Predictable Interrupt Management for Real Time Kernels over conventional PC Hardware[1]

[a]Luis E. Leyva-del-Foyo, [b]Pedro Mejia-Alvarez, and [c]Dionisio de Niz
[a] Universidad de Oriente, Depto de Computación, 90500 Santiago de Cuba, Cuba
[b] CINVESTAV-IPN, *Sección de Computación, Av. I.P.N 2508, 07300 México, D.F.*
[c] ITESO, *DESI. Periferico Sur 8585, Tlaquepaque, Jal. México*
e-mail: pmejia@cs.cinvestav.mx, luisleyva@acm.org, dionisio@iteso.mx

Abstract

In this paper we analyze the traditional model of interrupt management and its incapacity to incorporate reliability and the 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 interrupt hardware and the analysis of its overhead. Finally, experimental results are conducted to demonstrate the deterministic behavior of our integrated model and to quantify its overhead.

1. Introduction

The vast majority of microprocessors are now used for embedded systems, in which computer processors control physical, chemical, or biological processes or devices. Examples of such systems include telecommunication networks (e.g., wireless phone services), tele-medicine (e.g., remote surgery), manufacturing process automation (e.g., hot rolling mills), and defense applications (e.g., avionics mission computing systems). A typical embedded system is I/O-intensive and includes many external devices such as sensors, actuators, network interfaces or disks, requiring in many cases real-time guarantees during its operation. Embedded and real-time operating systems generally rely on interrupts to respond to asynchronous events generated by these devices.

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 where all tasks have timing requirements 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, mutexes, messages, mailboxes, etc.). The

[1] This research work has been supported in part by projects SEP-CONACyT 42151-Y, and CONACYT 42449-Y Mexico.

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.

In general-purpose systems the ISR disabling synchronization is adequate because no task ever need to preempt an ISR. However, in real-time systems it is possible for a task to have a higher priority than an ISR. Hence a selective protection against ISR preemptions is needed to allow some ISRs to preempt a task while disallowing others.

In consequence, we propose an alternate strategy that integrates both types of asynchronous activities, which opposes significantly to the schemes in traditional general purpose and real-time operating systems. The contributions of this work are the following:

- The proposal of an integrated strategy for the management of interrupts and tasks for embedded and real-time systems.
- The evaluation of the integrated scheme from the CPU's utilization point of view and the response time to external events. This evaluation will help system designers to find under which conditions this could be considered more adequate on this type of applications.
- The design of a portable low-level subsystem for interrupt management in real-time embedded systems based on the integrated model.

The rest of the work is organized as follows: In Section 2, we expose the disadvantages of using the traditional interrupt handling strategy for the development of reliable real-time systems. In Section 3, we introduce our integrated interrupt handling strategy with its advantages, when used on real-time systems. In Section 4, we conduct a schedulability analysis in order to compare both strategies. The design of a portable kernel subsystem based in this integrated model is presented in Section 5. The implementation over a conventional PC hardware is presented in Section 6, as well as the analysis of its overhead. In Section 7, experiments are conducted to demonstrate the deterministic behavior of our integrated model and to quantify its overhead. In section 8, related research work is exposed. Finally, in Section 9, our conclusions are presented.

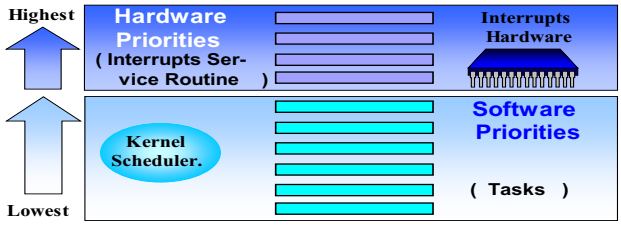

Figure 1- Priorities in the traditional model.

2. Difficulties of the traditional model in real-time systems

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 task 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 may not be executed due to temporal overloads provoked by an excessive number of executions of their associated ISRs (e.g. to get some input). This arrangement affects the capacity of meeting the real-time requirements of the system.

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 [4].

2.3. Weaken mutual exclusion synchronization

Interrupt disabling is frequently done by levels. That is, given interrupt priority levels (priorities) *1,2,…,i,…, n*, disabling interrupt level *i* blocks any interrupt with a priority level <=*i*. The use of this level-based disabling for mutual exclusion with tasks can lead to the breaking of critical section locks. For instance, consider a low priority task t_L that synchronizes with a low priority ISR I_L with interrupt priority level λ. To achieve mutual exclusion for a critical section in t_L, such a task must disable interrupt priority λ. Now suppose a high-priority ISR I_H is activated (given that has a higher priority) and in turn it activates a high priority task t_H. This activation would re-enable the interrupts again, including I_L, breaking the locking of the critical section in t_L. One way to fix this problem would be to keep the interrupt disabling level across context switches. However, this solution would affect the predictability of a task given that its execution may or may not suffer interrupt preemptions depending on which task was preempted (depending on whether the latter has disable interrupts or not). 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

ISRs frequently need to interact with the kernel to process incoming events. At the same time, these events can enable a blocked task. If such a task were urgent, it should, ideally, execute immediately, after which, the ISR should resume. However, allowing such immediate execution could leave the interrupt processing hardware in an unstable state (e.g. blocking other interrupts and losing events). The same problem can happen if the ISR calls parts of the kernel or uses libraries that take a long time to complete. As a result, sequencing restrictions are imposed on ISRs to force them to complete before a task (or calls to some parts of the kernel) is executed. All the existing solutions to solve this problem, which guarantee the logical correctness of the system, introduce an excessive priority inversion effect due to 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 hiding the interrupts at the lowest level of the kernel, which convert them into synchronization events using the abstractions of communication and synchronization among tasks. With this model, the ISRs become *Interrupt Service Tasks* (ISTs) and will remain idle until an interruption 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, an universal ISR, at the lowest level of the kernel, 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 ISRs, 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.

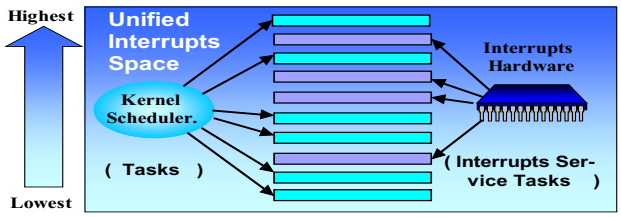

Figure 2 - Priorities in the integrated model.

The unification of the synchronization mechanism is a necessary but not a sufficient step. The integrated mechanism, illustrated in Figure 2, includes a unified and flexible space of dynamic priorities for all the activities (which are activated by hardware or software events). 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 [15].

In addition, the unification of the synchronization mechanism (ISTs can use any traditional sync mechanisms, e.g., semaphores) reduces the complexity of the synchronization constructs supporting 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.

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.

4.1. Decrease in the utilization bound

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

$$U_{\text{lub}} \geq U_i \qquad (1)$$

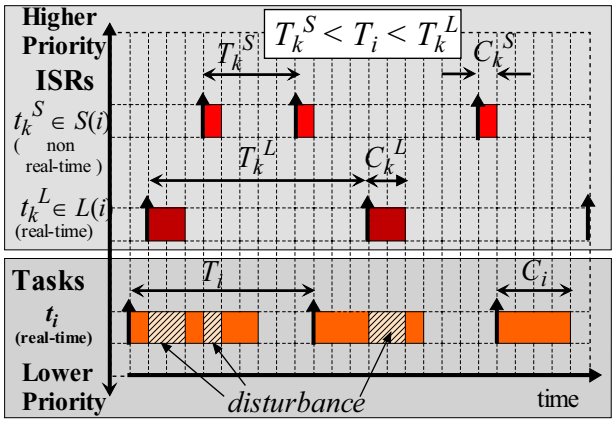

Figure 3 - Disturbances due to the separate space of tasks and interrupts.

where U_{lub} is the *least upper utilization bound*, which is $i(2^{1/i}-1)$ for Rate Monotonic Scheduling, or 1 if Earliest Deadline First is used. 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:

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

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 [9]. 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 *disturbance due to soft real-time tasks*. Let us denote *S(i)* to be the set of ISRs t_k^S with these characteristics, each one with computation time C_k^S and periods $T_k^S < T_i$. The utilization of an ISR t_k^S in $S(i)$ is C_k^S/T_k^S.
- 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 *rate monotonic priority inversion*. Let us denote *L(i)* as the set of ISRs t_k^L with these characteristics and C_k^L to its computation time. Since the inter-arrival times of these interrupts T_k^L, 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_k^S/T_i.

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

$$U_i = \left(\frac{C_i}{T_i} + \sum_{j \in P(i)} \frac{C_j}{T_j} \right) + \left(\sum_{k \in S(i)} \frac{C_k^S}{T_k^S} + \frac{1}{T_i} \sum_{k \in L(i)} C_k^L \right) \qquad (3)$$

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 re-written as follows:

$$U_{net} = U_{\text{lub}} - U_{iS} \quad (4)$$

where:

$$U_{iS} = \sum_{k \in S(i)} \frac{C_k^S}{T_k^S} + \frac{1}{T_i} \sum_{k \in L(i)} C_k^L \quad (5)$$

In order to minimize U_{iS}, the code of the ISRs (C_k^S, C_k^L) 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 tasks, under the control of the scheduler of the kernel, assigning a priority to the task according to the requirements of the application.

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 make 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 [13]. 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]:

$$R_i^n = C_i + B_i + \sum_{j \in P(i)} \left\lceil \frac{R_i^{n-1}}{T_j} \right\rceil C_j \quad (6)$$

where R_i^n denotes the *n-th* iterative value ($R_i^0 = C_i$), 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 $R_i^{n-1} = R_i^n$; or unsuccessfully when $R_i^n > D_i$. 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:

$$R_i^n = \left(C_i + B_i + \sum_{j \in P(i)} \left\lceil \frac{R_i^{n-1}}{T_j} \right\rceil C_j \right) + \left(\sum_{k \in S(i)} \left\lceil \frac{R_i^{n-1}}{T_j} \right\rceil C_k^S + \sum_{k \in L(i)} C_k^L \right) \quad (7)$$

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_i. 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 ISTs (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_j^H with execution time C_j^H and minimum inter-arrival time T_j^H (lower than period T_i of task t_i), which is handled by an ISR in the traditional model. Let δ^I 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_j^H 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 $C_j^H = c_j^H + \delta^I$. Therefore, Equation (2), including δ^I can be re-written as follows:

$$U_i^I = \frac{C_i}{T_i} + \sum_{j \in (P(i) - H(i))} \frac{C_j}{T_j} + \sum_{j \in H(i)} \frac{c_j^H + \delta^I}{T_J^H} \quad (8)$$

On the other hand, since in the integrated model all activities in the $H(i)$ set are treated as ISTs, their context switch time must be considered. Let δ^P be the context switch time. Then, the execution time C_j^H of an IST in the $H(i)$ set can be denoted by $C_j^H = c_j^H + 2\delta^P$. In consequence, Equation (2), including δ^P can be re-written as follows:

$$U_i^P = \frac{C_i}{T_i} + \sum_{j \in (P(i) - H(i))} \frac{C_j}{T_j} + \sum_{j \in H(i)} \frac{c_j^H + 2\delta^p}{T_j} \quad (9)$$

Therefore, the decrease in utilization U_i^{PI} due to the overhead produced by the activities in the $H(i)$ set as ISTs, is given by:

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

$$U_i^{PI} = \sum_{j \in H(i)} \frac{2\delta^P - \delta^I}{T_J^H} \quad (10)$$

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:

$$U_i^{PI} < U_{iS} \quad (11)$$

Following Equation 11, if we compare the decrease in the utilization bound of the traditional interrupt model U_{iS}

(Equation 5), against the decrease introduced by the integrated interrupt model U_i^{PI} (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 ISTs 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_i^{PI}.

5. Design of the low-level interrupt management system

In this section, we describe the design of the integrated interrupt management system mentioned above. *Figure 4* depicts the overall architecture of our system. The two most important modules are the Kernel Interrupt Management (KRNINT) and the Interrupt Hardware Abstraction Layer (INTHAL). The KRNLINT contains the hardware-independent management code, while the INTHAL contains the hardware-dependent management code. This two main modules are complemented with a Low Level Synchronization & Communication module that receives a generic signal call from the KRNLINT module and translate it into the proper dispatching (block/unblock tasks/ISTs) action in the scheduler.

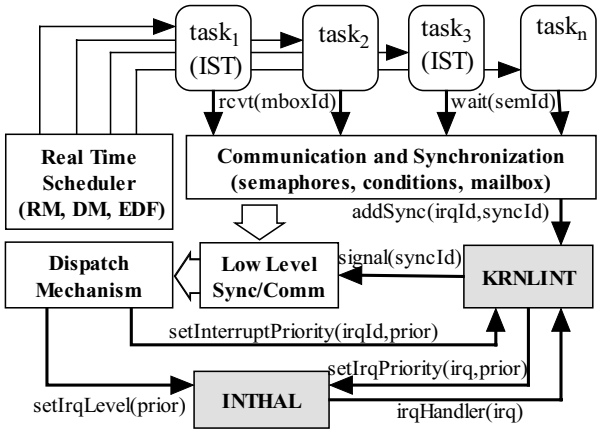

Figure 4 - Architecture of the kernel interrupt management subsystem

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 **syncId** 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 (**signal**(**syncId**) in *Figure 4*).
- To supply mechanisms for allowing the management of the interrupt priorities (**setInterruptPriority**() *Figure 4*).

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 the IRQx 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, all IRQs are in an *ignored* stated. An IRQ changes to a *captured* state if 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 (**IRQLevel**). 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 than 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 a **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 parameter.

6. Implementation over conventional PC interrupt hardware

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

An implementation in hardware can be realized using FPGA to implement a *Custom Programmable Interrupt Controller* that cooperates with the kernel scheduler to jointly schedule 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.

In order to provide an alternative to a hardware 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 *advanced PIC* (APIC) included in the most 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. This is possible by manipulating appropriately the Interrupt Service Register (ISR) and the Interrupt Mask Register (IMR) of the 8259A PIC (two in a modern PC). This 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) which is capable of supporting the integrated model. 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 δ^M be this overhead time. Then Equation (9) including δ^M can be rewritten as follows:

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

Now the decrease in utilization U^{PI*} due to the overhead of the integrate model will become:

$$U_i^{PI*} = \frac{2\delta^M}{T_i} + \sum_{j\in(P(i)-H(i))} \frac{2\delta^M}{T_j} + \sum_{j\in H(i)} \frac{2\delta^p + 2\delta^M - \delta^I}{T_j} \quad (12)$$

As may be noted, designing and implementing the system using the VCPIC, imposes a penalty in the performance of the system and even when U_i^{PI*} will not be smaller than U_{iS}, this scheme has the advantages discussed in Section 3.

In those systems where the temporal determinism is important, this implementation may be an attractive alternative to interrupts avoidance [10]. Many of this kind of applications would choose to add overhead to the system in order to achieve determinism and without sacrificing the benefits of the interrupt treatment.

7. Experimental results

Two types of experiments were conducted using our own experimental real-time kernel. The first type verifies experimentally the behavior of a single priority space. The second type measures the overhead of our implementation.

7.1. Behavior characterization

For the first type we use a task set consisting of the following tasks:

- t_1^S is an interrupt service routine (without hard real-time requirements) that attends the serial port (receiving 100 bytes per second) with a minimum interarrival 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.

Two experiments were run with this task set using an Intel Pentium 4 PC running at 3GHz with 1GB of Memory and 1MB of L2 cache.

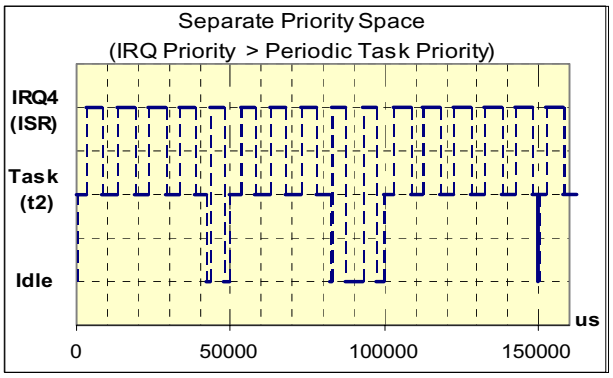

Figure 5 - Execution Trace – Separate Priorities

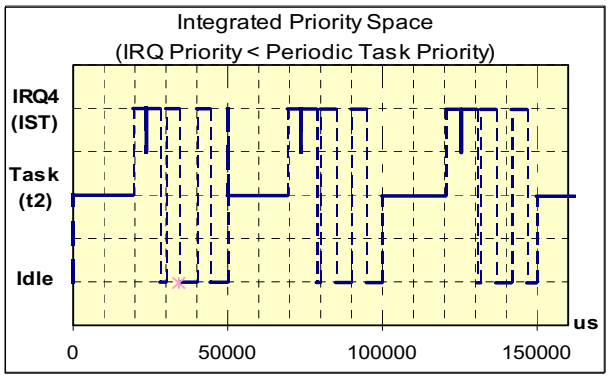

Figure 6 - Execution Trace - Integrated Priorities

In the first experiment, no integrated priority space is used and hence the t_1^S has a higher priority than t_2. *Figure 5* 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 a second experiment we used our integrated model. Here it is worth noting that (1) t_2 is given a higher priority than the IRQ (now an IST). Figure 6 depicts the execution trace of this experiment. In this case it is worth noting that t_2 is now not preempted by the IST and hence it can properly finish before its deadline in all instances shown in the figure (at 30000 µs, 80000 µs and 13000 µs), (2) At time 50000 µs, task t_2 preempts the IST, and (3) for each period of t_2 only 4 IRQs are serviced (instead of 5 IRQs of the traditional model). During the 20 ms of execution of t_2 two IRQs are issued but not attended because they have lower priority than the task. However, the hardware "remembers" one of them causing the back-to-back execution of the IST at 20000 µs, 70000 µs, and 12000 µs.

7.2. Overhead Measurement

The overhead of our implementation can be observed in the time measurements of two features: the interrupt latency and the context switch. For both of these timings we use two platforms, a Pentium 4 running at 3GHz with 1GB of memory and 1MB of L2 Cache and a Pentium MMX running at 200MHz. The experiments in the latter enable us to compare our measurements with those from other OS published in the past.

Context switching time. The difference in the context switching with and without our scheme δ^M is given by the worst-case execution time of the **setIRQLevel**() function. This function is responsable of reading and setting the 8259 IMR to selectively disable interrupts.

Table 1 presents the execution time of **setIRQLevel**() for the Pentium 4 and the Pentium MMX in both cases when the IMR needs to be changed and when it does not.

The figures in Table 1 for the Pentium MMX along with the data in [20] were used to project the potential cost of including our mechanism in other operating systems. Table 2 shows the context switch time and the projected cost on various operating systems. Cost(%) denotes the increment in percentage of the context switching time.

	P4 - 3 GHz		P-MMX 200MHz	
	w/o IMR writing	with IMR writing	w/o IMR writing	with IMR writing
Min	0.05781 µs	0.87218 µs	1.06140 µs	1.91750 µs
Avg	0.06249 µs	0.90800 µs	1.06140 µs	2.04367 µs
Max	0.09557µs	**1.04294** µs	1.14650 µs	**2.67849** µs

Table 1 - setIRQLevel execution times

OS	Min.	Ave.	Max.	Cost (%)
Hyperkernel 4.3	2.32 µs	5.64 µs	266.2 µs	1.0%
INTime 1.20	4.8 µs	5.47 µs	23.13 µs	11.6%
Windows CE.NET	7.0 µs	17.0 µs	37.0 µs	7.2%
QNX 6.1	2.1 µs	3.3 µs	11.5	23.3 %

Table 2 - Context switch times and projected cost

Interrupt Latency. The *interrupt latency* is the amount of time that elapses from the hardware raising an interrupt, to the execution of the first instruction of the interrupt handler set by the application. In system which the interrupt handling is at ISR level, this handler is a user ISR. Given that in our model ISRs are transformed into ISTs, the latency is divided into two figures:

- *Interrupt latency to the INTHAL:* Elapsed time from the interrupt raising to the execution of the first instruction of the kernel interrupt handler (**IRQHandler()**). This value reflects the overhead of the emulation mechanism to enable the integrated priority space (it depends on the emulation mode – EOI or AEOI).
- *Interrupt latency to the IST (IL):* Elapsed time from the interrupt raising to the execution of the first instruction of the corresponding IST. This time includes the context switching time due to our unified scheduling and synchronization of ISTs and tasks.

Table 3 shows the interrupt latency measurements performed using IRQ0 and 8253 channel 0 (averages performed over 1000 samples). All other measurements on

the paper were performed using the TSC (time stamp counter) of the Pentium Processor.

		Mode	Min.	Ave.	Max.
P4	INTHAL	EOI	3.352 µs	3.352 µs	3.352 µs
		AEOI	2.514 µs	2.514 µs	2.514 µs
	IST	EOI	5.029 µs	5.067 µs	5.867 µs
		AEOI	4.190 µs	4.972 µs	5.029 µs
PMMX	INTHAL	EOI	2.514 µs	2.599 µs	3.353 µs
		AEOI	1.676 µs	1.847 µs	2.514 µs
	IST	EOI	12.571 µs	12.587 µs	13.409 µs
		AEOI	11.733 µs	11.740 µs	12.571 µs

Table 3 - Interrupt latency

Interrupts switch out latency. A third important metric in our scheme is the interrupt switch out time (ISL). This time is the time it takes for an IST to switch to the next activity in the system (that may or may not be the interrupted activity – either a task or another IST). The ISL time value is in fact the kernel context switch time. Table 4 shows the switch out latency along with the interrupt latency (IL) for the Pentium 4 processor.

		IL	ISL	IL + ISL
P4-3GHz	EOI	4.088 µs	5.376 µs	9.464 µs
	AEOI	4.060 µs	4.687 µs	8.747 µs

Table 4 - Switch out Latency

7.3. Quantifying the utilization loss

In this subsection we use the figures from Subsection 7.2 and the equations from Section 4 to get an analytical figure of the utilization loss. Using equation (12) with the figures for the Pentium 4 processor we get:

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

$$U_i^{PI*} = \frac{2.086\,\mu s}{T_i} + \sum_{j\in(P(i)-H(i))} \frac{2.086\,\mu s}{T_j} + \sum_{j\in H(i)} \frac{8.747\,\mu s}{T_j}$$

Task	Period	Exec. Time
Keyboard IST	33 ms	3 µs
Serial Port IST	*Tx*	5 µs
Control task	1 s	0.5 s

Table 5 - Sample Task Set

Example: consider the task set shown in Table 5. The utilization loss (Equation 5) is given by:

$$U_{iS} = 0.00009 + 5\mu s/Tx$$

The overhead of the integrated model (Equation 12) for this example is given by:

$$U_i^{PI*} = 0.000064666 + 2.086\mu s/Tx$$

It is worth noting that that for all possible values of *Tx* $U_i^{PI*} < U_{iS}$. As a result, our model not only provides a predictable timing behavior but also yields a higher utilization bound.

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 [16] 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 [10]. Although this solution avoids the non-determinism associated to interrupts, its drawback is a low efficiency in the usage of the CPU, due to the busy wait in I/O operations. The advantage of our 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 [5] 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 [8] 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. The work in [13] introduces software and hardware solutions to prevent the overload caused by the interrupts. The integrated model proposed is able to handle this overload using any of the best-known scheduling techniques for these cases, such as the use of the sporadic servers [15].

In [7] and [3] different scheduling analysis are proposed which consider the interrupts as the activities with greatest priorities in the system. Other recent research works are: [14] that proposes a schedulability analysis which integrates static scheduling techniques and response time computation; [12] that modifies the exact response-time analysis with information about the tasks release

times and deadlines to obtain tighter response times; [2] that introduces static analysis techniques at the assembler level for interrupt-driven software. In [17] the exact schedulability equation [11] is extended to include the overhead of the interrupts in systems with static priorities and extended the model introduced in [15] 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 [15]. In [] an scheme is proposed to reserve CPU bandwidth for the execution of ISRs without preemption useful for legacy drivers.

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 [17] we extended the utilization bound equation [9] 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

In this paper we presented an integrated scheme of tasks and interrupts to provide predictable execution times to real-time systems. Its design and implementation over conventional PC hardware was discussed. An analysis to compute the disturbances on the utilization and the response times was presented. This analysis allowed us to identify situations in which a mix of ISRs and ISTs can provide the best results.

We presented a 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 synchronization mechanisms and reusable for various operating system implementations.

The component itself implements three different variants of the integrated model. All of them with different cost and efficiency trade-offs in their worst-case and average-case behavior. Even though the implementation in custom hardware yields the best worst-case and average-case performance, it cannot be used in many existing hardware platforms. As a result, an implementation on a conventional PC interrupt hardware is presented. Even though this implementation would not have a significant improvement in CPU utilization, its advantage in temporal determinism, ease of use, and reliability, are enough to favor it over existing solutions in many systems with strict real-time requirements.

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

[1] N. C. Audsley, A. Burns, M. F. Richardson y A. J. Wellings, "Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling*".Software Engineering Journal*, 8(5), 1993.

[2] D. Brylow, N. Damgaard, and J. Palsberg. "Static Checking of Interrupt-driven Software." *In Proc. of Intl. Conf. on Software Engineering*, pp. 47-56, May 2001.

[3] A. Burns, A. J. Wellings "Implementing Analyzable Hard real-time Sporadic Tasks in Ada 9X", *ACM Ada Letters*, Jan/Feb Volume XIV, Number 1, 1994.

[4] M. Carlsson, J. Engblom, A. Ermedahl, J. Lindblad, Björn Lisper. "Worst-Case Execution Time Analysis of Disable Interrupt Regions in a Commercial Real-Time Operating System", *Proc. Workshop on Real-Time Tools*, 2002.

[5] T. Facchinetti, G. Buttazzo, M. Marinoni, G. Guidi, "Non-Peemptive Interrupt Scheduling for Safe Reuse of Legacy Drivers in real-Time Systems", Proc. Of the EuroMicro RTS Conference (ECRTS'05), 2005.

[6] T. Hills, "Structured Interrupts", *Operating Systems Review* 27(1): 51-68, 1993.

[7] K. Jeffay, D. L. Stone, "Accounting for Interrupt Handling Cost in Dynamic Priority Task Systems", *Proc. of the IEEE Real-Time Systems Symposium*, pp. 212-221, Dec. 1993.

[8] S. Kleiman, J. Eykholt, "Interrupts as threads", *ACM SIGOPS Operating Systems Review*, Volume 29, Issue 2, Pages: 21 – 26, April 1995.

[9] M. H. Klein, et. al., "*A practitioner's handbook for real-time analysis*", Kluwer Academic Publishers, 1993.

[10] H. Kopetz, et-al. "Distributed Fault-Tolerant Real-Time Systems: The MARS Approach". *IEEE Micro*, 9(1), 1989.

[11] J. Lehoczky, L. Sha, Y. Ding, "The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behaviour," *Proc. of the IEEE Real-Time Systems Symposium*, pp. 166-171, December 1989.

[12] J. Mäki-Turja, G. Fohler, K. Sandström "Towards Efficient Analysis of Interrupts in Real-Time Systems". *EUROMICRO Conf. on Real-Time Systems*, May 1999.

[13] J. Regehr, U. Duongsaa, "Eliminating Interrupt Overload in Embedded Systems". Unpublished, 2004.

[14] K. Sandstrom, C. Erikssn, and G. Fohler, "Handling Interrupts with Static Scheduling in an Automotive Vehicle Control System". *Proc. Conference on Real-Time Computing Systems and Applications*, Oct. 1998.

[15] B. Sprunt. "Aperiodic Task Scheduling for Real-Time Systems." Ph.D. Thesis, CMU, August 1990.

[16] D. B. Stewart. "Twenty-Five-Most Commons Mistakes with Real-Time Software Development", *Proceedings of 1999 Embedded Systems Conference*, Sept. 1999.

[17] D. B. Stewart, G. Arora, "A Tool for Analyzing and Fine Tuning the Real-Time Properties of an Embedded System", *Trans. on Soft. Engineering*, V .29, N. 4, 2003.

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

[19] A. Zahir "An Integrated Concepts of Handling Preemptions and Interrupts for Automotive Real-Time Operating Systems", *Real-Time Magazine*, 99-3.

[20] Dedicated Sytems Encyclopaedia. http://www.dedicated-systems.com/encyc/BuyersGuide/RTOS/Evaluations/docspreview.asp.

0-7695-2516-4/06 $20.00 © 2006 IEEE