from a level A to a level B, the IRQs with priority levels between A and B are not really disabled so that these undesired IRQs can occur. If any of these IRQs occurs, then the IRQ is really masked to avoid future occurrences. In the case, the interrupt request is recorded so that it can be issued after the priority level is low enough. This avoids the interrupt masking overhead, because most of the times the interrupt request does not occur while the system is executing high priority task. Furthermore, when the system priority is decreased, it is necessary to verify whether an IRQ that has been masked could occur at the new level, and if this is the case, to modify the mask of those IRQs that should be enabled.

### 6.3. Analysis with virtual masking

When virtual masking is used, the masking of an IRQ may occur only if this IRQ really takes place (in undesired form), whereas the unmasking only takes place, if as part of a context switch, which exits some activity, it is needed to enable this IRQ again.

For the activities in the \( P(i) \) set, the worst-case situation occurs when all the IRQs in the \( H(i) \) subset occur in an undesired form while the activities in \( P(i) \) of greater priority than the corresponding IRQ are being executed. In this case, each of them would imply a first writing of the mask from its handler and a second writing when the preempted (in an undesirable way) activity in \( P(i) \) ends. However, since this writing takes place only when the IRQs occur, it should not be associated to each context switch in \( P(i) \). Instead, it is enough to associate two mask writings (\( 2\delta^M \)) to each possible activation of the activities in \( H(i) \).

However, now it is also necessary to take into account the disturbance associated to the potential execution of a small prologue dedicated to attend one of the IRQs associated to any of the activities in the \( S(i) \) and \( L(i) \) sets (not the handling activity itself). In fact, this prologue is the one who sets the real mask, so it can be executed only once. Also, in this case only one masking must be taken into account, because the context switch of any activity in \( P(i) \) never produces the unmasking of any of the IRQs associated to activities in \( S(i) \).
or $L(i)$.

Consequently, now equation (9) can be expressed as follows:

$$U^P_i = C_i + \gamma + \delta^M + \sum_{j \in (P(i), H(i))} C_j + \sum_{j \notin (H(i))} c''_j + 2\delta^P + \gamma + 2\delta^M$$

Where $\gamma$ is the execution time of the prologue associated to the undesired IRQs, which is needed to record their occurrence. Hence, the decrease in the utilization $U_{loss} = U^P_i \times T_C$ due to the overhead of the integrated model with virtual masking is:

$$U_{loss} = \frac{\gamma + \delta^M}{T_C} + \sum_{j \notin (H(i))} \left( 2\delta^P + \gamma + 2\delta^M \right)$$

(13)

Note that, different from the traditional scheme, which uses a minimal ISR and delegate the service at task level, this virtual masking scheme is temporally predictable. It introduces a priority inversion due to a small disturbance caused by the execution of a prologue of an undesired interrupt (given by $\gamma + \delta^M$). However, as showed in equation (13), this priority inversion is bounded. This scheme guarantees a predictable and efficient interrupts management with a very small utilization loss. Also, note that now $U_{loss}$ in Equation 13 depends only on the HATs in the $H(i)$ set and not on the SAT as occurs in Equation (12), making the system more scalable.

7. Experimental results

In this section we show experimental results that allow us to verify experimentally the feasibility and deterministic behavior of the implementation of the interrupt model with a single priority space and integrated synchronization over a conventional PC hardware. The experiments were executed in PARTEMOS², a experimental microkernel developed for experimental novel models of interrupt and exception handling for embedded and real-time system (see [10]). We are using an Intel Pentium 4 PC running at 3GHz with 1GB of memory and 1MB of L2 cache memory. All timing measurements were made using the Time Stamp Counter register of the processor.

We conduct three different experiments with different configuration of a task set consisting of the following tasks:

- $t_1$ is an IRQ handler (without hard real-time requirements) that attends the serial port (receiving 100 bytes per second) with a minimum inter-arrival time $T_{1}^{s}$ of 10 ms and a worst-case execution time $C_1^{s}$ of 5 ms (utilization $U_1^{s} = 0.5$).
- $t_2$ is a periodic hard real time task with a period $T_2$ of 50ms, a worst-case execution time $C_2$ of 20ms (with a utilization $U_2 = 0.4$), and a deadline of 30ms.

In all experiments, the traces for the start and end of both activities were logged, as well as, a trace for each time that the INTHAL LLIH is invoked passing as a parameter the IRQ associated to the serial port (and to the HAT $t_1^{s}$)

---

² Portable and Adaptable Real-Time and Embedded Microkernel Operating System.

---

In the first experiment, no integrated priority space is used and hence the $t_1$ has a higher priority than $t_2$. Figure 7 depicts the execution trace of the task set over a period of time. In this trace two things are worth noting. Firstly, after the first activation of task $t_2$ at time 0, it suffers multiple preemptions by the ISR. Secondly, these preemptions forces $t_2$ to miss its deadline at time 30000 μs. This situation is repeated for all activations of $t_2$ shown.

In the second and third experiments we used our integrated model which allows us to assign the periodic task $t_2$ a priority greater than the priority of the HAT $t_1$. Note that, for this particular task set, this priority configuration is the only one that guarantees the temporal requirements of the
periodic task. Also, this configuration is only possible with the integrated interrupt and task model.

In the second experiment we use the emulation of the VCPIC with the physical masking as first proposed in section 6. Figure 8 depicts the execution trace of this experiment. It is worth noting that: (1) the IRQ can not preemp the periodic task \( t_2 \), (there is not LLIH trace) hence, once it is activated (at \( 0 \mu s, 50000 \mu s, 10000 \mu s \) and \( 150000 \mu s \)) it runs without disturbance. Without this disturbance, task \( t_2 \) can finish properly before its deadline in all instances, as shown in the figure (at \( 30000 \mu s, 80000 \mu s \) and \( 130000 \mu s \)); (2) for each period of \( t_2 \) only 4 IRQs are accepted and handled (instead of 5 that should be accepted). During the 20 ms of execution of \( t_2 \) two IRQs are issued by the serial port, but they are not attended because they have lower priority than task \( t_2 \). However, the hardware “remembers” one of them causing the back-to-back execution of the HAT at \( 20000 \mu s, 70000 \mu s \), and \( 120000 \mu s \).

It is worth to make some comments about the losing of some interrupt request signals observed in Figure 9 and that it is caused by this priority configuration. The first comment is that at this point we have an unavoidable trade off: In this task set, the system cannot guarantee the processing of all interrupts and also guarantee the meeting of the deadline of the periodic hard real-time task. Indeed, this is the reason of this priority configuration: to guarantee the temporal requirements of those software activated tasks which have hard real time requirements, in spite of the overload caused by those hardware activated non real-time tasks. The second comment is that by the usage of a real-time analysis we certainly can guarantee that interrupts are never missed when all interrupt sources behave as expected and, at the same time, do not affect the timing requirements of the hard real time tasks.

In the third experiment we used the emulation of the VCPIC with virtual masking as proposed in section 6.2. Figure 9 depicts its execution trace. In this case it is worth mentioning that the HAT associated to the IRQ cannot preempt periodic task \( t_2 \), hence, similar to the previous experiment, task \( t_2 \) can finish before its deadline in all instances (as shown in the Figure at \( 30000 \mu s, 80000 \mu s \) and \( 130000 \mu s \)). However, in this case there is a difference. Now the IRQ really preempt the execution of \( t_2 \). This is shown in the Figure by the LLIH traces (depicted by diamonds), which occur a little later of \( 0 \mu s, 50000 \mu s, 100000 \mu s \) and \( 150000 \mu s \). Note that, here the corresponding HAT is not executed and that, this undesired interrupts are not served at those instants of time, but instead they are recorded (by the synchronization object) until the end of the periodic task (at \( 20000 \mu s, 70000 \mu s \) and \( 120000 \mu s \)). This is illustrated by the activations of the corresponding HAT (where the corresponding LLIH traces are not executed).

In this case it is important to note that, only one undesired interrupt by activation of \( t_2 \) is possible. Again, for each period of \( t_2 \) only 4 IRQs are accepted and handled (instead of the 5 IRQs that were issued by the serial port). On each execution of \( t_2 \) one IRQ is ignored. This restriction guarantees a small bound in the disturbance due to undesired interrupts as denoted by Equation (13).

8. Related work

Micro-kernels or even the latest versions of commercial embedded or real-time operating systems (i.e., RTLinux, Windows CE) provide service to interrupts and locate the device drivers at the user level. However, the interrupt priorities are not integrated with the task priorities, leaving task vulnerable to the priority inversions due to interrupt. These inversions lead to unpredictable execution times. In our case, the single priority model effectively eliminates these priority inversions providing predictable execution times for both interrupts and tasks.

Several research works propose alternatives to avoid the difficulties of the traditional interrupt model for real-time applications. In [15] the indiscriminate use of ISRs is considered as one of the most common errors in real-time programming. Several real-time operating systems have adopted radical solutions where all external interrupts are disabled, except for those that come from the timer and propose to treat all peripherals by polling [7]. Although this solution completely avoids the non-determinism associated to interrupts, it has as a fundamental disadvantage a low efficiency in the usage of the CPU, due to the busy wait in I/O operations. The advantage of our integrated scheme with respect to these proposals is that our scheme achieves temporal determinism without significantly affecting the usage of the CPU.

Several strategies have been proposed to obtain some degree of integration among the different types of asynchronous activities. In [3] a “structured” interrupts treatment scheme is proposed at the task level, introducing an interface independent from the synchronization mechanism which does not consider interrupts with dynamic priorities. In [5] a method is proposed where interrupts are treated as threads. Its proposal does not have as a fundamental goal to achieve temporal determinism, but the increase on the scalability of the system in multiprocessor architectures oriented to network servers operating systems. Consequently, the interrupt threads use a separate (not unified to tasks’) rank of priority levels. In [19] a scheme is proposed where the software priorities are overlapped within the space of interrupt priorities, executing the scheduler as part of an ISR invoked by hardware. Nevertheless, the ISRs priorities are static and the synchronization mechanism is not unified.

In [4] an scheduling analysis is proposed considering the interrupts as the activities with greatest priorities in the system. Other recent research works are: [13] that proposes a schedulability analysis which integrates static scheduling techniques and response time computation; that modifies the exact response-time analysis with information about the tasks release times and deadlines to obtain tighter response times; [11] that introduces static analysis techniques at the assembler level for interrupt-driven software. In [16] the exact schedulability equation [8] is extended to include the overhead of the interrupts in systems with static priorities and extended the model introduced in [14] to include the overhead of interrupt handling. The resulting equation evaluates the tradeoffs of performing the interrupt handling.
inside of an ISR vs. postponing most of the treatment to a sporadic server [14].

It is important to note that, unlike our integrated model, none of the previous research works solve all the problems stated in Section 2, and none of them provide an analysis that includes the disturbance on the utilization and the response-time caused by the use of two spaces of independent priorities. Unlike [16] we extended the utilization bound equation [5] and the response-time [1], and evaluated the possibility of completely eliminating the treatment of ISRs by integrating both types of asynchronous activities.

9. Conclusions

The handling of interrupts in real-time systems has always been a difficult and not overcome research issue. The separation of ISRs and tasks causes severe restrictions that appear on the services of the system that can be invoked within the ISRs. This separation causes the problem of an increase on the complexity of the design and implementation, which decreases the reliability of resulting software. In addition, real-time scheduling theory considers only one priority space which does not match with the current model implemented in real time operating systems. Due to this, the use of two spaces of independent priorities severely affects the determinism and the degree of feasibility in the scheduling of tasks with real-time requirements.

Many real-time operating systems have tried to maintain this traditional model introducing solutions that improve the determinism, including treatment to external events in two or even more levels, each one affecting differently to the response times, to the synchronization requirements, and to the temporal determinism of the system. Nevertheless, all these solutions jeopardize the efficiency and increase even more the complexity of the mechanism of synchronization between the different types of interrupt activities and tasks. All these solutions never achieve a truly deterministic behavior.

In this work, we have provided solid fundaments for the use of an integrated interrupts and tasks handling model for real-time systems executed on conventional PC hardware. An analysis is introduced to compute the disturbances on the utilization and the response times. This analysis evaluates the concrete situations under which our model is superior to the traditional model. In order to achieve the best possible results for a set of real-time tasks, with our analysis it could be possible to set up a configuration in which some interruption requests could be treated as HATs and others as ISRs.

We have designed a low level interrupt-handling component for an operating system based on the integrated model. An implementation of this component on a conventional PC interrupt hardware is presented. Even when this implementation would not have a significant improvement in the CPU utilization, its advantage in temporal determinism, easy of use and reliability, are sufficient to favor it. We improved this implementation with the incorporation of an optimistic interrupt protection scheme and showed its feasibility in a real-time context.

The complete integrated model proposed in this paper is an important step forward in the achievement of a seamless transition from the analysis model of a real-time system to its actual implementation.

10. References