FACULDADE DE CIÊNCIAS UNIVERSIDADE DE LISBOA

Faculdade de Ciências da Universidade de Lisboa

Instituto Superior Técnico

INSTITUTO SUPERIOR TÉCNICO

DARIO: Distributed Agency for Reliable Input/Output

Project FCT POSC/EIA/56041/2004

Securing the Timeliness of Input/Output Event Handling in Real-Time Kernels

DARIO Technical Report RT-05-01

C. Almeida, M. Coutinho, J. Rufino

June 2005

Faculdade de Ciências da Universidade de Lisboa

Instituto Superior Técnico

Securing the Timeliness of Input/Output Event Handling in Real-Time Kernels

To be submitted for publication: please do not distribute

Technical Report: DARIO RT-05-01
Authors: C. Almeida, M. Coutinho, J. Rufino
Date: June 2005

This work was partially supported by the FCT through Projects POSC/EIA/56041/2004 (DARIO) and the Large-Scale Informatic Systems Laboratory (LASIGE).

Securing the Timeliness of Input/Output Event Handling in Real-Time Kernels*

Carlos Almeida
IST-UTL†
cra@comp.ist.utl.pt

Manuel Coutinho
IST-UTL
mabeco@comp.ist.utl.pt

José Rufino
FCUL‡
ruf@di.fc.ul.pt

Abstract

Embedded control systems that need to interact with the real-world, performing input/output operations through sets of sensors and actuators, may be subjected to temporal uncertainties if not built with care. This happens because external events, which are usually asynchronous, are not completely controlled, and so, there is the possibility of having overload scenarios when a resource-adequacy policy is not used due to not being cost-effective. As most of these systems also have real-time requirements, it is of utmost importance to provide the means to ensure dependability in what concerns timeliness.

In this paper we address the problem of securing timeliness properties in the presence of input/output event handling. This work, performed within the scope of the DARIO (Distributed Agency for Reliable Input/Output) Project, is done as an extension to off-the-shelf real-time kernels, by incorporating temporal protection mechanisms. A case study using the real-time kernel RTEMS (Real-Time Executive for Multiprocessor Systems) is presented.

1 Introduction

Most embedded control systems need to interact with the real-world, by performing input/output operations through a set of sensors and actuators. This interaction with the environment may cause some uncertainty in the time domain because external events are not completely controlled. As most of these systems also have real-time requirements, this may pose a problem: how to ensure timeliness guarantees when the event rate associated with the interaction to the external world is not bounded (or at least can be higher than the assumed value at design time)?

In order to solve this problem we need to address the main methods to deal with input/output event handling. Namely, interrupt control, switching between interrupt and polling modes and tuning of polling cycle times, bounding of event handling rate (e.g. through event batching, event compression and pre-processing), are examples of actions to take to try and reach the desired goals.

As a general rule the system may include computational domains with soft or with no real-time requirements at all and domains where those requirements are extremely stringent. The overall objective is to prevent the contamination of real-time system domains with respect to timeliness by disturbances occurring on both the physical and computational environments.

This is achieved by building a *timing firewall* able to be integrated in the computational environments provided by off-the-shelf real-time systems and kernels.

In particular, we apply these concepts within the scope of the DARIO (Distributed Agency for Reliable Input/Output) Project, using the RTEMS (Real-Time Executive for Multiprocessor Systems) kernel.

The paper is organized as follows: in the next section we give an overview of DARIOS's architecture and its computational environment; in Section 3 we present some related work; in Section 4 input/output event handling is addressed, in Section 5 timeliness protection mechanisms are explained; in Section 6 a case study related to the incorporation of some of those timeliness protection mechanisms into the RTEMS real-time kernel is presented. The paper ends with the conclusions.

2 The DARIO Project

Embedded and distributed computer systems play nowadays a vital role in control applications as diverse as industrial processes, automotive, railways, avionics and aerospace, medical, etc. In this context, the relevance of standard communication networks such as fieldbus cannot be ignored. In the scope of the DARIO project, we plan to use the Controller Area Network (CAN) fieldbus as a communication infrastructure to build a distributed agency for reliable input/output operations.

To meet the required high-levels of reliability, the native CAN communication infrastructure needs to be complemented with a set of hardware/software additional mechanisms. The combination of a standard CAN layer with such dependability enhancement mechanisms, has been dubbed CAN Enhanced Layer (CANELy).

The services provided by CANELy (group communi-

*This work was partially supported by FCT, through Project POSC/EIA/56041/2004 (DARIO).

†Instituto Superior Técnico - Universidade Técnica de Lisboa, Avenida Rovisco Pais, 1049-001 Lisboa, Portugal. Tel: +351-21-8418397 - Fax: +351-21-8417499.

‡Faculdade de Ciências da Universidade de Lisboa, Campo Grande - Bloco C8, 1749-016 Lisboa, Portugal. Tel: +351-21-7500254 - Fax: +351-21-7500084.

cation, clock synchronization, node failure detection and membership) are of fundamental importance to the availability of mechanisms handling object replication, competition and cooperation management, useful constructs for fault-tolerant applications.

The DARIO project aims to develop a proof of concept concerning the use of CAN, the Controller Area Network fieldbus, in real-world highly fault-tolerant real-time applications. The main objectives are:

- extend/complement a state-of-the-art definition of a fault-tolerant CAN-based communication layer with enhanced and highly efficient mechanisms with respect to dependability and real-time attributes.
- provide an implementation of those mechanisms, dubbed CANELy (CAN Enhanced Layer), in a CAN-based infrastructure.
- definition and design of a novel CAN-oriented middleware layer satisfying the stringent requirements of (object-oriented) application levels.
- definition of an innovative uniform computational model for modular and reliable input/output operations, to handle sensors/actuators.
- application of such a model to a selected set of technologies relevant to dependable real-time industrial control: position control; robotics; power electronics; electro-pneumatics.

2.1 The DARIO Architecture

The DARIO architecture (Figure 1) also involves a need for a modular and generic approach to: the integration of physical input/output components; input/output event translation to/from a computational entity; uniform treatment of input/output events and message events information flows; provision of fault-tolerance and real-time guarantees. A modular approach should also be followed in the mapping and/or adaptation of the generic architecture to specific technologies, thus limiting the overall impact of technological aspects on system design. (In the context of DARIO project we will work together with some team members of the CORTEX project that also developed an architecture to handle generic events [21].)

On the other hand, the programmers of distributed control applications require constructs hiding as much as possible the implementation details of the underlying infrastructure (object-orientation).

Other aspect of application design concerns the possible partition and deployment of application components at several levels of the architecture (e.g. smart sensors configuration, resident robotics applications).

A solution to any of these problems in embedded environments, sometimes with scarce resources, do represent a set of real challenges that are being addressed in the context of the DARIO project.

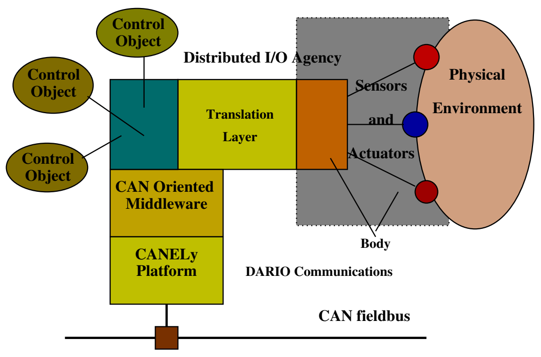

Figure 1. Architecture of DARIO

2.2 The DARIO Computational Environment

Besides the more abstract aspects of the generic architecture, there must be some computational environment to support the active entities and the management of resources. This means operating system support. Even if at application level we deal with more high level abstractions, and in the generic architecture there are also middleware components that offer a more object-oriented view of the system, we need nevertheless operating system support to build all this infrastructure. Concurrent applications will need support for multitasking, inter-task synchronization and communication. System resources such as memory and time will need to be managed in an efficient way, and input/output operations must be correctly handled to ensure the desired system dependability.

The overall system is a distributed system with several nodes. Each node is composed of several different layers, and may have specific characteristics. This means that it is possible to have several different types of operating systems. For example, if we only need to address more high level abstractions, a generic operating system able to support an object-oriented platform is adequate. However, if there are also real-time requirements, then we may need to use a system capable of supporting both real-time and more generic operations (e.g. RT-Linux [2]).

For some more specific components that need to deal with low-level input/output operations, providing real-time guarantees, then a real-time kernel (e.g. RTEMS [13]) may be more suitable.

In any case, these off-the-shelf real-time kernels may need to be extended with some specific timeliness control mechanisms when interacting with the external world, to avoid possible overload scenarios that could compromise the dependability and real-time properties of other parts of the global system.

If already available (for other reasons), or if the associated cost is low when compared to the benefit, specific hardware (e.g. FPGA – Field Programmable Gate-Array) may also be used, incorporating programmable functions that work as filters or validation mechanisms.

3 Related Work

The problem of ensuring timeliness properties is essential in order to support real-time applications. This must be addressed at several different levels, and depends on application characteristics and its interaction with the environment, and the overall system architecture. There has been some work related to these issues, both at architecture level and operating system level.

At the operating system level, there are mainly works related to scheduling. Besides more generic real-time scheduling algorithms found in the literature (e.g. RMS – Rate Monotonic Scheduling, EDF – Earliest Deadline First) [11, 10], that are mainly concerned with meeting the deadlines of processes without having much interference from input/output events, and assuming a worst-case scenario that is known *a priori*, there are some works that try to reserve specific percentages of CPU to critical activities thus providing a *temporal protection* to those critical activities in more dynamic environments where overload scenarios are not ruled-out [12, 6]. However, when there is also event overload, event handling must be addressed as well.

I/O throttling[1] is another technique that can be used to control the rate of input/output operations. In [18] this technique is used together with other techniques to control resource usage. They use a sliding window of recent events to compute the average rate for a target resource. The assigned limit is enforced by the simple expedient of putting application processes to sleep when they issue requests that would bring their resource utilization out of the allowable profile. However, dealing with external events implies a different approach to enforce temporal protection. Events must be filtered at the interface between the system and the environment.

At architecture level one fundamental distinction about approaches to use, concerns the way the system evolves. There are mainly two alternatives: the event-triggered approach and the time-triggered approach [22, 7]. As their names suggest, in the first case the system reacts to events, whereas in the second case it is the elapsing of time that regulates system behavior. Although the time-triggered approach makes it easier to reason about system properties in the time domain [9], it is not always realistic to assume this type of interaction with the real-world. In such cases, there must be some specific components in charge of isolating the system from the environment, and provide an interface that transforms events into state that will be accessed by the system at pre-defined time instants [8]. These components act as *temporal firewall* interfaces that make it possible to still reason in a time-triggered way when interconnecting several different components, thus facilitating the composability at design time. An event-triggered approach is, however, more flexible and, with the right control mechanisms, can be used most of the time.

Another work that addresses the problem of interconnecting components with different real-time properties is [16]. They propose the concept of *composite objects* as a way to integrate real-time and non-real-time computing into a single object-based framework. A composite object allows inter-operability between the real-time part and the non-real-time part, but acts as a timing firewall to ensure non-interference in the time domain. This is achieved by the management of priorities and having non-blocking communication.

From the point-of-view of operating system support to deal with real-time and non-real-time co-existence, an example is RT-Linux [2], where a real-time kernel is in charge of real-time activities, and runs Linux as a low-priority task. Linux interrupts are intercepted by the real-time kernel to avoid losing control.

The work presented in [17], is a recent work, developed in parallel with our work, that has a similar goal: to prevent interrupt overload. However, they are mainly concerned in bounding the interrupts, whereas we are also interested in a more complete characterization of event occurrence before deciding about the existence of an event overload. In [5] there is an effort to integrate task scheduling and interrupt scheduling. Although an important issue, it is not always possible to have this type of approach, due to specific system limitations.

4 Input/Output Event Handling in Real-Time Kernels

Besides all the aspects presented above, a specific problem concerns the way the operating system provides support to deal with the interaction with the environment. Namely, the handling of input / output. If not correctly handled, it may compromise the timeliness properties of the global system. In a generic way, it is important to deal with input / output in a modular fashion, so as to be able to integrate several different devices in a common way and provide reliable operation. But, when there are real-time requirements, modularity is not enough. One must ensure that the interaction with the environment, and the occurrence of events, do not cause uncontrolled interferences in the system, that may jeopardize application goals.

Dealing with the environment is always a difficult task because it implies interaction with components that are not completely controlled by the system. The occurrence of asynchronous events, if not bounded, may imply an overload that if not handled with care will make the meeting of deadlines an almost impossible mission.

The interface to external devices, to perform input / output operations, implies at least two different aspects: event detection, and event processing. The usual way to detect events uses polling or interrupts. Event processing

[1]I/O throttling: Short for input/output throttling, a technique used to more efficiently handle memory processing. During low-memory conditions, a system will slow down the processing of I/O memory requests, typically processing one sequence at a time in the order the request was received. I/O throttling slows down a system but typically will prevent the system from crashing.[1]

can be done at once, or split in several different phases.

In a multitasking kernel, a polling approach is achieved through the use of a task that periodically checks an input / output port to detect when an event has occurred and needs processing. Depending on the priority of the task and the periodicity of the check operation, it may be possible, or not, to detect in a timely fashion all events. If the task has a low priority, there is the possibility of not being able to perform the event handling at the right time or even miss some events. This can happen when there are other higher priority tasks ready to run, that will be chosen by the scheduler. However, if one chooses to have a high priority task to deal with event checking, then it is possible to waste a lot of CPU time just with the checking operation, without doing any useful work, and preventing other tasks from running. In order to minimize this effect, one possibility consists in increasing the time between event checks (increase task periodicity), and having the task sleep in the meantime. This approach has the drawback of not offering the guarantee of detecting all events. If the sleep time is too large, there is the possibility of missing some events.

Another approach to deal with event detection is to have them associated with interrupts. When an event occurs, an interrupt is generated that, as soon as possible, will be handled by a specific routine (interrupt handler). While polling the device provides a more controlled way to deal with the events, the use of interrupts provides a more flexible architecture. In scenarios where the occurrence of events is not uniformly distributed in the time domain, or where there are dynamic aspects in the overall system, an interrupt approach is usually more effective. There are even some situations where the use of interrupts is advisable in order to support a given functionality. For example, in embedded systems supported on batteries, the need to increase the autonomy (from the point-of-view of energy consumption) may imply the use of CPU low-power states (*sleep modes*). Recovering from these states may require the use of interrupts.

Although flexible, interrupts may introduce significant overhead and cause uncontrolled situations if the event rate is very high. In most multitasking kernels, interrupts have higher priority than any task running on the processor. If an interrupt overload situation takes place, the application timeliness may be in jeopardy. In order to have the flexibility of interrupts, but at the same time being able to preserve application timeliness properties, we need timeliness protection mechanisms in event handling to filter potential interrupt overloading. These overload situations may occur due to unanticipated load, or due to faulty scenarios.

In order to effectively support dependable real-time applications that need to deal with input / output operations, and secure their timeliness properties, it is necessary to incorporate timeliness protection mechanisms in the way the real-time kernels handle input / output.

5 Timeliness Protection Mechanisms

Asynchronous event handling, based on interrupts, introduces temporal uncertainty that may interfere with application timeliness. The rate at which interrupts are generated, if not bounded, may affect task execution time and, in a worst-case scenario, deadlines may be missed.

The main problem that needs to be solved is to avoid an unbounded rate of asynchronous events causing an overhead that was not anticipated at design time, and for which there are not enough resources to deal with. There must be some kind of "filter" if one wants to make sure that existing real-time tasks will get the assumed CPU execution time, and thus will not miss their deadlines.

When the event rate increases, interrupt processing or sample rate (depending on the used method) will also increase if we do not want to loose any event and do not change the way the events are handled. But, as we said above, this may be a problem if an overload situation is reached. And, in some cases, these extra events are not really important events. They may not add significant value to the application (depending on application timing granularity), or they can even be generated due to error conditions caused by accident (e.g. *stuck key*), or intentionally provoked by an intruder. If we want to have a dependable system, we must incorporate some fault-tolerant mechanisms to deal with such situations and avoid timing interferences to the other parts of the system.

If an event rate increase starts to menace the correct behavior of the overall system (application), then some form of flow control is needed. If all events are important, and the application has hard real-time requirements, then this is a design problem, and a resource adequacy policy is needed. Possible, the only thing that can be done is to preform some optimizations at event processing level, if there is still margin for that.

This optimization of event processing, can be related to different modes of operation, including graceful degradation of quality-of-service. For example, instead of processing immediately every event, some form of pre-processing can be done, compressing a set of events, or handling them in a batch mode (Figure 2 and Figure 3). This will reduce the overhead, but may not be able to ensure the desired level of temporal protection.

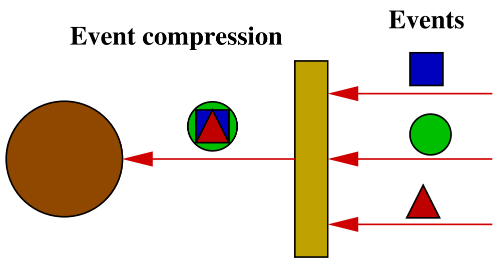

Figure 2. Using event pre-processing (compression) to reduce system load

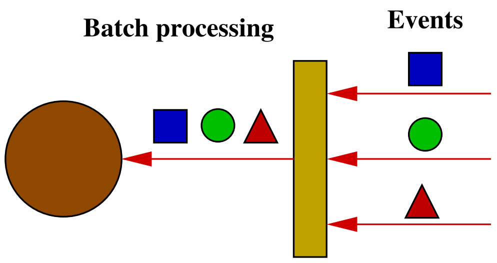

Figure 3. Processing several events together (batching) to reduce overhead

In a more drastic overload scenario, in order to ensure timeliness protection, we must limit the number of events that enter the system. If the application can tolerate to miss some events, because, for example, they are redundant or meaningless due to application time granularity, then it is possible to implement a "filter" reducing the number of events that will be handled and processed (Figure 4). This can be done by reducing the sample rate, through the modification of polling cycle time, or by disabling event interrupts during some time intervals. This is somehow similar to a "debounce" operation.

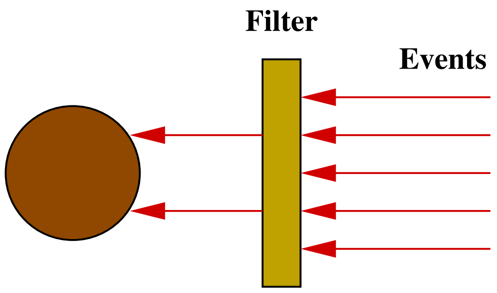

Figure 4. Applying a filter to reduce the number of events

Instead of having the system dealing directly with all environment interaction, another approach consists in having specialized subsystems in charge of those operations. Specific components, that may include both hardware and software, will preform a first level processing of events, presenting their results to the main system in a more controlled way. This is usually the approach used in a time-triggered architecture but may not be generic enough in many scenarios. However, when the available hardware architecture includes co-processors or symmetric multi-processors (SMP), and the used operating system has support for it, it may be an option to partition the system in specialized components.

Besides event detection, event processing plays an important role, mainly if the processing time required represents a significant load to the system. The ability to decompose that event processing in several different phases, may help in the integration with the scheduling of other processing activities, and minimize the overall impact on existing deadlines. For example, an interrupt handler will save vital information related to the event in a very fast way, and then an interrupt task will be in charge of the processing part. This way, being a task, it can be scheduled together with the other tasks, with a suitable priority that takes into account the overall system state. It is even possible to reserve a given amount of "CPU bandwidth" for specific activities, in order for them to not suffer interferences from the new activities.

In some cases, for example when dealing with legacy code, running on off-the-shelf operating systems, it would be desirable to be able to add some of those timeliness protection mechanisms with minimal impact. A modular approach providing the integration of interceptors to handle the first phase of event detection/processing would be extremely valuable.

The protection mechanism must intercept interrupts, determine the interrupt rate to evaluate if there is an overload situation, and, if necessary, disable interrupts and switch temporarily to a polling mode.

5.1 Event Characterization

In the implementation of those timeliness protection mechanisms, we need to have the means to characterize event occurrence, thus allowing to determine its rate and decide if there is, or not, an overload situation (Figure 5). Just measuring the time interval between any two interrupts may not be enough. Although it is useful in specific situations to detect a fault scenario if a pre-defined maximum inter-arrival time is exceeded, in a more generic situation it may be possible (from the point-of-view of the application) to have several interrupts in a given time interval, being that load tolerated. We would like to have more information about the event occurrence before making a decision.

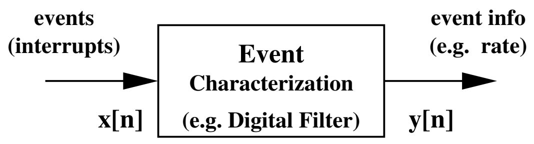

Figure 5. Generic module for event characterization

Being interrupts discrete events, in the context of our work, we decided to exploit discrete signal processing knowledge [20], with adaptations, to deal with this problem. The idea is to have a digital filter, that is applied to events and provides information about them. For now, we are mainly interested in obtaining the event rate, but in a more generic situation, depending on specific filter parameterization, it will be possible to obtain different type of information about the events. Another advantage of this approach is the possibility, if desired for performance reasons and cost-effective for a given application, to implement these filters using dedicated hardware (to be addressed in future work).

Discrete Event Processing

In order to apply the discrete event processing, we need to discretize time [15, 20]. Although we may consider the computing system internal clock as discrete (resolution $\sim 1\mu s$), in a macroscopic scale, from the point-of-view of event generation intervals, it can be considered as continuous. Discretization will be done through *sample/hold*. Assuming a given sampling period (T_{sample}), system state is periodically checked (*sample*) and kept (*hold*) until the next sampling. Continuous time t is thus transformed to a discrete time n through $t = nT_{sample}$. To represent that an event has occurred at discrete time n, a scalar discrete function $x[n]$ ($x : Z \rightarrow 0, 1$) is used:

$$x[n] = \begin{cases} 1, & \text{if an event occurs in } n \\ 0, & \text{otherwise} \end{cases} \tag{1}$$

Consider another discrete function $z[n]$ which represents the number of detected events until n

$$z[n] = \sum_{i=1}^{n} x[n] \tag{2}$$

assuming that no events occurred for $n < 1$ (that is, $x[n] = 0, \forall n < 1$). This is, in the discrete signal literature[20], described as *running sum* or *"discrete integral"*. It simply adds the number of events that occurred in the past plus if an event is occurring now, at discrete time n. In a recursive form, Equation 2 can be represented as

$$z[n] = z[n-1] + x[n] \tag{3}$$

The event rate can therefore be described as another discrete function $y[n]$ given by

$$y[n] = \frac{z[n]}{n} \tag{4}$$

which gives the average of the number of events occurred until the time n. For a recursive form of $y[n]$, the following equation can be derived from equations 3 and 4

$$y[n] = \frac{z[n-1] + x[n]}{n} = \left(1 - \frac{1}{n}\right) y[n-1] + \frac{1}{n} x[n] \tag{5}$$

Although the average number of events may be important from the point-of-view of event characterization, for our current purpose (overload detection), we would like $y[n]$ to give, not the average number of events since the origin of time, but information about a more instantaneous event rate generation. Otherwise, if the system has been running for a long time (large n), it will be very slow to react to an event overload. This happens because all events are assumed to have the same importance regardless of having occurred recently or a long time ago.

The most obvious solution to this problem consists in considering only the last events inside a temporal window (D). In the literature this is known as a FIR (Finite Impulse Response) filter, where $z[n]$ would be given by

$$z[n] = \sum_{i=n-D+1}^{n} x[n] \tag{6}$$

The parameter D is known as the FIR dimension and represents the number of samples of $x[n]$ to be considered in the past. The event rate is now determined by the average of the last D samples (note that this filter is equal to the first one presented if $D = n$).

$$y[n] = \frac{z[n]}{D} = \frac{1}{D} \sum_{i=n-D+1}^{n} x[n] \tag{7}$$

Another solution to have a more responsive system consists in weighting events based on the time of its occurrence. For that, we can use an IIR (Infinite Impulse Response) filter (recursive filter) and weight the events based on the event occurrence time. The event rate is given by

$$y[n] = \alpha y[n-1] + (1-\alpha)x[n] \tag{8}$$

where the parameter α belongs to the interval $]0; 1[$ in order to have a stable system and $y[n]$ positive. Note that this filter can also be converted to the first one if we make $\alpha = (1 - 1/n)$ (see Equation 5).

Using our methodology it is possible to implement specific filters to handle special cases such as a *maximum burst size*, for example. The filter of Equation 9 gives the number of events in a given interval of duration B.

$$y[n] = \sum_{i=n-B+1}^{n} x[n] \tag{9}$$

5.2 Overload Detection and Handling

Using the event characterization module, it is now possible to detect if/when there is an overload situation. The sampling of events (system state) to feed that component could be done using a specialized periodic task. However, that method is inefficient, implying to run the task even when there are no events and, if the polling period is too high, may also miss events. Moreover, if an overload situation is in progress, the polling task may be overrun by interrupts and so the goal for which it was designed is not achieved.

As we are mainly interested in overload situations caused by interrupts, detecting an overload situation inside the Interrupt Service Routine (ISR) is a preferred method. All events (interrupts) are accounted for, and there is only associated processing when a new event is generated. It also allows for a faster response to an overload situation by being able to immediately disable the interrupt. The downside of this method is the time overhead required inside an ISR to calculate the event rate, and possible restrictions about the use of floating-point operations. However, it is possible to make some optimizations, as will be explained later. The fact that the system is not sampled with a periodic rate (assumed in the discrete event processing literature [15, 20]), is easily overcome by registering the time of event occurrence and transform it to discrete time as explained before ($t = nT_{sample}$). When a new event occurs we know the elapsed time since the last registered event and thus are able to *"reconstruct"* the *sampling data* because we know that there were no events in the meantime.

When an overload is detected and the corresponding interrupt is disabled, two methods can be used to allow a graceful degradation of the services while maintaining a bounded interference with the rest of the system. By allowing interrupts in certain time intervals, the system can limit the interference and admit some interrupts to be processed. Adapting the time interval in which interrupts are enabled, the system can cope with transient situations and decide whether the overload has passed. A timer can be used to inform when to re-enable the interrupt.

A second solution consists in having a special periodic task that polls the device, checking for events and determining if an overload is still present. If not, the system returns to the normal state by enabling again the interrupt. The period and priority of this task must be such that it does not interfere with the deadlines of the rest of the system. The component to determine the current event rate to decide if there is still an overload situation, or not, can be the same as before. However, to provide a more stable environment, a hysteresis cycle can be used with the definition of two different thresholds M and m (see Figure 6), associated with mode switching. The value of the low threshold (m) is even much lower than M because when in polling mode there are events that might be missed (due to a larger sampling period). When controlling the overload of more than one interrupt source, the same task can be used acting as a cyclic executive calling the processing routines associated with each event.

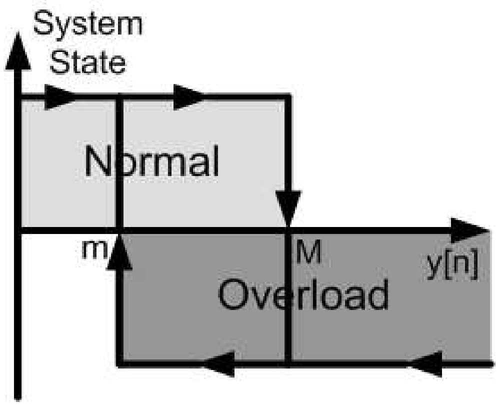

Figure 6. Using hysteresis for stability in establishing thresholds for interrupt and polling methods

5.3 Exploiting Kernel Native Facilities

The implementation of the timeliness control mechanisms described above, may benefit from the availability of specific kernel facilities. For example, the existence of system calls to enable/disable interrupts specifying different interrupt levels can give a high degree of flexibility in interrupt management.

Being able to define/install interrupt handlers, get information about existing ones, substitute and possible cascading new and old ones, allows to intercept events and perform some of the filtering and pre-processing explained above with minimal impact on the existent infrastructure.

In what concerns interrupt handling and event processing, the possibility of having different stages with different behavior with respect to interrupt acceptance and scheduling locking, may make a big difference in system tuning. Interrupt handlers that may, or may not, re-enable interrupts; special handlers that run with interrupts enabled, but with the scheduler locked; and interrupt tasks or threads that are scheduled as any other thread; are examples of mechanisms that increase the flexibility to deal with event handling without compromise specific timeliness requirements.

Priority inheritance mechanisms [19] may play also an important role in dealing with event processing and its relation with real-time requirements, by allowing the propagation of urgency to other components in such a way that unbounded durations are avoided.

In more specific scenarios, where a multiprocessor architecture is available (e.g. SMP), the ability to partition the system in different components with different properties (similar to the creation of virtual machines), can be used to reach a more controlled environment similar to the one used in a time-triggered architecture.

5.4 Implementation in a Real-Time Kernel

The mechanisms described above can be implemented in a real-time kernel with minimal impact on the existing infrastructure. The original Interrupt Service Routine (ISR) associated with the event is replaced by another one that, using the previous described components, determines the interrupt rate, decides if there is an overload, disabling the interrupt and switching to a polling mode if necessary, and calls the original ISR to process the event (see Figure 7). The pseudo-code of the new ISR, implementing an IIR filter to determine the interrupt rate, is described in Figure 8.

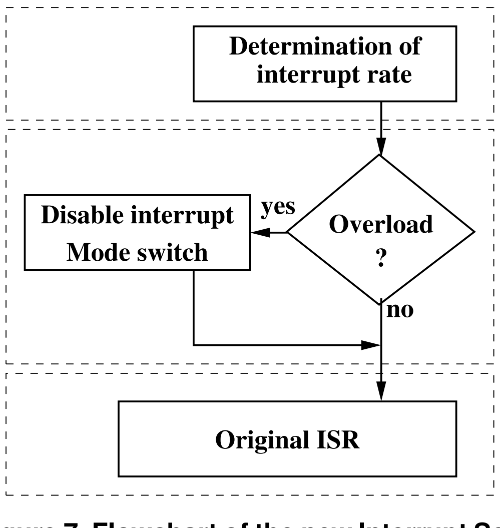

Figure 7. Flowchart of the new Interrupt Service Routine

```
newISR(){
                    // event rate determination
    t = getTime();
    n = sampleHold(t);
    y[n+1] = $\alpha^{n-last\_n}$ y[n] + (1-$\alpha$);
    last_n = n;
                    // overload detection and mode switching
    if( y[n+1] > M ){
        state = overload;
        disableInterrupt();
        switchPollMode();
    }
                    // event processing
    originalISR();
}
```

Figure 8. Pseudo-code of Interrupt Service Routine using an IIR filter

As explained before, due to the fact that events are registered in an asynchronous way (when an interrupt occurs) instead of periodically (as assumed in the discrete signal processing literature), some adaptations must be done. Continuous time must be converted to discrete time taking into account the sampling period. As system state (event occurrence) is only registered when events do occur, we must also account the event *non-occurrence* during the elapsed time. As $x[n] = 0$ when an event does not occur in n, $y[n]$ is given by only the first term of Equation 8. This way, during an interval N without events, $y[n]$ can be obtained by Equation 10.

$$y[n] = \alpha y[n-1] = \alpha^2 y[n-2] = ... = \alpha^N y[n-N] \quad (10)$$

As the term α^{n-last_n} is too computationally expensive to be determined at runtime in the ISR, a table is used (built during system initialization) to perform this calculation with only one memory access. The exponent, which is an integer, is used for indexing the correct table position. If the exponent is higher than the table length, a value of zero is used instead (recall that $\alpha < 1$).

A FIR filter is somewhat more complex to implement within an ISR. In order to optimize the processing time, when an interrupt occurs, the time of its occurrence is saved in a table containing the most recent interrupts (see Figure 9). In this filter decisions are made using a temporal window of size D. By making the table in the form of a *ring-buffer* of size D, the system can detect which interrupts to consider in $O(log(D))$ operations by sequentially dividing the array in two. It then advances the last interrupt pointer to the oldest interrupt within the last D sample periods. The sum of $x[n]$ is then given by the difference between the recent and the last interrupt pointers.

6 System Engineering: the RTEMS case study

In order to evaluate the ideas presented above we used the real-time kernel RTEMS. RTEMS (Real-Time Executive for Multiprocessor Systems) is a well-known real-time multitasking kernel, with a modular architecture, offering interesting characteristics to support the development of real-time embedded control applications [13]. It is an open source operating system, that is currently maintained by OAR (On-Line Applications Research Corporation – USA). It is available for several different platforms/architectures, including the PC386, which is the one we use in this work.

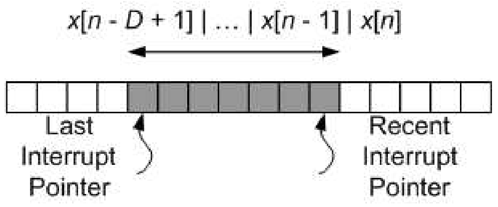

Figure 9. Ring-buffer for FIR filter memory

Although there are specific limitations associated with a given platform, as for example the granularity of interrupt level enable/disable operations in the PC386 platform [14], the basic functionality available, in what concerns I/O and interrupt management, makes it possible to build the desired temporal protection mechanisms.

Using RTEMS, we developed a simple text mode windows manager called VITRAL [4], that replaces the original RTEMS console driver, and takes into consideration timeliness properties (explained bellow), including a temporal protection mechanism associated with console input. With this mechanism we are able to tolerate overload situations such as those induced by a *stuck key*, without compromising the timeliness of other tasks. An earlier version of that work was presented in [3].

Although the main ideas used for the keyboard are also valid for other faster devices, for this paper we also made some experiments using the network Ethernet driver available in RTEMS, to show the use of our timeliness protection mechanisms in such scenarios. We intercept the interrupt handler, determine the interrupt rate, and, if it is above a given threshold, the interrupt is disabled for a given period of time. The results are presented bellow, in the results subsection.

6.1 VITRAL

The VITRAL (Portuguese word for Stained Glass Window) driver is a simple yet reliable multiple text windows manager [4]. It is completely compatible with standard I/O calls (stdio library) and each window can read from the keyboard and write to the output (Figure 10).

VITRAL, besides incorporating the timeliness protection mechanisms that we discuss in this paper, it is also integrated with the RTEMS core in such a way that minimizes the dependency on a specific RTEMS distribution. Figure 11 represents the architecture of that integration, presenting VITRAL, the new console driver, as an extension to the RTEMS core, to be used by the application. This is allowed by RTEMS since it is the application that

Figure 10. Aspect of VITRAL – a simple windows manager for RTEMS

indicates the needed drivers in a static manner.

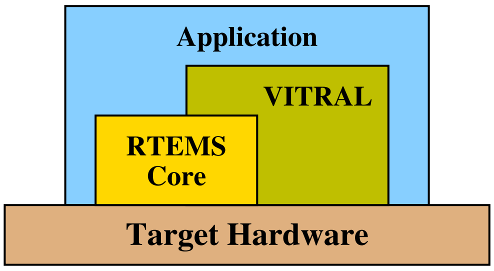

Figure 11. VITRAL architecture within the RTEMS scope

The internal VITRAL architecture includes a task that receives messages to create new windows or to process a *hot-key*. These *hot-keys* are used to select the desired input window and perform other functions such as hide or recover a specific window.

If not carefully addressed, VITRAL could present problems in what concerns timeliness properties. As the task that desires a window has to wait for its creation, if the VITRAL internal task is created with a low priority, then a priority inversion problem may occur: a medium priority task may occupy the processor and thus the window is never created, leaving a higher priority task blocked. If the VITRAL internal task has a high priority, then the processing of *hot-keys* may jeopardize other tasks timeliness.

To solve this problem one could take advantage of the priority inheritance algorithms provided by the kernel [13]. The chosen solution uses the priority ceiling algorithm in which the application must provide a constant that corresponds to the ceiling of the priority of all tasks that create a window. This ceiling becomes the priority of the VITRAL task during the window creation [19].

Another problem was related to input interrupt overhead and the ability to detect a failure such as a *stuck key*, preventing it from interfering with the timeliness properties of other tasks. Using the temporal protection mechanisms explained in this paper, the original interrupt handler associated with the keyboard was replaced by one that controls the number of allowed interrupts to determine from the interrupt rate if a failure has occurred, and act accordingly, before calling the original keyboard interrupt handler to process keys.

An important implementation aspect is the floating point operations during an ISR. One could take advantage of the FPU (Floating Point Unit) but RTEMS does not support by default (which is correct) the expensive context saving operations needed. Instead, a scale factor was added to allow the use of integers with little loss of resolution. For last, RTEMS only supports timers with the granularity of a clock tick, which is user defined but typically around $10ms$. This resolution is not suitable for this kind of filters due to the sampling time being of the same order as the minimum inter-arrival time of the interrupts, which is around $10 - 100\mu s$. So, a new function `getTime` was implemented based on the internal count of `Timer0` (in the Intel-386 architecture) offering a time resolution of $1\mu s$.

6.2 Results

To demonstrate the ability of the proposed control mechanisms to cope with overload scenarios, a simple test involving a rapid keyboard user and a deliberate *stuck key* is presented. Although the keyboard is not a very fast device (it can generate 33 interrupts per second), it can, nevertheless, be used as a representative input device. To determine the interrupt rate, we used both an IIR filter and a FIR filter with the following parameters: $T_{sample} = 1ms; M = 0,02; m = 0,002; \alpha = 0,999;$ $D = 1024$ (FIR only); $PollingPeriod = 300ms$ (polling task).

Figure 12 represents the situation with the IIR filter and where no protection mechanism is implemented. In the first 4000 samples, a user is pressing keys normally and $y[n]$ tends to stabilize to a relatively low value. In between $n = 7000$ and $n = 11000$ a *stuck key* is experimentally simulated, resulting in the rise of $y[n]$.

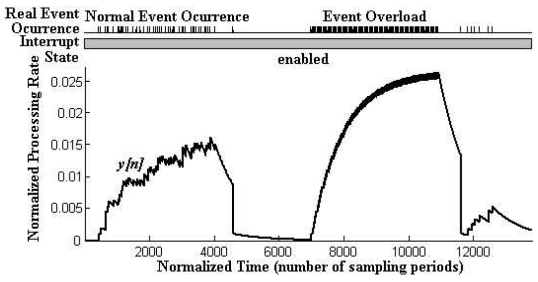

Figure 12. Event rate (IIR filter, no protection mechanism)

One can see that the user is not fast enough to trigger overload detection, whereas the rhythm of the stuck key is more than enough. At approximately $n = 4500$ and $n =$

11000, $y[n]$ decreases rapidly due to the way the function α^{n-last_n} is implemented in the ISR. The table used has a dimension of 200 sampling periods, meaning that if two interrupts are separated by more than this time interval, $y[n]$ is set to $(1-\alpha) = 0,001$. Besides reducing the size of the table, this is also a good mechanism to rapidly decrease $y[n]$ when the user stops pressing keys.

In Figure 13 the same scenario is represented, but now with the protection mechanism activated. When $y[n]$ goes above M, the system detects an overload, disabling keyboard interrupts and activating the polling task. During the overload, the system reads the keyboard and because there were pressed keys, the system stabilizes with approximately $y[n] = 1/300 = 0,0033$. When the overload is over ($y[n] < m$), the system enables the interrupts again and returns to normal mode.

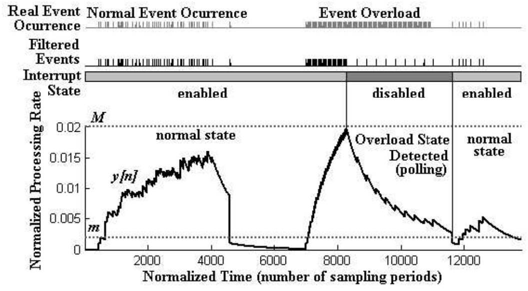

Figure 13. IIR Filter and protection mechanism

Similar results can be obtained using the FIR filter. Figures 14 and 15 represent the same scenarios without and with overload control mechanisms, respectively. Although both filters present a similar behavior, the FIR filter stabilizes more rapidly, especially when the interrupts are disabled. This happens because it follows an approximately linear curve while the IIR follows a decreasing exponential that has a slow variation when close to zero.

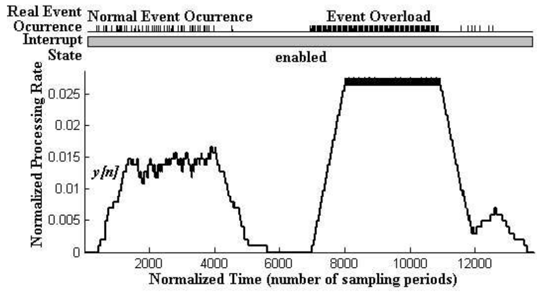

Figure 14. FIR filter without protection mechanism

Tuning the parameters

Decreasing α makes the system faster, detecting the *stuck key* in less time, but, on the other hand, it also allows the

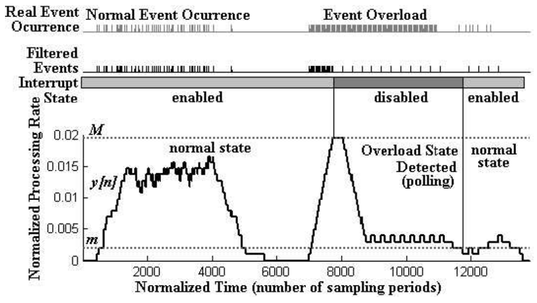

Figure 15. FIR Filter and protection mechanism

user to sometimes trigger a fault. The sample period must be such that only one interrupt may be triggered during that time. The determination of the polling period has to trade-of the CPU percentage usage and the time needed to determine if the failure has gone. Also, as said earlier, the m value must be a function of the polling frequency. In what concerns the threshold M, it can be defined taking into account the interrupt rate above which the system gets into overload, or one can choose another smaller value. A possibility is to choose the interrupt rate associated with a *stuck key*.

Controlling the load due to Ethernet

In order to evaluate, in practice, if the proposed timeliness protection mechanisms used with the keyboard could also be applied to a faster device, we made the following experience. We intercept the interrupt associated with the Ethernet driver, determine the interrupt rate, and, when the protection mechanism is active and the rate is above a given threshold (0.025 in the scenario presented in Figure 17), the interrupt is temporarily disabled, in order to avoid the extra load. Figure 16 represents the same situation without the protection mechanism activated. The network traffic was created using an application in another machine.

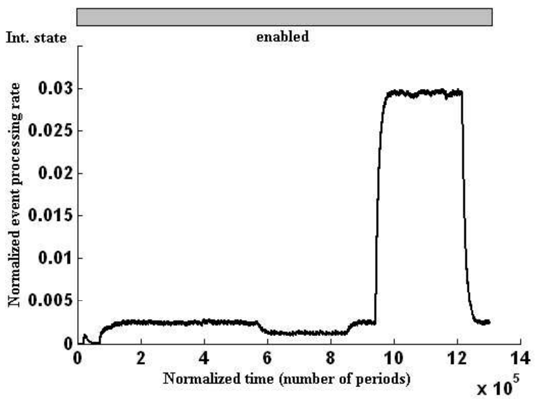

Figure 16. Interrupts associated with Ethernet without protection mechanism

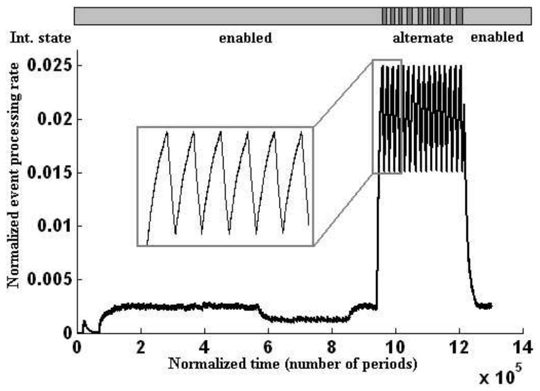

Figure 17. Interrupts associated with Ethernet with protection mechanism (enable/disable interrupt)

As shown, we are able to enforce that the interrupt load keeps bellow the specified threshold.

7 Conclusions and Future Work

Embedded control systems are used in many settings such as industrial and factory automation, automotive, railways, avionics and aerospace, medical applications, etc. Most of these applications have real-time requirements and, at the same time, need to interact with the real-world performing input/output operations through sets of sensors and actuators. This interaction with the external environment, and the related input/output event handling, may introduce some temporal uncertainty due to the fact that external events are not completely controlled by the system, which may imply overload scenarios.

In this paper we addressed the problem of ensuring timeliness guarantees through the use of protection mechanisms that prevent temporal interferences when dealing with input/output event handling.

Using the RTEMS real-time kernel, a simple windows manager called VITRAL, that replaces the original RTEMS console driver, was developed taking into account some of the referred temporal protection mechanisms. This is done as an RTEMS extension, with minimal impact on the existing infrastructure, and provides protection against such problems as a *stuck key* that would increase system load, possibly compromising task deadlines.

Although this case study uses a “console”, where the interaction is done with a human user, which is a “slow” interface, the same type of mechanisms can be applied to other input/output operations more related to device control or network traffic, for example, that have a smaller time granularity. That was experimentally demonstrated using the network Ethernet driver.

The protection mechanism presented works for any interrupt driven system, and can be implemented as a separate module so as not to modify each driver that uses this method. This can be done by replacing the ISR of the desired interrupt, do some calculations and calling the original ISR.

The ability to deal with such problems and adapt in a dynamic and flexible way is of utmost importance when supporting dependable real-time applications.

As future work, some improvements can be done by creating more elaborated algorithms to determine if a failure has occurred, and to better study the interaction with task scheduling.

References

[1] Webopedia - online dictionary for computer and internet technology definitions. http://www.webopedia.com. (http://www.webopedia.com/TERM/I/I_O_throttling.html).

[2] M. Barabanov and V. Yodaiken. Introducing real-time linux. *Linux Journal*, 1997(34), February 1997.

[3] M. Coutinho, J. Rufino, and C. Almeida. Control of event handling timeliness in RTEMS. In *Proceedings of the 17th IASTED International Conference on Paralel and Distributed Computing Systems - PDCS 2005*, Phoenix, Arizona, USA, Nov. 2005. IASTED.

[4] M. Coutinho, J. Rufino, and C. Almeida. VITRAL: A text mode windows manager for RTEMS. In *Terceiras Jornadas de Engenharia de Electrónica e Telecomunicações e de Computadores*, Lisboa, Portugal, Novembro 2005.

[5] L. E. L. del Foyo and P. Mejia-Alvarez. Custom interrupt management for real-time and embedded system kernels. In *Embedded and Real-Time Systems Implementation (ERTSI 2004) Workshop*, December 2004.

[6] M. B. Jones, D. Rosu, and M.-C. Rosu. Cpu reservations and time constraints: efficient, predictable scheduling of independent activities. In *Proceedings of the sixteenth ACM Symposium on Operating Systems Principles*, pages 198 – 211, Saint Malo, France, October 1997.

[7] H. Kopetz. Event-triggered versus time-triggered real-time systems. In *Proceedings of the International Workshop on Operating Systems of the 90s and Beyond*, volume 563 of *Lecture Notes In Computer Science*, pages 87–101. Springer-Verlag London, UK, 1991.

[8] H. Kopetz. Time-triggered real-time computing. In *Proceedings of the IFAC World Congress*, Barcelona, July 2002. IFAC Press.

[9] H. Kopetz and B. Gunther. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112–126, January 2003.

[10] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Proceedings IEEE Real-Time Systems Symposium*, pages 166–171, 1989.

[11] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *JACM*, 20(1):46–61, 1973.

[12] C. W. Mercer, R. Rajkumar, and J. Zelenka. Temporal protection in real-time operating systems. In *Proceedings of the 11th IEEE Workshop on Real-Time Operating Systems and Software*, May 1994.

[13] On-Line Applications Research Corporation (OAR). *RTEMS C User's Guide*, edition 4.6.2, for rtems 4.6.2 edition, August 2003. (The RTEMS Project is hosted at http://www.rtems.com.).

[14] On-Line Applications Research Corporation (OAR). *RTEMS Intel i386 Applications Supplement*, edition 4.6.2, for rtems 4.6.2 edition, August 2003.

[15] A. V. Oppenheim, R. W. Schafer, and J. R. Buck. *Discrete Time Signal Processing*. Prentice-Hall International, 2nd edition, 1999.

[16] A. Polze and L. Sha. Composite objects: Real-time programming with corba. In *Proceedings of 24th Euromicro Conference*, volume II, pages 997–1004, Vaesteras, Sweden, August 1998.

[17] J. Regehr and U. Duongsaa. Preventing interrupt overload. In *Proc. of the ACM Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2005)*, Chicago, IL, June 2005.

[18] K. D. Ryu, J. K. Hollingsworth, and P. J. Keleher. Efficient network and i/o throttling for fine-grain cycle stealing. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing*, Denver, Colorado, USA, 2001. ACM Press.

[19] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990.

[20] S. Smith. *The Scientist and Engineers Guide to Digital Signal Processing*. California Technical Publishing, San Diego, California, 2nd edition, 1999. Analog Devices Technical Library.

[21] P. Veríssimo, J. Kaiser, and A. Casimiro. An architecture to support interaction via generic events. In *Proceedings of the 24th IEEE Real-Time Systems Symposium. Work in Progress Sessions.*, Cancun, Mexico., December 2003.

[22] P. Veríssimo and H. Kopetz. Design of real-time systems. In S. Mullender, editor, *Distributed Systems, 2nd Edition*, ACM-Press, chapter 19, pages 491–536. Addison-Wesley, 1993.