Custom Interrupt Management for Real-Time and Embedded System Kernels[1]

Luis E. Leyva-del-Foyo
Facultad de Matemática y Computación, Universidad de Oriente, Cuba.
E-mail: lleyva@csd.uo.edu.cu

Pedro Mejia-Alvarez
CINVESTAV-IPN, *Sección de Computación*
Av. I.P.N 2508, México, D.F.
e-mail: pmejia@cs.cinvestav.mx

Abstract

In this paper, we make an analysis of the traditional model of interrupt management and its incapacity to incorporate reliability and the temporal determinism demanded on real-time systems. As a result of this analysis, we propose a model that integrates interrupts and tasks handling. Also, we make a schedulability analysis to evaluate and distinguish the circumstances under which this integrated model improves the traditional model. Finally, we propose the development of a Custom Interrupt Controller compatible with our integrated model, and its implementation in a FPGA architecture.

1. Introduction

The interrupts mechanism synchronizes the occurrence of the external asynchronous events and the Interrupt Service Routines (ISRs). The synchronization mechanism offered by the operating systems 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 in 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 introduces serious difficulties to the temporal predictability and reliability of the system.

The tasks are an abstraction of the model of concurrency 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 between tasks. On the other hand, the interrupts, are an abstraction of the hardware of the computer and the responsibility of their management lies in the hardware of interrupts. 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.

As a result, two fundamental forms of asynchronous activities are found: *tasks and ISRs*. Each one with independent scheduling and synchronization policies and mechanisms, and with exclusive and restrictive sets of primitives semantically and syntactically different.

The scheduling of the ISRs is responsibility of the hardware interrupts mechanism according to its priorities of hardware, whereas the tasks are scheduled by the kernel according to their software priorities. The hardware priorities are located over the software priorities (Figure 1). In general purpose systems, the 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 interruptions, avoiding data losses while other tasks are executing. Nevertheless, in real-time systems this scheme introduces a non-determinism that makes difficult to establish temporal guarantees to events.

The synchronization between 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 synchronization between ISRs is reduced to the mutual exclusion and it is achieved only 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* scheme, each ISR is executed as a critical section with respect to itself, to lower priority ISRs, and with respect to the tasks.

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

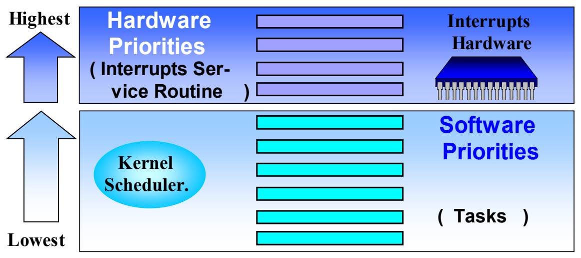

Figure 1: Priorities in the Traditional Model.

Although the ISRs are automatic critical sections with respect to the tasks, the opposite is not true. The mechanisms used to guarantee the exclusive access to critical sections between 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 to not affect the response time of the system to urgent interrupts, the CPU interrupt level must be raised only up to the level of priority of the ISR with which they could interfere.

In general purpose operating systems, this synchronization scheme is adequate because the execution of ISRs in *user mode* is not possible, and because the applications cannot modify the CPU interrupt level. This is only possible when the tasks execute code of the kernel in *supervisor mode*. Since the kernel is not preemptable, a context switch cannot occur if the actual CPU interrupt level is modified. In embedded systems, the diversity of devices used for the interaction with the environment makes necessary ISRs at the user level, so the fact that the kernel is non-preemptable severely affects the temporal determinism of the system.

In consequence, in the design of an experimental micro-kernel for embedded and real-time systems we propose an alternative strategy that integrates completely 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 proposal of a strategy completely integrated for the administration of interrupts and tasks for embedded and real-time systems.
- The evaluation of the integrated scheme from the point of view of the CPU utilization and the response time to external events. This evaluation, help system designers to demonstrate under which conditions could be considered more adequate for this type of applications.
- The proposal of a implementation of our integrated model using a customized hardware for interrupt handling.

The rest of the work is organized as follows: Section 2, exposes the disadvantages of using the traditional interrupt handling strategy for the development of reliable real-time systems. In Section 3, our integrated interrupt handling strategy is introduced with its advantages, when used on real-time systems. In Section 4, a schedulability analysis is conducted for comparing both strategies. In Section 5 related works are exposed. In Section 6, we propose the implementation of a Custom Programmable Interrupt Controller compatible with our integrated model, in a FPGA architecture. Finally, in Section 7, conclusions are presented.

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 small overhead. For this reasons it has been the method used in most of the operating systems for embedded and real-time systems. Nevertheless, its use in these environments causes serious difficulties, which we expose next.

2.1. Associated to the two priorities space.

In the traditional model, the assumption that states 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 requirements of some ISRs can be even in the same rank that those of the tasks with high activation frequencies. In this case, both spaces of priorities can interact so that they interfere among themselves. Specifically, the tasks with high priority are under the interference of hardware events necessary only for low priority tasks. On the other hand, low priority tasks associated to interrupts might not be executed due to temporal overloads, even though their associated ISRs are being executed. This affects the capacity to meet the real-time requirements of the system and produces a decrease in its utilization bound.

2.2. Associated to the interrupt latency

Perhaps the most significant argument against the traditional model can be found in its fundamental objective: to reduce to the minimum possible the interrupt latency. 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 response time of the system to the interrupts cannot be smaller than the maximum time that the interrupts are disabled anywhere in the system. Since the application can disable 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, the kernel certainly can establish a lower bound in the interrupt latency, but never will be able to guarantee its worst case.

2.3. Mutual exclusion mechanism

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

2.4. Conditional synchronization

Commonly an ISR will make at least one call to the kernel to indicate the occurrence of some event. This call can make ready a task of higher priority. If the context switching is executed, before the ISR finalizes, the rest of the ISR will not be executed until the interrupted task is executed; leaving the system in an unstable state. Consequently, if these services are invoked within an ISR, the kernel will have to postpone any context switching until the ISR finalizes. All the existing solutions to solve this problem, which guarantees the logical correction of the system, introduce an excessive priority inversion affect because of the context switching or exhibit a temporal behavior very difficult to model and hence to predict [16].

2.5. Diversity in synchronization mechanisms

The existing differences between the synchronization mechanisms, used according to the type of asynchronous activity, brings as a consequence 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 more probable the occurrence of design errors, affecting negatively the reliability of the system.

2.6. Associated to the exceptions mechanism

In languages such as ADA, where a structured exceptions mechanism with propagation by chain of calls to subprograms is used [11], an exception inside of an ISR would propagate to the exception handler of interrupted task. However, it is clear that this exception handler has no relation with the ISR that invoked the task.

A possible solution to this problem is to make the exceptions propagation mechanism to verify if the propagation goes out of the ISR, and if this is the case, to stop the propagation and abort the ISR. This produces the need to set an exception frame at the start of the ISR and to remove it at end of the ISR. In this case, the exceptions will be ignored, and the whole situation will affect negatively to the reliability of the system.

3. Integrated mechanism for tasks and interrupts handling

Given the difficulties discussed before, we propose a solution that consists of integrating both types of asynchronous activities (tasks and interrupts) through a 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 between tasks. With this model, the ISRs will now become *Interrupt Service Tasks* (ISTs), and will remain idle only until an interruption occurs. In this integrated model, the ISTs could be blocked 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 the interrupt occurs, a 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 to the kernel the low level details of the treatment of the interrupt, and eliminates the differences between the ISRs 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 an only type of asynchronous activity and a uniform synchronization and communication mechanisms between tasks and interrupts offers the following advantages:

- ISTs are executed in an environment where they can invoke without restrictions to any service of the kernel or of any library.
- Makes the development and maintenance of the system easier, because now there is only one mechanism for synchronization and communication between cooperating activities.
- Eliminates completely the need of the application to disable interrupts, allowing the kernel to guarantee the worst-case in the response time to external events (subsection 2.2).
- Eliminates the difficulty associated to the raising of exceptions inside of ISRs (subsection 2.5), because the ISRs cannot execute inside other tasks.
- Facilitates the development of re-entrant (re-usable) software components without disabling all the interrupts.

The unification of the synchronization mechanism is only a necessary but not a sufficient step. The integrated mechanism, illustrated in Figure 2, includes a space of dynamic priorities unified and flexible 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:

- The implementation of an enter/leave protocol to register the ISRs in the kernel is avoided, preventing from potential errors (subsection 2.3).
- Priority inversion associated to the independent priority space is avoided (subsection 2.1).
- 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 servers[13].

This completely integrated design eliminates the necessity to use the *busy wait* during the I/O operations, without sacrificing the temporal determinism of the system. Also, the decrease on the complexity of this integrated design favors the development of reliable systems. Overall, this scheme allows the development of robust, predictable and consequently verifiable systems. Consequently, in real-time system kernels, where the responses to events in time 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.

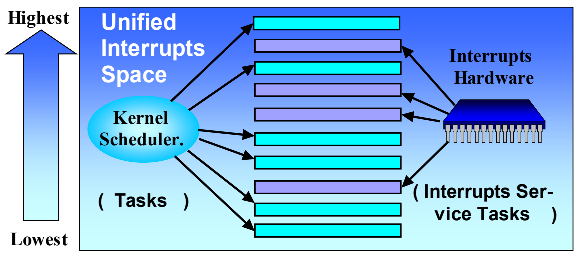

Figure 2. Priorities in the integrated model.

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 obtained in the integrated model. This analysis will allow us to evaluate the conditions under which some 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 is the following is met:

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

where U_{lub} is the *least upper utilization bound*, which is $i(2^{1/i}-1)$ for a static priority assignment (e.g., Rate Monotonic Scheduling), 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 task. This can be computed as follows:

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

The interference of the ISRs on the scheduling of task t_i can be described using the Generalized Rate-Monotonic Scheduling Theory [6]. There are two types of such interferences:

- The interference associated to interrupts, with minimum inter-arrival times inferior to those of task t_i, and linked to soft real-time tasks. We call this interference, *interference due to soft real-time tasks*. Let us denote *S(i)* to the set of ISRs t_k^S with this characteristics, each one with computation time

C_k^S and periods $T_i^S < T_i$. The utilization of an ISR t_k^S in $S(i)$ is C_k^S/T_k^S.

- The interference associated to ISRs with hard timing requirements, but with minimum inter-arrival times greater than those of task t_i. This interference is known as *rate monotonic priority inversion*. Let us denote $L(i)$ as the set of ISRs t_i^L with this characteristics and C_i^L to its computation time. Since the inter-arrival times of this interrupts T_i^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 interferences 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 second and third 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:

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

where:

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

Until now, the interference due to interrupt disabling has not being considered. Let I_L be the maximum time used for disabling interrupts anywhere in the system. Then Equation (3) can be extended as follows:

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

The decrease in the utilization bound considering the disabling of interrupts U_{iS}^* is computed by:

$$U_{iS}^* = \sum_{k \in S(i)} \frac{C_k^S}{T_k^S} + \frac{1}{T_i} \left(\sum_{k \in L(i)} C_k^L + I_L \right) \qquad (6)$$

In order to minimize U_{iS}, and U_{iS}^* the code of the ISRs (C_k^S, C_k^L) must be maintained to a minimum. 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.

Now that the delays from the ISRs have been reduced, the tasks execution times will be more predictable. However, note that since the kernel cannot guarantee the minimum inter-arrival times of the interrupts, still the execution of the ISRs still could introduce a non-bounded delay.

4.2. Increment in the response time

On 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]:

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

where R_i^n denotes the n^st^. iterative value (where $R_i^0 = C_i$), 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 (7) denotes the total interference suffered by t_i from tasks in the $P(i)$ set. This iterative process terminates successfully when $R_i = R_i^{n-1} = R_i^n$; 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 (7) the interference of the ISR sets $S(i)$ and $L(i)$, to the response time of task t_i. Adding this interference to Equation (7) we have the following:

$$R_i^n = \left(C_i + B_i + \sum_{j \in P(i)} \left\lceil \frac{R_i^{n-1}}{T_j} \right\rceil C_j \right) + \left(\sum_{k \in S(i)} \left\lceil \frac{R_i^{n-1}}{T_j} \right\rceil C_k^S + \sum_{k \in L(i)} C_k^L \right) \qquad (8)$$

The first section of Equation (8) includes three terms identical to those of Equation (7). The remaining terms (second section) denote the interference of using an independent space of priorities on the response time R_i. However, since Equation (8) 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 integrated model proposed is the overhead introduced by the context switching of the ISTs (that before were treated 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, needed to save and restore the state of the CPU, keep track of the nesting of the ISRs (and establish an exceptions frame, if an structured exceptions mechanism is used, as discussed in subsection 2.3). 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:

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

On the other hand, in the integrated model all activities in the $H(i)$ set are treated as ISTs. 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:

$$U_i^P = \frac{C_i}{T_i} + \sum_{j \in (P(i)-H(i))} \frac{C_j}{T_j} + \sum_{j \in H(i)} \frac{c_j^H + 2\delta^p}{T_j} \qquad (10)$$

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:

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

$$U_i^{PI} = \sum_{j \in H(i)} \frac{\delta}{T_J^H} \qquad (11)$$

where $\delta = 2\delta^P - \delta^I$ is the difference in computation time between two context switch activities and the computation time used by the enter/leave protocol to the ISR.

The overhead of the integrated model will be smaller than the priority inversion effect of the traditional model if the following condition is met,

$$U_i^{PI} < U_{iS} \qquad (12)$$

Therefore, following Equation 12, if we compare the decrease in the utilization bound of the traditional interrupt model U_{iS} (Equation 5) and U_{iS}^* (Equation 6), against the decrease introduced by the integrated interrupt model U_i^{PI} (Equation 9) 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 produced.

In any case, it could be possible to design an hybrid model with a configuration in which some activities are treated as ISRs and others as ISTs to satisfy the condition stated in Equation (12). For example, since the timer interrupt will have always the highest priority in the system and will never be handled by the application, it could be considered as an ISR. This cause a reduction in the $H(i)$ set therefore reducing U_i^{PI}.

5. Hardware implementation

The lack of flexibility of the available interrupt hardware, produces a difficulty to the integrated model of interrupts and tasks management proposed here. This is due to the requirements of our model to set dynamically and independently the priorities of each of the hardware interrupt request lines, and to locate these priorities within the same space of priorities of the scheduler.

In general, commercially available interrupt controllers (e.g., Intel 8259) provide support only for the traditional scheme (Figure 1). Although in some hardware architectures it is possible to develop software to emulate the functionality of the proposed model, its implementation would introduce an overhead to the system, which would cause some changes to Equation (11) affecting its feasibility.

Given the recent advances in programmable logic devices, such as FPGAs, we propose the implementation of an interrupt controller in hardware with all the necessary flexibility to adequate completely its space of priorities of hardware with the priorities provided by the scheduler (Figure 3). This way, the scheduling of ordinary tasks (activated by software) and ISTs (activated by hardware) now would become a common responsibility of the scheduler of the kernel and of the *Custom Programmable Interrupts Controller (CPIC)*. Both components cooperate to the achievement of a completely integrated priority scheme (Figure 2).

Under this configuration, the CPU (executing under the control of the scheduler) may write in a set of command registers of the CPIC, in order to provide dynamically the priorities of each one of the interrupt request lines and of the actual interrupt level. The priority of the task in execution will always be the same as that of the actual interrupt level. The proposed CPIC then will have the goal of directing an interrupt request to the CPU, only if its associated priority is higher than the actual interrupt level imposed by the scheduler.

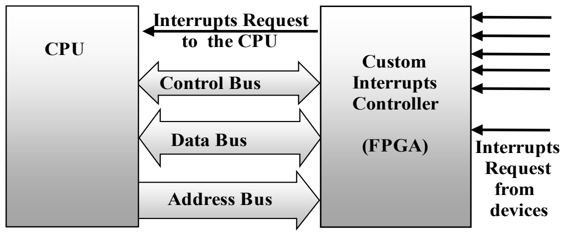
CPU
Interrupts Request to the CPU
Control Bus
Data Bus
Address Bus
Custom Interrupts Controller
(FPGA)
Interrupts Request from devices

Figure 3. Interrupt controller implemented in FPGA

From Figure 3 note that, the control bus between the CPU and the FPGA support the interrupt request/acknowledge protocol of the CPU, and the data and address bus allows the programming of the CPIC.

We propose a software/hardware codesign strategy where the CPIC design is conducted using a high level hardware description language such as VHDL or Verilog. This strategy lead us to the possibility of:

- To parameterize some design aspect (i.e., the number of priority bits) in such a way that this parameters reflect any scheduling scheme used by the real-time system kernel.
- To configure some logic aspects (i.e., the interrupt request/acknowledge protocol) so that it matches with the CPU used in the embedded system.

In this co-design, only at compiling and synthesis time the adequate values would be set for such parameters and configuration options. This way, we could get the gate-level logic and the appropriate FPGA connections and routes for a programmable interrupt controller completely customized for the CPU and scheduling policy specifically used.

6. Related Work

Several research works propose alternatives to avoid the difficulties of the traditional interrupt model for real-time applications. In [14] the indiscriminate use of ISRs is considered as one of the most common errors in real-time programming. Several real-time operating systems have adopted radical solutions where all external interrupts are disabled, except for those that come from the timer and propose to treat all peripherals by polling [7]. Although this solution completely avoids the non-determinism associated to interrupts, has as a fundamental disadvantage a low efficiency in the usage of the CPU, due to the busy wait in I/O operations. The advantage of our integrated scheme with respect to these proposals, is that our scheme achieves temporal determinism without significantly affecting the usage of the CPU.

Several strategies have been proposed to obtain some degree of integration between the different types of asynchronous activities. In [3] an "structured" interrupts treatment scheme its proposed at the task level, but introducing an interface independent of the synchronization mechanism, and do not consider interrupts with dynamic priorities. In [5] a method is proposed where interrupts are treated as threads. Its proposal does not have as a fundamental goal to achieve temporal determinism, but the increase on the scalability of the system in multiprocessor architectures oriented to network servers operating systems. As a consequence, the interrupt threads use a specific rank of priority levels. In [17] 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 priorities of the ISRs are static and the synchronization mechanism is not unified. The work in [10] introduces software and hardware solutions to prevent the overload caused by the interrupts. The integrated model proposed in this paper 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 [13].

In [5] 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: [12] that proposes an schedulability analysis which integrates static scheduling techniques and response time computation; [10] 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 [15] the exact schedulability equation [8] is extended to include the overhead of the interrupts in systems with static priorities, and extended the model introduced in [13] to include the overhead of interrupt handling. The resulting equation evaluates the trade-off of doing the interrupt handling inside of an ISR or postponing most of the treatment to an sporadic server [13].

None of the previous research works provides an analysis that includes the interference on the utilization and the response-time caused by the use of two spaces of independent priorities. Unlike [15] we extended the utilization bound equation [6] and the response-time

[1], and evaluated the possibility of eliminating completely the treatment of ISRs by integrating both types of asynchronous activities.

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 on the complexity of the design and implementation, which decreases the reliability of resulting software. In addition, the use of two spaces of independent priorities severely affects the determinism and the degree of feasibility in the scheduling of tasks with real-time requirements.

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

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

The integrated model proposed in this paper has been implemented and incorporated as part of an experimental micro-kernel for embedded and real-time applications and it is actually being implemented using FPGAs.

8. 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 Proceedings of ICSE'01, 23rd International Conference 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] T. Hills, "Structured Interrupts" Operating Systems Review 27(1): 51-68, 1993

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

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

[7] M. H. Klein, T. Ralya, B. Pollak, R. Obenza, M. González Harbour, "A practitioner's handbook for real-time analysis", Kluwer Academic Publishers, 1993

[8] H. Kopez, A. Damm, C. Koza, M. Mulazzani, W. Schwabla, C. Senft y R. Zainlinger. "Distributed Fault-Tolerant Real-Time Systems: The MARS Approach". IEEE Micro, 9(1), Feb. 1989.

[9] J. Lehoczky, L. Sha, Y. Ding, "The rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior," Proceedings of 10th IEEE Real-Time Systems Symposium, pp. 166-171, December 1989.

[10] J. Mäki-Turja , G. Fohler , K. Sandström "Towards Efficient Analysis of Interrupts in Real-Time Systems". 11th EUROMICRO Conference on Real-Time Systems, York, England. May 1999.

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

[12] J. Ruiz, J. de la Puente, J. Zamorano, R. Fernández-Marina, "Exception Support for the Ravenscar Profile", ACM SIGAda Ada Letters, Volume XXI, Issue 3, Pages: 76-79, September 2001.

[13] K. Sandstrom, C. Erikssn, and G. Fohler, "Handling Interrupts with Static Scheduling in an Automotive Vehicle Control System". Proceedings of the Conference on Real-Time Computing Systems and Applications, Hiroshima, Japan, October 1998.

[14] B. Sprunt. "Aperiodic Task Scheduling for Real-Time Systems." Ph.D. Thesis, Carnegie-Mellon University, August 1990.

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

[16] D. B. Stewart, G. Arora, "A tool for Analyzing and Fine Tuning the Real-Time Properties of an Embedded System", IEEE Transactions on Software Engineering, Vol. 29, Nr. 4, April 2003.

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

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