A Latency Model of Linux 2.6 for Digital Signal Processing in Real Time

Sergio A. Rodriguez
Dept. Telecomm. and Control Engineering
Escola Politecnica, University of Sao Paulo
Itacolomi,423 CEP 01239-020 Sao Paulo SP, Brazil
sergio.rodriguez@poli.usp.br

Phillip M. S. Burt
Dept. Telecomm. and Control Engineering
Escola Politecnica, University of Sao Paulo
PTC/EPUSP 05508-900 Sao Paulo SP, Brazil
phillip@lcs.poli.usp.br

Abstract

This paper develops a new qualitative latency model of the Linux 2.6 OS for the Intel x86 architecture. The proposed model analyses aspects related to digital signal processing in real time. In this context, the study identifies all latency sources since the arrival of a signal sample (or block of samples) up to the execution of the first instruction related to processing that sample. The interrupt latency is divided into seven components, including hardware latency sources such as microprocessor operation and interrupt controller queue. The dispatch latency is divided into six components, adding new components such as interrupts stack latency and deferrable functions latency to the known sources such as scheduler latency and switch context latency. The paper also identifies further overhead sources such as memory allocation by demand paging. Finally some of the new latency components are measured using an instrumented kernel. The measures are obtained with and with out computational load for the purpose of analyzing its influence in the latency components.

1 Introduction

During the last years there has been an increase in the processing capacity of personal microcomputers. This increase suggests the possibility of using microcomputers instead of specific-use processors in several applications. An example of great practical interest would be the use of microcomputers compatible with the IBM-PC (Intel x86) running Linux for real time digital signal processing (DSP) applications, instead of using digital signal processors. However, the performance of such a system would be limited due to the fact that although the improvements for real time in version 2.6, Linux was not developed taking into account the requirements of real time DSP applications [1, 2].

In real time DSP applications one important factor is the sampling period, since each digital sample (or block of samples) has to be fully treated in this fixed time period. For this reason, the purpose of this paper is to study the Linux latency for DSP applications in real time. This latency establishes the lower bound for the sampling period; consequently it establishes an upper bound for the frequency of the signals that can be processed.

This work consists of two main parts. In the first part a qualitative model for the Linux latency in a DSP real time application is proposed. This model divides the latency into several components, considering the delays of both the microprocessor and the operating system. In the second part the Linux kernel is modified in order to allow measuring some of

the components defined in the model.

2 Background and main assumptions

2.1 Real time DSP program structure

Programs that perform real time digital signal processing have a particular structure, as shown in Figure 1. The central part of the program is basically a loop where 4 operations are executed: read the input sample, process the input sample, generate an output datum and check the end of the processing. This structure is similar to the one of a real time program [3, 4]. However, there is an important specificity: the input sample must be processed in a fixed time period known as the sampling period. The sampling period is limited by the frequency content of the input signal, according to the Sampling Theorem [5].

From the discussion above it follows that a real time DSP program needs a time mechanism to signalize the sampling time instant to it. Although Linux has software timers, reference [6] recommends the use of hardware timers in real time applications in order to be more precise. Consequently, it is assumed in this work that an interrupt request signal (IRQ) signalizes the sampling time instant. This interrupt is named IRQsa.

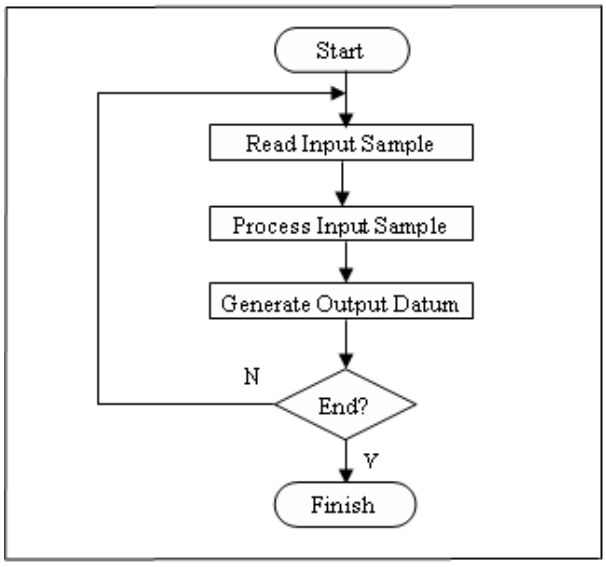

FIGURE 1: *DSP program block diagram*

2.2 Functional Model

For greater simplicity, a two-process functional model is assumed for a real time DSP application running in a multiprocessing operating system such as Linux. The first process is the DSP process. The second process ("NDSP process") is an abstraction that encloses all the other processes and kernel threads that are sharing the system. Figure 2 shows the dynamic of this model.

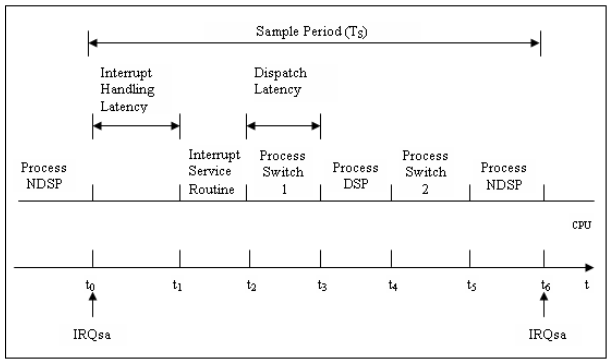

FIGURE 2: *Functional model*

The dynamic may be explained as follows:

1) At instant t0 a sampling signal is raised through the IRQsa. At this instant the DSP process is in the blocked state and the NDSP process is in the execution state. The main goal of the operating system is to put the DSP process into the execution state in a short time interval, because it is assumed that the digital processing is in real time.

2) The system requires a period of time to start the handling of the IRQsa. This time interval is known as Interrupt Handling latency.

3) At instant t1 the interrupt service routine (ISR) starts to be executed. The functions of this routine are: to read the digital sample (or block of samples) from the external device, to change the state of the DSP process into the ready state and to inform the operating systems that a process change is needed.

4) At instant t2 the ISR finishes and the operating system starts the process change. The time interval since t2 up to the instant when the DSP process passes into the execution state (t3) is known as Dispatch latency. This latency is historically divided into two components: Scheduler latency and Switch Context latency [7, 8].

5) The DSP process deals with the input sample between the instants t3 and t4.

6) At instant t4 the DSP process releases the Central Processing Unit (CPU), so it passes to the blocked state and calls the scheduler. The reason for this is that the DSP process needs to wait for the next sampling signal.

7) The operating system exchanges the processes between the instants t4 and t5. As the model assumes that there are only two processes in the system, at instant t5 the NDSP process passes to the execution state.

Two remarks about this model are in order:

1) The model implicitly assumes that the digital signal processing is performed by a process running in User Mode.

2) It is necessary that the digital signal processing finishes before the arrival of the next sampling signal. In case this doesn't happen the system is in an under-sampling condition.

3 Qualitative Model for the Linux Latency

Let the Linux latency be divided in three parts. The first part is the latency before the ISR execution, which, as previously defined in Section 2.1, is called Interrupt Handling latency. The second part is the latency after the ISR execution, called Dispatch latency. The third part is the latency due to the loss of the DSP process computation time caused by some operating system tasks (Section 3.3), which will be called Execution latency.

In the following the Handling Interrupt latency and the Dispatch latency are analyzed in detail. Consequently, a new model for the Linux latency is established.

3.1 Interrupt Handling Latency

This latency is divided into seven components as explained below.

3.1.1 Interrupt Checking Latency

A microprocessor of the x86 family checks whether an interrupt signal has been raised only when an instruction is completely executed [9, 10]. Therefore, there is a time interval between the interrupt signal raising and the microprocessor checking it. This time interval is defined as Interrupt Checking latency. This latency depends on the execution time of the instruction that is being processed by the CPU when the interrupt is raised.

The x86 family has a CISC (Complex Instruction Set Computer) architecture; consequently it has instructions with different execution times. Besides this, the execution time of some instructions depends on the values of the operands.

Both these factors lead to a greater variance of the Interrupt Checking latency, if compared with the variance which would be obtained with specific-use processors or RISC (Reduced Instruction Set Computer) architecture [11] processors.

3.1.2 Maskable Interrupt Latency

In a microprocessor of the x86 family, a maskable interrupt can be in two states: masked or unmasked. A masked interrupt is ignored by the control unit of the microprocessor as long as it remains masked [9]. The Maskable Interrupt latency is defined as the time interval during which an interrupt is ignored because it is masked.

In Intel microprocessors the interrupts are masked in two ways. In the first one, when an interrupt is being handled the microprocessor automatically masks all the interrupts (cleaning the IF bit of the EFLAGS register). The purpose of this is to avoid that more than one interrupt is treated by the system at the same time [10] until the operating system unmasks the interrupts.

In the second one, the operating system masks the interrupts by handling the IF bit of the EFLAGS register by means of appropriated functions (local_irq_enable and local_irq_disable). The purpose of this is to avoid a race condition and to execute critical regions of the kernel code without disturbance. Normally this happens when an exception is handled or a deferrable function is being executed [6].

3.1.3 PIC Queue Latency

The I/O devices demand an interruption of the microprocessor through a signal called interrupt request (IRQ). The IRQ lines are connected to a circuit called Programmable Interrupt Controller (PIC). This device monitors the IRQ lines and signalizes the microprocessor when an interrupt is raised [10].

When an IRQ line is activated, the PIC executes the following actions [10]:

1) Stores the interrupt vector in a PIC I/O port, thus allowing the CPU to read it via the data bus.

2) Raises a signal to the microprocessor INTR pin.

3) Waits until the CPU acknowledges the interrupt signal.

4) Clears the INTR line.

It can be observed that there is a time interval between the IRQ signal activation and the release of the INTR. In case an interrupt is raised during this time interval,the controller puts it into a pending state and deals with it when the INTR line is released. The controller establishes a priority, so that if one or more interrupts are pending, the one with the greater priority will be the first to be attended.

The PIC Queue latency is defined as the time during which an interrupt remains pending in the controller queue to be attended.

3.1.4 Microprocessor Operating Latency

When the microprocessor detects an interrupt, it needs to execute some instructions before starting the execution of the Linux interrupt handler. In microprocessors of the x86 family, the most important operations are [10]:

1) Save the state of the EFLAGS, CS and EIP registers in the stack.

2) Load the CS and EIP registers with the values that define the logical address of the routine which handles the interrupt.

The Microprocessor Operating latency is defined as the time interval demanded by these operations.

3.1.5 Registers Saving Latency

The microprocessors of the x86 family don't save all the common use registers. So the first operation of the Linux interrupt handler is to save all the registers that were not automatically saved by the microprocessor [6].

In Linux all interrupts raised by I/O devices are directed to the same address. This is performed by putting the same address on every one of IDT (Interrupt Descriptor Table) entrances that belong to interrupts raised by I/O devices [6]. The aim of this procedure is to deal with this type of interrupt in the same way.

The Registers Saving latency is defined as the time interval demanded for this saving.

3.1.6 Interrupt Handler Entrance Latency

Linux implements software layers to abstract certain hardware characteristics such as the following: different types of PIC, dynamic allocation of IRQ lines and sharing of an IRQ line by several I/O devices.

These software layers delay the beginning of the ISR execution and consequently they produce latency.This delay is caused by several operations such as [6]:

a) Check the interrupt number.

b) Control of nested interrupt handler execution.

c) Change from kernel stack to IRQ stack.

d) Update fields of the interrupt vector descriptor.

The Interrupt Handler Entrance latency is defined as the time interval demanded by these operations performed by Linux before the ISR execution.

3.1.7 Shared IRQ Lines Latency

Linux supports the sharing of IRQ lines by more than one I/O hardware device. When there is a sharing, the interrupt code doesn't identify the I/O device anymore. Consequently when a shared IRQ is raised, the Linux needs to the check all devices that signalize by means of this interrupt [6].

This is performed by a loop that runs along a linked list of descriptors which contain information about each ones of the I/O devices associated with the IRQ line.

The Shared IRQ Lines latency is defined as the time interval demanded for this loop.

3.2 Dispatch Latency

In this section, the delay after the ISR execution is analyzed. As in the previous section, this latency is divided into several components.

3.2.1 Interrupt Handler Exit Latency

This latency is caused by operations performed after the ISR execution. As already analyzed in Section 3.1.6, these operations are caused by the software abstraction layers implemented by Linux to handle an interrupt. Examples of these operations are [6]:

a) Check if the device is a source of random events.

b) Check if the interrupt was handled. The purpose of this is to detect spurious interrupts raised by malfunction of the hardware.

c) Update fields of the interrupt vector descriptor.

d) Control of nested interrupt handler execution.

Interrupt Handler Exit latency is defined as the time interval demanded for these operations performed at the end of the interrupt handling.

3.2.2 IRQ Handlers Stack Latency

Linux allows interrupt nesting but it doesn't allow kernel preemption while an interrupt is being treated [6]. In this way it is possible that when an IRQsa is raised, the operating system is treating one or more interrupts. This creates a stack of interrupts being treated, called IRQ Handlers stack. A necessary requirement for this is that all the interrupts in the stack could be treated with the unmasked interrupts [6]. A field in the interrupt descriptor signalizes to the operating system if the handler is executed with the interrupts enabled or disabled.

In this scenario a process change is only performed when all interrupts in the stack have been treated. The IRQ Handlers Stack latency is defined as the time interval demanded to empty the IRQ Handlers stack.

3.2.3 Deferrable Functions Latency

Linux has a mechanism called softirqs or deferrable functions that it is used to execute operations that need to be delayed. A common example is the implementation of software timers. In several points of the kernel code, Linux verifies if some of these functions need to be processed [6].

One of these checking points is in the interrupt handler, when all the interrupts in the IRQ Handlers stack have been treated (Section 3.2.2). Consequently, a time interval is demanded to check and execute the deferrable functions. This time interval is defined as Deferrable Functions latency.

3.2.4 Kernel Preemption Disable Latency

Version 2.6 of Linux could be compiled to have or not kernel preemption by means of the compilation parameter CONFIG_PREEMPT. Even when the kernel is compiled to have kernel preemption, the operating system disables the kernel preemption to execute some procedures. An example is during the execution of a deferrable function [6].

As a consequence, at the end of all interrupt treatments Linux needs to check whether the kernel preemption is enabled or not. Therefore, if the IRQsa is raised when the kernel preemption is disabled, the process change will only occur when the operating system enables it again. This delay in the process change is defined as Kernel Preemption Disable latency.

3.2.5 Scheduler Latency

The interrupt service routine that treated the IRQsa signalizes that the scheduler has to be called by the operating system because a process change is required. This occurs due to the fact that the DSP process was changed into the running state (Section 2.1).

The time needed by the scheduler routine to find the DSP process in the waiting queues is defined as Scheduler latency.

3.2.6 Context Switch Latency

After the DSP process is found in the waiting queues, the operating system needs to switch the data structures of the current process to the ones of the DSP process. The time required for this update is defined as Context Switch latency.

3.3 Execution Latency

Execution latency can be defined as the difference between the time spent in the input sample processing and the time that would be spent if the procedure were executed as an atomic procedure. The factors that can delay the input sample processing are described below.

3.3.1 I/O Interrupt

The DSP process can be delayed by the raise of hardware interrupts, since the operating system uses computational time of the DSP process to treat them.

3.3.2 Deferrable Functions

Linux runs the deferrable functions using processing time of the current process. Linux checks if any of these functions needs to be processed at the end of

either an exception or an interrupt handling (Reference [6] mentions that although there are standard checking points, there are possibly other checking points depending on the version and the distribution). Consequently, it is possible that when the DSP process executes a system call, a deferrable function is executed using CPU time of the DSP process.

3.3.3 Virtual Memory

As Linux implements virtual memory, it is possible that when the IRQsa is raised part of the program or part of the data is not in the RAM (Random Access Memory) memory. In this case, a Page Fault exception is raised and the operating system has to load the memory page from the hard disk, causing a great impact in the processing time latency.

4 Latency Measurement Method in Linux

4.1 Basic Idea of the Method

The basic idea of the measurement method proposed in this paper can be divided in two parts:

1) Introduction of time markers in the kernel routines that treat the interrupts. Theses markers read timing circuits of the system, marking both the initial and final instants of execution of the kernel code portion under analysis. Latency can be calculated by the difference between those instants.

2) Development of a program that behaves as DSP program and manages the measurement, obtains the markers values, calculates and stores the latencies in the user space memory. Before finishing, the process writes a text file with the data. This process is called Management process.

The advantage of kernel instrumentation is the direct measuring of the execution time. The disadvantage is that the use of the markers implies alterations in the kernel code. These alterations cause systematic errors due to the processing time of the markers. These errors can be minimized by measuring the execution time of the markers and making the corresponding corrections.

As this method is based only on software, the markers can be placed only after the registers saving. Due to this, the latencies due to the hardware can only be measured aggregately. Consequently, the Hardware latency is defined as being the sum of the following latencies: Interrupt Checking, Maskable Interrupt, PIC Queue, Microprocessor Operating and Registers Saving.

The measurement of the Hardware latency needs synchronization between the external device that raises the interrupt and the kernel. The proposed method satisfies this requirement by using the IRQ0 (timer interrupt) to measure the Hardware latency. The IRQ0 is raised when the PIT (Programmable Interrupt Controller) timer expires. Thus, it is enough for the first marker to read the PIT to obtain the Hardware latency.

An advantage of using the IRQ0 is that it is a periodical interrupt, allowing periodical measurements. The disadvantage is that the IRQ0 is the interrupt with the highest priority so it has PIC Queue latency equal to zero. This limitation can be overcome measuring the Hardware latency by using the interrupt raised by the CPU Local timer in a similar way as it is done with IRQ0.

This method can been modified to use the interrupt raised by the HPET (High Precision Event Timer) in systems which use this timer circuit instead of the PIT.

4.2 Implementation Aspects

A data structure was created in the Linux 2.6 kernel. Its main purpose is to store the data measured by the markers in the kernel space memory. The structure also has fields to control the measurement and avoid wrong measures in case of nested interrupts. It is initialized during the load of the operating system.

There are two type of markers. The first one was already mentioned and uses the PIT timer to measure the hardware latency. The second one uses the TSC (Time Stamp Counter) register and is used to measure the other latencies originated by the kernel routines.

Two system calls were developed. The first one starts and ends a set of measures. Its functionality is similar to the open and close file commands. The second system call blocks the management process, placing it in a waiting queue. It also transfers the values from the data structure in the kernel space memory to a structure in the user space memory of the management process.

The service interrupt routine that attends the IRQ0 was altered. The two purposes of this alteration are to put the management process into the running state and to signalize the operating system that the scheduler has to be called.

5 Experimental Methodology

The microcomputer that was used was a Pentium III, with 700 MHz clock frequency and 640 MB RAM.

In the Linux 2.6.19 kernel, markers were placed to measure the following latencies: Hardware, Interrupt Handler Entrance, Interrupt Handler Exit and Dispatch.

The kernel was compiled with the standard values for the compilation parameters and to load the standard daemons. It is important to mention that in this configuration Linux has no kernel preemption.

An additional control program was developed. This control program makes 100 calls to the Management program (item 4). The time interval between the calls is 120s, controlled by a Linux timer (SIGALARM). The Management program performs 100 measures that are saved in a text file. Following this method, 10.000 measures were obtained in approximately 3 hours and 20 minutes. The purpose of the control program is to obtain a greater time interval between the measures blocks than it would be possible using only the IRQ0 signal. In this way there is equilibrium between the time interval and the data size.

The measurement was performed with the system in two conditions: loaded and unloaded. In the unloaded condition the control program was executed directly from the shell. In the loaded condition the control program was executed from the graphic environment. An extra CPU bound program was executed as a way of loading the system. In both cases the standard scheduling of the Linux was used.

This test can be considered a worst case because the kernel has no preemption and the real time scheduling of Linux 2.6 was not used.

6 Experimental Results

This section shows the CDF (Cumulative Distribution Function) and the mean of four latency components: Hardware, Interrupt Handler Entrance, Interrupt Handler Exit and Dispatch. A data analysis is also presented.

6.1 Results

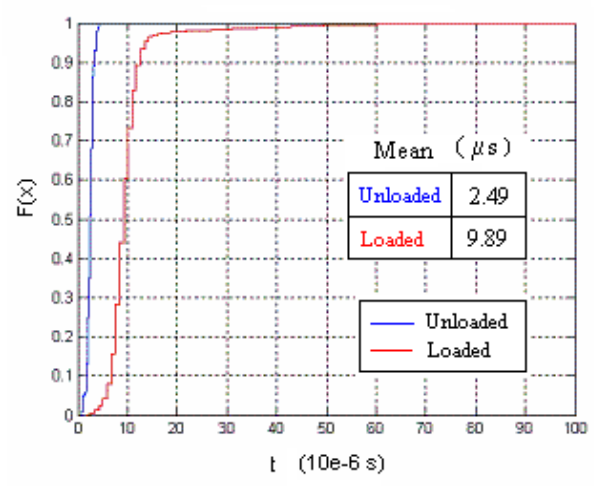

FIGURE 3: *CDF Hardware Latency*

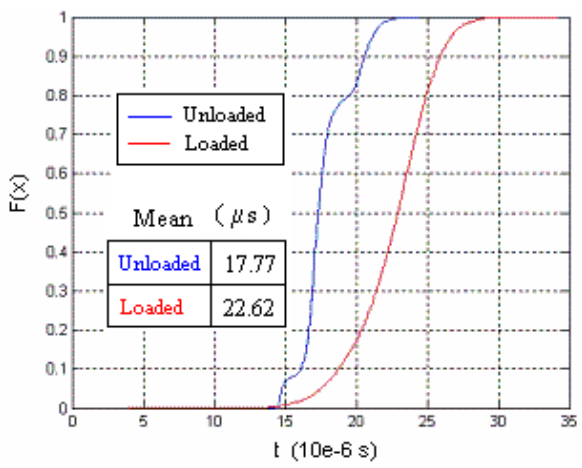

FIGURE 4: *CDF Interrupt Handler Entrance Latency*

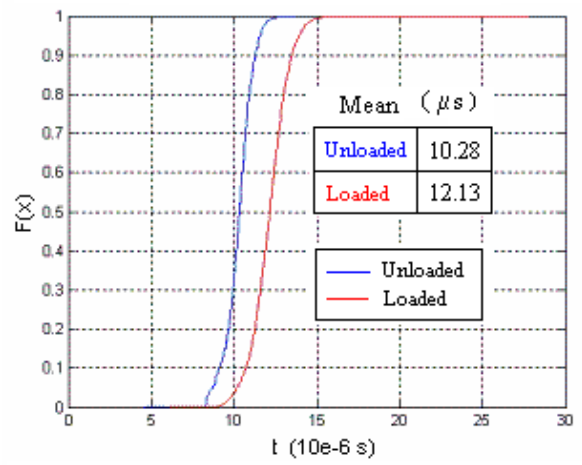

FIGURE 5: *CDF Interrupt Handler Exit Latency*

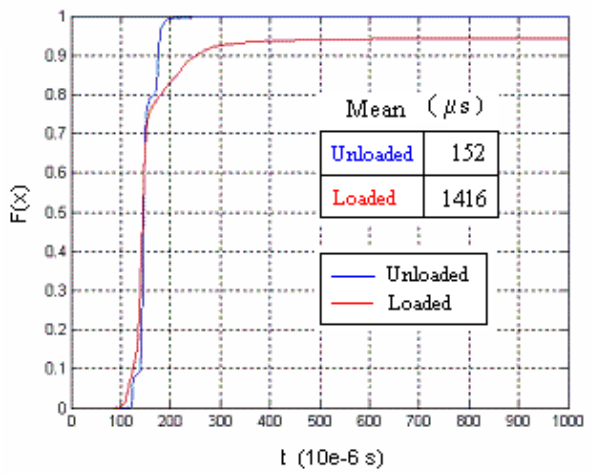

FIGURE 6: *CDF Dispatch Latency*

6.2 Data Analysis

1) The mean of the Dispatch latency approximately increased in 800% with load. Such an increase may be explained by two factors: the kernel had no preemption and the real time mechanism of the scheduler was not used.

2) The mean of the Hardware latency approximately increased in 300% with load. As an I/O bound program was not used, such an increase may be principally explained by the synchronization mechanisms of the kernel than unable the interrupts.

3) The load also influences the execution of the interrupt handler. This may be observed by the 20% increase in the means of the Handler Interrupt Entrance latency and the Handler Interrupt Exit latency. This increase is expected because with load the cache hits diminish.

7 Conclusions

This paper proposed a new qualitative latency model of the Linux 2.6 OS for the Intel x86 architecture. This work also developed a latency measurement method based on an instrumented kernel.

References

[1] *The Evolution of Real-Time Linux*, Dietrich S, Walker D, Nov 2005,Proceedings of 7th Real-Time Linux Workshop, Lille, France

[2] *A Real-Time Linux*, Yodaiken V, Barabanov M, Jan 1997,Proceedings of Linux Applications Development and Deployment Conference (USELINUX), Anahein, USA

[3] *Software Engineering*, Sommervile I, 2007, Pearson Education Limited

[4] *Real-Time Systems Design and Analysis*, Laplante P A, 2004, IEEE Press

[5] *Signals & Systems*, Oppenheim A, Willsky A, Nawab S, 1996, Pearson Education

[6] *Understanding the Linux Kernel*, Bovet P D, Cesati M, 2005, OReilly Media Inc.

[7] *A Measurement-Based Analysis of the Real-Time Perfomance of Linux*, Abeni L, Goel A, Krasic C, Snow J, Walpole J, Set 2002,Proceedings of the 8th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)

[8] *Linux Scheduler Latency*, Clark W, 2002,Red Hat, Inc.

[9] *Intel 64 and IA-32 Architectures Software Developers Manual Volume 3A: System Programming Guide, Part 1*, Nov 2006,Intel Corporation

[10] *Architectures Software Developers Manual Volume 3B: System Programming Guide, Part 2*, Nov 2006,Intel Corporation

[11] *Microprocessors Outperform DSPs 2:1*, Blalock G, Dez 1996Microprocessor Report The Insiders Guide to Microprocessor Hardware