Predictable Interrupt Scheduling with Low Overhead for Real-Time Kernels

[a]Luis E. Leyva-del-Foyo, [b]Pedro Mejia-Alvarez, [c]Dionisio de Niz
[a]*Dpto. Computación, Universidad de Oriente, 90500, Santiago de Cuba, Cuba*
[b]*Departamento de Computación, CINVESTAV-IPN, Av. I.P.N. 2508, México, D.F., 07300*
[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 inability to incorporate the reliability and temporal predictability demanded by real-time systems. As a result of this analysis, we propose a model that integrates interrupts and tasks handling. We introduce a novel implementation of this model that uses an adaptation of the optimistic interrupt protection technique [12] for achieving predictability and low overhead. The detailed design of a flexible and portable kernel interrupt subsystem for this integrated optimistic model is presented. We make a schedulability analysis to evaluate the optimistic integrated model and perform experiments to verify its deterministic behavior and its overhead

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 [11]. As a result, the interrupt mechanism needs to meet this predictability.

The scheme used for interrupts management in general purpose operating systems is designed to provide a fast response to external events disregarding temporal requirements of the interrupted tasks. Although this scheme is adequate in systems with high processing demands, as those found in database and networking operating systems, it constitutes one of the main causes of temporal unpredictability. Some drawbacks of this model are:

(1) There is a mismatch between some of the assumptions of the scheduling analysis and the runtime behavior which, among other things, includes assuming negligible interrupt execution time. This mismatch has been the focus of study in [6][15] where an analysis is introduced to take into account the effect of interrupt activity.
(2) Timing requirements are jeopardized due to temporal input/output overload. In [9] and [1] different approaches to cope with this interrupt overload are proposed.
(3) There is an excessive memory consumption due to the stack usage in the *Interrupt Service Routine* (ISR). An approach to cope with this problem is proposed in [10].
(4) There is a lack of preemption control over sections of the interrupt handlers. An approach to schedule non-preemptive interrupt handlers is presented in [4].

Although these research works provide solutions to some aspects of the problem, they deal with different manifestations of the same underlying problem: the lack of predictability of a mechanism not designed for real-time systems. On the other hand, these solutions miss another important issue: the use of interrupts in the traditional model is inherently error-prone (due to different task-to-task and interrupt-to-task synchronization schemes) [14] which can easily lead to unreliable systems. An example of this synchronization problem is when a medium-priority ISR that blocks low-priority interrupts is interrupted by a high-priority ISR. If the latter blocks all interrupts at the beginning and unblocks them at the end it will leave the medium-priority ISR exposed to low-priority interrupts for which it is not prepared. Concerning temporal predictability the most common example is the livelock induced by the reception of an infinite stream of network packets that causes all processing time (or an important portion) to be spent on the ISR as discussed in [8].

In order to cope with these drawbacks, in [7] a completely integrated mechanism for interrupt and task management was introduced. This mechanism allows a direct application of the real-time scheduling and concurrency theory while achieving a predictable and dependable behavior without sacrificing the advantages of the asynchronous handling of external events. This scheme allows us to avoid the explicit disabling of interrupts avoiding, in turn, the synchronization problem between ISRs mentioned earlier. In addition, because it blocks interrupts with lower priority than the current task, it also avoids livelocks and unpredictable temporal disruptions.

However, the implementation proposed in [7] has an important overhead that causes a utilization loss, an interrupt latency higher than the one that can be achieved with the traditional interrupt mechanism, and a worse average-case performance. In practical systems, where both hard and soft real-time tasks are included, average performance is as important as the worst-case response.

In this context, the contributions of this work are:

- The introduction of an efficient implementation of the *integrated model for the management of interrupts and tasks* proposed in [7].
- The introduction of an approach for the *optimistic interrupt masking* presented in [12] that is adequate for its use in real-time systems and its schedulabilty analysis.
- A detailed design of an interrupt management subsystem based on the integrated model. This subsystem can be

customized to a wide range of embedded applications with different cost and real-time requirements.

This work is organized as follows: Section 2 briefly presents the integrated interrupt and task model proposed in [7] highlighting its overhead. In Section 3, the optimistic interrupt masking, as introduced in [12], is presented. This is followed by our adaptation to a real-time systems context. Section 4 discusses the design alternatives of the integrated model using virtual masking and the algorithms for its implementation. In Section 5 an analysis of the overhead of this implementation is presented. The experimental results to highlight the advantages of this implementation are presented in Section 6. Finally, Section 7 presents our conclusions.

2. The Integrated Interrupt and Task Management Model

In this section we introduce the integrated interrupt and task management model as well as its analysis and implementation over traditional interrupts hardware.

2.1. Notation Used in this Paper

Throughout this paper we will use the following notation:

- δ^I Total CPU time for the code to enter and leave the ISR. This includes the time to save and restore the state of the CPU, send an *end-of-interrupt signal* to the hardware interrupt controller, and other kernel requirements.
- δ^M Time needed to change, by software, the hardware interrupt priority level (in the hardware interrupt controller).
- δ^P The context-switch time without changing the hardware interrupt priority level.
- c_x Worst-case execution time (without kernel or system overhead) of an asynchronous activity t_x (task or ISR).
- T_x Period or minimum inter-arrival time of the asynchronous activity t_x (task or ISR).
- C_x Worst-case execution time (including any execution time for kernel or system overhead) of the asynchronous activity t_x (task or ISR): For an ISR $C_x = c_x + \delta^I$. For a Task $C_x = c_x + 2\delta^P$.
- p_x Priority of task t_x.
- t_i Hard real-time task with period (or minimum inter-arrival time) T_i, execution time C_i and priority p_i.
- *P(i)* Activity set (ISRs or tasks) with priority higher than the priority p_i of the task t_i.
- *U(i)* Task set with priority lower than the priority p_i of the task t_i.
- *S(i)* ISR set t_k^S with no hard real-time requirement that has minimum inter-arrival times T_k^S smaller than those of task t_i and computation time C_k^S.
- *L(i)* ISR set t_k^L with hard real-time requirement that has minimum inter-arrival times T_k^L greater than those of task t_i and computation time C_k^L.
- *H(i)* IRQ handler set t_j^H with hard real-time requirement that has minimum inter-arrival times T_j^H lower than those of task t_i and computation time C_j^L.

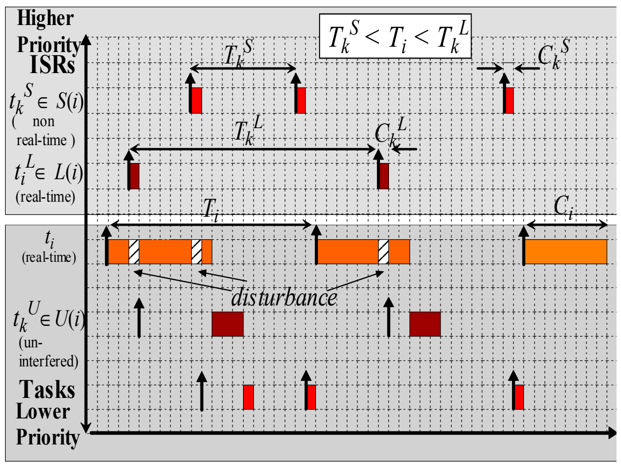

Figure 1 – Traditional Interrupt Handling Model

2.2. Motivation of the Integrated Model

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

$$U_{\text{lub}} \geq U_i \tag{1}$$

where U_{lub} is the *least upper utilization bound*, which is $i(2^{1/i}-1)$ for a static rate-monotonic priority assignment, or 1 if a dynamic priority assignment scheme is used (e.g., Earliest Deadline First). It is assumed that U_i is the CPU utilization due to task t_i, plus the utilization from the *interference* of higher priority tasks. This can be computed as follows:

$$U_i = \frac{C_i}{T_i} + \sum_{j \in P(i)} \frac{C_j}{T_j} \tag{2}$$

As shown in Figure 1, Equation 2 does not take into account the disturbances from interrupts. This Figure shows the temporal behavior of a group of asynchronous activities (ISRs and Tasks) in a real-time operating system that uses the traditional model of interrupts management (i.e., Windows CE 3.0+ [13], Real Time Linux [5], etc.) In this model all ISRs hold a priority above the task priorities, therefore *H(i)* ∈ *P(i)*, *S(i)* ∈ *P(i)*, *L(i)* ∈ *P(i)*. As a result, the ISRs associated to *S(i)* and *L(i)* introduce a *disturbance* in the execution of task t_i. This *disturbance* is manifested as a decrease of the *least upper utilization bound* U_{loss}:

$$U_{\text{lub}} - U_{loss} \geq U_i \tag{3}$$

In the traditional model, $U_{loss} = U_{iS}$, where U_{iS} can be obtained as follows [7]:

$$U_{iS} = \sum_{k \in S(i)} \frac{C_k^S}{T_k^S} + \frac{1}{T_i} \sum_{k \in L(i)} C_k^L \tag{4}$$

In order to minimize U_{iS}, the code of the ISRs (C_k^S, C_k^L) must be maintained to a minimum. Therefore, in the traditional model, the interrupt handling is executed in two phases (see Figure 1): in the first phase, an ISR will perform

IEEE COMPUTER SOCIETY

the processing necessary to avoid data losses and to activate an *interrupt service task* (IST). Once activated, this task will execute, as other tasks, under the control of the scheduler, assigning a priority to the task (according to the requirements of the real-time application). The actual response to the interrupt occurs in the second phase as part of this IST.

Although this approach minimizes the disturbance produced by the ISRs, it faces two important drawbacks:

The remainder disturbance (given by Equation 4) is not negligible: the time needed to service an ISR is dominated by the input/output operations. Yet with this scheme the ISR has to do at least two port operations: one for servicing the device that issues the interrupt request and another for acknowledging the hardware interrupt controller.

It does not solve the predictability problem. The predictability problem originates from the inability to predict the frequency of the interrupts from all devices in the system. Too many interrupts occurring during a short time interval make the system unpredictable and may cause deadline misses.

In order to remove these drawbacks (and many other related to the synchronization between ISRs and Tasks) an integrated model for task and interrupt management was introduced in [7].

2.3. The Integrated Model

The integrated interrupt and task model unifies two important aspects of the interrupt and task management:

(1) *The scheduling and priority space*: all activities in the system (ISRs and Tasks) share a unified and flexible space of dynamic priorities and are scheduled cooperatively by the kernel and the interrupt hardware.

(2) *The synchronization mechanism*: all IRQs are handled by a *Low Level Interrupt Handler* (LLIH) at the lowest level of the kernel, which converts them into synchronization events using the abstractions of communication and synchronization among tasks (i.e., semaphores, mailboxes, etc).

With this model we have *Software Activated Tasks* (SAT) and *Hardware Activated Tasks* (HAT)[1]. HATs are used to handle interrupts instead of the traditional ISRs. Both kinds of tasks are the same type of asynchronous activities and allow the following: $S(i) \in U(\mathrm{i})$, $L(i) \in U(\mathrm{i})$. As $U(i) \cup P(i) = \emptyset$, the disturbance associated to $S(i)$ and $L(i)$ is completely avoided and $U_{iS} = 0$.

2.4. Low Level Interrupt Management

The implementation of this model lies on two software components: the *Kernel Interrupt Management Component* (KRNLINT) which unifies the synchronization mechanism and the *Interrupt Hardware Abstraction Layer* (INTHAL) which unifies the priority space. The UML diagram in Figure 2 shows the relationships between these two components. The **iKRNLINT** interface allows the communication between the interrupt management subsystems and the rest of the kernel. This interface is implemented by the KRNLINT using the interface **iINTHAL** provided by the INTHAL.

[1] Although in [7] the term IST was used, here we prefer to use the term HAT to emphasize the difference against the ISTs present in other systems (Windows CE 3+) which do not integrate the priority space for IRQs and tasks

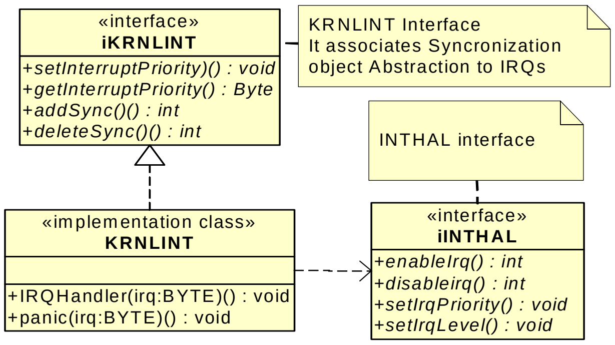

Figure 2 – INTHAL component Interfaces

2.4.1. Interrupt Hardware Abstraction Layer.

When the system is started, 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 produces 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).

2.5. Original Emulation

Computer systems using Intel family processors and compliant with the industry standard, use standard interrupt hardware composed by two *Programmable Interrupts Controllers* (PIC) 8259A chips (or equivalent inside the motherboard chipset) connected in cascade. This configuration provides 16 IRQ lines (IRQ0…IRQ15) ordered by priorities in the following way: IRQ0 (highest priority), IRQ1, IRQ8, IRQ9,…, IRQ15, IRQ3 ,…, IRQ7 (lowest priority). This hardware (and the modern advanced PIC included in the most recent PCs) does not provide the necessary flexibility to implement the integrated interrupt and task priority space. To address this problem in the emulation presented in [7], the INTHAL provides a *Virtual Custom Programmable Interrupt Controller* (VCPIC) which manipulates the *Interrupt Service Register* (ISR) and the *Interrupt Mask Register* (IMR) of the two PIC 8259A [3]. This emulation is performed in two stages:

(1) *Cancellation of the PICs automatic priority handling*: assuring 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, it 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 to enable the others.

2.6. Overhead of this Implementation

To implement the integrated model, the kernel must call the service **setIRQLevel**() (which yields a worst-case execution time of δ^M), as part of any context switch to inform the VCPIC about the current priority level. This service must calculate the IMR for the current interrupt level and set it executing an expensive port operation. This introduces an important overhead in the context switch time, causing a decrease in the *least upper utilization bound*. This decrease $U_{loss} = U_i^{PI^*}$ can be expressed by the following equation [7]:

$$U_i^{PI^*} = \frac{2\delta^M}{T_i} + \sum_{j \in (P(i)-H(i))} \frac{2\delta^M}{T_j} + \sum_{j \in H(i)} \frac{2\delta^p + 2\delta^M - \delta^I}{T_j} \quad (5)$$

Modern processors use deep pipeline and superscalar execution. In these processors, any modification to the interrupt mask typically requires a pipeline flush which limits the capacity of simultaneous execution, limiting the performance severely. Also, most of the processors only incorporate part of the interruption masking logic in the chip and the rest is implemented by a hardware controller outside the processor package requiring a potentially expensive off-chip access.

In this context we identify two fundamental problems: (1) the increase in speed and complexity of the processors implies that any modification to the current interrupt priority level is relatively more expensive, causing an important contribution of δ^M in the above equation (for example, using a Pentium 4 3GHz the **setIRQLevel**() service has a worst case execution time of 1.04292 μs [7]); (2) the decrease in the utilization bound does not depend only on the HATs ($H(i)$ set) but also on all the tasks in the $P(i)$ set. This dependence affects the scalability of the model considerably.

3. The Virtual Interrupt Masking

In this section, we introduce a novel approach for interrupt handling named Optimistic Interrupt Masking and then we present its adaptations for real-time systems.

3.1. Optimistic Interrupt Masking

Kernels of general purpose operating systems must often disable interrupts to avoid preemptions while certain code sections that modify critical data structures are being executed. When the execution of this critical code finishes, the kernel must enable the interrupts again. With the aim of speeding up this entry/leave protocol to the critical sections in these systems, a technique called *optimistic interrupt protection* was introduced in [12].

The idea of optimistic interrupt protection consists of the following steps: (1) When entering to a critical section inside the kernel the protocol sets a *software interrupt mask* to indicate what interrupts must be masked. The hardware interrupt mask is not changed. (2) A prologue code section is located at the entry of all interrupt handlers. This prologue code checks the software interrupt mask to verify if the issued interrupt is logically masked, and if so the execution of the remainder of the ISR is deferred to a later moment. (3) When leaving the critical section the protocol checks if there are any pending interrupts. If this is the case, the control is transferred to the corresponding interrupt handler before resuming the "normal" computation.

In order to simplify the code, optimistic masking recommends the following: In the event of a logically masked interrupt, besides remembering the interrupt request and before returning the control the interrupt prologue should update the hardware mask as specified in the software interrupt mask. In this case, after the deferred interrupts are handled as part of the leaving protocol of the critical section, the hardware interrupt mask must also be restored to its original level.

3.2. Adapting to Real-Time Systems

The performance penalty analyzed in section 2.6 can be diminished greatly if the optimistic interrupt handling approach is adapted to our integrated model. With this technique, 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 occur, then the IRQ is really masked to avoid future occurrences.

Unlike the original idea of optimistic protection, in this case, the masking of undesired IRQs after their occurrence is not optional (with the purpose of simplifying the implementation logic) but rather becomes obligatory in order to guarantee the temporal predictability. In this 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 when the system is executing high priority tasks. 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, modify the mask of those IRQs that should be enabled.

4. Implementing The Integrated Model with Virtual Masking

This section presents the detailed design of the interrupt subsystem for a real-time kernel that is compliant with the integrated interrupt and task model. This subsystem can be configured in different emulation modes to satisfy the different trade-offs between cost and temporal predictability.

We introduce an analysis of the implementation alternatives for the virtual masking mode and discuss its implementation algorithms. Our implementation was made in a custom real-time kernel that runs on a PC running in real mode.

4.1. INTHAL Status Data

In order to maintain the state of the VCPIC, it is necessary to maintain a group of internal variables that are described next.

The following arrays, with one element for each of the 16 IRQs in the system, are maintained:

- **IRQ_Priority**: An array of 16 elements of a type compatible with the type of the system priorities (byte or word) that holds the priority of each IRQ in the system.
- **IRQ_Mask**: An array of words (16 bits). Each word keeps one mask to be set in the IMR registers of both 8259s when the corresponding IRQ occurs. This word unmasks the IRQ of greater priority than the corresponding IRQ and masks all remaining IRQs (including the corresponding IRQ).

In addition the following variables are maintained:

- **Virtual_Mask_Mode**: Flag that signals the masking mode in operation: virtual (TRUE) or physical (FALSE).
- **IRQ_Level**: Byte or word (according to the system) that keeps the current (unified) system priority level.
- **Logical_Mask**: Interrupt mask (word value) that corresponds to the current system priority (**IRQ_Level**).
- **Physical_Mask**: Interrupt mask (word value) set in the IMR registers of both 8259s. In the physical masking mode, this mask will always be equal to **Logical_Mask**, however, in virtual masking mode it may be different to **Logical_Mask**.

4.2. General Emulation Logic and Implementation Alternatives

In the virtual masking mode, the occurrence of *undesired interrupts* is possible. Any undesired interrupt I with priority P_i must fulfill the condition $P_i \leq$ **IRQ_Level**. If this happens, this IRQ is physically masked in the IMR register of the corresponding 8259 (the **Physical_Mask** is updated to reflect this fact), so that the occurrence of a second undesired interrupt is avoided. Also, the occurrence of this IRQ must be recorded. In this case the current interrupt level (value of **IRQ_Level**) is not modified.

4.2.1. Mechanism for masking an undesired IRQ

To guarantee temporal predictability in the virtual mask mode the masking of the undesired IRQ I inside of the LLIH (section 3.2) is required. There are three possible ways to carry out this masking, all of them guaranteeing a maximum bound in the priority inversion due to the disturbance of these undesired IRQs:

Masking only the undesired IRQ I that took place: This masking involves computing and setting a mask that disables the specific IRQ (without modifying the others).

Masking all IRQs with priorities below or equal to P_i: this option has two advantages: (1) it is easy to implement, because it is only needed to set the mask located in **IRQ_Mask[I]** (pre-calculated by **setIrqPriority**()); (2) when **setIrqLevel**() is called to rise the priority level, it does not have to calculate the mask corresponding to the new priority level (it must be done only if it is called to decrease the priority level). Since **setIrqLevel**() must be executed at each context switch, advantage (2) implies a smaller context switch overhead. However, this option has the drawback of allowing the occurrence of other undesired IRQs (all those x that fulfill the condition **IRQ_Level** $< P_x < P_i$). This not only would cause the masking of other IRQs, but also would produce a larger worst-case disturbance due to undesired interrupts.

Masking all IRQs with priorities below or equal to **IRQ_Level**: this option is equivalent to set the physical IRQ level (physical mask) equal to the system (logical) priority level (logical mask). With this option, the service **setIrqLevel**() must compute the interrupt mask that corresponds to each level. This must be done even if the invocation raises or diminishes the current system priority level. The drawback here is a higher context switch overhead. However, it guarantees that once an undesired interrupt occurs, any other one will not occur when the system priority level is higher or equal to the current level (the first undesired IRQ masks all the others). This provides the best possible worst-case in the disturbance due to undesired interrupts.

4.2.2. Mechanism for recording an undesired IRQ

After an undesired interrupt occurred and it is masked, it must be recorded to allow its occurrence only when the system priority goes below the priority of this interrupt. Here, the integrated model of interrupt management allows two options:

a). VCPIC (INTHAL) recording. This option is equivalent to the original idea of the interrupt prologue in the optimistic interrupt masking. In order to achieve this it is necessary to keep an occurrence flag for each possible IRQ (called continuation in [12]) to record the occurrence of the undesired interrupts. When the system interrupt level goes down, the interrupt occurrence is simulated executing **IRQHandler(irq)** in the service **setIrqLevel**() of the INTHAL. This option has the advantage of making the kernel independent of the masking modes in the INTHAL.

It is worth noting that this option is the only one available when a traditional interrupt management scheme is used. Also, note that it causes an increase in the execution time of **setIrqLevel**() δ^M. This increase does not cause any problem in traditional interrupt management systems neither in general purpose operating systems (for which the optimistic masking was designed). This is because **setIrqLevel**() is not executed at every context switching (as in the integrated model), but as part of the entry/leave protocol of the kernel's critical sections. In fact, this service is less expensive than handling the interrupt level directly.

In the context of the integrated scheme of interrupt and task management and for real-time operating systems, the previous arguments are not valid. This is due to a complete change in the scheme and the design objectives of the kernel:

1. Now **setIrqLevel**() is not called as part of the entry/leave protocol to the kernel critical sections, but as part of each context switch. In fact, the context switch itself constitutes a critical section, hence it is not possible to simulate an interrupt within **setIrqLevel**(). It must be noted that, using the proposed integrated model eliminates the need to disable the interrupts inside the kernel. The critical sections inside the kernel are protected simply by disabling the preemption (which is modeled using the immediate priority ceiling protocol [1]).
2. The efficiency of the integrated scheme and the schedulability that can be achieved, is very sensitive to the worst-case execution time of the **setIrqLevel**() service δ^M. Hence, any small increase in δ^M is an important disadvantage.

b). Kernel (KRNLINT) recording. A solution to the difficulties of using the optimistic masking with the integrated model can be found in the model itself. Since this model integrates the communication and synchronization mechanism between interrupt handlers and tasks, then the VCPIC does not need to hide the occurrence of an undesired interrupt from the kernel. Therefore, it is feasible now to notify this event to the kernel for its recording. However, it must be noted that the VCPIC does not need to notify the kernel if an interrupt is desired or not. The kernel itself has enough information to differentiate desired from undesired interrupts (using the system priority level).

The fact is that, when an interrupt occurs (**IRQHandler**(**irq**) is called) the kernel signals the synchronization objects associated to that IRQ and calls the scheduler. Due to the restriction that only signal-recording objects (semaphores, mailbox, etc) can be associated to IRQs the recording of an undesired interrupt is achieved in a transparent form. In this case, a desired interrupt would cause the preemption of the current task to execute the HAT that waits for the IRQ (because it has higher priority), while an undesired interrupt would set ready the associated HAT without preempting the current task (because its HAT has lower priority). The HAT is automatically scheduled when the system priority level goes down.

4.3. Algorithms for the Main VCPIC Services

The implementation of the VCPIC is carried out by two INTHAL components: the priority management services and the LLIH for the captured IRQs.

4.3.1. Priority management services

These services achieve the goal of having IRQs with dynamic priorities within the same priority space of the kernel scheduler. The interface services provided are **setIrqPriority**() and **setIrqLevel**() and the auxiliary services provided are **set8259IMR**() and **setIRQMask**().

The auxiliary service **set8259IMR**(...) must be invoked whenever it is necessary to set the mask in the interrupt hardware. It allows keeping the current value of the mask registers in the **Physical_Mask** variable so that, whenever the new mask matches the mask already set, the expensive input/output operations are avoided. The algorithm for this service is shown in Figure 3. First, it verifies if the physical mask is different from the new mask. If this is true, it sets the masks in both 8259 IMR registers and updates the values of **Logical_Mask** and **Physical_Mask**.

The **setIRQMask**(…) service provides support for virtual masking. As shown in Figure 3, its behavior is related to the masking mode stored in the state variable **Virtual_Mask_Mode**:

- **Physical masking**: this service only calls the **set8259IMR**() service to set the mask in both 8259s.
- **Virtual masking**: this service sets the mask only if it causes an IRQ enable (unmasking). When the new mask causes an IRQ disabling (masking) then only the value of the logical mask (variable **Logical_Mask**) is updated.

When operating in physical masking mode, the value of **Logical_Mask** is always equal to the value of the IMR registers of both 8259s. On the other hand, in virtual masking mode, the value of **Logical_Mask** may be equal or different to **Physical_Mask** (and the IMR registers). However, this mode must always fulfill the condition that the bits in 1 in the **Physical_Mask** must be a subset of the bits in 1 of the **Logical_Mask**. In other words, the following condition must always be true:

(NOT **Logical_Mask** AND **Physical_Mask**) = 0

```
set8259IMR (mask) {
  if ( mask ≠ Fisical_Mask )  {
     Logical_Mask ← mask
     Physical_Mask ← mask
     IMR registers of both 8259 ← mask
  }

setIRQMask (mask) {
  if ( VirtualMaskMode =  TRUE)  {
     if ( Physical_Mask AND ( NOT mask ) )
       set8259IMR(mask)
     else
       Logical_Mask = mask
  }
  else {
    set8259IMR (mask)
  }
}
```

Figure 3 – Auxiliary Services Set8259IMR and setIRQMask

Service **setIrqPriority** (**irq, priority**). This service allows the setting of the priority level of an IRQ. Its function is to establish a correspondence between the priorities assigned to each IRQ (within the system priority space) with the value of the mask to be set in the interrupt hardware (IMR registers of both 8259).

Figure 4 shows **setIrqPriority**() algorithm. This algorithm modifies the **IRQ_Priority** and **IRQ_Mask** arrays (see subsection 4.1). At each invocation, the entry in **IRQ_Priority** that corresponds with the IRQ being changed is updated. Next, it obtains the mask associated to each IRQ from the new priority configuration. These masks are stored in the **IRQ_Mask** array. In addition, if this IRQ goes from enabled to disabled or vice versa, then **setIRQMask**() is called to update the interrupt mask.

Service **setIrqLevel**(**priority**). This service sets the current system interrupt level and maintains the **IRQ_Level** variable. As shown in Figure 5, it uses the **IRQ_Priority** and **IRQ_Mask** arrays to determine the mask to be set for the new priority level. This mask is the one that disables all IRQs with a priority level less than or equal to the **IRQ_Level**. After this mask has been computed, if this implies the masking or the unmasking of some IRQ then it is set according to the masking mode (physical or virtual); with the help of **setIRQMask**().

```
setIrqPriority (irq, priority) {
  /* Compute all masks for the priority setting*/
  IRQ_Priority[irq] ← priority

  - Obtain IRQ masks in IRQ_Masks[] array

  /*  Update de current 8259 mask */
  irqMaskBit ← (1 << irq  /* 1 in IRQ position*/
  if ( priority ≤ irqLevel ) { /* mask IRQ */
    setIRQMask( Phyisical_Mask  OR  irqMaskBit )
  }
  else { /* IRQ activation*/
   setIRQMask(Physical_Mask AND NOT irqMaskBit)
  }
}
```

Figure 4 – Service setIrqPriority

```
setIrqLevel (priority) {
  /* Get the new interrupt mask */
    - obtain the mask which corresponds to the
      actual priorities configuration
      (IRQ_Priority[]  and IRQ_Mask[] arrays)
       and the priority parameter
  /* Set the mask in accordance to the
     masking  mode */
  setIRQMask( mask );
}
```

Figure 5 – Service setIrqLevel

4.3.2. INTHAL Low Level Interrupt Handler (LLIH)

The INTHAL has a LLIH for each possible ISR state (captured or ignored). The control is transferred to these handlers whenever an IRQ with the associated state takes place. The handler gets the requested IRQ as an argument. The CAPTURED_ENTRY algorithm (which is the LLIH associated to the captured interrupts), is shown in Figure 6. Its main responsibilities are: (1) to cancel the PICs traditional priority scheme, (2) to enforce the unified priority space and (3) to transfer the control to the kernel interrupt handler (**IRQHandler**()).

```
CAPTURED_ENTRY(irq) {
  - Save CPU registers
  if ( Virtual_Mask_Mode = TRUE ) {
    if (IRQ_Priority[irq] ≤ irqLevel) {
      Set8259IMR (Logical_Mask)
    } else {
      Logical_Mask ← IRQ_Mask[irq]
      IRQ_Level ← IRQ_Priority[irq]
    }
  } else {
    Set8259IMR(IRQ_Mask[irq])
    IRQ_Level ← IRQ_Priority[irq]
  }
  sendEOI
  IRQHAndler(irq) /* Enter to the kernel */
  - Restore used CPU registers
}
```

Figure 6 – LLIH for captured IRQs

When operating in the physical masking mode, if an interrupt *I* with priority P_i, occurs, it is because the condition $P_i >$ **IRQ_Level** is satisfied. In this case, all IRQs with priority less than or equal to **IRQ_Level** are masked and the system priority level is raised, to set it equal to P_i. Finally, the kernel handler (**IRQHandler**()) is called.

When operating in the virtual masking mode, in spite of the current system priority level **IRQ_Level**, the occurrence of any interrupt is possible. In this case, two situations may occur:

- *An interrupt I with priority P_i* > **IRQ_Level**, *took place (desired interrupt)*. In this case, the same operations that took place in the physical masking mode must be carried out, with the exception of the IRQ masking.
- *An interrupt I with priority P_i* ≤ **IRQ_Level** *took place (undesired interrupt)*. In this case, to avoid the occurrence of a second undesired interrupt, the IMR register of both 8259s must be set using the mask which correspond to the priority level active when the interrupt took place (in addition, the physical mask is updated). Finally, the **IRQHandler**() is invoked to record its occurrence (even though the interrupt is undesired, see subsection 4.2.2). Note that in this case, the current interrupt level (**IRQ_Level**) is not modified.

5. 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 U_{loss} (equation 4) can be expressed as follows:

$$U_i^P = \frac{C_i+\gamma+\delta^M}{T_i} + \sum_{j\in(P(i)-H(i))}\frac{C_j}{T_j} + \sum_{j\in H(i)}\frac{c_j^H+2\delta^p+\gamma+2\delta^M}{T_j}$$

IEEE COMPUTER SOCIETY

Where γ 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 $U_{loss} = U_i^{PI*}$ due to the overhead of the integrated model with virtual masking is:

$$U_i^{PI*} = \frac{\gamma + \delta^M}{T_i} + \sum_{j \in H(i)} \frac{\gamma + 2\delta^p + 2\delta^M}{T_j} \qquad (6)$$

Note that, different from the traditional scheme, which uses a minimal ISR and delegate the service at task level (subsection 2 and Figure 1), 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 (6), 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_i^{PI*} in Equation 6 depends only on the HATs in the $H(i)$ set and not on the SAT (as occurs in Equation 5), making the system more scalable.

6. Experimental Results

Two types of experiments were conducted. The first type allows us to verify experimentally the deterministic behavior of the implementation of a single priority space with and without virtual masking. The second type was developed to compare the overhead of this new implementation against the overhead of the implementation using physical masking. All experiments were executed using an Intel Pentium 4 PC running at 2.8GHz with 1GB of memory and 1MB of L2 cache memory. All timing measurements were made using the Time Stamp Counter register of the Pentium processor.

6.1. Behavior Characterization

In the first type of experiments we used a task set composed of the following:

- t_1^S 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.

A task set like this can be found in digital control systems where t_2 executes the control loop, while t_1 attends the communications with a remote system (e.g., for reporting or configuration).

Two experiments were executed assigning the periodic task t_2 a priority greater than the priority of the HAT t_1^S. 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 both experiments, the traces for the start and end of both tasks were logged, and 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).

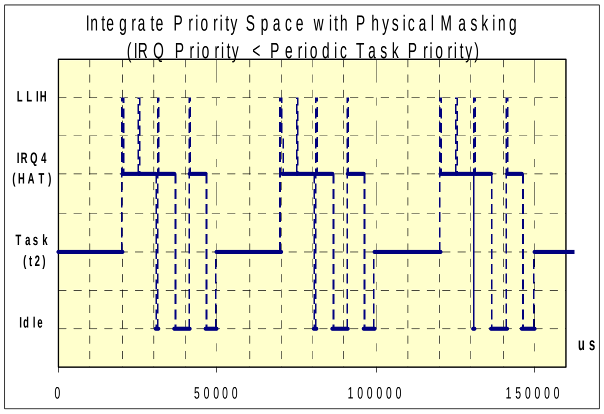

Figure 7 - Execution Trace – Physical Masking Mode

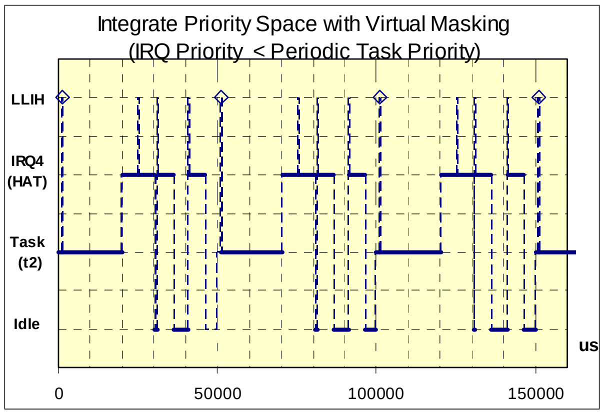

Figure 8 - Execution Trace – Virtual Masking Mode

In the first experiment we use the emulation of the VCPIC with the physical masking mode as proposed in [7]. Figure 7 depicts the execution trace of this experiment. It is worth noting that: (1) the IRQ can not preempt the periodic task t_2, (there is not LLIH trace) hence, once it is actived (at 0 µs, 50000 µs, 10000 µs and 150000 µs) 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 µs, 80000 µs and 13000 µs); (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 µs, 70000 µs, and 12000 µs.

It is worth to make some comments about the losing of some interrupt request signals observed in Figure 7 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 second experiment we used the emulation of the VCPIC with virtual masking mode. Figure 8 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 µs, 80000 µs and 13000 µs). However, in this case there is a difference with respect to the execution of the physical masking mode. 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 µs, 50000 µs, 100000µs and 150000 µs. Note that, here the corresponding HAT is not executed and that, this undesired interrupts are not serviced at those instants of time, but instead they are recorded (by the synchronization object) until the end of the periodic task (at 20000µs, 70000µs and 120000µs). 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 6).

6.2. Overhead Measurement

In this section, we analyze the implementation overhead of the integrated interrupt and task model. In the integrated model, the *interrupt latency* includes the time to setup the priority in the interrupt controller, the time to signal to the interrupt synchronization object (i.e. semaphore) and the context switch time from the interrupted task to the HAT (this timings are not included in the traditional model). Therefore the interrupt latency (of the HAT) is the main overhead indicator.

In our measurements, we used a task with an endless loop that logged a trace with an identification code and a time stamp. Also we used a HAT associated to the IRQ, logging a trace with another identification code and the current time stamp. The values were collected for the four combinations of the two emulation modes (Physical or Virtual Masking) and the two EOI modes (Explicit or Automatic). Figure 9 plots 1000 latency samples[2].

In all modes the behavior of the interrupt latency is practically stable around the average latency: 4.869 µs (for EOI with physical masking), 4.158 µs (for AEOI with physical masking), 3.469 µs (for EOI with virtual masking) and (2.716 µs for AEOI with virtual masking). This is a very important factor for real-time systems. Another observation is that the average values for the virtual masking modes are a 72.9% and a 67% of the corresponding values for the physical masking. This represents a significant reduction in the overhead of this implementation compared with the implementation proposed in [7]. The two virtual masking modes yield better results than the two physical masking modes. The automatic EOI mode with virtual masking shows the best performance with a very low result for the worst-case interrupt latency of 3.06 µs.

It is interesting to analyze how the results for the interrupt latency for each emulation mode are related with the number of writing operations to the input/output port for that mode. This relation is shown in Table 1, where it is easy to identify that the number of port access is the dominant factor for the kernel interrupt latency.

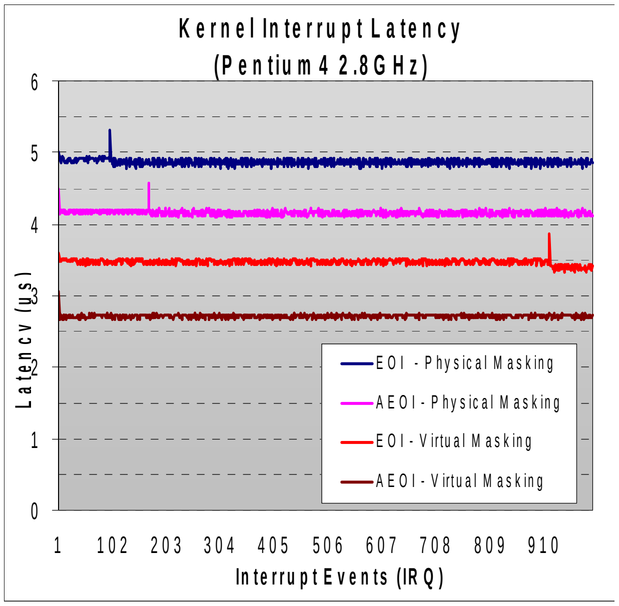

Figure 9 - Execution Trace – Kernel Interrupt Latency

Table 1. Interrupt latency figures and port operations

Emu. Mode		I/0 Write			Interrupt Latency (µs)		
EOI	**Mask**	**EOI**	**IMR**	**#**	**Min**	**Ave**	**Max**
Explí-cit	Phy-sical	1	2^1	**3**	4.781 µs	4.869 µs	5.318 µs
Auto-matic	Phy-sical	0	2^1	**2**	4.097 µs	4.158 µs	4.564 µs
Explí-Cit	Vir-tual	1	0^2	**1**	3.340 µs	3.469 µs	3.877 µs
Auto-matic	Vir-tual	0	0^2	**0**	2.672 µs	2.716 µs	3.060 µs

1. It assumes that it has been necessary to set the IMR in both 8259. Really, often it is only necessary to establish a single IMR. However, this measurement was done with an implementation that always sets the IMR in both 8259 (worst case).
2. A desired IRQ is assumed. In case of an undesired one, the LLIH writes both masks. However, this is not included into the interrupt latency because the ISR is not invoked.

[2] We believe the spikes in the plots are due to speculative execution and/or cache misses.

6.3. Evaluation of the Experimental Results

As final comments of our experiments, we will compare the overhead of the port access between the integrated and the traditional model.

1. In the case of the traditional scheme, the interrupt latency is not affected by the overhead of any port access. Nevertheless, before leaving the ISR, it is incurred in this overhead due to the need to issue an EOI command. Consequently, the port access overhead is indeed reflected in the disturbance (or interference) causing a decrease in the utilization bound.
2. In the case of the integrated model using the automatic EOI mode, the port operations are included in the interrupt latency (due to the need of setting the IMR). Nevertheless, the EOI writing is eliminated completely. Consequently, since the port writing is the dominant factor in the execution time, we expect that the overhead introduced by our scheme to be of the same order as the traditional interrupt management scheme.
3. If the automatic EOI mode is used in combination with the virtual masking then (for desired interrupts) there are no port writings (neither at the entry, nor at the exit of the HAT). Consequently, due to the need to use explicit EOI in the traditional interrupt management, we expect that the overhead introduced by the integrated model indeed be lower than that of the traditional model.

In summary, as demonstrated by equation 6 and the experimental results, the implementation of the integrated model using the AEOI and virtual masking emulation mode allows (in modern architectures) an interrupt management scheme completely predictable and without overhead. Also, the decrease on the complexity of this integrated design favors the development of reliable 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. Therefore, our proposed unified design is well adapted for real-time systems.

7. Conclusions

The details in the implementation of the interrupts handling have a dramatic impact in the design and use of the synchronization mechanisms in real-time and non-real-time operating systems. As a result of the separation of ISRs and tasks, severe restrictions appear on the services of the system that can be invoked within the ISRs. This causes the problem of an increase in the complexity of the design and implementation, which decreases the reliability of resulting software. In addition, the real-time scheduling theory considers only one priority space which conflicts with the actual model implemented in real-time operating systems that has one space for interrupts and another for tasks. The use of these two spaces of independent priorities severely affects the determinism and utilization level in the scheduling of tasks with real-time requirements.

This work eliminates the overhead of the integrated interrupt and task priority space when implemented as in [7]. We introduced a detailed design of a low level interrupt handling component for an operating system based on the integrated model. This component is portable to various hardware platforms, adaptable to different scheduling and synchronizations mechanisms for various operating system implementations. We improved significantly the average case behavior of this implementation with the adaptation of the optimistic interrupt protection scheme to the integrated model and provide a schedulabilty analysis that showed its feasibility in a real-time context. We implemented the various emulation modes presented in this paper for the integrated model as part of an experimental micro-kernel for embedded and real-time applications. Using this implementation we collected experimental evidences that showed its predictable behavior as well as its low overhead. This new implementation is well suited for real-time systems where average case behavior is an important factor.

8. References

[1] Luca Abeni. "Coping with interrupt execution time in RT kernels: A non-intrusive approach". IEEE Real-Time Systems Symposium, Work in Progress, 2001

[2] Theodore P. Baker, "Protected Records, Time Management and Distribution", ACM ADA Letters X(9), pp. 17-28.

[3] Intel, "8259A Programmable Interrupt Controller", Intel , 1988

[4] T. Facchinetti, G. Buttazzo, M. Marinoni, G. Guidi, "Non-preemptive interrupt scheduling for safe reuse of legacy drivers in real-time systems." 17th Euromicro Conference on Real-Time Systems, (ECRTS 2005) 6-8 July 2005

[5] A. C. Heursch; D. Grambow; D. Roedel and H. Rzehak: "Time-critical tasks in Linux 2.6 - Concepts to increase the preemptability of the Linux kernel;" Linux Automation Konferenz, University of Hannover, Germany, March 2004.

[6] Kevin Jeffay, Donald L. Stone, "Accounting for Interrupt Handling Cost in Dynamic Priority Task Systems", Proceedings of the IEEE Real-Time Systems Symposium, Raleigh-Durham, NC, December 1993. pp. 212-221.

[7] L. E. Leyva-del-Foyo, P. Mejia-Alvarez, D. de Niz, "Predictable Interrupt Management for Real Time Kernels over conventional PC Hardware", "Proceedings of IEEE Real-Time and Embedded Technology and Applications Symposium" (RTAS06), San Jose, Cal., April 4 - 7, 2006.

[8] Jeffrey C. Mogul , K. K. Ramakrishnan, "Eliminating receive livelock in an interrupt-driven kernel", ACM Transactions on Computer Systems (TOCS), v.15 n.3, p.217-252, Aug. 1997.

[9] John Regehr , Usit Duongsaa, "Preventing interrupt overload", ACM SIGPLAN Notices, v.40 n.7, July 2005

[10] John Regehr, Alastair Reid, Kirk Webb, "Eliminating stack overflow by abstract interpretation", In Proc. 3rd International Conference on Embedded Software, pp. 306–322, Oct., 2003

[11] John A. Stankovic, "Misconceptions About Real-Time Computing." IEEE Computer, 21 (19), pp 10-19, Oct. 1988.

[12] D. Stodolsky, J. B. Chen, B. N. Bershad, "Fast Interrupt Priority Management in Operating System Kernels" USENIX Symp. Micro-Kernels and others Kernel Architectures", 1993.

[13] Bart Van Beneden, "Executive Summary of the Evaluation Report of Windows CE 3.0 from Microsoft Corporation", Dedicated Systems Magazine, 2001 Q3, 2001.

[14] Ian Sommerville, "Software Engineering", 6th Edition, Addison-Wesley 2001.

[15] D. B. Stewart and G. Arora, "A Tool for Analyzing and Fine Tuning the Real-Time Properties of an Embedded System", IEEE Trans. on Software Engineering, Vol. 29, No. 4, 2003.