Universidade de Aveiro Departamento de Electrónica, Telecomunicações e Informática,
2011

Jorge Manuel Coelho Amado de Azevedo

Xenomai Lab - Uma Plataforma para Controlo Digital em Tempo-Real

Xenomai Lab - A Platform for Digital Real-Time Control

Jorge Manuel Coelho Amado de Azevedo

Xenomai Lab - Uma Plataforma para Controlo Digital em Tempo-Real

Xenomai Lab - A Platform for Digital Real-Time Control

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requesitos necessários à obtenção do grau de Mestre em Engenharia Electrónica e de Telecomunicações, realizada sob a orientação científica de Dr. Alexandre Mota, Professor do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro

o júri / the jury

presidente / president

Prof. Doutor José Alberto Gouveia Fonseca
Professor Associado da Universidade de Aveiro

vogais / examiners committee

Prof. Doutor Alexandre Manuel Moutela Nunes da Mota
Professor Associado da Universidade de Aveiro (orientador)

Prof. Doutor Paulo Bacelar Reis Pedreiras
Professor Auxiliar da Universidade de Aveiro (co-orientador)

Prof^{a}. Doutora Ana Luisa Lopes Antunes
Professora Adjunta do Departamento de Engenharia Eletrotécnica da Escola Superior de Tecnologia de Setúbal do Instituto Politécnico de Setúbal

agradecimentos / acknowledgements

Ao Prof. Doutor Alexandre Mota por me ter sugerido este tópico para a dissertação. Sem essa primeira abordagem, nenhum deste trabalho teria sido possível. Embora haja uma divergência clara ao nível do gosto em rock clássico, há um equilíbrio nos nossos interesses técnicos que fez o trabalho ser produtivo e relaxado ao mesmo tempo ao longo destes meses.

Ao Prof. Doutor Paulo Pedreiras pela orientação e disponibilidade praticamente constante. Por me mostrar, vez após vez, onde estava errado e pacientemente me colocar no caminho certo. Infelizmente não partilhamos um gosto por rock clássico.

Ao Prof. Doutor Rui Escadas por não ter hesitado em dispensar parte do seu tempo para me ajudar a montar o circuito de benchmarking usado nesta dissertação. Também divergimos ao nível do rock clássico, mas consideravelmente menos do que no caso do Prof. Doutor Alexandre Mota. São escolas diferentes, no fundo.

Ao Diego Mendes por me ter disponibilizado a sua biblioteca de matrizes e, claro, por ter coragem de usar a minha aplicação para fazer a dissertação dele. É de homem.

Ao Tiago Gonçalves por me ter ajudado a montar e ter ensinado como fazer o PCB para o pêndulo invertido.

Ao Bruno César Almeida por ter desenhado o logotipo do Xenomai Lab e me ter revisto parte do grafismo nesta dissertação. Para um ignorante do pantone, foi uma ajuda fundamental.

Aos meus pais pois sem eles nada disto seria possível. À minha irmã por estar sempre e incondicionalmente lá.

Palavras-Chave Xenomai, Temp-Real, Controlo Digital, Sistemas de Controlo, Diagrama de Blocos

Resumo

O Xenomai Lab é uma plataforma *open-source* que permite a um utilizador projectar gráficamente um sistema de controlo recorrendo a um diagrama de blocos. O sistema projectado pode ser executado em tempo-real a uma frequência de operação de até 10KHz pela *framework* de tempo-real Xenomai. Execução pode ser uma mera simulação numérica, ou uma interacção com o mundo real recorrendo a blocos de input e output. A instalação traz de origem vários blocos potencialmente úteis, como um osciloscópio, um gerador de sinais, interface com perfis de *setpoint* feitos em MATLAB, entre outros. É também incluída documentação e alguns exemplos ilustrativos.

O desenvolvimento do Xenomai Lab teve por base uma pesquisa exaustiva de sistemas operativos de tempo-real baseados em GNU/Linux. As performances de Linux, do patch PREEEMPT_RT, do RTAI e do Xenomai foram medidas recorrendo a um mesmo teste. Desta forma, tornou-se possível fazer uma comparação directa entre as diferentes tecnologias. De acordo com os nossos testes, o Xenomai apresenta um balanço ideal entre performance e facilidade de utilização. O *jitter* de escalonamento esteve sempre abaixo de $35\mu s$ num computador de secretária.

O Xenomai Lab foi desenvolvido de forma a ser fácil de utilizar. Esta é a característica chave que o distingue de *software* semelhante. Algoritmos de controlo são programados em linguagem C, não sendo necessário nenhum conhecimento específico de Xenomai ou mesmo de sistemas de tempo-real em geral. Assim, o Xenomai Lab é adequado para engenheiros da área de controlo sem experiência em GNU/Linux ou sistemas operativos de tempo-real ou mesmo estudantes de engenharia de controlo, robótica e outras áreas técnicas. Utilizadores avançados sentir-se-ão imediatamente em casa.

Keywords Xenomai, Real-Time, Digital Control, Control Systems, Block Diagrams

Abstract

Xenomai Lab is a free software suite that allows a user to graphically design control systems using block diagrams. The designed system can be executed in real-time with operating frequencies of up to $10KHz$ using the Xenomai framework. Execution can be merely a numerical simulation or an interaction with the real-world via input/output blocks. Several useful blocks are included in the default installation, such as an oscilloscope, a signal generator, MATLAB setpoint profile loader, and others. A rich set of documentation and examples is also provided.

Development of Xenomai Lab was supported by a thorough study of real-time operating systems based on GNU/Linux. The performances of standard Linux, the PREEEMPT_RT patchset, RTAI and Xenomai were benchmarked using a standard test. This allowed for a direct comparison between them. Xenomai was found to have the ideal balance between performance and ease of use, with scheduling jitter bellow $35\mu s$ on a desktop computer.

Ease of use was one of Xenomai Lab's main goals. This distinguishes it from alternatives. Control algorithms are programmed in C and no prior knowledge of Xenomai, or real-time operating systems in general for that matter, is needed. This makes our system adequate for use by control engineers unfamiliar with GNU/Linux and by entry level students of control engineering, robotics, and other equally technical areas. Advanced users will feel right at home.

Contents

Bibliography **151**

List of Figures

Acronyms

AD Analog to Digital

ADEOS Adaptive Domain Environment for Operating Systems

API Application Programming Interface

APIC Advanced Programmable Interrupt Controller

ARM Advanced RISC Machine

ASCII American Standard Code for Information Interchange

CD Compact Disc

CFS Completely Fair Scheduler

CPU Central Processing Unit

CTW Cascading Timer Wheel

DA Digital to Analog

DIAPM Dipartimento de Ingegneria Aerospaziale Politecnico di Milano

DMA Direct Memory Access

EDF Earliest Deadline First

FAQ Frequently Asked Questions

FIFO First-In First-Out

FSR Full-Scale Range

GNU GNU's Not Unix

GPOS General Purpose Operating System

GUI Graphical User Interface

HAL Hardware Abstraction Unit

IDE Integrated Development Environment

IPC Inter-Process Communication

IRQ Interrupt Request Line

ISR Interrupt Service Routine

KISS Keep It Simple, Stupid

MVC Model-View-Controller

NRT Non Real-Time

OCO Oven Controlled Oscillator

OS Operating System

PC Personal Computer

PCB Printed Circuit Board

PCI Peripheral Component Interconnect

PIT Programmable Interrupt Timer

POSIX Portable Operating System Interface

PWM Pulse-Width Modulation

RAM Random-Access Memory

RCU Read-Copy-Update

RM Rate Monotonic

ROM Read-Only Memory

RT Real-Time

RTAI Real-Time Application Interface

RTDM Real-Time Driver Model

RTOS Real-Time Operating System

SAL System Abstraction Layer

SHM Shared Memory

TSC Timestamp Counter

TTL Transistor-Transistor Logic

USART Universal Serial Asynchronous Receiver Transmitter

XL Xenomai Lab

Part I

Introduction

Chapter 1

Motivation

A **real-time operating system** (RTOS) is an operating system specially designed to support applications with very precise timing requirements. RTOSs are used in embedded systems to provide reliable multimedia, reliable network communications and other distinct, time sensitive operations. One of the possible uses of RTOSs is for control systems.

Control is an engineering discipline that tries to manipulate the behavior of a given system, called the *plant*, to behave according to a predefined rule. This objective is fulfilled by monitoring the plant's outputs and manipulating its inputs, the relation between which is defined by a *controller*. Control systems are abundant in nature. As an example, consider the balance of the human body. To remain upright, one needs to adjust his position constantly. If one were to let the muscle fully relax, the body would naturally fall to the ground. Control systems are also abundant in our daily lives. Pre-heating an oven requires that we merely adjust the dial to the desired temperature. The built-in control system of the oven will make sure that the heating resistances stay on until the desired temperature is reached, and then turned off when the temperature is exceed, and turned back on when the temperature drop bellow a certain threshold, and so on. In the earlier part of the 20th century it was common to implement a controller as a machine or an electrical circuit. Since then, due to the availability of quality digital computers at a low price, controllers are mostly implemented in software and interact with the real word through an interface [1].

The availability of free software RTOSs based on **GNU/Linux** make them ideal candidates for control engineers to build their systems upon. Reality, however, tells us a different story. These OSs are seldom used within the control community, while other, proprietary, OSs are commonplace. To understand why this happens, let us first take a brief look at

what free software is, and where it came from.

The **free software** movement began in the early 80's when Richard Stallman lead an initiative to build a free implementation of UNIX. By free implementation Stallman did not mean free as in "free beer" but rather free as in *freedom*. The source code was to be available free of charge and be free to study, modify and distribute. It was named **GNU**, a recursive acronym for GNU's Not Unix [2].

The project intended to develop everything from a compiler to a kernel, even games. By the early 90's, most of the major pieces of a complete OS were built, except for one - the kernel. In 1991 Linus Torvalds announced to an unsuspecting world that he had completed a working version of his Linux kernel [3]. **Linux** was a monolithic and simple kernel that offered very little in the way of features, but was immediately incorporated into GNU and has developed at a very fast rate ever since.

GNU/Linux has seen mass adoption in certain areas, while in others it lacks penetration. Almost 90% of super computers run some modified version of GNU/Linux [4], over 60% of the Internet's servers run GNU/Linux [5] and Android, the number one smartphone operating system in the world, runs Linux in the background. In the desktop, adoption is usually held to be around 1%[6]. Justifying why these numbers are as they are is a matter of opinion. It is the opinion of the author and the team behind this work that the high adoption of GNU/Linux among computer scientists and software engineers attests for its superior quality. On the other hand, the lack of adoption in less computer literate sectors attests for its lack of user friendliness. By this we mean that GNU/Linux has a **steep learning curve** to install and to use.

Having said that, let us return to the issue at hand. There is an enormous potential in utilizing GNU/Linux for real-time control, yet this potential has never really been fulfilled. Xenomai Lab and this document are an attempt to bridge the gap between control engineers and real-time GNU/Linux. The effort is two fold.

First, we intend to assert the expected **real-time performance** on a desktop computer. This allows an engineer to easily assert if the system is suitable for his needs. Prior work has been done in this field and we intend our work to confirm and build upon these past results.

Having established the performance, our second effort consists in building an application where control systems can be easily modeled and tested in real-time. Control systems are usually projected by use of a block diagram, where different elements of the system are modeled by use of a transfer function. **Xenomai Lab** allows the user to **project his own**

block diagram with pre-built blocks or by programming his own. The user can test his system by simulating a plant, or he can use one of the computer's I/O ports (such as the parallel, or centronics, port) to control a real plant. All of this work is done using the C language and is mostly decoupled from any specific notions of RTOSs. In addition to all this, our objective is to make installation as painless as possible by use of intelligent packaging. The end result is that any control engineer can jump right in with almost no additional knowledge. The knowledge that is needed, we intend to fully document with a rich set of examples and guides.

In a nutshell, our objective is to completely bridge the gap between control engineers and real-time GNU/Linux. By reducing the entry barrier to almost zero, it is our hope that GNU/Linux can take a greater position in both the teaching of control systems and the control industry.

1.1 Objectives

The formal objectives of this dissertation can be summarized in the following points:

- To study the most important real-time Linux solutions available today;
- To put forth a quantitative analysis of operational jitter in signal generation;
- To build a platform where control systems can be easily designed and tested.

1.2 Organization

This dissertation has been divided into four distinct logical components.

In **Part I**, the theoretical fundamentals of control systems and RTOSs are presented as support to later stages of the work. In chapter 1 we state the motivation and outline the objectives for our work. In chapter 2 we present the elements of digital control systems, while in chapter 3 we present a brief overview of the most important real-time concepts in operating system design.

Part II sees Linux and all the main approaches to improving its real-time behavior analyzed. Chapter 4 presents the most important elements of Linux's architecture and benchmarks the high resolution timer mechanism and the PREEMPT_RT patchset. Chapter 5 is dedicated to the study of the co-kernel approach to performance improvement. In this chapter we explain the historical context, introduce and benchmark RTAI

and Xenomai. Finally, chapter 6 compares and contrasts the different results obtained and aligns them with previous results obtained in the field.

Part III is dedicated to Xenomai Lab. Chapter 7 provides an introduction to the software suite, explaining the fundamental design choices and analyzing some of its alternatives. Chapter 8 contains an in-depth look at the architecture of the program. In chapter 9 the program is tested under different circumstances and an attempt is made at controlling an inverted pendulum via the parallel port.

Part IV is our conclusion. Chapter 10 gathers the most important conclusions of the entire document and presents opportunities for future work.

Part V contains the appendixes. These are documents that are important enough to be present, but are either too specific or inappropriate to be placed in the main body of work.

Chapter 2

Control Systems

In this chapter we present an overview of the most important concepts associated with control systems. A fairly in-depth look at the process of analog to digital and digital to analog is presented. An emphasis is given on the negative consequences of jitter in the sampling frequency.

2.1 Overview

In chapter 1 we've seen how control systems are abundant in nature and in our modern lives. These intuitive notions are important to familiarize one's self with the fundamental concepts of control, but a more formal definition is necessary to gain further understanding of the subject.

A **control system** is built as an interaction between several smaller systems to fulfill an objective. A **system** can be defined as [7]

> "A combination of components that act together to perform a function not possible with any of the individual parts."

There are many types of systems with varying degrees of complexity. The engine of a car is an example of a mechanical system, a hi-fi audio amplifier an example of an electronic system, etc.

In general, all systems have inputs and outputs and impose a relation between the two. The collection of mathematical equations that define the input/output relation of a system is that system's **mathematical model**. Rarely are mathematical models exact, the more common case is that the model is an *approximation* of the system [8]. A very complex

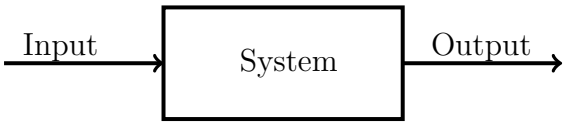


Figure 2.1: Graphical representation of a system as a block.

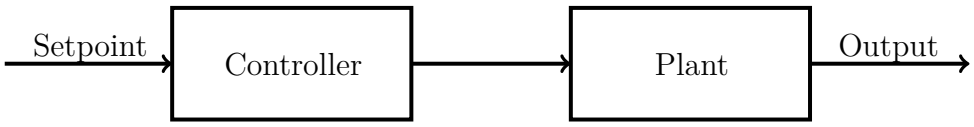


Figure 2.2: Block diagram of an open-loop control system

model can be composed of a large number of equations and go beyond the I/O relation and reveal the inner workings of the system. The behavior is predicted with a high degree of precision. A simpler model can be a coarse approximation of the behavior but be easier to understand and manipulate. Models come in very different forms and are used according to the precision required.

Mathematical modeling is a very powerful tool for understanding and ultimately manipulating the natural world. Fig. 2.1 shows a simplified representation of a system as a block. A block acts as a black box that, given an input, or a *cause*, produces an output, or an *effect*, according to a transfer function.

A **transfer function** is the I/O relation of the system. They are commonly written as differential equations in time, although other options exist. A transfer function reveals nothing about the *nature* of the block. The same transfer function can represent mechanical, hydraulic, thermal or even electrical phenomena. Even though their nature seems radically different at a first glance, they abide by the same rules. One might say that all systems were created equal in the eyes of mathematics, if one were so biblically inclined.

In a control system, the system under control, the *plant*, has its inputs manipulated in such a way as to produce a desired output. Fig. 2.2 illustrates this as a block diagram, where both elements are represented as system blocks and their connections by directed lines. This variety of control system is the simplest form of control and is usually called **open-loop** control, as no loop is formed.

In this system, the output of the plant does not affect the action of the controller. A conceptual example of such a system is a hand dryer operated by a timed button. The desired output is dried hands, and by pushing the button the air heater is on for 15 seconds. The fact that one's hands aren't dried by the end of the cycle does not make the hand

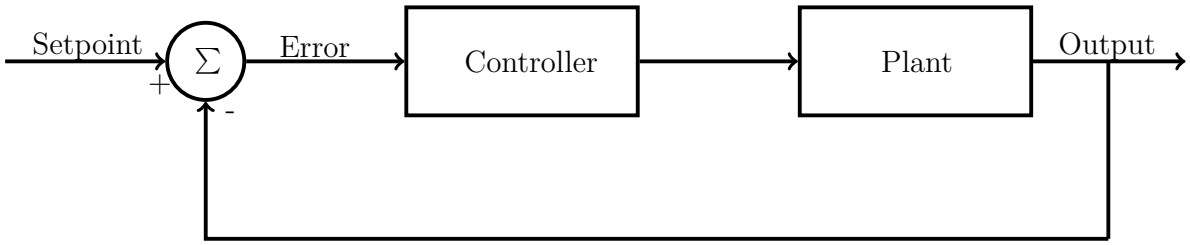


Figure 2.3: Block diagram of a closed-loop control system

dryer stay on any longer. It is a one way street.

Fig. 2.3 illustrates a **closed-loop** system. Here the output is fed back to the controller after being subtracted to the desired output, called the *reference* or *setpoint*. The result of the subtraction is the difference between the desired and the current output, called the *error*.

The controller acts upon the plant so as to make the error equal zero, at which point the desired output has been achieved. A system such as this is a **negative feedback loop**. An example of such a system is a hot-air balloon. In it, the operator of the burner keeps adjusting the hot-air production until the desired altitude is reached. If the altitude suddenly decreases, the operator re-ignites the burner until the altitude is reached once again.

If the error signal were to be made to grow continuously it would be a **positive feedback loop**. It is one of these loops that occurs when there's "feedback" during a live music show. What happens is that the microphones on stage pick up the sound coming out of the loudspeakers. A loop is formed where the amplified sound is amplified again and again. This causes the audio equipment to ultimately saturate, with the resulting sound being a loud high-pitch noise at the frequency of resonance of the equipment.

Each of these strategies has its advantages and disadvantages. An open loop is cheaper because it has less components but is limited in function and requires a very precise knowledge of the plant. Within a closed loop, a controller can adjust its operation to account for imperfections in the plant at a cost of greater system complexity. This makes a closed loop system much less sensitive to external disturbances or internal variations of the plant's parameters [1].

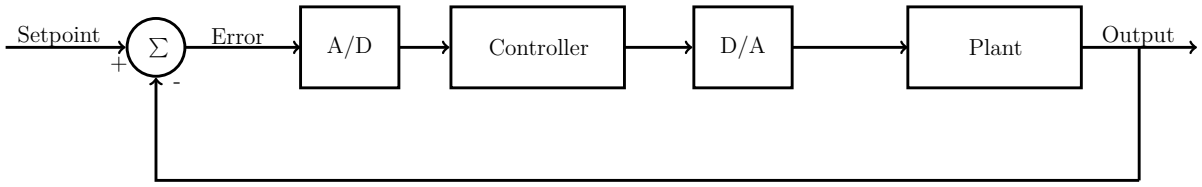


Figure 2.4: Block diagram of a closed-loop control system

2.2 Digital Control

Prior to the invention and democratization of digital electronics and computers, control systems were implemented with analog elements. In the analog world, variables are represented by continuous physical quantities or signals. A controller could be electronic and use amplifiers to manipulate signals, or an electromechanical servomechanism and interact with machines directly, to name but a few examples.

The rise of cheap and powerful computers and micro-controllers brought in the age of **digital control**. Implementing a controller programmatically in the form of logical operations on abstract numbers brings plenty of advantages. It makes for more flexibility, as the controller can be reprogrammed as needed. Digital computers also bring lower sensibility to noise, a smaller size, a more rugged construction, all at a lower price [8].

Fig. 2.4 illustrates a simple digital control system. The controller is enclosed in an analog-to-digital and a digital-to-analog converter[1]. Control is no longer direct. First the signal must be acquired, which in this context is a process called **data acquisition**, and then a signal must be produced, which is called **actuation**.

Conversion between the analog and digital domains is a nontrivial exercise. A complete description of the various conversion algorithms is out of scope of this work. Nevertheless, a brief presentation of the process's fundamental mechanisms and fragilities will prove beneficial to our current discussion and later chapters.

The conversion from **analog to digital** (A/D) consists in the following three steps:

1. **Sampling**, which converts a continuous-time signal to a discrete time base;

2. **Quantization**, that approximates the continuous values of signal amplitude to a set of discrete steps;

[1]This diagram isn't ultimate and final. There are multiple ways of implementing a digital controller but our simple example is still perfectly valid for the sake of argument.

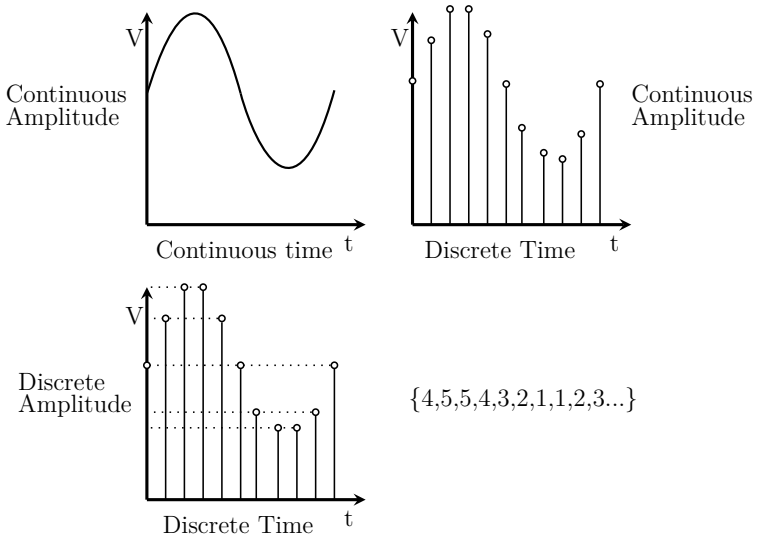


Figure 2.5: The four stages in an analog to digital conversion

3. **Encoding**, by which the quantized result is encoded to a binary (or digital) representation.

These steps will lead to the four stages seen in fig. 2.5.

An A/D converter will accept values within a given range and encode them with a number of bits n. In doing so, the limited amplitude range will be divided into discrete steps. The number of steps will be the total number of binary combinations 2^n. The difference between each step is the **quantization level** Q, and is defined as [8]

$$Q = \frac{FSR}{2^n} \tag{2.1}$$

Where FSR stands for Full Scale Range, i.e. the accepted input amplitude range.

The error introduced by this process is called **quantization error** and varies between 0 and $\pm\frac{Q}{2}$. By increasing the number of bits, the quantization error goes down.

The opposite conversion, from **digital to analog** (D/A), is illustrated in fig. 2.6 and consists also in three steps:

1. **Reconstruction**, which converts the binary digits to discrete voltage steps;

2. **Hold**, which holds the voltage until the next sample, converting the signal to a continuous time base;

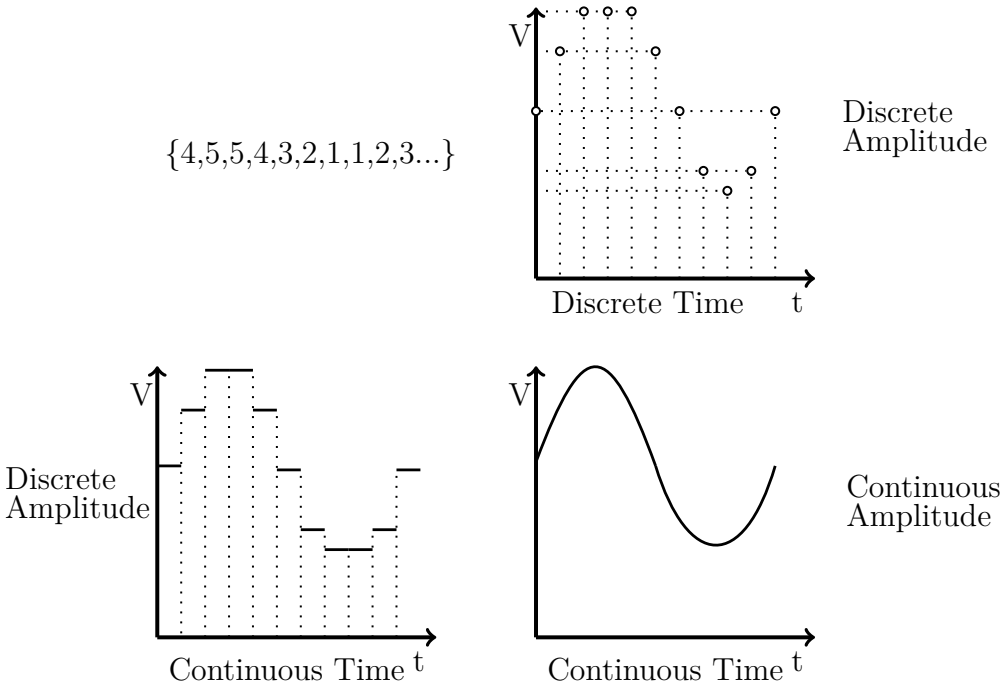


Figure 2.6: The four stages in a digital to analog conversion

3. **Interpolation**, which generates continuous voltage values between the samples.

An important aspect of both these processes is the **sampling frequency**. According to the Nyquist theorem, no loss of information occurs if the sampling frequency is equal to double the bandwidth of signal (or in other words, the highest frequency present in the signal) [8]

$$f_s \geq 2f_{sig} \tag{2.2}$$

Another critical timing consideration is the regularity of the sampling instants. No signal is perfect, and the signal that clocks the D/A and the A/D conversions will suffer variations in period, called *jitter*, that affect the overall precision of the conversion.

Jitter has a nefarious influence on both conversions. Fig. 2.7 presents a detailed look at the A/D conversion. Each sample is taken T seconds apart, and on the right hand side, a closer look at a sample illustrates the effects of a small variation Δt of period T.

As an example, consider that our input signal is a sine wave of the form

$$v(t) = Asin(\omega t) \tag{2.3}$$

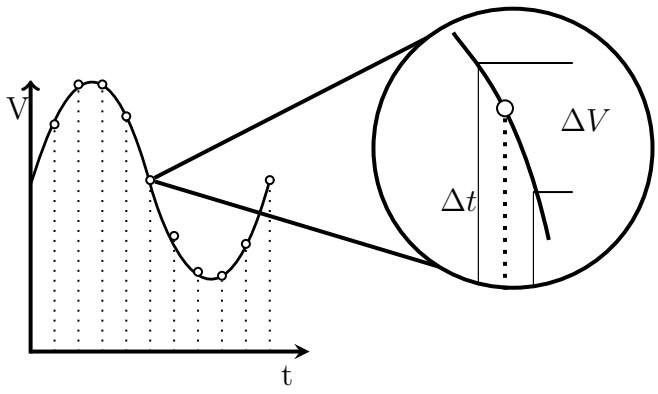


Figure 2.7: Detail of the consequences of jitter during sampling

The sampling frequency here can be as low as double the frequency of the wave.

Let us consider the effects of our sampling period being off by a small variation Δt. As can be seen in fig. 2.7, to a variation in T corresponds a variation of amplitude ΔV.

The worst-case ΔV happens when the derivative of our wave is maximum, and, therefore, the point where the amplitude changes more rapidly.

$$\dot{v}(t) = \omega A cos(\omega t) \tag{2.4}$$

$$Max\{\dot{v}(\mathrm{t})\} = \omega A,\ \omega t = \pm k\pi,\ k\epsilon\mathbb{N} \tag{2.5}$$

At $\omega t = 0$, sin is 0. This is the result we expected, a sine wave "drops" faster when it crosses zero, then slows down towards the peaks. By expanding our Δ into a subtraction, we have

$$V_2 - V_1 = Asin(\omega t_2) - Asin(\omega t_1) \tag{2.6}$$

For small variations of Δt, the linear approximation $sin(\omega t) \approxeq \omega t$ is valid, by substituting in equation 2.6, we have that

$$V_2 - V_1 = A\omega t_2 - A\omega t_1 \tag{2.7}$$

$$\Delta V = A\omega\Delta t \tag{2.8}$$

As we can see, there is a linear relation between ΔV and Δt for small variations of Δt.

If we express Δt as a percentage j_i of the sampling period T_S

$$\Delta t = j_i T_S, \quad j_i \, \epsilon \, [0, 1] \tag{2.9}$$

and define our sampling frequency as a multiple k of the bandwidth of the signal

$$f_s = k f_{sig}, \quad k \geq 2 \tag{2.10}$$

By substituting accordingly in equation (2.8), we obtain the result

$$\Delta V = A 2\pi \frac{j_i}{k} \tag{2.11}$$

To a higher jitter j_i corresponds a higher error in amplitude. If we increase the sampling frequency multiple k and j_i remains the same, then ΔV varies in inverse proportion.

As we've just seen, the presence of signal jitter causes a considerable degradation of reliability in data acquisition. It also has serious consequences on actuation.

Controllers are projected not by experimentation but by mathematical calculations based around the transfer function of the plant. This transfer function is calculated with the assumption that the system does not change or changes predictably over time. Consider the transfer function of an aeroplane, which traces the relation between engine power and altitude. The relation depends upon the weight of the plane, as more weight requires more engine power to sustain the same altitude. On a first look, one would think the plane has a constant weight. In reality, as the plane flies it burns fuels. Since fuel has weight, as time goes by the plane gets *lighter*. A controller of the plane's altitude can then adjust engine power as the plane burns fuel.

If the plane were to be damaged and, say, leak fuel, then our model of the plane would not be valid anymore. The prediction of the weight would be wrong, and hence, control of the altitude would fail.

It can be shown that actuation jitter will alter the control system in this very way [9]. The result will be equivalent to actuation with no jitter on a system that's time-varying in a random way. If jitter is very high, the control system can become unstable.

Control systems have stringent timing requirements for this very reason. A reliable control system must behave predictably in time. And the operating system that supports the control system must behave as well.

Chapter 3

Real Time Operating Systems

In this chapter the fundamental tenets of real-time and non real-time operating systems are presented. Concepts such as latency, jitter, deadlines and scheduling are discussed at length.

3.1 In Tune and On Time

The canonical definition of a real-time system, according to the real-time computing FAQ, is the following [10]:

> "A real-time system is one in which the correctness of the computations not only depends upon the logical correctness of the computation but also upon the time at which the result is produced. If the timing constraints of the system are not met, system failure is said to have occurred."

In less technical terms, real-time computing is when what matters isn't solely the result, but *when* the result is produced. Compressing a folder into a zip file is an example of non-real time computing. We want the result to be logically correct, i.e. the files compressed without errors. If the operation takes two seconds instead of one, the quality of service has been degraded but the result is still correct. The system didn't perform as well but it didn't *fail*. Watching a video, on the other hand, requires an image displayed every 40ms[1]. If a frame is late then it's not worth displaying anymore. A late result, is, in essence, equivalent to no result at all. If many frames are lost, the result is no longer a video and the system is said to have failed.

[1] This is equivalent to a frame rate of 25 frames per second.

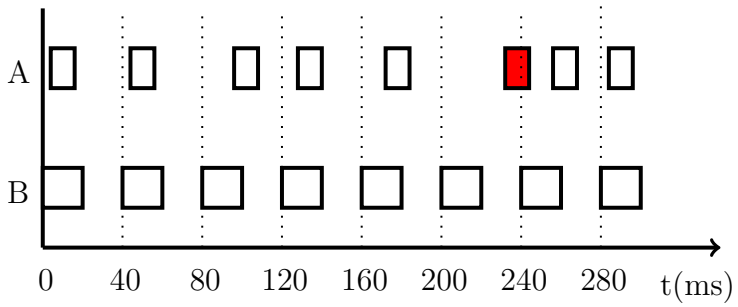


Figure 3.1: A non real-time system A and a real-time system B periodically producing a frame. A missed deadline is marked red.

Examples like the one just given illustrate the point but tend to generate a miss-conception. One might think that to be real-time, and hence "keep up", a system has be *fast performing*. In reality, a system needs only to be *deterministic* and hence keep its deadlines to be real-time [11]. It is actually the case that real-time system usually *under-perform* when compared to non real-time counterparts.

To further clarify this point, let us look at the video example once again. An image needs to be produced every 40ms and let us suppose we have two systems performing this operation: non real-time A and real-time B. A can produce the image in 5ms but B needs 10ms. A is clearly in the lead, but what happens if we put the system under stress? If A is busy over 35 ms in any of the 40 ms periods, a frame will be dropped. System B, however, will continue working 10ms out of each 40ms to produce frames because it always keeps its deadlines. When we say that a real-time system is deterministic, we mean that **temporal performance is decoupled from system load and all other aspects of system operation**.

Fig. 3.1 illustrates this failure. While system B behaves predictably in time, A does not and ultimately produces a frame too late.

Determinism is a common metric of real-time systems. It is common for people in the field to separate systems in two categories based on this metric: hard real-time and soft real-time [12].

Hard real-time systems are designed to meet their deadlines 100% of the time. No matter what the system load, a deadline will always be met. In hard real-time systems, missing a single deadline brings catastrophic consequences. An example of hard real-time is the flap system of an aeroplane, where missing a deadline during landing can mean the destruction of the system, or worse yet, the people in it. Most hard real-time systems don't

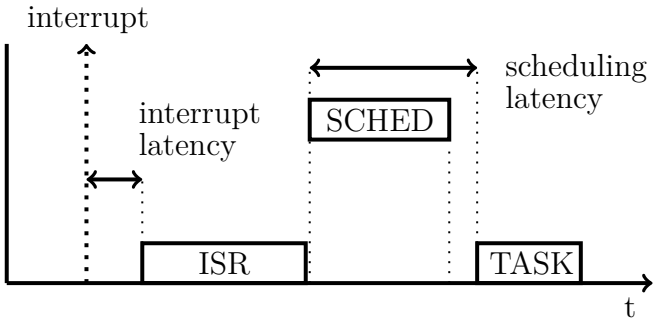


Figure 3.2: Common latencies during system operation.

have failures as deadly as our example, but need their deadlines met nonetheless.

Soft real-time system can meet their deadlines *most* of the time, but not always. In soft real-time systems missing a deadline means a degradation of quality of service, but nothing catastrophic. Our video example falls in this category.

3.2 Real-Time Essentials

As we've seen, real-time systems are defined by their temporal performance. There are many ways to characterize a system's behavior in time, and it's helpful for our purposes to review the most important definitions.

Latency, or response time, can be defined as the time elapsed between a given stimulus and its appropriate response [13]. Latency is used as a measure of system **responsiveness**, or, in other words, how quickly the system responds.

Scheduling latency is the latency measured between when an action was scheduled to execute and when it actually executes. In the video example of the previous section, we saw how an increase in scheduling latency will eventually mean the loss of deadlines.

Interrupt latency is the time between when the interrupt line in the CPU goes active and the correspondent Interrupt Service Routine (ISR) executes. This measures how quickly a system can respond to an external request.

Jitter is a variation of latency. A real-time system may have higher or lower interrupt latency, but it *must* have low jitter because it *must* be deterministic.

Another important notion to keep in real-time systems is the **worst-case** value. A system may have an average scheduling latency of $15us$. If, for whatever reason, that value sometimes doubles to $30us$, that is the worst-case scheduling latency of the system. As

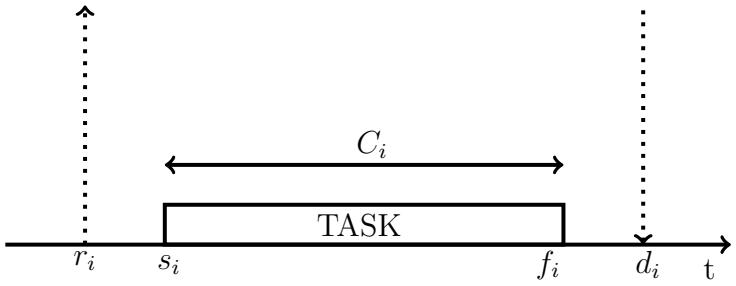


Figure 3.3: The most important figures characterizing a real-time task.

we've seen, for hard real-time system where a deadline can *never* be missed, the fact that on average the scheduling latency is $15us$ is irrelevant because it cannot be guaranteed 100% of the time.

Fig. 3.2 attempts to illustrate these concepts. When an interrupt occurs, a certain interrupt latency elapses before the ISR is ran. When the ISR finishes, the scheduler is called, and after some time a task of interest starts executing.

These metrics characterize the system running our real-time tasks, but don't characterize the tasks themselves. As we've seen in the previous section, real-time tasks are of a different nature than non real-time. Recalling our video example, what jumps to mind is that real-time tasks are periodic. This needn't be the case, as a task can be aperiodic and still be real-time. The key characteristic is that real-time tasks have **deadlines**, or in other words, a point in time after which the result is undesirable or useless.

Fig. 3.3 shows the most important characteristics of a generic real-time task i. The task becomes ready for execution at r_i, the **release time**, but will experience a delay and only commence execution at s_i, its **start time**. The task then takes a certain amount of time C_i to complete, called **computation time**. This leads to the task finishing at f_i, the **finishing time**. For correct operation f_i must happen before d_i, the **deadline**. Some more metrics can be established for a real-time task, but these few are sufficient for our purposes.

Our video was an example of a task with a periodic release time and a relative deadline equal to the period.

3.3 Operating Systems and Purposes

An RTOS is an OS built towards achieving temporal determinism and the support of real-time operations. Examples of RTOSs are OSs such as Vx-Works [14] or QNX [15]. On the other end of the spectrum we have General Purpose Operating Systems (GPOS), such as Linux or Windows.

This distinction is important because GPOS and RTOS have opposing goals. By exploring this opposition, it will become apparent why transforming Linux into an RTOS is such a complex task.

An operating system provides an abstraction layer between the system's hardware and the application programs that interest the user. It strives to be transparent, to decouple the usage of a given user program from the hardware. This way, an application is built for an OS and will run in that OS regardless of the underlying hardware. In providing this functionality an operating system implements three fundamental mechanisms [16]:

1. **Abstraction**. By providing an abstract interface to concrete hardware, an OS allows for simpler and portable application design. An example of such an abstraction is a read operation. This allows an application to read a variable number of bytes from a given file and deposit them in some location. The read operation is specified in the same way whether the file is on a flash disk, a hard drive or a CD-ROM. The read call hides the inherent complexity of talking to all the different I/O controllers with specific timings and handle the multitude of errors that may arise.

2. **Virtualization**. Most OSs support execution of multiple applications simultaneously, even on systems with a single CPU. The available resources must then be *shared* among applications. OSs provide this feature by assigning a virtual machine for each application. This allows applications to be written as though they have the resources all to themselves, when in reality they are competing for them. An example of this is virtual memory. Each application gets from the OS a continuous block of memory that in reality may be split among the RAM, the CPU cache or the swap file on disk.

3. **Resource Management**. Since the applications are isolated from the underlying hardware, it is up to the OS to manage the concurrent access to all the available resources. The management is done in a way that will maximize overall system performance while assuring that no application gets neglected.

The difference between OSs lie in the fine details of the implementation of these three basic concepts. For instance, part of the resource management duties of an OS is to schedule programs for execution in the CPU according to a given rule. This process is called **scheduling**, and the rule is usually called a **scheduling policy**. A GPOS will try to schedule tasks so that on average each task spends a fair amount of time executing. "Fair" is not a precise concept. As such, an ideal policy does not exist as it depends on numerous factors, such as what kind of tasks are being ran by the user. An RTOS, on the other hand, is an OS where the applications that interest the user are real-time tasks (as defined in the previous section). Scheduling should then be done in such a way that all real-time tasks are serviced before their deadlines no matter what.

Let us consider that an RTOS is trying to schedule a set of n periodic real-time tasks, n being an integer greater than zero. Each individual task i belonging to the set is of period T_i and computation time C_i. Let's assume that these parameters do not change during system operation. If we normalize C_i in relation to T_i and add them for all our set, then we will obtain the CPU utilization U, hence defined as [17]

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \tag{3.1}$$

As an example, consider that, when combined, our set of n tasks require $0.7s$ of computation time each second. This yields a U of 70%. With this metric it is immediate to see if the hardware is suitable for the intended operation. If a set of tasks require a U of over 100%, then a faster CPU is needed to cope with the load. Even if U is bellow 100%, it is not guaranteed that the set can be successfully scheduled. Let us look at two basic examples of real-time scheduling policies and their impact in CPU utilization.

Rate Monotonic Scheduling (RM) is a scheduling policy where tasks with higher frequency are given higher priority. If we assume that task frequencies don't change mid-system operation, RM is an example of a fixed priority scheduling policy.

It is possible to assert if a given group of real-time tasks can be successfully scheduled by RM with no missed deadlines. If we were to assume that in addition to constant T_i and C_i, the relative deadline d_i is equal to the period T_i and that our tasks are independent from one another and fully preemptible, then it has been shown that a sufficient condition for successful scheduling is [17]

$$U \leq n(2^{1/n} - 1) \tag{3.2}$$

For an increase in n, U tends towards $ln2 \approxeq 70\%$. This means that if our set of real-time tasks, whatever they may be, don't need CPU more than 70% of the time, an RM scheduling policy will successfully schedule the set. This leaves 30% of the CPU idle. This can be quite wasteful, and our next example improves upon this result.

Earliest Deadline First (EDF) is a scheduling policy in which priorities are assigned based not on the *length* of the deadline, but rather in its *proximity* to the present time. This means that priorities are continuously being updated to reflect imminent deadlines, which makes EDF a dynamic-priority assignment policy. It has been shown that for the same assumptions we've established for RM, a necessary and sufficient condition for successful scheduling under EDF is

$$U \leq 1 \tag{3.3}$$

EDF guarantees that if the computation requirement of the task set does not exceed the capacity of the CPU, the tasks can be successfully scheduled.

Having finished this analysis, the more astute reader will probably ask – Why can't such scheduling policies be integrated in a GPOS? Real-time tasks could remain high priority and be scheduled by RM while the idle time could be managed by some other scheduling policy.

Firstly, the more astute reader would be advised to lower his voice as his thoughts may be trespassing intellectual property [18]. Secondly, merging policies is perfectly valid, it is done in Linux and Windows, for example. However, it solves only *part* of the problem. The fission between a GPOS and an RTOS is far greater than merely the scheduling policy, as the CPU isn't the only resource managed by the OS. Let us look at an example.

A valid strategy for a GPOS to implement virtual memory is to use **paging** [16]. By dividing each memory space into 1-16KB blocks, or *pages*, the OS can easily manage which memory is currently in RAM or stored in swap. If a program needs access to a page that is not in RAM, a page fault occurs and the OS responds by trying to load the respective page into memory. If no memory is available, the process may sleep indefinitely [16].

In an RTOS, the performance emphasis is on temporal determinism and reduced latencies. A paging strategy as we've just described is, therefore, not acceptable. With it, the computation time of a task is dependent on a page being on memory. If a page fault occurs, the time at which the task resumes execution is unknown. There are many other examples of GPOS strategies of resource usage optimization that directly conflict with real-time objectives. We will look at this issue further when discussing Linux' implementation.

Part II

Real-Time Linux

Chapter 4

Linux

In this chapter a complete overview of the most important Linux concepts is presented. The major attempts at improving Linux' real-time performance are analyzed and benchmarked. These are the hrtimers mechanism and the PREEMPT_RT patchset. The chapter closes with a direct comparison of the benchmarking results obtained.

4.1 Linux 101 - An Introduction

Linux is a monolithic kernel. It runs as a single process in a single address space. This does not mean that the kernel needs to be compiled as one big static binary. Linux supports loading and unloading components (called **modules**) during execution time. Typically, the more essential kernel systems are statically compiled, while hardware support (i.e. drivers) are compiled as modules and loaded during boot. The essential kernel systems are components such as a process scheduler to coordinate access to processor execution time, memory management, interrupt handlers, networking, etc.

The kernel process runs in an elevated privilege mode, while secondary processes lay on top of the kernel in an unprivileged mode. The rationale is that potentially destructive operations such as direct hardware access are reserved for the kernel. Any other process needing elevated privileges operations asks the kernel to do it *for them*, using a mechanism called **system calls**. The kernel then acts as a proxy between the untrusted process and the operation it wants done on a restricted system, e.g. write 1024 bytes to a file on disk.

This dichotomy between privilege and lack thereof is the difference between **kernel space** and **user space**.

Fig. 4.1 illustrates the previous points.

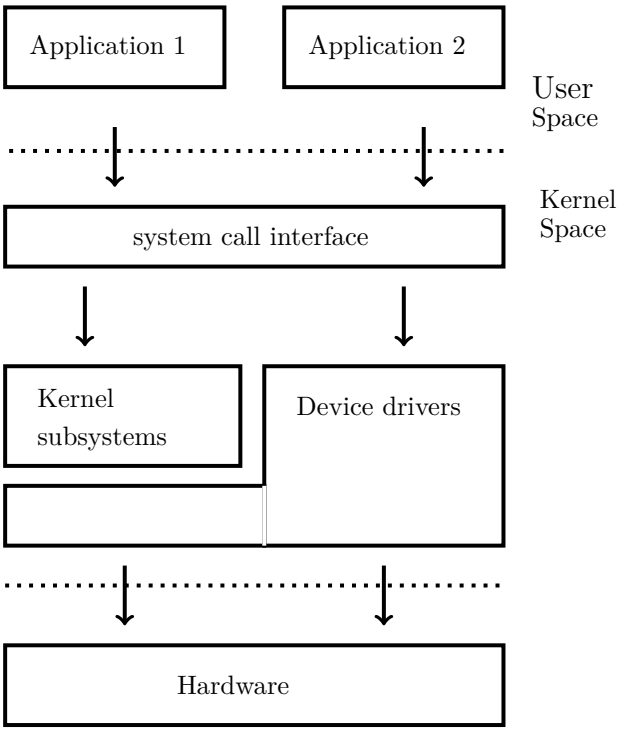


Figure 4.1: Conceptual structure of the operating system.

4.1.1 User Space vs. Kernel Space

Kernel space is the privileged mode in which the kernel runs. For any program to run in kernel space it must be either compiled into the kernel binary or as a separate module and loaded in runtime. The main characteristic of kernel space is that there is no memory protection and no paging. Since everything in kernel space shares the same address space, an invalid memory operation, e.g. dereferencing a NULL pointer, generates a kernel panic. This brings the whole system to a halt. In addition to this, every allocated byte of memory stays in physical memory (RAM). It never gets swapped to disk to make space for other processes.

User space is the regular, unprivileged mode in which every other executable runs. All modern features of an operating system are exposed to user space: per-process virtual memory, inter process communication, etc. The caveat is no direct use of low-level functions, user space uses the kernel as a proxy for its actions by means of system calls.

4.1.2 Processes and Scheduling

In Linux, a process is made of two parts: the **active task**, meaning the binary executable, and its related **resources** such as its priority, current state, opened files, identity number, etc. Both of them are brought together in the **process descriptor**, a data structure that holds all the process' associated variables. Notice that when we speak of a process we imply execution, not a program stored in disk to be executed in the future.

Linux supports multitasking, that is, the concurrent execution of different processes. Since there are usually many more processes than CPUs to execute them, processing time has to be shared. To coordinate this concurrent access to the processor is the purpose of the **scheduler**. As the name implies, the scheduler schedules processes for execution. It does this by assigning a lease of processor time to each process, usually called a **timeslice**. Different processes get different timeslices according to a given norm, called the **scheduling policy**. When execution switches from one process to another, the previous state of the CPU (the *context*) has to be saved and a new state has to be loaded[1]. This is called a **context switch**. A context switch from one process to another may be voluntary or involuntary. When a process *yields* its execution to another by directly calling the scheduler we call it **yielding**. When a process is *interrupted* by the scheduler so that another can take its place, we say that the process has been preempted by the scheduler, and hence we call it **preemption**.

Examples of both yielding and preemption are numerous throughout the kernel. For instance, when a process issues a read call to read information stored on the disk, it will sleep until the disk retrieves the information. To do this, the process will remove itself from the list of schedulable processes and directly call the scheduler so that another process can take its place on the CPU. This leaves the task **blocked** (or *sleeping*) and is an example of *yielding* execution. Preemption is more common. For instance, when an interrupt occurs whatever process is running is preempted as to allow the interrupt handler to run.

The heart of the scheduler is the scheduling policy. It is beyond the scope of this work to analyze this subject at depth since it is an area of constant flux – algorithms are adopted and discarded fairly rapidly. The problem of scheduling offers no perfect solution since it is inherently contradictory. To maximize efficiency we need to minimize context switches, to reduce latency and improve interactivity we need more preemption.

It is important to refer that Linux supports multiple scheduling policies at the same

[1]The state of the CPU includes all of the CPU's register and other important elements, such as the virtual memory mapping and stack information, among others.

time and switching between them at runtime. There are three "normal" scheduling policies – SCHED_OTHER, SCHED_BATCH and SCHED_IDLE – and two real-time scheduling policies – SCHED_FIFO and SCHED_RR. A process can assign itself a scheduling policy and a static priority between 0 and 99.

Processes assigned to **SCHED_FIFO** or **SCHED_RR** must have a static priority between and 1 and 99, while for the other scheduling policies the priority is always 0. This means that real-time processes will *always* preempt any normal process that happens to be running. The difference between the two real-time policies is that a SCHED_FIFO process runs until it decides to yield, while SCHED_RR processes are assigned a timeslice after which they are preempted.

SCHED_BATCH is intended for "batch" type processes, i.e. processes that require no user interaction and usually run in the background. **SCHED_IDLE** are for processes with the lowest possible priority and will only run when no other processes are available.

SCHED_OTHER is the default scheduling policy which most user space applications use. This policy implements an algorithm that can be defined as a boot parameter. Typical choices are the Completely Fair Schedule (CFS), O(1), and others.

Given that yielding generally only happens when a task intends to block, most of the context switches and calls to the scheduler occur via preemption. However, some areas of the kernel are *not* preemptible. This fact greatly affects Linux's real-time performance. If a timer is to go off at a time when the kernel is not preemptible the latency will be higher than during a preemptible section. This variability is a clear degradation of performance.

User space preemption happens in two situations: when the execution is returning to user space from a system call and from an interrupt handler. In other words, whenever there's a system call or an interrupt, the scheduler is called. This vulnerability to interrupts means that user space is *fully preemptable* as long as interrupts are not explicitly deactivated (which is a rare and looked down upon practice).

In kernel space we have a different story. It is preemptible, but not *fully* preemptible. In general, the kernel is not preemptible if it holds a spinlock (which means it is executing a critical section) or if preemption is explicitly disabled. Therefore, the kernel is subject to preemption when an interrupt handler exits, when it releases a lock and when preemption is explicitly re-enabled. All of these result in calls to the scheduler. We will look further into kernel preemption points later in this chapter.

4.1.3 Interrupts

Interrupts are signals asynchronously generated by external hardware that interrupt the regular processing flow of the CPU. Upon receiving an interrupt on its Interrupt Request Line (IRQ), the CPU will make a context switch to a predefined interrupt handler (also called an Interrupt Service Routine, or ISR for short). Interrupts can also be generated by software using a specific processor instruction, in which case they're called software interrupts. Upon completion of the interrupt handler, the system resumes execution of the previous state.

Linux separates the interrupt handler into two separate parts: the top half and the bottom half. The **top half** is the actual ISR. It does only the absolutely essential work (e.g. acknowledge the interrupt, copy available information into memory). The **bottom half** is the non-urgent part of the work that can be postponed for execution at a more convenient time. Note that because interrupts can arrive at any moment, they may interfere with potentially important and time-sensitive operations. By separating ISR's in two components, the interference is kept to a minimum.

Bottom halves have three distinct implementations: softirqs, tasklets and work queues.

Softirqs are statically defined bottom halves. They are registered only during kernel compilation and cannot be dynamically created by, say, a kernel module. Top halfs usually mark their corresponding softirq for execution (called *raising* a softirq) before exiting. At a more convenient time, the system checks for raised softirqs and runs them. Note that softirqs are not processes. The scheduler does not see them and they **cannot block**. This means that once a softirq begins execution it cannot be preempted for rescheduling. Softirqs only *yield* execution when they terminate. If the system is under load and many softirqs are raised, user space starvation can occur. When, then, should softirqs be run? Softirqs can run when the top half returns, they can be explicitly called from code in the kernel and by per-cpu **ksoftirq threads**. These threads are an attempt to curb the possible starvation due to softirq overload. When a softirq raises itself (which happens, for instance, in the network softirq) it will only be executed when the corresponding ksoftirq thread is scheduled. Because the thread is marked as low priority, the system won't starve *as much*. This is a very important point that will explain some of the performance results obtained later in this chapter, so it is worth pointing out again: due to their non-preemptive nature, **softirqs may induce user space starvation**.

Due to all of the above, softirq usage is very limited. In order of priority, the systems that use them in the kernel under test are: high priority tasklets, kernel timers, networking,

block devices, regular tasklets, scheduler, high resolution timers and RCU locking [3]. System strain can then be induced by overload of tasklets, timers and heavy networking and disk I/O operation (which is what our benchmarking suite does).

Tasklets are very similar to softirqs. The main difference is that they can be created dynamically. The implementation defines two softirqs associated with groups of high and low priority tasklets. When one of these softirqs is able to run, its group of associated tasklets also runs. And like softirqs, they cannot block.

Finally, **work queues** are implemented as kernel threads. They run with the same scheduling constraints of any other process under SCHED_OTHER. Since they are independent of softirqs, they suffer none of their problems (they can block, for instance), but also have none of their benefits.

4.1.4 Timers

Introduced in version 2.0, the classic Linux timing architecture is based upon the notion of a periodic tick. At boot, the system timer[2] is programmed to generate an interrupt according to the predefined **tick rate**. This rate is defined at compile time by a static preprocessor define called HZ. Typical values of HZ are 100, 250 and 1000 Hz. The timer interrupt handler would then be responsible for operations such as updating the system's uptime and wall time, updating resource and processor usage statistics, and run any kernel timers that might have expired.

Kernel timers are the basic kernel support for timing operations. They operate solely in one-shot mode. In fact, the timer does not have a period but an absolute expiration date measured in jiffies. jiffies is a 64 bit integer that counts the number of *ticks* since boot. Therefore, the maximum frequency achievable with these timers is equal to or a fraction of HZ. Precision wise, these timers offer no guarantees. The only guarantee is that the timer callback function will not execute *before* the expiration date. The delay can be as high as the next tick [3].

[2]The exact hardware varies between architectures. On x86 the programmable interrupt timer (PIT) and the Advanced Programmable Interrupt Controller (APIC) are the usual suspects. The processor's time stamp counter (TSC) can also be used to keep track of time.

4.2 Real-time Isn't Fair

All of Linux' mechanisms were developed with desktop and server usage in mind. This means high throughput and fair access to hardware resources for all processes. But, as we've previously seen, real-time isn't fair, and its requirements are usually in direct contradiction with other computing paradigms. As a general purpose OS, Linux excels. As a real-time OS, however, the situation is rather different.

Linux' scheduling of processes is temporally non-deterministic. It is highly dependent on system load and promotes fair access to processor time. By registering a process with the SCHED_FIFO policy we mostly bypass the later problem, but the fact that kernel space isn't fully preemptible keeps the first problem in place.

Within kernel space, the situation is better but still poor. Softirqs, spinlocks and other non-preemptable sections of the kernel are fundamentally destructive of real-time predictability. In addition to this, the timing architecture offers a deficient support for periodic operation and its architectural simplicity brings serious problems. Firstly, although the periodic tick conceptualization seems like a natural way for an operating system to keep track of time, it is highly inefficient. The system keeps *ticking* even when it is idle, which represents unnecessary power consumption. The tick rate itself is a tricky and inflexible trade-off since the whole infrastructure relies on it. Too low and the timing is very coarse, too high and the CPU might spend an unreasonable amount of time executing the timer interrupt handler. Secondly, the fact that timers operate not in time units but in jiffies makes precise timing operations unportable and unreliable. The *API* allows setting an expiration date 2ms from now, but on a system with HZ=100, the expiration date will be silently rounded off to 10ms.

Over the past 10 years, several solutions were developed that greatly improved Linux' real-time performance and overall predictability. The High Resolution Timers project redesigned the timing architecture and is now a standard of the mainline kernel. The PREEMPT_RT patchset tries to address the lack of preemptability of the kernel with an eye for reducing latency. Other projects such as RTAI and Xenomai completely bypass Linux and its problems. Instead, they introduce a micro kernel between Linux and the hardware that directly handles interrupts and scheduling [19][20]. Several other solutions exist, such as ARTiS which splits tasks between real-time and non-real-time CPUs [21]. We will dedicate a full chapter to the dual kernel approach. For now, let us look at the first two approaches.

4.3 High Resolution Timers

High resolution timers (*hrtimers*) were introduced in kernel version 2.6.16 [22]. They are part of an effort lead by Thomas Gleixner and Ingo Molnar to completely redesign the timing infrastructure [23]. Some of the objectives of this undertaking were:

- To engineer a new abstraction of time that will minimize platform specific code and, hence, maximize maintainability.
- An infrastructure that can support both ticked and tickless (or dynamic) operation
- A new timer infrastructure that supports high resolution timing, measured in nano seconds, and complements the legacy concepts but is entirely independent of them.

The project was an enormous success and brought a much needed refresh to the dated code, not to mention welcomed functionality and performance improvements. The fruits of this work also made it to user space in the form of itimers and an implementation of the POSIX 1003.1b real-time clock/timer specification [24]. These include a timer functionality and the nanosleep system call.

This new architecture is rather complex (due to all its platform-agnostic abstractions) and a complete description of it is out of the scope of this work. We present, however, some of the most important highlights.

The decision to introduce a new API for high resolution timing instead of upgrading the existing timing infrastructure to, say, a *sub-jiffie* granularity was the fruit of some careful and practical considerations. Two distinct use cases of timers were identified. *Timeouts* are used to detect a rare failure. They do not require high resolution, and are almost always deleted before expiring. Much like a *watchdog timer*, a timeout isn't supposed to go off, but it's there to react in case of a system failure. It's very commonly used in the networking stack for protocol timeouts, for instance. *Timers* are the opposite usecase. They're used to schedule events, run periodic functions, and other carefully timed scenarios. They require high resolution and usually expire.

This distinction is important because they call for *different* supporting implementations [25]. The classical architecture is called the Cascading Timer Wheel (CTW) and works phenomenally well for timeouts. It has low overhead for inserting and removing timers ($O(1)$) because it sorts timers based on their expiration date very coarsely. Every so often, however, the timer wheel has to be fully sorted, a very time consuming operation that

completely disrupts any ambition of precise timing. *hrtimers* keep timers sorted in a red-black tree[3]. It has higher overhead for inserting and removing timers ($O(log(n))$), but the sorting is already taken care of. In a nutshell, the CTW tries to push sorting timers as far off into the future as possible. The red-black tree sorts timers as soon as they arrive. The former lends itself well for timers that are usually removed before they expire, the latter is ideal for the opposite.

Why use both? Maintaining the legacy architecture keeps compatibility with older code and avoids an unnecessary rewrite of several core sections of the kernel. But besides compatibility, notice that the bigger the red-black tree, the slower the insertion and removal times. By separating the architecture into two components, both the CTW and the red-black tree are kept in a sizable form. The solution developed was to tie the CTW not to an interrupt issued by a system timer, but to an hrtimer [26]. This frees the hardware from CTW's hands, and makes way for an easier implementation of tickless operation and nanosecond timers. Which is exactly what was done by the same team for Linux 2.6.21 and remains valid until the time of writing.

4.3.1 Performance

To assert the performance of hrtimers (and all systems henceforth) a standard testsuite was developed. It was aptly named "The Testsuite" and its implementation can be reviewed in appendix A.

In a nutshell, the system under test is programmed to generate a square wave on the parallel port with period $2ms$. Then, the jitter of the wave is measured using an external microcontroller. The jitter is measured in four distinct situations. In **Idle**, the system is essentialy in stand-by, in **CPU** a video is played that occupies 100% of the CPU, in **I/O** a file is constantly moved between partitions, in **Net** a file is constantly moved between the computer running the test to another computer via ethernet. The result is subject to an uncertainty of $\pm 100ns$ and a systematic error of -1.3μs, which is immediately corrected.

4.3.2 Kernel Space

The testsuite was performed on an hrtimer running in kernel space. The code used is presented in listing C.1.

[3]A red-black tree is a type of self-balancing binary search tree. This means that the data it contains is continuously sorted.

Fig. 4.2 shows the results obtained. Fig. 4.2a plots the jitter distribution while 4.2b presents a statistical approach highlighting the worst-case jitter. Table 4.2c presents the precise values used to plot fig. 4.2b

The average jitter is about $-200ns$ on all cases. This means that the wave is in reality $200ns$ shorter than what it should be. Under Idle, I/O and Net the exact period is only measured 15-20% of times, evidenced by the peaks. Under CPU, that value drops bellow 5%.

Although fig. 4.2a shows jitter between -4 and 4 μs, the worst-case jitter is much higher. In fig. 4.2b we can see that it lies within the 100-200μs range. The lower jitter is naturally in Idle, the higher ones are I/O and Net. This is to be expected because both tests overload the system with softirqs.

As we've seen in section 4.1.3, hrtimers, I/O and the network interface all use softirqs. The hrtimer softirq, being of inferior priority than both I/O and Net experiences greater delays. The interrupt latency can double, as evidenced by the doubling in worst-case jitter.

In spite of this, we see that even though the system is overloaded, the hrtimer is never starved. Delay can double, but it is still serviced in a reasonable amount of time.

4.3.3 User Space

Recalling section 4.3, hrtimers are exposed to user space as an implementation of POSIX norm 1003.1b. This norm contains the timer that was used to generate the wave, the code for which can be reviewed in listing C.2. The program was scheduled with the SCHED_FIFO policy for maximum real-time performance.

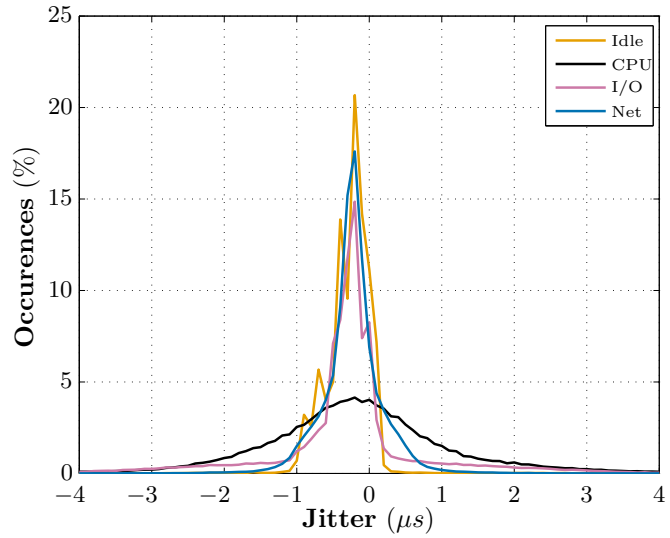
Fig. 4.3 shows the results obtained, similar in presentation to our previous results[4]

As can be immediately seen, although hrtimers are also being used, in user space the scenario is strikingly different. The fact that the timer handler is executed in process context makes all the difference.

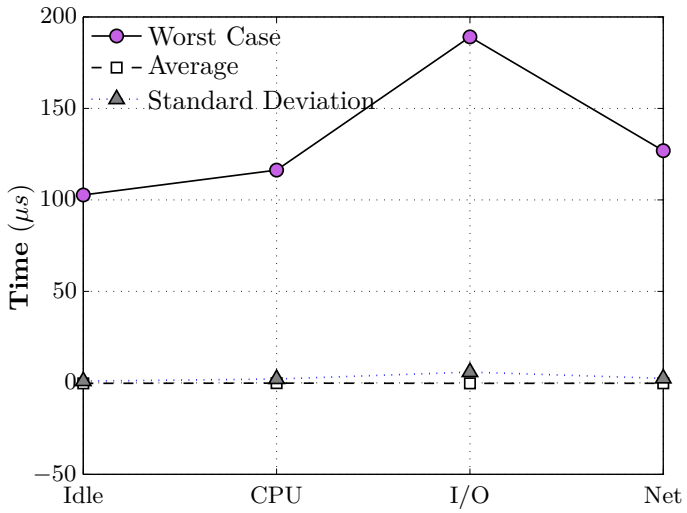
As we can see in fig.4.3a there is a small peak under the exact period, but jitter is mostly spread out between -1 and -1μs. Under CPU the graph goes almost flat.

In fig. 4.3b we can see that worst case is much higher than in our previous analysis. In Idle and CPU, the worst-case is held between 500/800μs. In I/O, the worst-case sky

[4]Unfortunately, the network load is not presented due to an irrecoverable glitch in the measurements. Due to time considerations, it was not possible to re-run the testsuite. Fortunately this is the only glitch in the +24 hours of testing undertaken for this dissertation and no such errors will be present beyond this point.



(a) Distribution.



(b) Statistical analysis.

Load	Worst Case	Average	Standard Deviation
Idle	102.7	-0.2821	0.9606
CPU	116.3	-0.1570	2.0857
I/O	189.1	-0.2659	5.8801
Net	126.9	-0.2271	2.4467

(c) Statistical analysis. All values in μs

Figure 4.2: Jitter analysis for hrtimers.

rockets to over $2ms$

Contrary to the kernel space case, user space timers starve in case of softirq overload. Recalling our discussion of operating system purposes in chapter 3, Linux, being general purpose, does not optimize for real time. Even though a timer is set, the fact that it resides in user space means it *must* be less important than system activity.

In practice, to toggle the parallel port bit the hrtimer handler needs to run, then the scheduler must be called and schedule the user-space handler. Our previous analysis covered the jitter in the first step in this chain. Under I/O and Net, the other two steps are constantly preempted by interrupts and delayed by the corresponding softirqs.

4.4 PREEMPT_RT

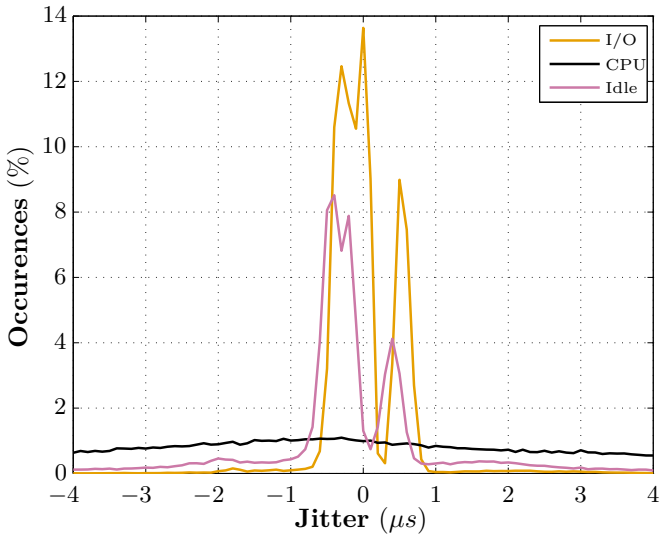
The PREEMPT_RT patchset is an on-going project lead by Ingo Molnar, Thomas Gleixner and Steven Rostedt [27]. Its main objective is to make the Linux kernel fully preemptable. A lot of minor changes are done to the source code, but the most important architectural changes are done to **spinlocks**, **semaphores** and **interrupt handlers** [28]. Let us now look at each of these changes individually.

4.4.1 Spinlocks and semaphores

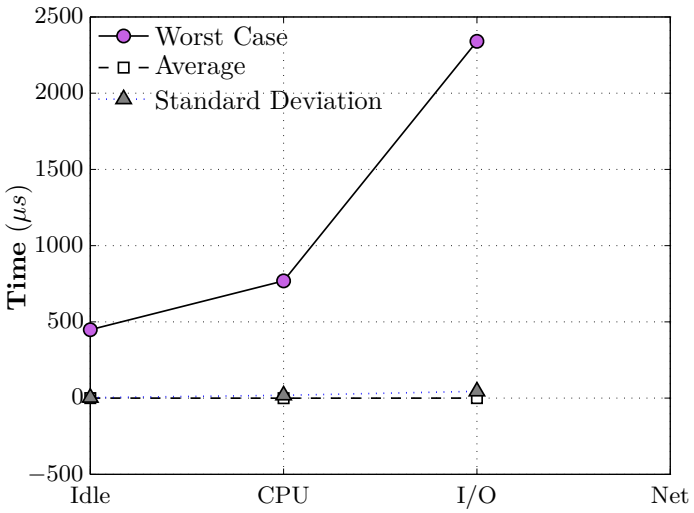
Spinlocks are a kernel mechanism to ensure mutual exclusion on a given critical section. They are called spinlocks because instead of sleeping they *spin*, i.e enter a busy wait cycle continuously testing the lock. This differentiates them from semaphores which sleep instead of spinning. The rationale for using one instead of the other is as follows. Semaphores are to be used when the wait period is supposed to be long. Spinlocks, on the other hand, are to be used when the wait period is short. In this case the cost of context switching is an unreasonable detractor to performance.

In practice, spinlocks disable kernel preemption while allowing hardware interrupts. This means that no other process will take its place in the CPU until the critical section is finished and the lock is released, but interrupts will still be service as they arrive.

PREEMPT_RT changes this behavior. The patch separates spinlocks in two types: spinlocks and atomic_spinlocks [29]. The first sleep while the later do not. Since all the kernel code uses the default spinlock type, the patch immediately makes all critical sections preemptable. The problem is that while in most cases this is perfectly acceptable, in some



(a) Distribution.



(b) Statistical analysis.

Load	Worst Case	Average	Standard Deviation
Idle	448.5	0.0000	2.9
CPU	768.8	-0.3	18.4
I/O	2340.1	0.1	44.4
Net			

(c) Statistical analysis. All values in μs

Figure 4.3: Jitter analysis for POSIX timers.

specific cases *it is not*. Some sections really do need to disable preemption, and these sections are altered to use atomic_spinlock instead of the default type. The few examples are the scheduler, PCI management and architecture specific code, among others [29, 28].

The fact that spinlocks are now preemptable makes them vulnerable to priority inversion and this affects real-time performance. Consider the following situation:

- Low priority task A gains the lock
- Medium priority task B preempts task A and executes
- High priority task C preempts task B and executes, but when it attempts to aqcuire the lock held by task A, it blocks

As we can see, a high priority task is being conditioned by a low priority one. This is why the scenario is called priority *inversion*. PREEMPT_RT solves this issue by making spinlocks and semaphores use priority inheritance. With this algorithm instead of task C waiting indefinitely for task A, task A's priority is temporally boosted until it executes long enough to release the lock. When the lock is released, the priority is reset and task C can resume execution.

4.4.2 Interrupt Handlers

As we've seen in previous sections, some core interrupt handlers are implemented as softirqs or tasklets and these are not preemptible. PREEMPT_RT forces all interrupt handling to be done in process context. The processes are registered with the SCHED_FIFO scheduling policy and priority 50. This includes both Top Halves and Bottom Halves, which are referred to as **Hard IRQs** and **Soft IRQs** [30]. To explain this, let's accompany what happens from the interrupt request to the actual execution of the corresponding handler.

When an interrupt arrives, the basic interrupt handler is called. From this handler a kernel thread is created for the corresponding interrupt line if it doesn't already exist. There can be only one Hard IRQ kernel thread per IRQ and they're called [IRQ-x], where x is the corresponding IRQ number. An interrupt can be flagged as to not run in process context, in which case instead of creating a kernel thread, the ISR will be run directly. The most notable example of this exception is the timer interrupt. When the Hard IRQ finishes, it can either return or instantiate a Soft IRQ if it has further work to do. All Soft

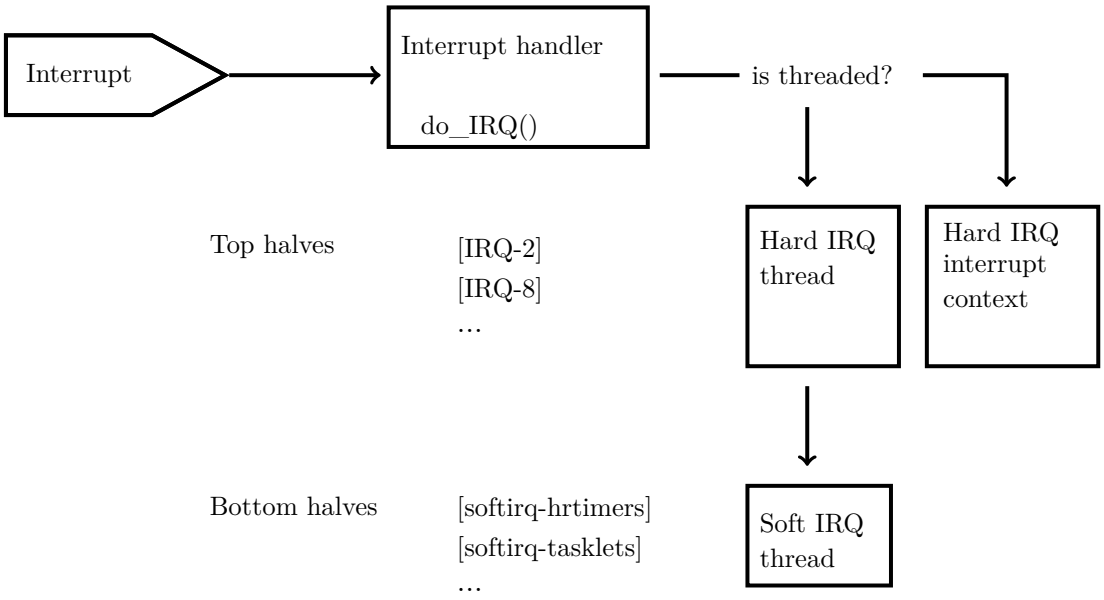


Figure 4.4: PREEMPT_RT implementation of interrupt handling.

IRQs are run as threads called [softirq-label]. By default they all have the same priority, but this can be manually set.

Fig. 4.4 illustrates this architecture and hopefully clarifies some of the unnecessary confusion of similar, but not quite equal, nomenclatures.

4.4.3 Usage

The PREEMPT_RT is delivered as a patch against the mainline kernel. While it provides some interface changes in terms of kernel development, it does not affect user space or hrtimers. A kernel binary image pre-patched with PREEMPT_RT is present in the repositories of all major Linux distribution (Ubuntu, Fedora, OpenSUSE, etc..) so installation can be as simple as installing the respective package.

4.4.4 Performance

To assert the performance of PREEMPT_RT, the same exact programs using hrtimers (C.1) and POSIX timer (C.2) were used to run the testsuite.

The difference in performance between the same programs running on an unpatched (or *vanilla*) kernel will reveal the effects of the patchset.

4.4.5 Kernel Space

Fig. 4.5 shows the full set of results obtained. In fig. 4.5a we can see that in the Idle and I/O loads the distribution is fairly tight around 0/-100ns, with peaks around the 25-40% band. In Net and CPU the distribution spreads and the peak drops bellows 10% in the first case, and bellow 2% in the second.

Looking at the worst-case distribution in fig. 4.5b we see a strange behavior. While on average the performance is better or approximately that of vanilla kernel, the worst-case behavior is *much worse*. Under I/O the worst-case jumps to over 600μs, while before it barely reached 200μs.

Although it also consists in interrupt overload, the Net sees worst-case at one sixth the value of I/O. This might be explained by deficiency in the disk controller driver, as all Soft IRQs have the same priority.

4.4.6 User Space

Fig. 4.6 presents the results for user space. The result is approximately the same. This is unsurprising as, recalling section 4.4.2, interrupt handlers are implemented as processes. This means that both user space programs and kernel space interrupts and scheduled by the Linux scheduler. In the jitter distribution in fig. 4.6a we see very similar wave forms, albeit with peaks about 10% lower. In the worst-case in fig. 4.6a, the results are very similar.

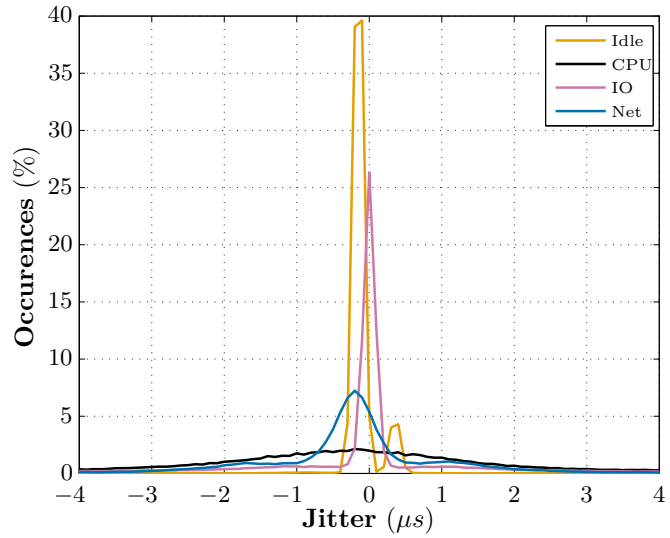
The reasons that made user space starvation an issue in the vanilla kernel are gone. By the same token, the scheduling and processing advantage of softirqs are also gone. Hence, the behavior is the same.

4.5 Conclusion

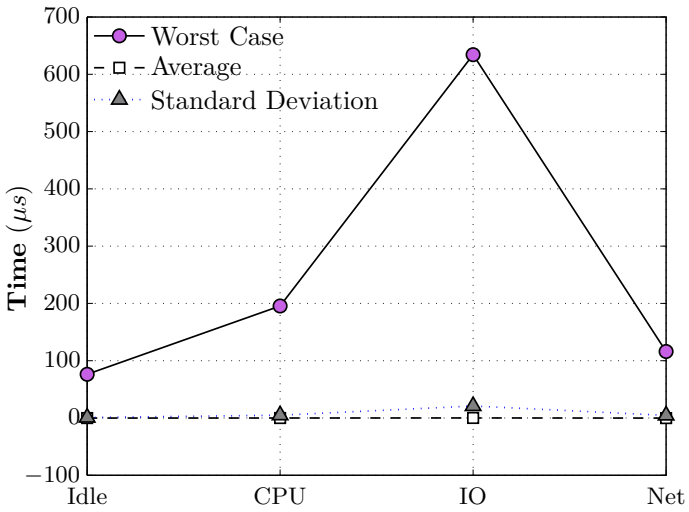
Fig. 4.7 condenses the performance results obtained in this chapter.

Because PREEMPT_RT implements softirqs as processes, they are scheduled like any other process, be it user space or not. This balances the performance and nullifies the kernel space advantage.

It is important to point out the following. In all our tests so far, the cause for higher worst case was always I/O. In PREEMPT_RT it is quite aberrant. If it weren't for this detail, PREEMPT_RT would have achieved the kernel space vanilla performance in both



(a) Distribution.

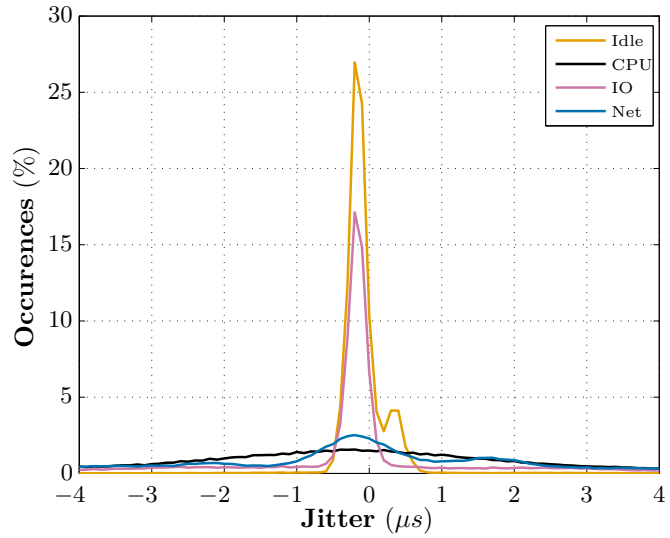


(b) Statistical analysis.

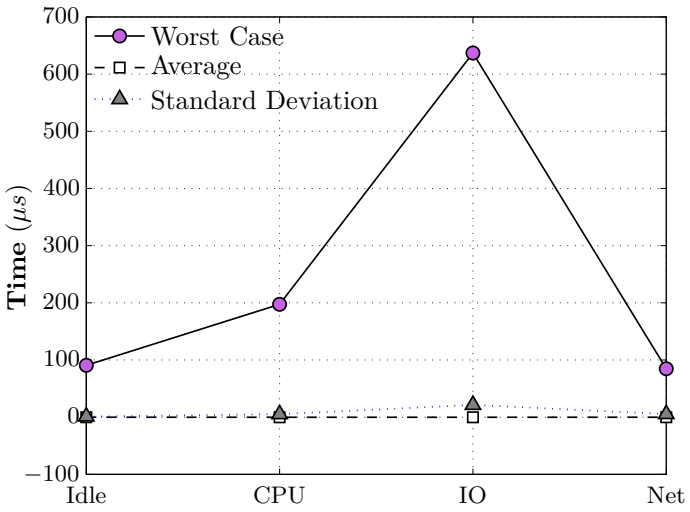
Load	Worst Case	Average	Standard Deviation
Idle	76.3	-0.1020	0.8328
CPU	195.5	-0.0320	4.8170
I/O	634.2	0.0544	20.9579
Net	116.1	-0.1785	4.4808

(c) Statistical analysis. All values in μs

Figure 4.5: Jitter analysis for hrtimers (PREEMPT_RT).



(a) Distribution.



(b) Statistical analysis.

Load	Worst Case	Average	Standard Deviation
Idle	90.7000	-0.0834	1.1671
CPU	197.4000	-0.1506	5.2077
I/O	636.9000	-0.1516	21.4575
Net	84.7000	-0.1015	5.4419

(c) Statistical analysis. All values in μs

Figure 4.6: Jitter analysis for POSIX timers.

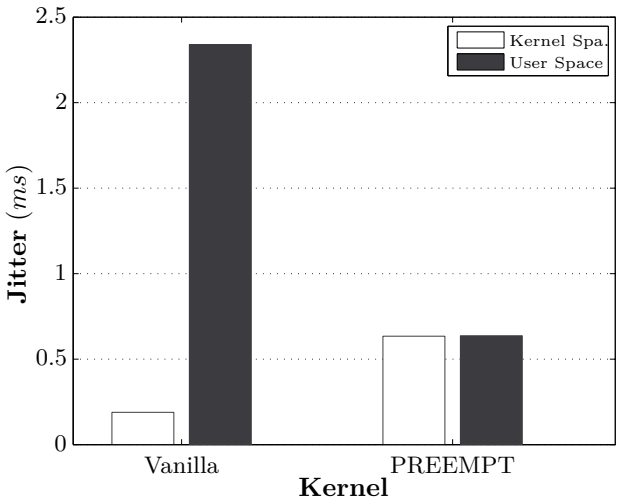


Figure 4.7: Worst-case jitter for vanilla and PREEMPT_RT kernels.

kernel and user space. Our contention is that PREEMPT_RT exposes some fault in the disk driver code. It might be the case that some of the sections aren't preemptable. That would fully explain the results obtained and point towards a way to fixing the issue.

Chapter 5

The Dual Kernel Approach

In this chapter the dual kernel approach to improving Linux' real-time performance is studied at depth. An historical introduction positions RTLinux, RTAI and Xenomai in relation to each other. The later two are presented at length and their performance is benchmarked.

5.1 A Brief History of Real-Time

We've seen how much of the work of the PREEMPT_RT team has improved Linux real-time performance, but that improvement is not enough for a great number of applications. The best performance today can be achieved by use of the **dual kernel approach**. It consists in impeding Linux of directly controlling the hardware. Instead, a micro or nano kernel manages the hardware and provides real-time scheduling and other specific real-time features to isolated real-time tasks. Today, a fair number of projects employ this principle but after almost 15 years of development a lot of misconceptions and obsolete ideas and projects are widespread. In this section we'll hopefully clear some of those misconceptions.

The first real successful attempt at providing hard real-time capabilities to Linux was the **RT Linux** project in 1997. It was developed by Michael Barabanov under the supervision of Victor Yodaiken in the New Mexico Institute of Mining and Technology as part of Michael's Masters in Computer Science [31]. Here, Michael implemented the novel idea of introducing a Hardware Abstraction Layer (HAL) called the RT-HAL (or RT-Executive) between Linux and the hardware. Using the HAL, RTLinux was able to escape the consequences of the overuse of cli and sti[1] for synchronization in the Linux kernel.

[1] cli and sti are x86 instructions. cli clears the interrupt flag, which disables interrupts at processor

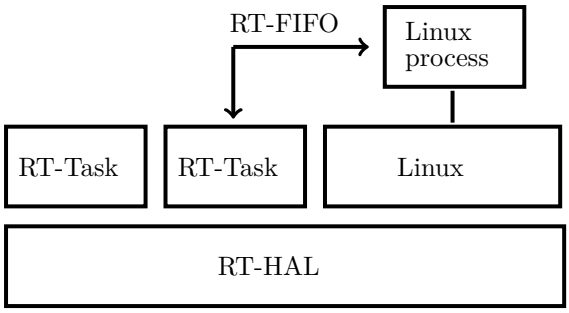


Figure 5.1: The original RTLinux architecture

The RT-Executive provided support for RT-Tasks scheduled by the RT-Scheduler. Inter-Process Communication (IPC) between RT-Tasks and Linux was assured by RT-FIFOs. Both RT-Tasks and the RT-Scheduler were implemented as kernel modules. Linux was also considered an RT-Task, but it was the *idle task*, the task that only runs when no others are available. The paradigm was that any RT application would be split in the RT component that ran in kernel space, and the non RT component that would live in user space and communicate with the RT task by an RT-FIFO. This is illustrated in fig. 5.1.

This work would lead to Victor's filing of U.S. Patent 5,995,745 later that year. This patents covers the notion of adding real-time support to a GPOS by means of a real-time micro-kernel that runs the GPOS as the idle task using interrupt emulation [18]. The patent would be accepted in 1999[2], the same year Victor would found FSMLabs, a company dedicated to the development and marketing of RTLinux. It split the OS into their own property version and a free version - RTLinux/Free.

Earlier that same year, in April 1999, the first version of **RTAI** is published. The Real-Time Application Interface (RTAI) was developped by the Dipartimento de Ingegneria Aerospaziale Politecnico di Milano (DIAPM) under supervision of Paolo Mantegazza [32]. The DIAPM was looking for ways to do complex control and ended up using the RT-HAL but with a different system on top, as RTLinux proved to be too simplistic for their needs. RTAI introduced a much richer interface with semaphores, shared memory, other IPC mechanisms and user space real-time support with the LXRT extension.

In 2001, Philipe Gerum begins work on Xenoadaptor, a development framework to facilitate migration from proprietary RTOS to Linux [33]. The concept was to mimic their

level, and sti sets the interrupt flag.

[2]Notice that since this is a software patent it is invalid in Europe and enforced for the most part only in the United States of America. The validity of such software patents is left as an intelectual exercise to the reader.

APIs so that migration to a free alternative involves as few code changes as possible. In 2002 the project is renamed to **Xenomai** and is released as an extension to RTAI [34].

During this period, a concern about RTLinux's patent encumbrance begins to materialize [35]. A disagreement on licensing generated a continuous dispute between RTLinux and RTAI [36, 37, 38]. In 2000 the seminal paper "Adaptive Domain Environment for Operating Systems" put forth a new mechanism for hardware sharing between kernels that would lend itself perfectly to enabling hard real-time capabilities to Linux [39, 40]. To decouple his project from any legal issues, in June 2002 Phillipe Gerum implements the proposed architecture as **ADEOS** and builds his Xenomai project on top of it [40, 34]. By March 2003 RTAI is also ported to ADEOS and completely abandons RT-HAL [41].

In 2004 Xenomai is renamed **Fusion** and becomes an integral part of RTAI [42]. In 2005 Xenomai abandons RTAI an builds its own implementation of the real-time co-kernel [34].

This brings us to the present day.

RTLinux has been bought by WindRiver[3] and is now sold as Wind River Real-Time Core. RTLinux-free is still available under the same conditions and has become mostly irrelevant.

RTAI is still maintained and developed but has become mostly stagnant, with few releases since 2008. Xenomai is still actively developed, with multiple releases per year. We shall now look at these two in depth.

5.2 Xenomai

Xenomai has used the same stable architecture since it abandoned the RTAI project in 2005. The architecture is designed to be easily portable to other processor architectures. Fig. 5.2 illustrates the architecture.

ADEOS sits on top of the hardware. It intercepts synchronous events such as traps and exceptions, and asynchronous events such as interrupts. ADEOS then establishes an event pipeline (I-PIPE) that propagates the events to each of its domains in order of priority. The Xenomai co-kernel is the primary domain. It is implemented as a kernel module that, when loaded, registers itself with ADEOS as the domain with highest priority. The Linux kernel is left as the secondary domain. This means that Xenomai has a chance to handle the events *before* Linux. Not only that, but ADEOS stops Linux from disabling interrupts

[3]Windriver is the developer of VxWorks, a proprietary RTOS industry standard.

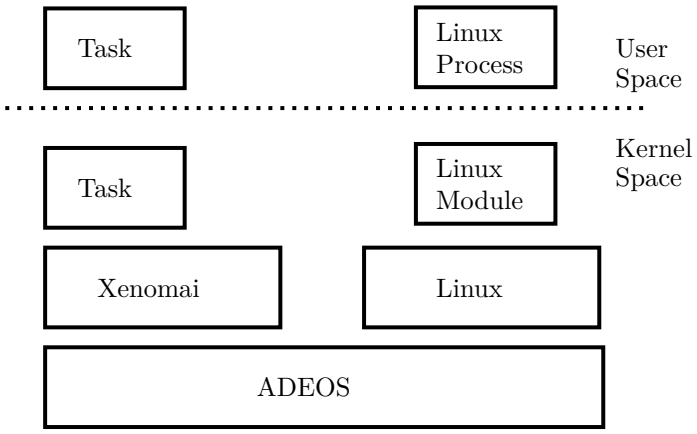


Figure 5.2: The Xenomai architecture

directly with the CPU. This alone means with almost no cost, Linux is fully preemptable by the real-time tasks. It is both these facts that give Xenomai the stringent real-time capabilities.

The Xenomai co-kernel has been designed to ensure portability and extensibility. Like the original Xenoadaptor project, Xenomai emulates various real-time APIs such as POSIX threads and proprietary RTOS like VxWorks. To support this, Xenomai implements a *real-time nucleus* that provides the common primitives with which the different APIs can be implemented. These different APIs are called *skins*. Xenomai has its own skin, called the *native skin*, that tries to implement a saner and easier to use real-time API.

As can be seen in fig. 5.3, the nucleus does not run directly on top of ADEOS. Instead a Hardware and a Systems Abstraction Layer (HAL/SAL) decouples the nucleus and higher levels from any machine dependent code [11]. This means that the nucleus can run unchanged on any machine supported by ADEOS/HAL/SAL, but it also means that the nucleus can be built on top of a *simulator*. An event driven simulator is used to run Xenomai instead of bare metal to easily develop and debug new skins and features.

By keeping every component of the system neatly encapsulated, Xenomai becomes easier to develop and maintain. It also becomes easier to port to other process architectures, evidenced by the high number of them already supported [20].

5.2.1 Features

Xenomai packs a rich feature-set and a reasonably well documented API. The native skin leverages all the capabilities of the underlying real-time nucleus. These are [43].

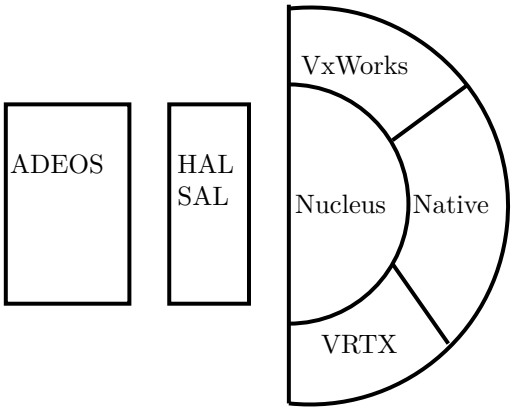


Figure 5.3: Xenomai Nucleus

- Real-time task management
- Timing services
- IPC between real-time tasks and regular Linux tasks with message queues, pipes and memory heaps
- Synchronization between tasks using semaphores, mutexes, condition variables and event flag groups
- Device I/O Handling

This API is symmetric between user space and kernel space. This means that the functions and data structures associated with the API are virtually the same in both contexts. Because the difference in performance between user space and kernel space is minimal (as we will see), the Xenomai team does not recommend using the API from kernel space. Instead, Xenomai provides a specific API to develop device drivers called the Real-Time Driver Model (RTDM). The idea is that kernel space should be reserved for real-time device drivers, while user space should be reserved to regular real-time tasks.

Xenomai has a strong focus on embedded platforms and is available for x86, PowerPC, ARM, Blackfin and Nios II.

5.2.2 Usage

Xenomai is delivered as a set of libraries and a complete Linux patch which includes ADEOS and the Xenomai co-kernel. Documentation of the installation procedure is avail-

able at the official website [20], but a more focused guide was produced for this dissertation and can be reviewed in appendix D.

5.2.3 Performance

To test Xenomai's performance, two applications were programmed using the native skin to produce the $2ms$ square wave on the parallel port. Both applications operate in much the same way, except one is a kernel module while the other is a user space application. The former can be review in listing C.3 while the later in listing C.4.

5.2.4 Kernel Space

Fig. 5.4 shows the results obtained for Xenomai kernel space.

The jitter distribution in fig. 5.4a stands rather precisely around the 0/-100ns gap in all cases. The graph starts with a high 50% peak in Idle, and gradually the peak drops for I/O, Net and CPU, where it drops bellow 5%. Looking at the statistical analysis in fig. 5.4, we can see a countinuous increase in worst-case jitter from Idle to Net. The overall worst-case lays between the 12.4/23.1 μs interval.

5.2.5 User Space

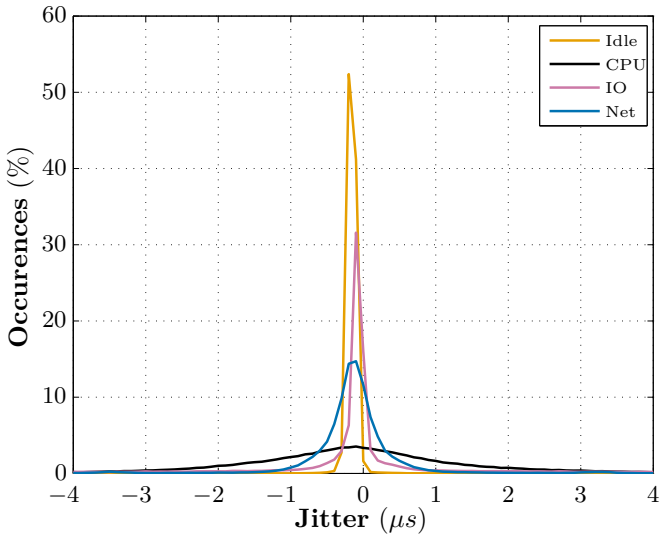
Fig. 5.5 presents the same analysis for Xenomai user space.

We can see that the behavior is mostly the same. The distribution of jitter and its analysis follows the same overall form of the results obtained for kernel space, with one small caveat.

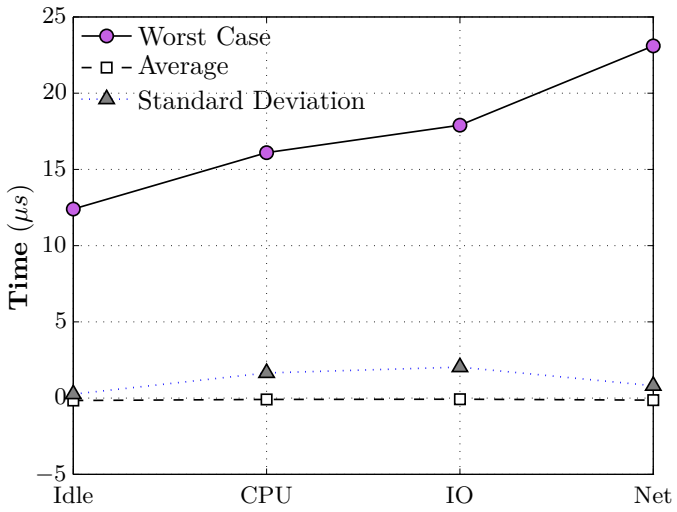
In Fig. 5.5b we can see that although the shape is very similar, it has a vertical offset. There is an increase in worst-case jitter of approximately 10μs on all conditions. Although in terms of determinism the situation is very similar, in absolute time user space pays a price. In Idle, worst-case jumps from 12.4 to 21.8μs, almost twice the amount of jitter.

5.3 RTAI

Upon the change to ADEOS, RTAI used the same architecture as Xenomai uses today. It would eventually change, however, with the objective of pushing real-time performance even further. As can be seen in fig. 5.6, RTAI does not use ADEOS to intercept interrupts.



(a) Distribution.

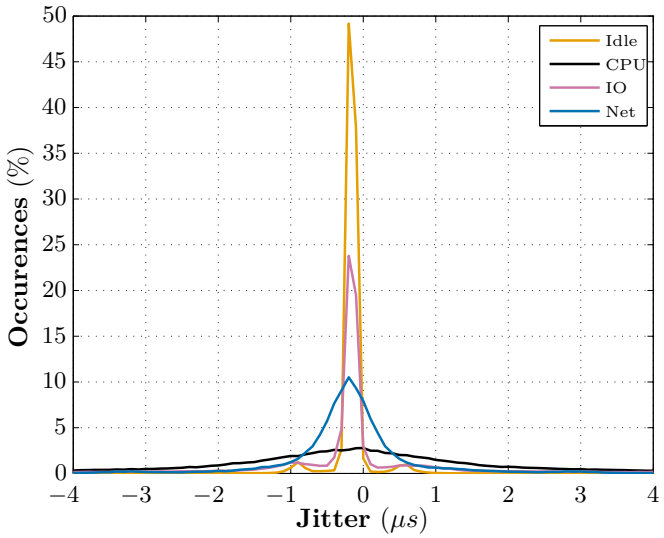


(b) Statistical analysis.

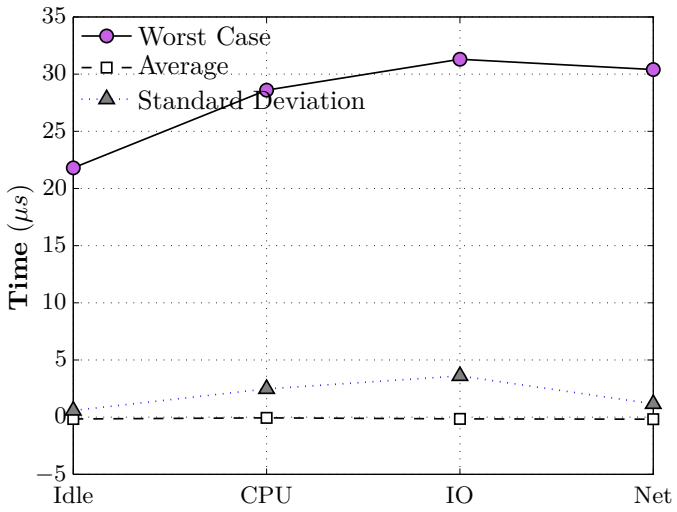
Load	Worst Case	Average	Standard Deviation
Idle	12.4	-0.1565	0.2593
CPU	16.1	-0.0875	1.6494
I/O	17.9	-0.0743	2.0319
Net	23.1	-0.1320	0.8129

(c) Statistical analysis. All values in μs

Figure 5.4: Jitter analysis for Xenomai (Kernel Space)



(a) Distribution.



(b) Statistical analysis.

Load	Worst Case	Average	Standard Deviation
Idle	21.8	-0.1590	0.5830
CPU	28.6	-0.0627	2.4738
I/O	31.3	-0.1510	3.6059
Net	30.4	-0.1766	1.1612

(c) Statistical analysis. All values in μs

Figure 5.5: Jitter analysis for Xenomai (User Space)

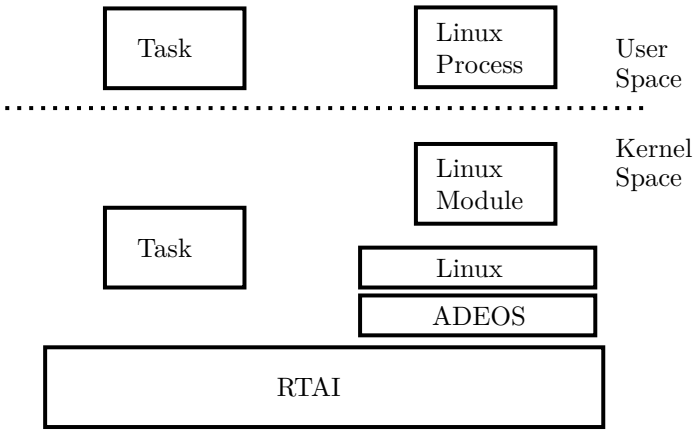


Figure 5.6: The RTAI architecture

Instead, RTAI intercepts interrupts directly and uses ADEOS to propagate these to Linux [44]. This reduces interrupt latency to a bare minimum since the "middleman" has been cut out - if an interrupt arrives that needs servicing by a real-time ISR, it will be serviced immediately. The I-PIPE is only used when no real-time ISRs exist.

This minor difference carries a significant trade-off. By interacting directly with the hardware, RTAI bypasses the ADEOS overhead and achieves the minimum possible latency. At the same time, RTAI becomes architecture dependent and more difficult to maintain.

5.3.1 Features

RTAI provides the full feature-set one would expect from an RTOS encapsulated into several kernel modules. These modules are to be loaded as needed, and then removed when the application that uses them terminates. The most important features are [45]:

- Real-time task management
- Timing services
- IPC between real-time tasks and regular linux tasks using mailboxes, FIFOs, shared memory, semaphores, RPCs and POSIX mutexes, conditional variables and message queues.
- Synchronization between tasks using semaphores, event flags, signals and tasklets.

RTAI supports real-time scheduling in user space by using the LXRT kernel module. The API strives to be as similar as possible between kernel space and user space. Contrary to

Xenomai, kernel space is still considered the way to go if extra performance is required. As far as API support is concerned, RTAI supports its own API and some POSIX extensions. It is available for x86, PowerPC, ARM and m68k architectures [19].

5.3.2 Usage

RTAI is delivered as a set of libraries and a kernel patch. Documentation for installation is sparse and difficult to grasp. The major difficulty is in patching the kernel. The kernel needs to be configured in a specific way for each machine. This is a major problem as a configuration for one machine will generally not work in another one. A guide containing a step by step procedure and a collection of tips and tricks for kernel configuration is available in appendix E.

5.3.3 Performance

Similarly to the treatment given to Xenomai, a kernel and user space incarnation of the parallel port wave generator was programmed. The code is displayed in listings C.5 and C.6.

5.3.4 Kernel Space

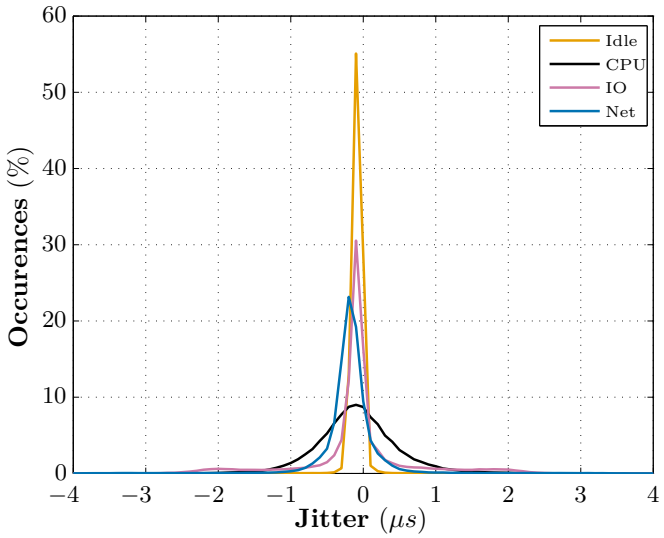
Fig. 5.7 illustrates our usual analysis.

The distribution in fig.5.7a is very similar to Xenomai's. Since their architectures are so similar, this was to be expected.

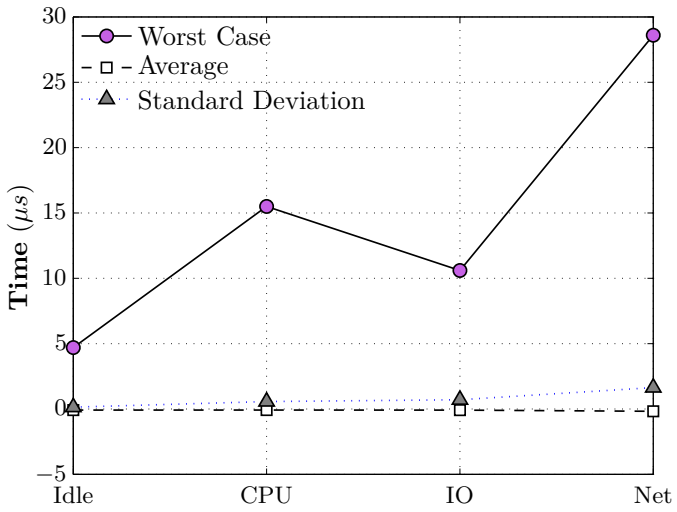
In fig. 5.7b we can see where the similarities end and RTAI's attention to performance pays off. In Idle, the worst case jitter is barely $5\mu s$. It is, however, more inconsistent as in Net the worst-case jumps to $30\mu s$.

5.3.5 User Space

Fig. 5.9 shows the equivalent analysis for user space. As expected, the distribution remains unchanged. The worst-case jitter shows, like in Xenomai, a small penalty. The offset in this case is about 5 μs, half of what we've seen for Xenomai.



(a) Distribution.

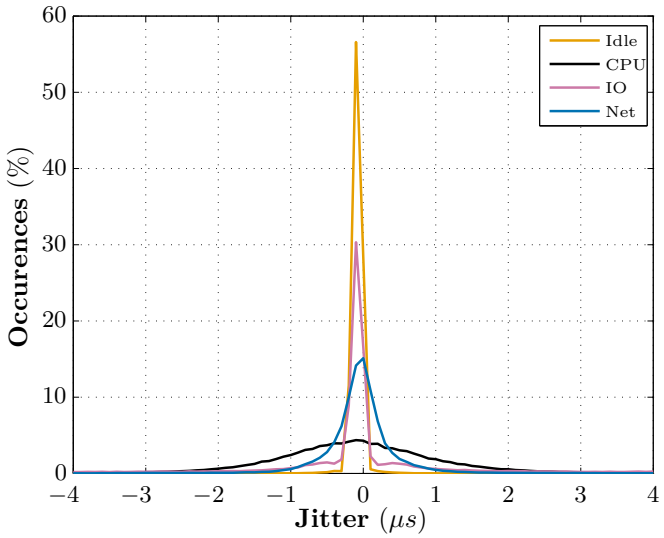


(b) Statistical analysis.

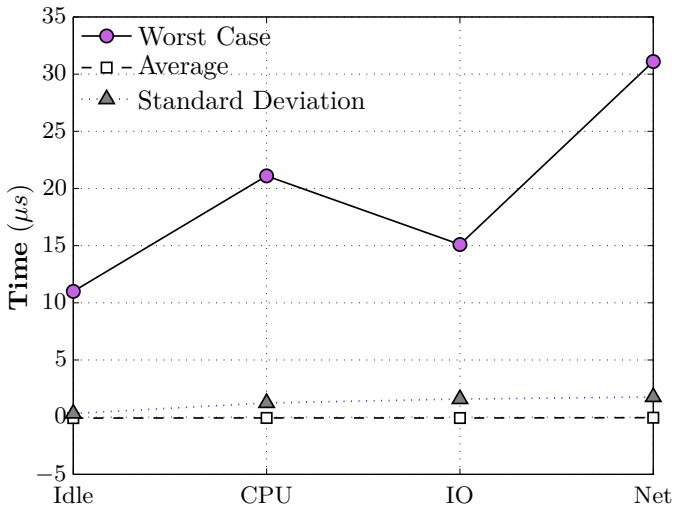
Load	Worst Case	Average	Standard Deviation
Idle	4.7	-0.0830	0.1403
CPU	15.5	-0.0801	0.5689
I/O	10.6	-0.0856	0.7084
Net	28.6	-0.1767	1.6397

(c) Statistical analysis. All values in μs

Figure 5.7: Jitter analysis for RTAI (Kernel Space)



(a) Distribution.



(b) Statistical analysis.

Load	Worst Case	Average	Standard Deviation
Idle	11.0	-0.0818	0.3203
CPU	21.1	-0.0666	1.2435
I/O	15.1	-0.0750	1.5865
Net	31.1	-0.0453	1.7615

(c) Statistical analysis. All values in μs

Figure 5.8: Jitter analysis for RTAI (User Space)

5.4 Conclusion

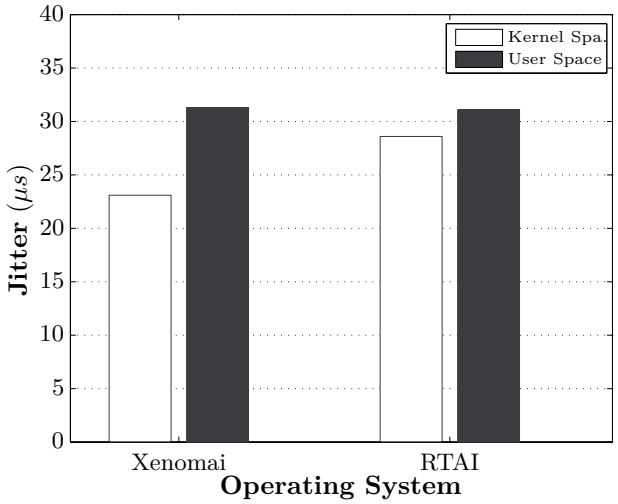
Fig. 5.9a presents a comparison between worst-case jitter performance between Xenomai and RTAI. The results are somewhat surprising.

Although in our tests Xenomai kernel space sees a distinct advantage, the safer assumption is that it is merely a statistical oddity. If the testsuite were allowed to run longer, the conclusion would most probably be that the worst-case jitter was about $30\mu s$ for both operating system in both execution spaces.

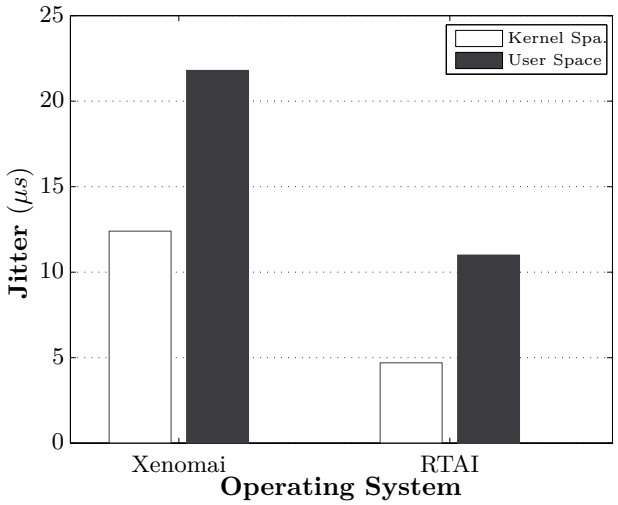
This $30\mu s$ mark is the worst-case in all our tests during a Net load. It never changes, unlike all the other cases that are slightly worse in user-space than in kernel-space. This is especially obvious in RTAI kernel space, where the Net worst-case is six times higher than when idle, and triple the value under I/O.

Since this $30\mu s$ wall is independent of the operating system and the space, it can be attributed to either a common flaw in both designs or to the network card itself and the way it interacts with the motherboard. The first option is possible since both architectures are indeed very similar. However, we should notice how the difference in receiving an I/O interrupt or a Net interrupt should be none whatsoever. In both cases the OS simply runs an ISR where the decision is to defer execution to Linux or not, and in Linux whatever handling happens is preemptable by definition. Because of this, it is more likely that the $30\mu s$ wall is due to the network card.

To assert this possibility, the testsuite should be run in a similar desktop computer but with a different motherboard and network card. Nevertheless, fig. 5.9b shows the worst-case jitter for the Idle load, where this possible network interference does not exist. The results make much more sense, as the tendency there illustrated is valid for both the CPU and I/O loads.



(a) Worst-case jitter for RTAI and Xenomai.



(b) Worst-case jitter for RTAI and Xenomai in Idle.

Figure 5.9: A comparisson of worst case jitter for Xenomai and RTAI.

Chapter 6

Conclusion

In this chapter conclusions encompassing the whole of Part II are presented. The most important benchmarking results are compiled into a Linux real-time scale – an ordered list of worst-performing to best-performing system. As closure, the results obtained thus far are compared to other work done in the field.

6.1 Results

In this second part of the dissertation, a total of 32 different performance tests were performed with the help of our testsuite. With all this data in hand, we are able to built the Linux real-time scale presented in fig. 6.1.

On the x axis we have the target's initials. On the right a U means "User Space" while K means "Kernel Space". On the left, V stands for "Vanilla", P for "PREEMPT_RT", X for "Xenomai" and R for "RTAI".

From this scale, we can draw some immediate conclusions. Focusing on regular Linux first, we can see that Linux user space offers the worst performance. The best performance can be obtained with vanilla kernel space, although on an Idle computer PREEMPT_RT would offer equal performance on user space. The large gap between Idle and worst overall in PREEMPT_RT might be explained by a deficiency in the disk driver, since it is high I/O that's pushing the worst-case so high.

Be that as it may, regular Linux can not bring worst-case bellow the 100s of micro seconds. With the dual kernel approach we can lower this value to the tens of micro seconds – an order of magnitude lower.

The best in class for this category is RTAI running in kernel space. 5 μs was the lowest

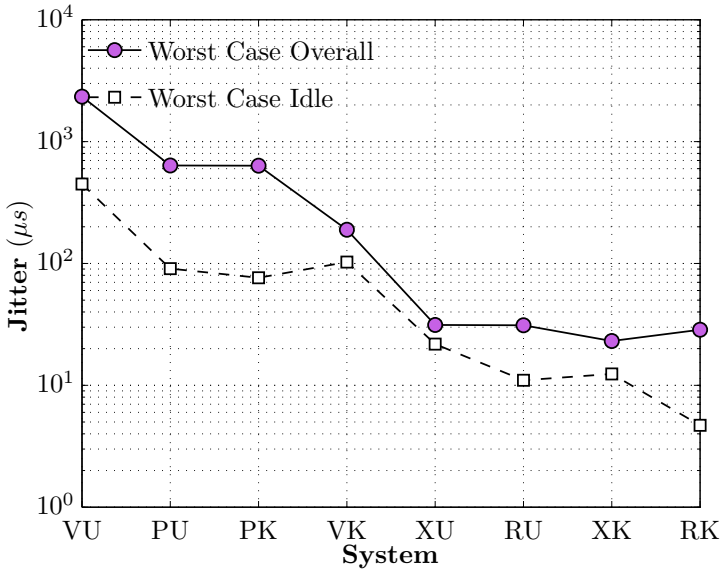


Figure 6.1: The Linux Real-Time Scale

jitter measured in our tests. Xenomai user space was always worst-performing, although, as we can infer from the vertical distance between the worst case overall and idle, it was the most deterministic, hovering around $30\mu s$ in all the scenarios.

The worst-case for all implementations of the dual kernel approach were all around $30\mu s$. This is a strange result that might be explained by the network card strangling the motherboard since it is independent of all our variables, OS and space.

6.2 Conclusion

As we've mentioned in the first chapter, part of our objective was to confirm and build upon other results in this field. Towards that goal, we will review some key papers of the last few years regarding Linux performance.

In 2007, RTAI 3.5 and Xenomai 2.3.3 had their scheduling jitter's measured and compared in [46]. A desktop computer with an Intel 1GHz processor was used and results were measured with a RTDAC4/PCI card. As a load, the author used concurring real-time tasks. Without this load, the worst case jitter was around 40 μs for RTAI and about $50\mu s$ for Xenomai. The values obtained by our testing fall within the same order of magnitude, yet show a welcomed increase in performance.

In 2008 a comparison between RTAI, Xenomai, Linux and the proprietary RTOS VxWorks was performed in [44]. The OSs were installed on single board Motorola board with a PowerPC CPU and the resulting interrupt latency was measured. As we opted for measuring scheduling jitter instead of interrupt latency, our results are unrelated.

In 2009 the difference in scheduling jitter between vanilla and PREEMP_RT was tested in [47]. Linux version 2.6.25 was installed on a machine with a 2.66GHz Core2Duo E6750. A user space task was made periodic by the use of POSIX timers and CPU and I/O system loads were prepared. The results are quite interesting. The jitter was measured by calling gettimeofday() and storing the task activation time. Vanilla kernel jitter stood between 20 μs and 18 ms under I/O, while PREEMPT_RT stood between 70 μs and 300μs. The 2.6.32 Linux version used for our tests has greatly improved native real-time performance. For PREEMPT_RT we can induce that our values are similar to those obtained, as those were merely the scheduling jitter, and ours had the additional jitter of producing a signal on the parallel port.

In 2010 RTAI's scheduling jitter was analyzed in [48] on a Pentium 4 machine similar to ours. Code was auto-generated by Scicos and hence ran in user space. Results were obtained by applying varying I/O load using data acquisition boards. RTAI responded with worst-case jitter from 30 μs on minimum I/O Load to about 400μs. This exposed a sensitivity to the over use of the acquisition board. Our I/O load would be comparable to the minimum I/O load used by that team, so our values are well in agreement.

Although the Linux real-time scale we present in fig. 6.1 is, by no means, written in stone, the overall picture it paints is in line with previous research on the matter. PREEMPT_RT improves upon vanilla kernel and obtains jitter values on the hundreds of μs range. Xenomai and RTAI bring this value down to the tens of μs, and RTAI usually outperforms Xenomai.

Part III

Xenomai Lab

Chapter 7

Introduction

In this chapter, the conceptual basis for Xenomai Lab is discussed. The reasoning behind its architecture as well as the main technological choices that steered its development are presented.

7.1 Keep it Simple, Stupid

Xenomai Lab is a fresh approach to designing control systems in Linux. It allows the user to graphically design a block diagram of a control system and execute it in real-time. It also provides an open application programming interface (API) to easily program new blocks. Control algorithms are programmed in straight C, with library support for matrices. Any engineer will feel right at home.

The platform was designed with the intent of decoupling control algorithms from RTOS concepts, while, above all, keeping it simple. A simple system in the KISS sense is simple to use and simple in the implementation. This makes it easy to debug and expand with new features. Xenomai Lab is distinctive because it is a pure Linux affair - it uses native widgets for the interface and plays well with package management. In Debian systems such as Ubuntu, installation is as easy as a double click[1]. These things matter because usability matters. An application that is easy to install is much more likely to be tried than one that puts up an initial barrier.

Having presented the application, let us now try to answer the most important questions regarding motivation and design decisions.

[1] At this moment, only with Xenomai already installed.

7.2 Why build something new ?

Digital control systems have been designed for more than half a century. The longevity of the area inevitably means that it has industry standards and established workflows. Designing a control systems usually consists in the following steps:

1. A mathematical model of the plant is deduced, and a controller is projected that achieves the desired performance metrics. The system is tested via numerical simulation, which means the results are merely theoretical approximations. This modeling is normally done in proprietary industry standards such as MATLAB/Simulink or LabVIEW.

2. Communication with the plant is built and tested. This means setting up the proper sensors, actuators, A/D and D/A converters. Usually an acquisition board is used. Some custom electronics may need to be designed in this process to filter the signals coming from the sensors to maximize efficiency in the data acquisition.

3. The theoretical controller is tested with the real plant instead of its mathematical model. As the plant model is merely an approximation, inevitably the designer needs to adapt his controller and/or hardware to make it work as intended.

The controller can be implemented in a desktop computer, in an embedded system or in a microcontroller. The control system can be monolithic, distributed, network controlled, or any other. Whatever the case may be, the workflow remains essentially unchanged.

To develop a system using this workflow in Linux, one has but a few options. The more mature one would be the RTAI-Lab/SciLab/SciCos combo. **SciLab** is a software package similar to MATLAB. It provides a high-level language similar to MATLAB's and **SciCos** is akin to Simulink. SciLab sits in a software uncanny valley where everything is vaguely familiar but *not quite*. One might argue that in engineering functionality, and not concepts such as usability or user friendly, is the most important thing. In the opinion of the team behind this work, the low adoption rates of SciLab/SciCos speak for themselves on this matter. **RTAI-Lab** implements code generation from SciCos to RTAI executables and has a graphical application with scopes and other such graphical utilities to monitor execution.

An engineer's time is worth money. Although a software package like RTAI-Lab and SciLab/SciCos can provide almost the same functionality by no price, they cost time. Time

to install, learn how to use, test and debug. This can easily incur in higher costs then using industrial grade tools. The fact that proprietary software has thorough documentation and is widely used overshadows the price. RTAI-Lab was last released in 2006 and its documentation remains unchanged since 2008.

The other option would be to continue modeling using whatever software is used, but then implement the control system using Xenomai or RTAI's API. In reality, this is no option at all. Block diagrams provide function decomposition and expose a system's inner signals. By programming a monolithic program, the internals are not exposed and decomposition is lost.

Xenomai Lab tries to be the RTAI-Lab/SciCos combo for Xenomai, but done in a different way, with none of its problems. Application related coding is done directly in C instead of another language. This means that no new language has to be learned, as all engineers inevitably know C. The fact that the application-related code is not auto-generated means that it's faster and easier to debug. The interface is simplified and is specific to the application. It does less things which means it is *focused*. There is almost nothing between the user and the control algorithms.

7.3 Why Xenomai ?

Both RTAI and Xenomai can deliver real-time performance suitable for control applications. However, a choice had to be made between the two, and it is our objective in this section to explain why we chose Xenomai.

RTAI is the obvious choice for applications requiring maximum performance. The $5\mu s$ worst-case jitter obtained for a kernel space module in section 5.3.4 is an impressive figure of merit. It is the opinion of the team behind this work is that the biggest issue with RTAI is that it is difficult to use. Granted, these are relative notions. For any advanced Linux user there is nothing in RTAI that is particularly out of the ordinary. Difficulty is necessarily in the "eye of the beholder". Be that as it may, the knowledge required to use RTAI without major issues is very specific. Software has to be manually compiled and installed through the command line, and not only that, real-time applications are not executed directly by a call to the binary. The RTAI paradigm is that a loader script loads the necessary kernel support before the binary is executed, and unloads them after the program completed.

Xenomai, on the other hand, strives to be flexible and easy to use. What Xenomai does

not have in performance, it compensates in usability. This is much more important for our purposes. If a user's main concern is maximum achievable performance, then Xenomai Lab isn't a good option to begin with because it increases execution latency and uses much more resources than programming a monolithic C program. Of course, a monolithic C program requires much more work than the graphical diagram approach. Organizing a system in blocks lends itself naturally to code reuse, which at the end of the day means greater functionality with less coding.

Xenomai's installation procedure has been adapted for Debian systems. Rather then running a typical old-school make install command, Xenomai can be compiled into a Debian package. This makes Xenomai integrate extremely well with the system. The usual channels for installing and uninstalling can be used. In fact, the Xenomai libraries can be directly installed from the Ubuntu Software Center as they are present in the repositories. A kernel must still be manually compiled, however.

Since we intend to develop our application in Ubuntu, which is a Debian system, Xenomai is a much more attractive solution.

In addition to this, Xenomai 3 (latest version at the time of writing is 2.6) is just around the corner and will bring an unbeatable feature - the implementation of the Xenomai API over a PREEMPT_RT kernel. This nullifies the difficulty in installation and will make Xenomai Lab installation truly a double-click affair. Of course, the dual kernel approach will still be available for those who need its performance.

7.4 Why Qt ?

Qt is a cross-platform C++ application framework that runs on Linux, Mac OS and Windows, among other platforms [49]. Qt is one of the major toolkits for building Linux applications. The other major toolkit is GTK, which is an integral part of the GNU project. GTK is the toolkit used by the GNOME desktop, while Qt is the toolkit behind KDE, the other major Linux desktop.

Xenomai Lab was programmed in Qt for various reasons. It integrates better with the GNOME and KDE desktops than the other way around, it natively uses C++ rather than plain C which is a major advantage when programming graphically, but above all, it has an Integrated Development Environment (IDE). Qt Creator provides in a single application the ability to code, build, debug, control versions, consult documentation, and every other feature one would expect from a modern IDE. No such application exists for

GTK development. The only IDE for programming GTK is the Mono framework, which is a reimplementation of the Microsoft .NET and C# technology.

There were other options, such as Java, but Qt provides the ideal balance between performance and ease of use for our purposes.

Chapter 8

Xenomai Lab

In this chapter a thorough analysis of Xenomai Lab's usage and implementation is presented, as well as experimental results of both simulation and real-world interaction. By the end of this chapter, the reader will be able to easily program his own blocks or even dwell into hacking new functions into the lab.

8.1 Head First

The easiest way to understand what Xenomai Lab (XL) does is to see it in action.

Fig. 8.1 shows XL's main window. It consists of a big white area, the **canvas**, where block diagrams can be drawn. To the left is the **block list**, where two set of blocks can be used: real-time and non real-time. Across the top rest three **toolbars** for interaction with the block diagram.

Blocks can be chosen from the block list and dropped on the canvas. Lines can then be drawn between blocks. Blocks can accept any number of inputs. They can also output a result to any number of blocks.

As an example, let us consider we intend to project a controller for a plant defined by the following transfer function:

$$G(s) = \frac{100}{s + 100} \tag{8.1}$$

For a sampling period $h = 1ms$, the transfer function becomes

$$H(q) = \frac{0.0952}{q - 0.9048} \tag{8.2}$$

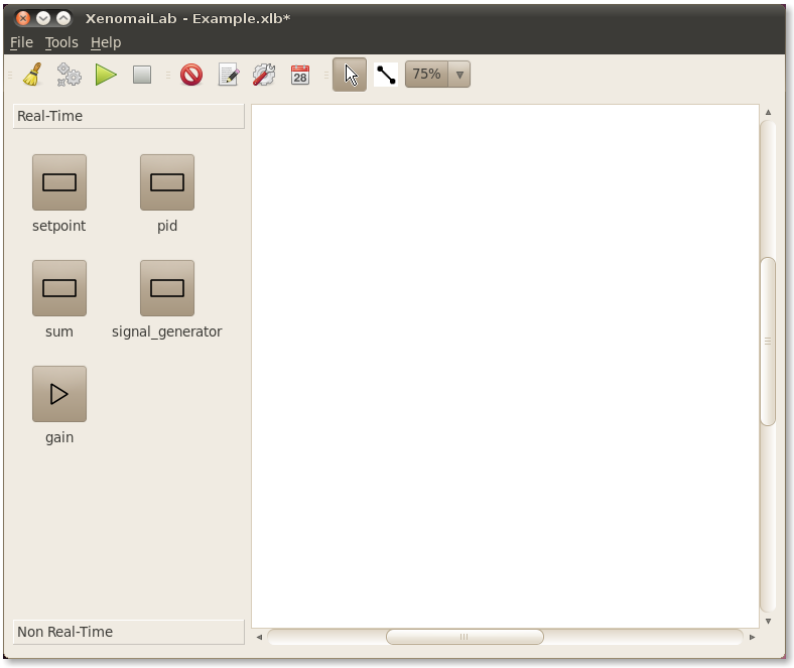


Figure 8.1: Xenomai Lab

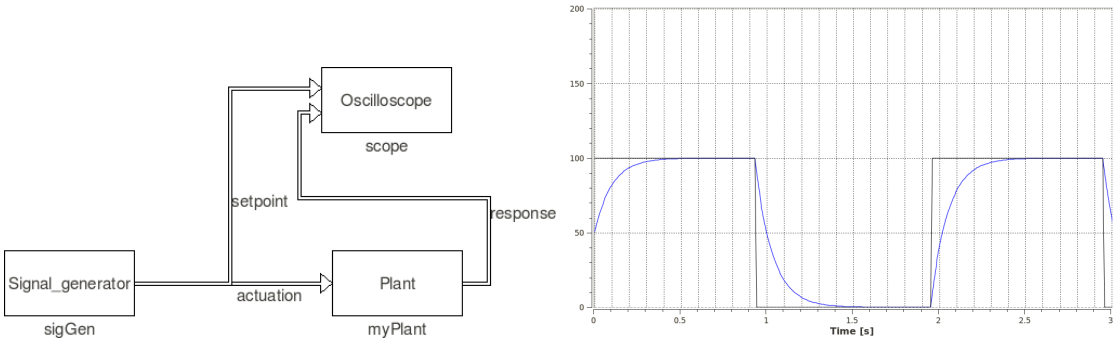
At this stage it can be inserted into XL as a plant. Fig. 8.2a shows a diagram to test the plant's step response. The figure contains only the canvas, the rest of the interface has been omitted for simplicity's sake. The output of the signal generator and of the plant have been connected to an oscilloscope. The result can be seen in fig. 8.2b.

The plant is stable as expected, but it has a slow rise time. As we've seen in section 2.1, if we close the system in a negative feedback loop and introduce a controller, then performance may be improved.

Fig. 8.3a shows an attempt at just that using a PID controller. By configuring the PID with $K_p = 3, T_d = 0.5$ and $T_i = 0.5$, the rising time was greatly improved and oscillation and overshoot kept to a minimum. The result can be seen in fig. 8.3b.

Xenomai Lab comes built-in with several useful blocks. The real-time blocks include

- **Signal_generator** - outputs sine, triangular and PWM waves.
- **Setpoint** - outputs a setpoint value.
- **MLoad** - loads a setpoint profile from MATLAB.
- **Gain** - multiplies input by a double.



(a) Diagram.

(b) Scope. Setpoint and response are plotted as black and blue, respectively.

Figure 8.2: An Open Loop System

- **Sum** - adds inputs.
- **PID** - A simple PID controller.
- **Plant** - Applies a discrete transfer function to the input.

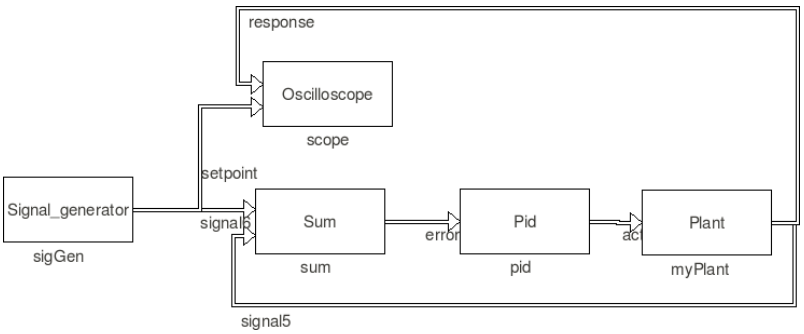
The non real-time include

- **Oscilloscope** - A 5 channel oscilloscope.
- **Display** - Prints input signals to stdout.
- **MSave** - Saves input signal to a MATLAB file.

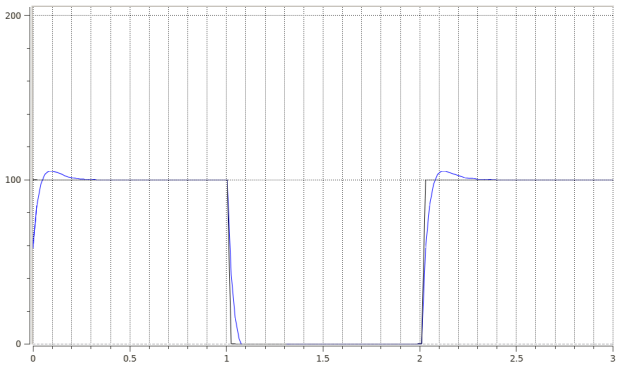
Part of XL's power lays in the fact that customizing existing blocks or programming completely new ones is quite simple. In fact, XL is composed of two separate, yet tightly integrated components: the **Blocks** and the **Lab**. The C code for any block can be easily edited and compiled from within the lab. To fully understand how this works, we must look at how blocks and the lab are implemented and how they work together.

8.2 Blocks

A **Xenomai Lab block** is essentially a small C program that instantiates a Xenomai real-time task. All blocks are separate executables, and hence, separate processes. They are independently and directly scheduled by the Xenomai co-kernel.



(a) Diagram.



(b) Scope. Setpoint and response are plotted in black and blue, respectively.

Figure 8.3: A Closed Loop System with a PID controller.

Each block has a list of input and output channels. The channels are passed as input arguments to the executable. Consider the following call to the gain block

```
./gain −i error −o plant
```

Here, gain will read the error channel, manipulate it and write the result to the plant channel. There is no limit in the number of channels, a block accepts a list of comma separated channel names of any size.

```
./sum −i setpoint,ad,noise −o error
```

Here, sum will add the values of setpoint, ad and noise and write the result to the error channel.

A **channel** is a communication bridge between two blocks. Blocks use channels to exchange information. The information exchanged are floating point numbers, but instead of using double or float, blocks communicate using a custom **Matrix** type. The Matrix type is part of a matrix library developed by Diego Mendes who was kind enough to allow integration of his work with Xenomai Lab. By using Matrix, blocks are able to exchange a group of floating point numbers with double precision. Scalars, vectors and matrices can all be encapsulated into the Matrix type. The library also comprises a set of standard matrix operations such as calculating determinants, eigen values, etc. These functions can be used within the block to perform common computations.

Not all elements in a block diagram implement control functions. Some blocks exist solely for **monitoring** purposes. We might want to save a signal to a file and analyze it later, for example. These operations don't have the strict real-time requirements that control operations have. To support this fundamental difference, blocks can be either **real-time** or **non real-time**. Non real-time blocks operate in much the same way as their real-time counterparts.

```
./display −i setpoint,error,actuation
```

Here, display is reading 3 input channels which will then be displayed to the user. By definition, non real-time blocks have no outputs in the context of Xenomai Lab. Nevertheless, non real-time blocks are regular Linux C programs detached from any Xenomai specificities. This means that, in reality, they can be programmed to do anything with the information they get from the input channels. The block can service a web server with

real-time data from a process, or send experimental results through a UNIX socket to the other side of the globe. No restrictions are imposed by XL.

Communication between real-time and non real-time blocks is strictly unidirectional, from real-time to non real-time. A special kind of channel, called a **pipe**, is used for this special case.

```
./gain -i error -o plant -p actuation
```

Here, gain outputs its result to an extra non real-time channel called actuation. A non real-time block can then open it like in the previous example.

Blocks have parameters, or operational **settings**. A PID controller, for example, has its operation conditioned by three gains – K_p, T_d and T_i. Nothing stops these parameters from being hardcoded in the controller implementation, but this approach brings several problems.

One block diagram might have multiple blocks of the same type. For example, it is perfectly common to have a block diagram with several gain blocks. If the value of gain were hardcoded in the code, this means that every gain block would apply the same gain. It would be unfeasible to design a diagram of complexity above trivial with such a system. One would need as many gain blocks as many different gain values were needed. A gain block of gain -1, another of gain 4, and so forth.

Bearing this in mind and following the UNIX tradition, each block has a **configuration file** that defines its operational settings. This way one single block, (meaning one single executable and one single source file) can run with different settings depending on the configuration file. A single block executable can have multiple **instances**, and hence, we can also refer to the configuration file as a block instance.

```
./gain -i error -o plant -p actuation Gain4
```

Here, we added an instance to the end of the gain call. gain would then look for a Gain4.conf file and load its parameters from it.

The problem doesn't end here however. A reasonable expectation of such a system would be the ability to change these parameters in *runtime*, when the block diagram is running in real-time. If a plant is being controlled by a PID, for instance, it is essential to be able to change the parameters in real-time and watch how the variation affects the control.

Towards this goal, each block has a Graphical User Interface (*GUI*) that allows real-time interaction, in the form of a **settings application**. This is a minimal Qt application that allows block settings to be changed before, during or after a diagram is executing.

8.2.1 Anatomy of a Block

A block lives inside a directory with a strict structure. Listing 8.4 shows the structure using the gain block as an example.

```
Makefile
gain*
gain_settings*
gain.c
gain.conf
gain_settings.c
gain_settings.h
gain_settings_proj/
    gain_settings_proj/
        main.cpp
        mainwindow.cpp
        mainwindow.h
        gain_settings_proj.pro
```

Listing 8.4: File structure of the gain block. An asterisk marks the executables.

The structure is precisely this for every block. Because of this rigidness we can continue our review of the block using gain as an example without loss of generality. A setpoint block, for instance, would live in a setpoint directory, with a setpoint.c and a setpoint_settings_proj folder, etc.

A block is comprised of seven source files and one configuration file. These define only what is *specific* to a block. The *common* functionality is hidden behind a library. This separation makes programming blocks a breeze – a system designer can focus on the algorithms and associated parameters. The algorithm is programmed in the real-time block executable which is but one source file, while settings comprise the rest of the structure.

8.2.2 The Real-Time Block Executable

gain.c contains the definition of the block executable. It is responsible for instantiating the real-time task. This task is nothing more than a simple thread. The code contains a main() function that is standard to all blocks, so we won't review it here. Suffice to say that main() is responsible for opening the communication channels and starting the real-time task. The entry point for the task is the loop function, which then calls a periodic function, as shown in listing 8.5. This is where the actual *gain* functionality of the block is implemented.

```
Matrix periodic_function(Matrix* inputChannel,short numChannels){
    Matrix ret;
    ret=matrix_mul_double(&inputChannel[0], gs->gain);
    return ret;
}

void loop(void *arg){
    Matrix outputMatrix;
    while (running) {
        read_inputs();
        outputMatrix=periodic_function(io.input_result,io.input_num);
        write_outputs(outputMatrix);
    }
}
```

Listing 8.5: gain.c (detail)

Inside loop() we have the main while cycle. This will read the input channels defined in the program arguments, obtain an output result and write it to the output channels. Notice that most of the time the input channels are empty. read_inputs() is a blocking function and thus spends most of the time sleeping, only waking up when there is information in the channel. This is why the cycle doesn't burn 100% CPU and periodic_function() is called *periodic*.

periodic_function() is called with a Matrix array and its size. The array contains the reading from each channel. In this example, the periodic function multiplies the matrix in input channel 0 by gs->gain. gs stands for global settings. It's a data structure that holds

the block parameters. This brings us to the next section

8.2.3 Settings

The Settings functionality encompasses several files. Luckily, they are quite simple. The configuration file contains the settings and can be seen in listing 8.6.

```
[Operation]
Gain=-1

[Task]
Priority=99
```

Listing 8.6: gain.conf

This configuration file is implemented using the "Application Settings Management" library [50]. Parameters can have any alphanumerical name, and can have **integers**, **doubles**, **strings** or **Matrices** as data. Each block comes with a built-in priority parameter for the real-time task, which can vary between a minimum of 0 and a maximum of 99. This is useful to assert which block runs first in case of contention.

The block_settings.h file contains the data structure associated with the .conf and can be seen in listing 8.7

```
struct global_settings{
    double gain;
    int task_prio;
};

extern struct global_settings* gs;
```

Listing 8.7: gain_settings.h (Detail)

In this example, gain is of type double even though we wrote it as an integer in the .conf. The block_settings.c contains the loader and unloader functions for the parameters and the declaration of gs;

get_double() is used when the block starts and the variable is loaded from the .conf file. store_double() is the complementary function, it saves the variable to the .conf if it

```
struct global_settings* gs;

void load_gs(void){
    get_double("Operation","Gain",&gs->gain);
    get_int("Task","Priority",&gs->task_prio);
}

void unload_gs(void){
    store_double("Operation", "Gain", gs->gain);
    store_int("Task","Priority",gs->task_prio);
}
```

Listing 8.8: gain_settings.c (Detail)

was changed during execution. Variations of these functions exist for integer, strings and matrices.

In block_settings_proj, we have a small Qt project for the settings interface. The main.cpp for the project is completely standard Qt boilerplate code. The only difference is a call to a static function from the library that supports the settings interface. This instantiates the required data structures and handles the executable inputs. Settings need an instance passed to it as input

./gain_settings gain1

Here, gain_settings will look for gain1.conf in the same way the block does. By default instances should be in the **workspace** directory, which we will talk about in the next section.

Once settings has an instance, it creates the window seen in fig. 8.9. As is common in Qt, the main window of an application is built using a MainWindow class.

As can be seen in listing 8.10, MainWindow derives from BlockBase, the class that implements the settings functionality common to all blocks and encapsulates everything about settings creation and management. The MainWindow constructor in listing 8.10 calls fillDialog with the name of the block and a small description. This description accepts basic html notation, like a
 tag for line breaks, or for bold text.

setSettings is a pure virtual member of BlockBase, which means it must be implemented by any derived classes. This function is akin to load_gs() and unload_gs() in

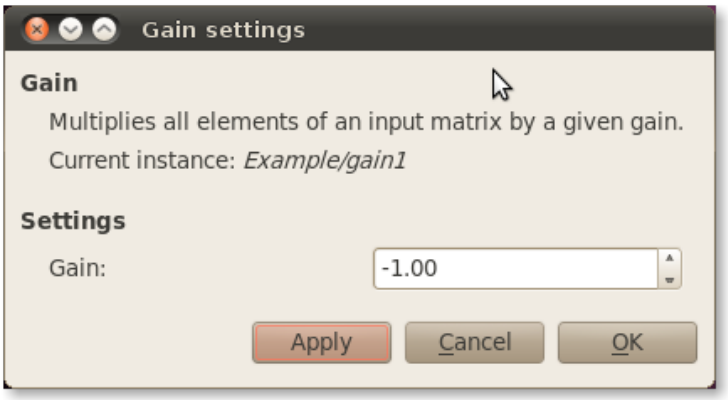


Figure 8.9: Settings interface for the gain block.

```
MainWindow::MainWindow(const QString &execName,
                       const QString &name,
                       QWidget *parent) :
    BlockBase(parent,execName,name)
{
    fillDialog("Gain","Multiplies all elements [...]");
}
void MainWindow::setSettings()
{
    newEntry("Gain:",&gs->gain);
}
```

Listing 8.10: mainwindow.cpp (Detail)

block_settings.c, but all within one function. We establish a new parameter entry and pass it the pointer to the respective gs variable.

This concludes our walkthrough of the block stack. There is no more code. The complete set of data to define a block is then

- A name
- A periodic function
- A collection of parameters and functions to load and unload them to and from memory
- A small description

The real magic lays behind the curtain, hidden in the library files. A brief walkthrough of the main library components can be found in appendix B. As a side note, it is important to mention that a template block is shipped along with the regular blocks. This blocks comes accompanied with a script that automatically renames it to a chosen name. This is currently the preferred way to programming new blocks.

8.3 The Lab

Xenomai Lab is nothing more than an abstract way of interacting with the blocks. There is nothing the Lab does that one could not do by hand in the terminal. For this to work, Xenomai Lab expects blocks to be positioned in specific directories.

The heart of Xenomai Lab lies in the ~/.xenomailab folder[1]. This directory is structured as can be seen in listing 8.11.

The **blocks** folder contains blocks individualized in separate directories. Inside that directory lives the structure we've presented in the previous section. **blocks.conf** registers the blocks with a symbol. An excerpt is presented in listing 8.12.

The **include** folder contains the library files that blocks need. These are described at length in appendix B.

The **examples** directory contains sample projects and reference implementations of I/O blocks. These blocks were developed as part of validation of the program. They are distributed along with the application as a reference. Since they are specific to custom hardware they are useless to users, but very useful as an example.

[1] ~/ is the user's home folder.

```
blocks/
    blocks.conf
    newblock.sh
    display/
    signal_generator/
    template/
    tick/
    ..
include/
    blockbase.cpp
    blockbase.h
    mtrx.c
    mtrx.h
    nrt_block_io.c
    nrt_block_io.h
    rt_block_io.c
    rt_block_io.h
workspace/
examples/
```

Listing 8.11: File structure of .xenomailab

```
[Real-Time]
signal_generator=square
gain=triangle

[Non Real-Time]
display=square
```

Listing 8.12: blocks.conf

The **workspace** is where project folders and block instances live.

8.3.1 Functionality

Before dwelling into the implementation, let us first review how Xenomai Lab works. Since the project clocks in at a hefty 10,000 lines of code spread across almost 40 files, this approach will make it easier to understand what's going on.

When the application starts, a new project called "Untitled" is created. A **project** consists in a folder in the workspace. It acts as a place holder for all the blocks instances in the diagram. At this point, the workspace tree looks like this

```
workspace/
        Untitled/
                diagram.conf
```

diagram.conf is a descriptive file of the current diagram. Since the diagram is initially empty, diagram.conf is also empty.

Having established a project, we arrive at the application's main window, which we've already seen in fig. 8.1. Let us look at the fine details.

The block list, on the left-hand side, is populated by reading blocks/blocks.conf. In this initial implementation, blocks can only be squares or triangles.

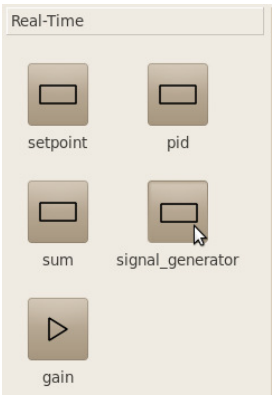
By clicking once in a block we select it. By clicking in an empty space on the canvas we instantiate a block. The application will ask for a name and then copy the .conf from the block folder to the Untitled workspace. An entry will be added to the diagram.conf with the coordinates, instance name and block type. This is illustrated in fig. 8.13

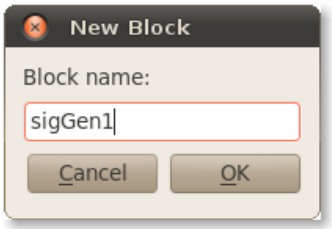
This is fairly useless as is. But by adding a new block we can then make a connection and have our block diagram to something useful.

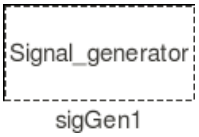
A **line** is a connection between blocks. The connection process is illustrated in fig. 8.14. The application decides if it is a real-time or non real-time connection on the destination block. For the user, the issue is handled transparently. Upon execution, the two blocks will be called like so:

```
./signal_generator -p signal1 Untitled/sig
./display -i signal1 Untitled/display1
```

At least they would be, if things were as simple as they seem. As we've seen in the previous sections, a block is a non periodic task that blocks on read_inputs() until data is



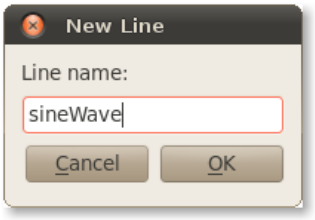


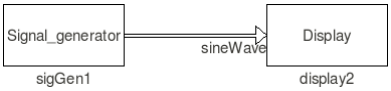



```
[Block0]
Name=sigGen1
Type=signal_generator
X=566.667
Y=724

[Diagram]
Blocks=1
```

Figure 8.13: Placing a signal generator block.






```
[Block0]
Name=sigGen1
Type=signal_generator
X=566
Y=724

[Block1]
Name=display1
Type=display
X=876
Y=724
```

```
[Diagram]
Blocks=2
Lines=1

[Line0]
Value=[0]
Name=sineWave
Destiny=display1
Origin=sigGen1
```

Figure 8.14: Making a connection between 2 blocks.

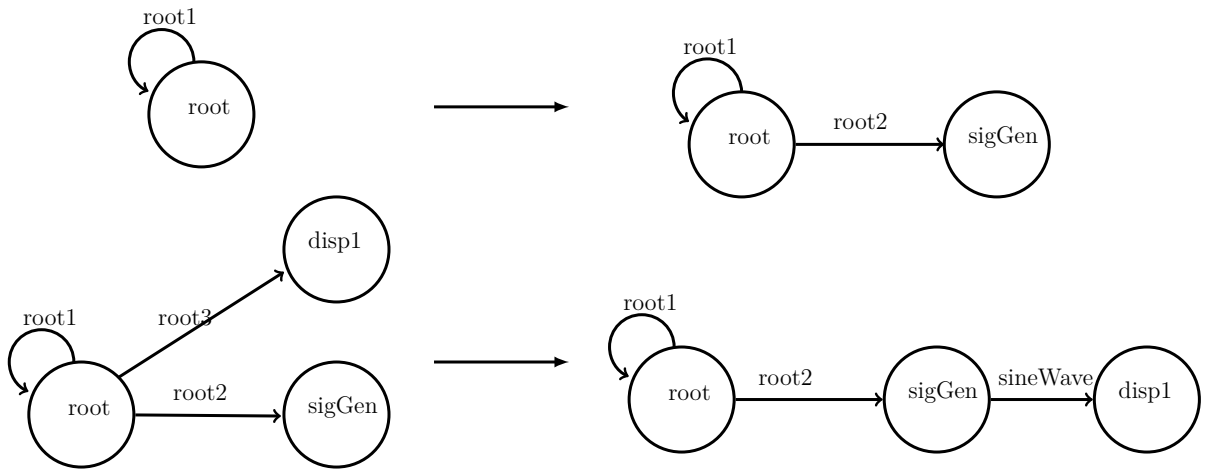


Figure 8.15: The evolution of the logic graph when creating a basic block diagram.

available in its inputs channel. signal_generator is a perfectly regular block, so executing the above would produce nothing of desirable. There's more to the diagram than meets the eye

A block diagram is nothing more than a logic graph with a set of vertices and directional edges. Each block can therefore be treated as a vertex in a graph. Every block diagram designed in XL has an *implicit* vertex called the **root** vertex. When we designed our example diagram in fig. 8.14, behind the scenes the graph shown in fig. 8.15 was being drawn.

When we first started the application, even though we omitted it for the sake of simplicity, a root instance of type tick was instantiated. It is connected to itself because there can be no orphan vertices in the graph. By orphan, we mean a vertex that doesn't have any inputs.

When we added the signal_generator and it was the only block in the diagram, it wasn't orphan. In reality, it was connected to root, and so was display when first added. By drawing a line between signal_generator and display we create an edge between signalGen1 and display1. Because it now has an input, display is no longer an orphan and so its connection to root is deleted.

So, what does root do? root defines the sampling period of the system. It is an instance of the **tick** block, which, contrary to regular blocks, contains a periodic task that only calls write_outputs(). In practice, root is responsible for periodically waking up the orphans. This is how signal_generator is awoken periodically, it blocks on the read queue until root wakes and writes a dummy Matrix on that queue.

So, if we were to run this diagram, the following would be executed.

```
./tick −i root1 −o root1, root2 Untitled/root.conf
./signal_generator −i root2 −p testSignal Untitled/signalGen1
./display −i testSignal Untitled/display
```

Having understood how the diagram works, let us now look at the way we can interact with it.

- **Clean**. runs “make clean” on each block folder
- **Build**. runs “make” on each block folder.
- **Run**. executes the diagram by calling each block executable with the appropriate argument and instance.
- **Stop**. Sends SIGKILL to every block.

Figure 8.16: Diagram actions toolbar

- **Delete**. Deletes the selected block or line.
- **Edit**. opens block.c with vim. The default editor can be changed.
- **Settings**. Calls block_settings for the selected block
- **Sampling period**. Calls tick_settings for the root instance.

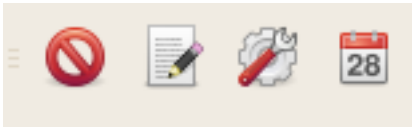


Figure 8.17: Block actions toolbar

- **Edit/Settings**. Already covered
- **Run in terminal**. This flags a block to execute inside an xterm. This allows the user to read the block's stdout stream and is useful for debugging.
- **Run as sudo**. Instead of runing the block directly, the block will be run as super user by preceding it with the sudo command. XL will ask for the user's password.

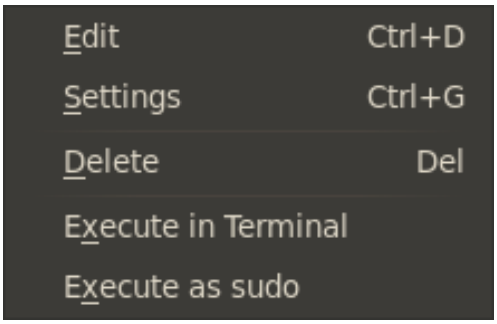


Figure 8.18: Block context menu

- **Save**. Tars the current workspace and saves it under a project name with a .xlb extension.
- **Save as**. Same as save but with a different project name.
- **Open**. Untars a .xlb into the workspace folder and iterates over diagram.conf so as to populate the diagram.

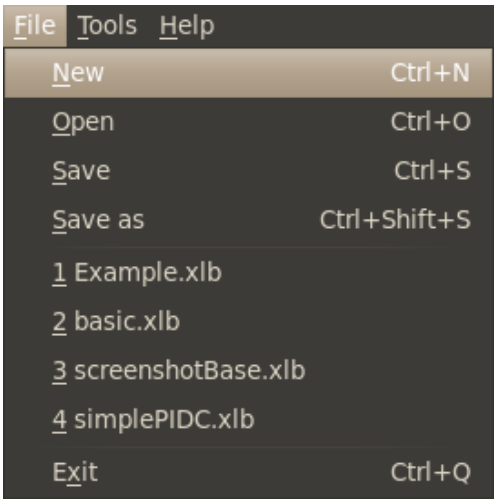


Figure 8.19: File menu

Having reviewed all of XL's current functionalities, we are now in a much more comfortable position to detail the implementation.

8.4 Implementation

Xenomai Lab implements a fairly standard design pattern known as the Model-View-Controller or MVC for short.

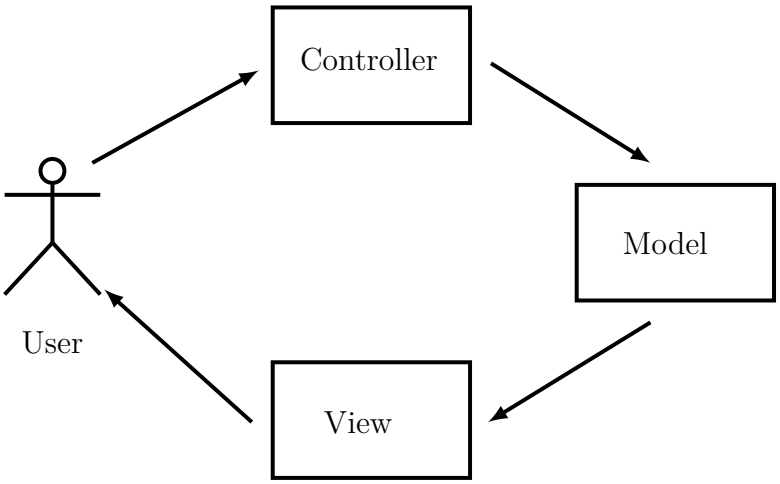
The **MVC** pattern separates an application into a **logic model** that holds some data, a **view** that represents the model for the user, and a **controller** that allows the user to manipulate the data model.

Xenomai Lab uses the canvas to represent its data model. Manipulation is done via the surrounding buttons and direct interaction with the canvas. The data model is the block diagram, that is in reality two models in one. The **abstract block diagram** is the logic graph with vertices and edges, the **concrete block diagram** is the workspace with block instances. Fig. 8.21.

8.4.1 Model

Abstract Block Diagram

The abstract block diagram is implemented in the **BlockDiagram** class. It encapsulates the creation and management of blocks and lines.



1

Figure 8.20: The basic diagram of the MVC design pattern.

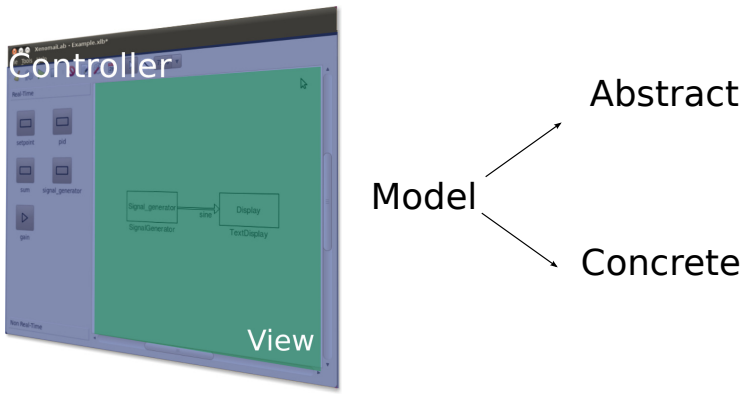


Figure 8.21: MVC redux.

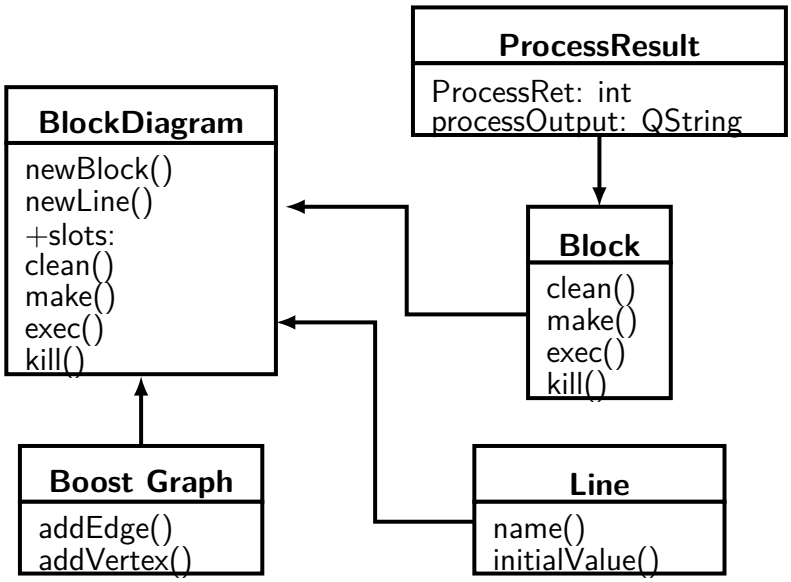


Figure 8.22: BlockDiagram and supporting classes.

Block Diagram uses the C++ boost graph library. **Boost** is an established C++ library known for providing enterprise quality extensions to C++ [51]. It was a natural option to implement this functionality. The boost graph holds edges and vertices. Each vertex is defined as a Block object and each edge as a Line object. The basic graph is then a logically ordered collection of pointers to these objects.

The **Block** class contains the essential block data. Each block has a name (the instance name), a type, an rt/non rt flag, flags to condition execution, etc.

Block also implements a group of actions. The clean, exec, make and kill are synchronous calls that instantiate a process that does these actions in the block folder-

These functions return a **ProcessResult**, a basic data structure that holds the result of the process. The result of the process is its stdout or stderr and its return value.

The **Line** class holds the name and initial values as matrices.

To interact with the diagram, BlockDiagram has slots called clean, make, exec and kill.

These slots take advantage of a functionality of the Boost graph. It allows us to iterate over every vertex in the graph according to a certain rule. BlockDiagram does a breadth first search on the graph[2] and calls the appropriate function on the Block object. The Block responds with a ProcessResult which is then analyzed. If an error occurred, the search is

[2]In our case, a breadth first search starts in the root, then the siblings of root, then the siblings of those siblings and so on.

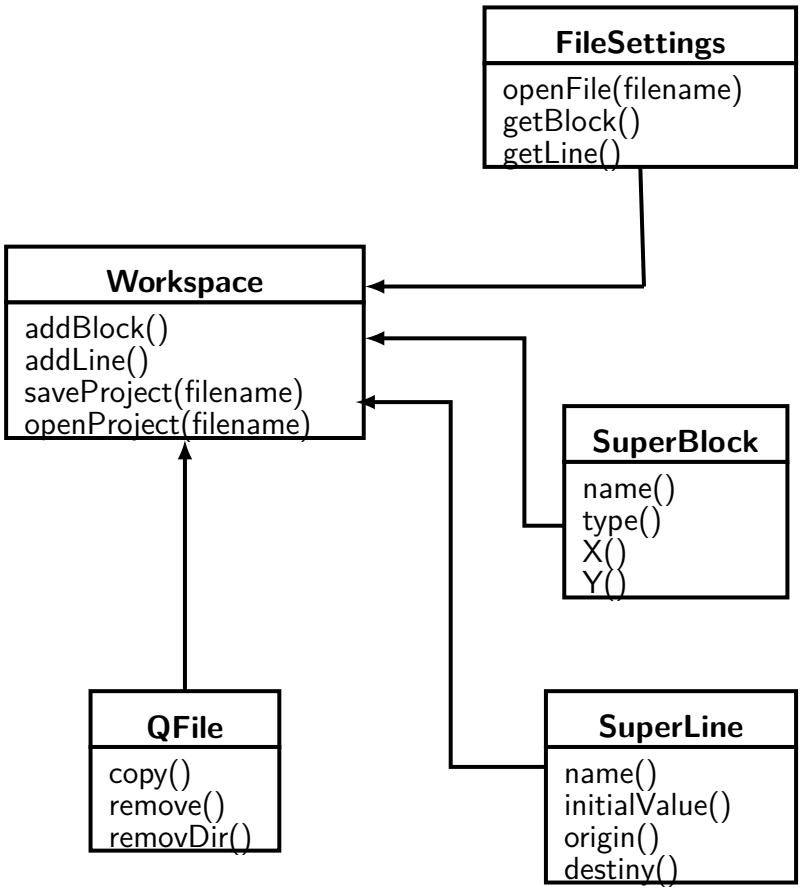


Figure 8.23: Workspace and supporting classes.

aborted and already executed blocks are killed. The ProcessResult is unconditionally sent to the main window as a progress report of how the requested action is coming along. This is how XL shows a loading bar during build and execution.

It is worth mentioning that when executing, BlockDiagram uses boost functions to get the names and initial values of all lines going in to a block (the vertex's in_edges) and all lines going out (the out_edges). This is how the block argument is formed.

Concrete Block Diagram

The concrete Block Diagram is implemented the Workspace class and can be seen in fig.8.23.

Workspace maintains a list of SuperBlocks and Superlines and uses FileSettings to interact with diagram.conf. It can open and save .xlb project files, which as we've mentioned is nothing more than taring and untaring the workspace folder. It can then load the

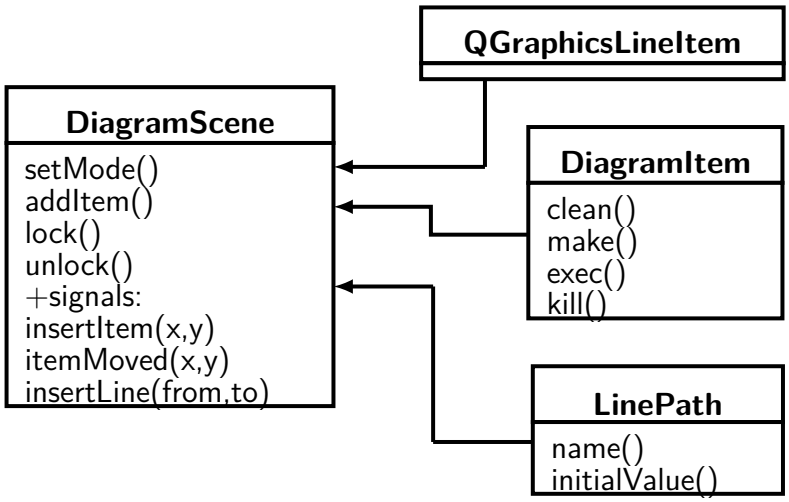


Figure 8.24: DiagramScene and supporting classes.

diagram.conf of the project and instantiate the respective SuperBlocks and SuperLines.

QFile is used to manage block instances. It is used to copy the block.conf from the block folder to the current workspace and rename it to blockname.conf.

Notice that **SuperBlocks** and **SuperLines** are nothing more than basic data structures. They hold the complete information needed to instantiate these types. That's why they're *Super*. The Block in BlockDiagram does not have coordinates, for instance, since these are useless to execution and cater only to graphical representation.

FileSettings is an encapsulation of the same settings library used to implement the .conf of each block. It provides an abstract interface to query data from diagram.conf

8.4.2 View

The view is XL's white canvas. As per fig 8.24, the view is implemented in the DiagramScene class, which derives from QGraphicsScene. It manages and plots a collection of DiagramItems and LinePaths. It also plots a temporary QGraphicsLineItem when the user begins dragging a connection from one block to another.

DiagramScene isn't merely a representation of the data model as it offers its own manipulation options. You can, for instance, directly move a block by dragging it in the scene. Although DiagramScene doubles as a view and a controller, it always forwards events to the controller. This way, our system's sanity is preserved. We can see in fig. 8.24 how DiagramScene emits signals for the main events on the canvas. MainWindow catches these

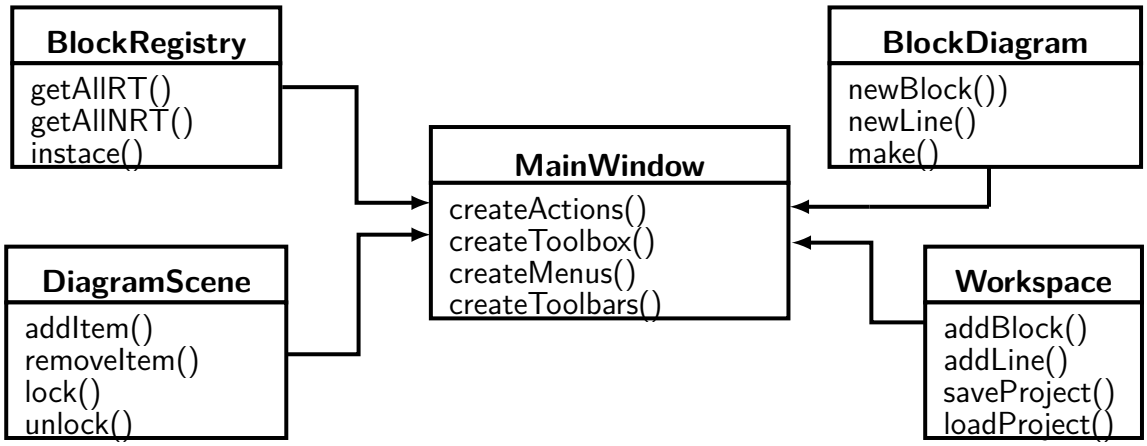


Figure 8.25: MainWindow and supporting classes.

signals and acts accordingly.

Blocks in this context are represented by a **DiagramItem** while lines by **LinePath**. DiagramItem contains a list of incoming and outgoing LinePaths, as well as a polygon, and two text labels. LinePath contains a text label and an arrow, which is implemented as a polygon that draws the contour of an arrow. This approach proved itself heavier than was expected, and is by far the biggest performance bottle neck in the plot of the block diagram.

8.4.3 Controller

The main window is a subclass of QMainWindow, the basic Qt class for application windows. It implements the file menu, toolbar, side bar, about dialog, error messages, etc.

As can be seen in fig. 8.25, **MainWindow** is the controller because it is the central point where other classes meet. Workspace doesn't know about BlockDiagram, but MainWindow knows about both and keeps them in sync.

It is a useful exercise to accompany how MainWindow is constructed. This will bring together what we've been discussing up until this point.

MainWindow starts by registering available blocks with **BlockRegistry**. This will interpret blocks.conf and create the data structures that will populate the side bar. If no blocks.conf is found, the applications throws an error,

Afterwards, instances of QGraphicsScene, BlockDiagram and Workspace are created. This prepares an empty canvas, a graph with the a root vertex and a workspace called Untitled with an empty diagram.conf and a tick instance called root.

Chapter 9

Experiments

In this chapter block diagrams that actuate on and acquire data from the real-world are presented. The experiments were conducted with the objective of verifying Xenomai Lab's real-time performance and establishing it as a validated control platform.

9.1 Are you experienced ?

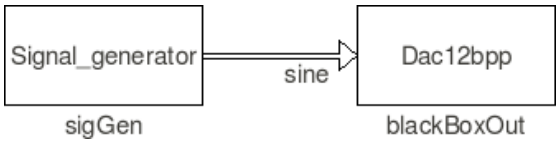
It is important to test XL's capabilities as a digital control platform. Towards this goal, two experiments we named the "Black Box" and the "Inverted Pendulum" were conducted. We shall review these in turn.

9.2 Black Box

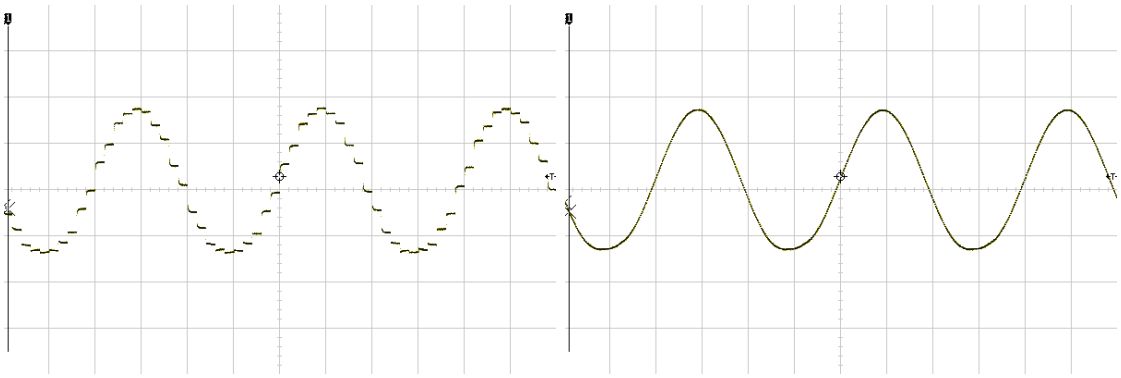
The black box consists in a 12 bit A/D D/A combo enclosed in a single (black) package. The box connects to a computer via parallel port. It was developed *in-house* in the gone year of 1998, when DOS and its lack of parallelism was still a familiar site instead of an old joke.

The developer put together a small driver that consisted in 2 functions: one to write a value between 0 and 4096 on the D/A, one to read an int from the A/D.

To test it with XenomaiLab two blocks were developed: dac12bpp and adc12bpp. This took less than one hour of coding. Let us look at the some of the results



(a) dac12bpp block test.



(b) T=1000μs (c) T=100$u\mu s$

Figure 9.1: BlackBox signal generation. Oscilloscope displayed with 5ms and 0.5V/div.

9.2.1 Signal Generator

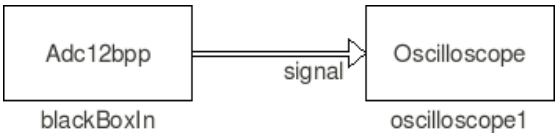
The block diagram in fig. 9.1 was put together to test the D/A. Fig. 9.1b shows a $50Hz$ sine wave sampled at $1KHz$ frequency, and 9.1c the same wave at $10KHz$. This was the maximum frequency achievable by the system. Above the system the system completely froze.

The results shown were obtained using a digital oscilloscope.

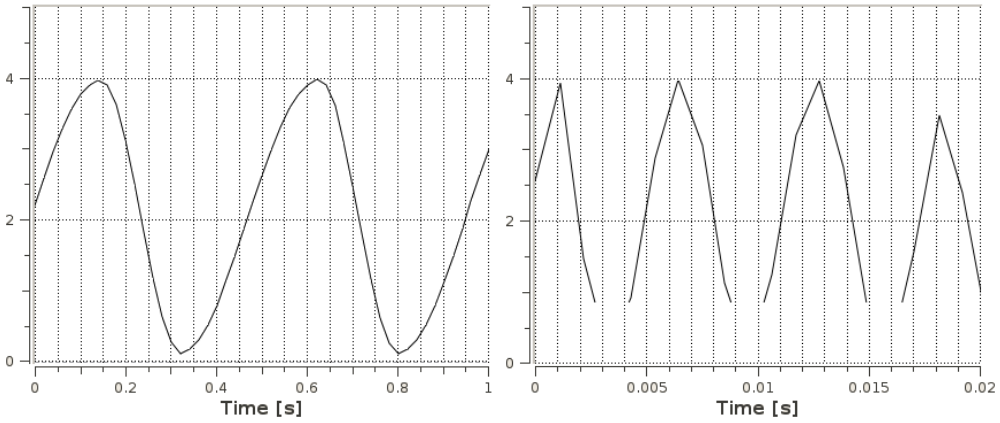
9.2.2 Oscilloscope

To test the other block, an external function generator was hooked to the A/D input and the block diagram in fig. 9.2a executed.

As can be seen in fig. 9.2b, the XL oscilloscope could successfully measure the signal coming from the outside world. Unfortunately, since the oscilloscope is not real-time, its bandwidth is very limited. At 200 Hz, the oscilloscope could barely keep up, as seen in fig 9.2c. The maximum sampling period the oscilloscope can sustain is $1ms$. An improved version of the oscilloscope would be beneficial for scenarios such as this one.



(a) adc12bpp block test.



(b) Oscilloscope reading a 2Hz sine wave (c) Oscilloscope reading a 250 Hz sine wave

Figure 9.2: BlackBox signal input.

9.3 Inverted Pendulum

Having established that XL can generate signals and acquire them in real-time, the next logical step would be to control a real plant.

The **inverted pendulum** is a textbook example of a fairly complicated control system. A movable cart tries to maintain a bar in the upright position. Assuming that the system starts in equilibrium and is subject to minimal interference, the mathematical analysis can be greatly simplified. This is why it is a typical, entry level, control example.

A picture of the setup can be seen in fig. 9.3. A servo mechanism controls the position of the cart. Holding the bar, a metal piece connected to an optical encoder reads the current angle. An additional encoder is connected to the servomechanism to monitor the cart's movement. Both encoders are connected to 16-bit counters (two hctl 2016 chips) that expose the current value of the encoder in 8-bit TTL values. The setup was built for use with microcontrollers. As such, much of the circuitry had to be buffered so that it could connect with the computer's parallel port. Three blocks were developed that interact with the parallel port using the parapin library [52]: cart_pos, hctl_angle, hctl_cart.

The system was initially controlled ignoring the cart's position. Fig. 9.4a shows the block diagram and 9.4b shows a screenshot of the scope mid system operation. The setpoint

Figure 9.3: Picture of the inverted pendulum setup.

is implicitly 0. The PID was configured with $K_p = 2, T_d = 0.5$ and $T_i = 0$. It can be seen in the scope that the angle is kept close to 0, as was intended.

To push this system further, a printed circuit board (PCB) was projected to interface the pendulum. The schematic and final PCB layout can be seen in Appendix F. The circuit has standard I/O buffers for the parallel port and provides a button to reset the hctl counters. More detailed information is available in the appendix.

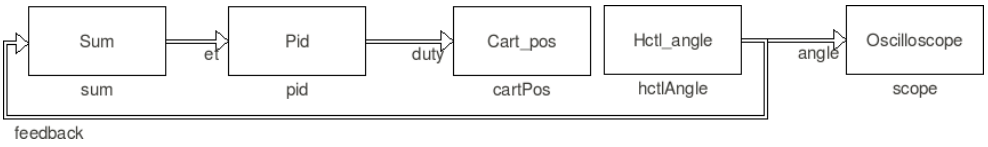
The final block diagram and an example measurement can be seen in fig. 9.5.

This final control stage was unsuccessful. Data acquisition was working perfectly and the problem was unrelated to XL. The problem encountered was that the hctl counters would reset on seemingly random basis. A correlation was discovered between spikes in current caused by the servomechanism and the spurious resets[1]. This happened even though a separate power source was used for the servo. This can be clearly seen in fig. 9.5b. The angle is plotted in black, and the position in blue. As the bar starts falling the angle moves away from zero, the position of the cart tries to compensate the fall. After a peak in velocity both readings are reset to zero at approximately $7.75s$, and the setpoints are lost.

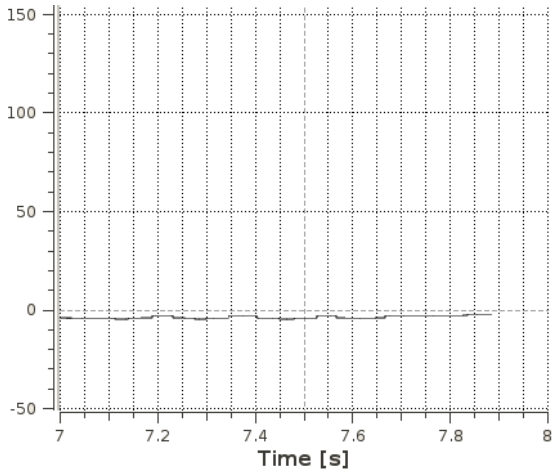
Since the measurements of the position and angle would constantly reset, the design of the control system was severely impaired. The issue was also present when the cart position was ignored, but it was manageable. With the second hctl in place, the hardware fault was much more noticeable as the cart suffers much greater acceleration. Due to lack of time, the hardware was left as is and this final stage of control abandoned.

Nevertheless, our objective was to put XL to the test. The real-time behavior of XL was perfectly adequate for these examples.

[1]Correlation does not imply causation, however.

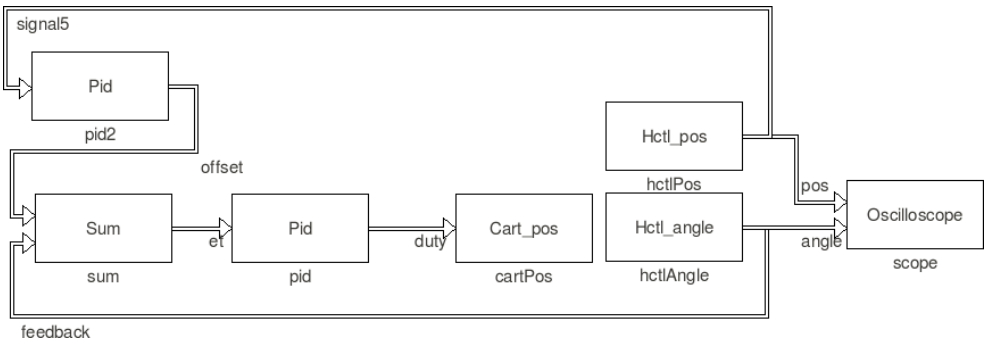


(a) Diagram.

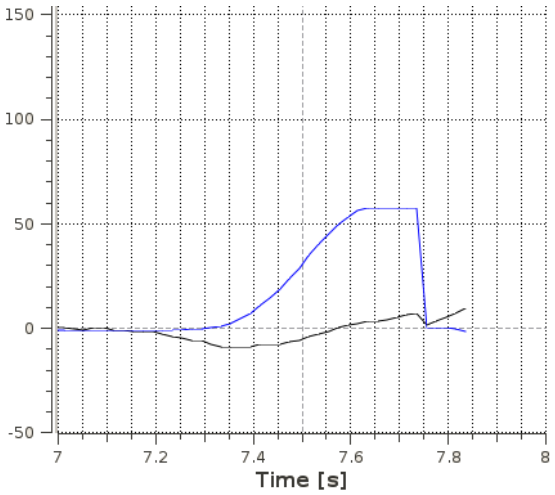


(b) Scope. Angle measurements are plotted in blue color.

Figure 9.4: A simple control system for the inverted pendulum.



(a) Diagram.



(b) Scope. Angle and position measurements are plotted in black and blue color, respectively.

Figure 9.5: A more advanced control system for the inverted pendulum.

Part IV

Conclusion

Chapter 10

Conclusion

The main objective of this dissertation was two-fold: to study the real-time performance achievable today with Linux and to present a platform for digital control that decouples control algorithms from RTOSs concepts.

Regarding the first objective, a quantitative analysis was made of all major players in the real-time Linux world. The analysis was based on the **jitter of a square wave** produced on the parallel port. By putting the systems under various types of loads, a more complete picture was painted. Several fragilities were thus identified. A **real-time Linux scale** was established that helps a systems designer position himself in the midst of so many different solutions.

Regarding the second objective, a platform for digital control using Xenomai was developed. Xenomai Lab includes a **graphical application** to design block diagrams, and a **block API** to easily program new blocks. The development cycle was unfortunately not entirely finished as it takes more time than a dissertation allows. The platform needs some more work and feedback from people in the field. Such things are lurking on the horizon, however, as a dissertation to implement **adaptive control algorithms** using Xenomai Lab is already in its early stages and will be completed in 2012. In the University of Wroclaw, Poland, a **teaching package** using Xenomai Lab is being prepared. The project uses several pandaboards (a single board computer with a dual-core ARM chip) running Ubuntu 11.04 and challenges students to use them to control some in-class experimental setups. Students can program a real-time driver to interface with the setup and then use Xenomai Lab to deal with higher level control. Work on this is on-going, Xenomai Lab should make its debut in polish classrooms as early as February 2012. Contacts are also being made to mention Xenomai Lab in the official Xenomai wiki, which would bring a

much needed exposure to the platform.

The validation of Xenomai Lab with a real-world setup, an inverted pendulum in a cart was not entirely successful. This was not due to some intrinsic fault of Xenomai Lab but due to hardware issues and a lacking control algorithm. With some more time invested the issue would probably have been solved. Nevertheless, the successful experiments were enough to prove that Xenomai Lab's performance is sufficient. Reading and producing electrical signals was demonstrated to work, and a simplified inverted pendulum setup was successfully controlled.

10.1 Future Work

On benchmarking real-time performance, the same test could be run on the latest versions of every operating system and track improvements and possible regressions. A similar test could also be run to benchmark interrupt latency from the parallel port. This would provide the last major real-time benchmark.

Xenomai Lab presents endless possibilities for future work. The sky is indeed the limit. New features that are simple to implement and would benefit the lab greatly include:

- A way to export block diagrams as self-contained executables for easier deployment.
- A tighter monitoring of block execution for better error handling
- An improved oscilloscope with higher bandwidth and more advanced features such as signal memory navigation, auto triggering and auto scale
- Direct deployment to remote targets, as to enable development of networked controlled systems. This can be implemented with clever scripting.
- The concept of blockset would be beneficial to separate blocks into categories other than real and non real-time. As the application grows in exposure, the number of blocks might grow rapidly.
- Profiling of computation time for each block, with relevant static values (mean, minimum, maximum, covariance, etc...) presented within the lab.

Part V

Appendixes

Appendix A

The Testsuite

A.1 Rationale

Given that we intend to implement a control system using a desktop computer, it makes sense to benchmark the OS's scheduling jitter. However, to produce an electrical signal outside of the computer, the scheduling jitter is only part of the equation. We intend to use the parallel port for I/O, and this means the signal has to cross the motherboard before it reaches the outside world. The motherboard is filled with asynchronous activities to the CPU such as DMA transfers and operations on the PCI bus. These will delay the signal for seemingly random intervals. If we were to generate a square wave on the parallel port and measure its jitter, we would have an approximation of precisely this compound jitter.

Several aspects interest us when analyzing the results. For a hard real-time system, the worst-case jitter is the definitive figure of merit. Hard real-time systems have strict deadlines, so the maximum delay has to be within well-defined limits. The worst-case jitter does not, however, characterize a system extensively. It can be that a system has a higher worst-case, but a better average and a smaller standard deviation. It can be that a system has a lower worst-case at a first glance, but may be disrupted by a high network load. Different systems call for different solutions, and a worst-case oriented analysis paints too small a picture.

Bearing all this in mind, the performance test we devised consists in producing a square wave on the parallel port and then measure the time between transitions under different usage scenarios: Idle, under CPU stress, under I/O stress and under Network stress. By doing this we can see how the determinism of each system holds up in the face of common desktop operations.

A.2 Experimental setup

Ubuntu 10.04.2 was installed on a single-core Pentium 4 machine with 1.5GB of RAM and an ATI Radeon 9200 video card. The following versions of the Linux kernel were also installed:

- Linux 2.6.32-32
- Linux 2.6.33 with the PREEMPT_RT patchset
- Linux 2.6.32-20 with Xenomai 2.5.5.2
- Linux 2.6.32-11 with RTAI 3.8.1

Eight different applications (one kernel module and one user space program for each system) were prepared for generating a square wave, the code for which is available in appendix C. Each application uses methods that describe in the respective chapters to execute the following function every millisecond

```
void toggle(void){
    static char byte=1;

    outb(byte,0x378);
    byte=~byte;
}
```

As was explained in the previous section, the computer is put under four different types of loads.

- **Idle**: The system is cold booted and the test is launched.
- **I/O**: A large file is continuously moved between partitions until the end of the test.
- **CPU**: The classic horror film "Night of the Living Dead"[1] in 1080p is played in the VLC player as to load 100% of the CPU.
- **NET**: A large file is continuously moved from the real-time computer to the second computer via Ethernet.

[1]Public domain. Available free of charge from the Internet Archive at www.archive.org

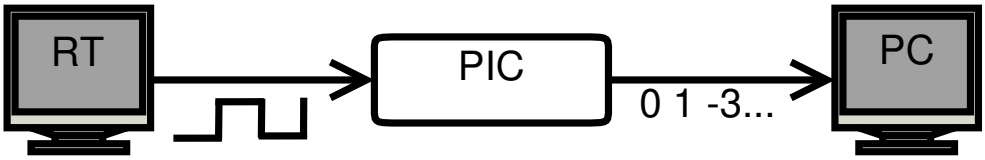


Figure A.1: Experimental setup

One of the output pins of the parallel port was connected to an interrupt line of a PIC18F258 micro-controller. The *PIC* then periodically calculates the wave's jitter and sends the result to a second PC through a serial line. The PC is running a program that continuously polls the serial port for new data and writes the information to a plain text file. Figure A.1 illustrates this description.

A.3 PIC

Using a micro-controller to perform time measurements has clear benefits. It does not interfere with the real-time system and provides a reasonably high precision. Yet it is important not to forget that it has some inherent imperfections that will skew the results in subtle ways. To fully understand these effects, we need to analyze how the *PIC* was programmed to perform these measurements.

The *PIC*'s interrupt line was configured to generate interrupts on both the falling and the rising edge of the wave. We then use one of the built-in 16-bit timers to count the time elapsed between two consecutive interrupts and the expected value is subtracted from the measurement. As an example, let us consider that the time between interrupts was not $1ms$ as it should, but $0.99ms$. The jitter in this case is $-10\mu s$. Since the timer runs at $10MHz$ (the frequency of the crystal), this method allows us to obtain a value of jitter with a resolution of $100ns$.

The interrupt service routine (*ISR*) then needs to perform the following logical steps

- Stop timer
- Read timer value
- Calculate jitter
- Reset timer value
- Start timer

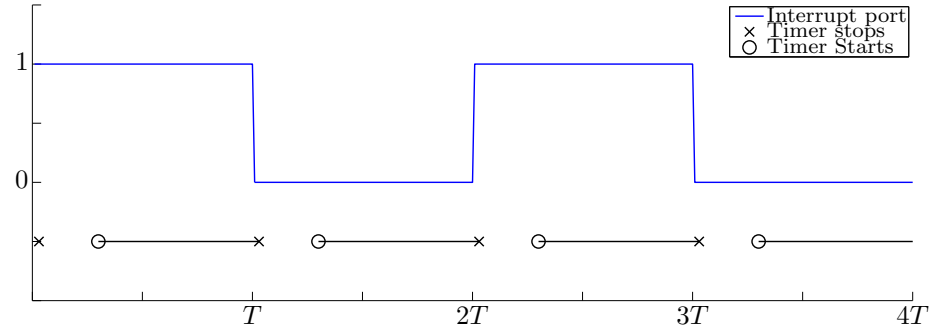


Figure A.2: The difference between the wave duration and what the PIC measures.

- Send jitter through *USART*

Reality, however, is seldom as simple as one hopes. Each of these steps takes a certain amount of time to execute and we need to account for this time. For instance, sending the calculated jitter through the *USART* line was measured to take over $2ms$. This happens because of a heavy *itoa* operation that converts a 16-bit integer to a sequence of characters that spell out the value in *ASCII* encoding. $2ms$ is the time is takes for two interrupts to be generated. This means that accessing the *USART* cannot be done in every iteration of the *ISR*. In fact, outputting the value needs to be done separately from the time measurements since it completely disrupts them.

The time between stopping and restarting the timer is also non-neglectable. This means that the time measured will be shorter than the time elapsed between interrupts. In fact, the time the timer is off will be a *systematic error* across all measurements. Figure A.2 shows a diagram (not to scale) illustrating the issue. The square wave generated by the parallel port and connected to the interrupt line is shown in blue. The *ISR* runs on every state transition. A black cross illustrates when the timer stops counting, and a black circle when it starts counting again. The line connecting the circle to the cross illustrates the time measured by the timer, much shorter than what it should be.

The systematic error is then the time elapsed between stopping the timer and restarting it. To minimize the error, we need to minimize the number of operations we execute between stopping and restarting the timer.

Taking all the above into account, the simple ISR outlined above is implemented as the following pseudo code.

```
stop_timer();
if(iter==29){
```

```
        read_timer();
        calculate_jitter();
        usart_send();
        iter=0;
}
else{
    reset_timer();
    start_timer();
}

iter++;
```

Each 30 iterations, instead of measuring time, we read a measurement, calculate its jitter and send it through the USART line. This way, accessing the serial line never interferes with our measurements.

This *ISR* was measured to yield a systematic error of $1.3\mu s$. This means that the if statement and the reset_timer() function take 13 instruction cycles to execute. For an unoptimized compiler, this figure is perfectly within reason.

A.4 Validation

To validate our system, a reference wave of known period needs to be correctly measured by our device. Towards that end, an Oven Controlled Oscillator (OCO) of frequency equal to that of the *PIC*'s crystal ($10MHz$) was connected to a binary counter that divided the wave's frequency 2^{16} times, down to a period of $6,553,600ns$. The *OCO* used was a device developed *in-house* to achieve minimum jitter and drift, in the other of *ps*. Considering the accuracy of the PIC, the *OCO's* jitter is in practical terms null. The *PIC* was then used to measured this period during an hour long experiment. The expected result would be half the wave's period minus the systematic error of $1.3\mu s$, $3,275,500ns$. Fig. A.3 shows the results obtained. In over 90% of the cases the half-period was in fact $100ns$ short. This fact reveals both the nature of interrupt processing in the *PIC* and the differences between the system's internal crystals.

The time between the state transition on the interrupt line and the time at which the *ISR* begins execution is called *interrupt latency*. For external interrupt sources such as ours, the *PIC*'s interrupt latency is not fixed but variable between 3 to 4 instructions cycles

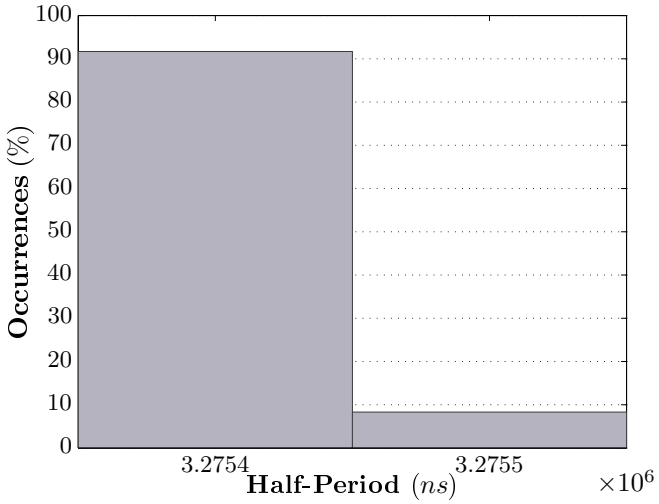


Figure A.3: *OCO* half-period as measured by our setup.

[53]. The fourth cycle accounts for the asynchronous nature of both systems [54]. Since the *PIC* can only interrupt execution at the *end* of an instruction cycle, if the interrupt happens *during* an instruction cycle it will only be serviced when the cycle ends.

The result we obtained is the time between *two interrupts* and each interrupt will be subject to a variable interrupt latency. Depending on the duration of the wave, the different permutations of latency in each edge will either add or subtract 100ns. For a given time between interrupts T, the *PIC* will measure $T \pm 100ns$.

The particular distribution of results obtained with our experiment can be explained by the difference between the frequency of the *PIC*'s crystal and the *OCO*'s crystals. Using a high precision frequency counter we were able to assert that while the *OCO* uses a high precision crystal of period $100.00008ns$, the *PIC* has a slower crystal of period $100.002ns$. Since the *PIC*'s period is *longer* and time is measured as multiples of this period, we need *less* of them to represent the same amount of time.

In conclusion, every measurement done by the *PIC* is subject to an uncertainty of $\pm 100ns$ and a systematic error of $-1.3\mu s$. The results obtained with the *OCO* have confirmed our expectations since they fall within the expected boundaries, and our measurement apparatus is therefore considered to be fully validated.

Appendix B

The Xenomai Lab Block Library

A system that is easy to use is usually complicated to implement. XL tries to provide elaborate features while maintaining complexity at a manageable level. Fig. B.1 illustrates the structure of each block. It follows the operational description of section 8.2. Let's accompany the structure.

The real-time block executable is one independent process with 2 real-time Xenomai threads: the main thread and the loop thread that runs periodic_function(). The periodic function operates over the input channels using global settings in shared memory and writes the result to all its output channels.

The global settings are read from the .conf using the block library and are published to a shared memory instance common to both processes – the gs variable.

The settings project is the second process running on a single Xenomai real-time thread. The settings window is a real-time task so that it can interact with the shared memory, which uses Xenomai primitives exclusive to real-time tasks. As it sleeps when it's not being used, the interference with the real-time system is minimal.

The settings project uses the library functions to interact with the .conf (for offline editing) and get the shared memory address (for runtime editing).

rt_block_io

This header file and its respective .c implement the details of all we've just described. A complete description is available directly in the source code as comments. Nonetheless, we will review the more important functions as a future reference.

The library is logically divided into the following sections

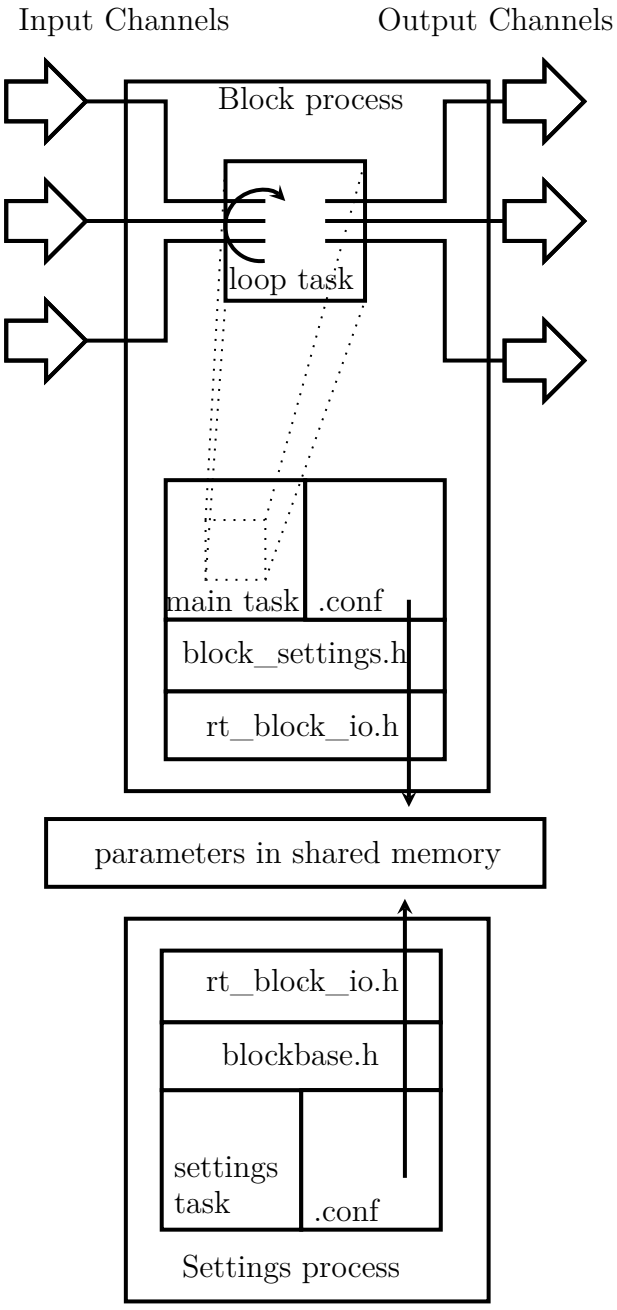


Figure B.1: Graphical representation of the architecture of a block.

```
void parse_args(int argc, char* argv[]);
void free_args(void);

void create_io(void);
void free_io(void);

void read_inputs();
void write_outputs(Matrix sample);

struct ioelements {
        RT_PIPE *output_pipes;
        RT_QUEUE *input_queues,*output_queues;
        char** input_strings,**output_strings,
            **outputp_strings;
        short input_num,output_num,outputp_num;
        Matrix input_result[10];
        char *config_file,*block_name;
};
```

Listing B.2: I/O functions of rt_block_io.h

This set of functions take the argument of the executable and produce the respective data structures needed for real-time I/O, as seen in listing B.2.

The input arguments are handled by using the standard GNU library Argp[1]. The resulting strings are then parsed in parse_args and the *_strings, *_num[2], config_file and block_name variables from ioelements are populated.

create_io() then takes the names of the channels and registers them with Xenomai. The real-time channels are RT_QUEUEs while non real-time channels are RT_PIPEs. Both of these are FIFOs with the size of exactly one Matrix. The only nuance in this process is that channels are used for communication between two blocks. The block that first accesses the channel should *create* the channel, while the last block should merely *bind* to

[1]A more seasoned Linux user will surely recognize the help screen by calling ./block - -help

[2]Here, asterisk is used as the wildcard for "anything"

the channel. This can quickly become a problem when we add feedback to a diagram or if we launch blocks out of order. The solution is quite simple. Each block tries to create the RT_QUEUE, if it fails it tries to bind, and if it that fails the block exits with an error. If the block creates the RT_QUEUE, it immediately writes a dummy Matrix so that the queue is never empty. This nullifies the precedence problem created by feedback loops.

Task

```
RT_TASK loop_task,main_task;

void wait_for_task_end();
void start_task(int priority, void* task_function);
```

Listing B.3: Task functions of rt_block_io.h

These functions are used to create the loop and the main task. Each task is based on an RT_TASK data structure. Ideally, both loop_task and main_task should be inside the ioelements struct instead of being global variables and thereby polluting the namespace. Unfortunately, this is a Xenomai limitation – some variables *must* be global.

Settings

This set of functions is responsible for managing the shared memory structure and interacting with the .conf. The SHM instance is protected by an RT_MUTEX from concurrent access from the GUI and the block. This is how we assure that parameters don't change mid periodic_function() execution – read_inputs() locks the settings and write_outputs() unlocks them.

Much like in the I/O case, the settings feature presents a complicated problem of precedence. Whoever runs first, the block or the GUI, must parse the .conf, allocate the global settings structure in shared memory and create the mutex. Who comes after must merely bind to the structure. So far so good, but due to Xenomai's implementation of shared memory, whoever created it *must* delete it. This means that the first to execute must always be the last to exit.

Like in the I/O case, precedence is established by creating and in case of failure binding to the shared memory. This information is stored in the variable settings_owner and is

```
int settings_owner;
RT_HEAP settings_heap;
Settings *settings;
RT_MUTEX gs_mtx;

void get_double(char* section, char* key, double* value);
void store_double(char* section, char* key, double value);

int load_settings(char* config_file, size_t size);
int save_settings(char* config_file);
int update_settings(char* config_file);

void settings_lock(RT_MUTEX* mtx);
void settings_unlock(RT_MUTEX* mtx);
```

Listing B.4: Settings functions of rt_block_io.h

maintained until the end of execution. This way each program can always assert if it should create or bind, unbind or delete, or just wait until it is safe to delete.

Stop

```
int running=1;
void stop(int signum);
```

Listing B.5: Functions of rt_block_io.h related to stopping execution.

This is a handler to the SIGKILL and SIGTERM signals. By catching the signal, we cancel the running variable. This will make the block break from the while cycle in loop and exit gracefully.

BlockBase

This is the base class for each block setting's MainWindow. BlockBase uses the same functions from rt_block_io.h to access and manage shared memory, so most of the com-

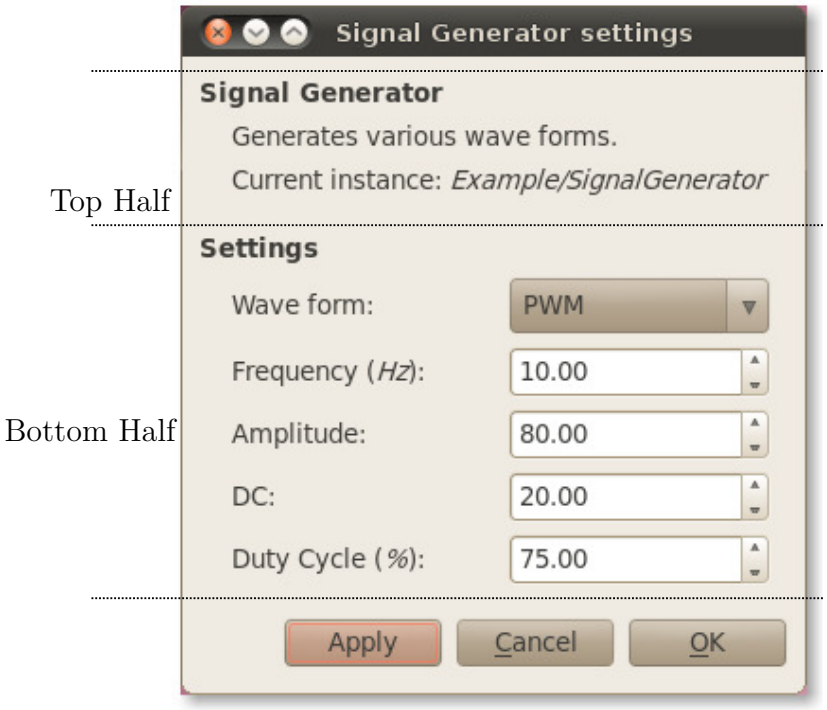


Figure B.6: Graphical representation of the architecture of a block.

plexity has already been explained. In any case, let us present a quick overview of the class.

The interface is divided into two components, as can be seen in fig. B.6.

The **top half** contains two labels: one with the name of the block and one with the description. These are defined in the fillDialog() function we call from MainWindow's contructor.

The **bottom half** is a vertical stack of various variable editors. The possible functions to generate these entries can be seen in listing B.7

```
void newEntry(const QString& text,int* value);
void newEntry(const QString& text,double* value);
void newComboBox(const QString& text,
        const QStringList& options,char* select);
void newMatrix(const QString& text,Matrix* M1);
```

Listing B.7: Functions in blockbase.h to generate entries.

Each entry contains a text QString, the short text label to the left hand side, and a pointer to the data in question. For a string parameter the situation is marginally different

as the user must create a string list with the different options that will appear in the combo box. In the case of fig. B.6, a QStringList with “PWM”, “Sine” and “Triangular”is given to the newComboBox() call.

Notice that by writing in these entries, one is writing directly into the gs structure and hence to shared memory. This may be a source of confusion due to the Apply, Cancel and OK buttons at the bottom.

The buttons refer to the instance file. By clicking **Apply**, whatever values are in shared memory are committed to the .conf. The **Cancel** button resets the shared memory to the original values. **OK** does Apply, then exits.

B.0.1 Non real-time blocks

NRT blocks don’t use any Xenomai concepts. They are simple Linux applications that read input pipes and do something with the value. Current usage is mostly for logging purposes, i.e., save a signal to a file, display using printf, etc.

The supporting nrt_block_io library implements the same API but using linux concepts. read_inputs, for instance, uses a standard read call.

Appendix C

Sources

Listing C.1: Linux Kernel Space hrtimers

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/hrtimer.h>
#include <linux/ktime.h>
#include <asm/io.h>

MODULE_LICENSE("GPL");

#define TICK_PERIOD 1000000 //ns
#define LPT 0x378

static struct hrtimer hr_timer;

enum hrtimer_restart my_hrtimer_callback(struct hrtimer *timer){
    static char byte=0x00;

    outb(byte,LPT);
    byte=~byte;

    hrtimer_forward(timer, ktime_get(), ktime_set(0,TICK_PERIOD));
    return HRTIMER_RESTART;
}
```

```
int init_module(void){
    ktime_t ktime;

    ktime = ktime_set(0, TICK_PERIOD);
    hrtimer_init(&hr_timer, CLOCK_MONOTONIC, HRTIMER_MODE_REL);
    hr_timer.function = &my_hrtimer_callback;

    hrtimer_start(&hr_timer, ktime, HRTIMER_MODE_REL);
    return 0;
}

void cleanup_module(void){
    hrtimer_cancel( &hr_timer );
    return;
}
```

Listing C.2: Linux User Space POSIX timers

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/io.h>
#include <sys/time.h>
#include <signal.h>
#include <time.h>
#include <sched.h>

#define LPT 0x378
#define TICK_PERIOD 1000000

short stopflag=1;

void stop(int signum){

    printf("\nCaught␣signal␣%d,␣exiting\n", signum);
    stopflag=0;
}
```

```
static void square_wave(int signum){
    static char byte=0x00;

    outb(byte,LPT);
    byte=~byte;
}

int main(void){
    struct sched_param proc_sched;
    struct itimerspec it1;
    struct sigaction s1,s2;
    timer_t timer1;

    //On SIGINT (Ctrl-C), call stop()
    signal(SIGINT, stop);
    signal(SIGTERM, stop);

    if (ioperm(LPT,1,1))
        fprintf(stderr, "Couldn't get the port at %x, are you root?\n", LPT), exit(1);

    //what is the maximum priority number for a SCHED_FIFO process?
    proc_sched.sched_priority = sched_get_priority_max(SCHED_FIFO);
    //set SCHED_FIFO and set max priority for process
    sched_setscheduler(0, SCHED_FIFO, &proc_sched);

    //get current scheduler
    if(sched_getscheduler(0)!=SCHED_FIFO)
        fprintf(stderr, "Couldn't set scheduling policy to SCHED_FIFO\n"), exit(1);

    //Define sigalarm handler (event generated by itimer)
    s1.sa_handler=square_wave;
    sigemptyset(&s1.sa_mask);
    s1.sa_flags=0;
    sigaction( SIGALRM, &s1, &s2);
```

```
    //Define itimer to peridodicaly activate task
    it1.it_value.tv_nsec=TICK_PERIOD;
    it1.it_value.tv_sec=0;
    it1.it_interval.tv_nsec=TICK_PERIOD;
    it1.it_interval.tv_sec=0;

    timer_create(CLOCK_REALTIME, NULL, &timer1);
    timer_settime(timer1, 0, &it1, NULL);

    while(stopflag){
        pause();
    }
    timer_delete(timer1);

    return 0;
}
```

Listing C.3: Xenomai Kernel Space

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <asm/io.h>
#include <nucleus/module.h> //xenomai module functionality
#include <native/task.h> //same as user space

MODULE_LICENSE("GPL");
#define TICK_PERIOD 1000000 //ns
#define LPT 0x378

RT_TASK square_wave_task;

void square_wave(void *arg)
{
    char byte=0x00;

    rt_task_set_periodic(NULL, TM_NOW,TICK_PERIOD);
```

```
    while (1) {
        rt_task_wait_period(NULL);

        outb(byte,LPT);
        byte=~byte;
    }
}

int init_module(void){

    rt_task_create(&square_wave_task, "trivial", 0, 99, 0);
    rt_task_start(&square_wave_task, &square_wave, NULL);

    return 0;
}

void cleanup_module(void){

    rt_task_delete(&square_wave_task);

}
```

Listing C.4: Xenomai User Space

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/io.h>
#include <stdlib.h>
#include <native/task.h>

#define TASK_PRIO 99 /* Highest RT priority */
#define TASK_MODE 0 /* No flags */
#define TASK_STKSZ 0 /* Stack size (use default one) */
#define LPT 0x378
#define TICK_PERIOD 10000000
```

```
RT_TASK square_wave_task;
int stopflag=1;

void stop(int signum){
    printf("\nCaught␣signal␣%d,␣exiting\n", signum);
    stopflag=0;
}

void square_wave(void *arg){
    char byte=0x00;
    short i=0;

    rt_task_set_periodic(NULL, TM_NOW,TICK_PERIOD);

    while (stopflag) {
        rt_task_wait_period(NULL);

        outb(byte,LPT);
        byte=~byte;
    }
}

int main(int argc, char* argv[])
{
    //On SIGINT (Ctrl-C), call stop()
    signal(SIGTERM, stop);
    signal(SIGINT, stop);

    if (ioperm(LPT,1,1))
        fprintf(stderr, "Couldn't␣get␣the␣port␣at␣%x,␣are␣you␣root?\n", LPT), exit(1);

    /* Avoids memory swapping for this program */
    mlockall(MCL_CURRENT|MCL_FUTURE);

    rt_task_create(&square_wave_task, "trivial",TASK_STKSZ,TASK_PRIO,TASK_MODE);
```

```
    rt_task_start(&square_wave_task, &square_wave, NULL);

    pause();

    rt_task_delete(&square_wave_task);
    return 0;
}
```

Listing C.5: RTAI Kernel Space

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <asm/io.h> //Routines for accessing I/O ports
#include <rtai.h>
#include <rtai_sched.h> //rt_task* and *rt_timer

MODULE_LICENSE("GPL");

#define LPT 0x378 //Parallel port address
#define STACK_SIZE 4000
#define TICK_PERIOD 1000000 /* ns */

static RT_TASK thread;

static void square_wave(long int xis){
    char byte=0x00;

    while(1){
        rt_task_wait_period();

        outb(byte,LPT);
        byte=~byte;

    }
}

int init_module(void)
```

```
{

    rt_task_init(&thread, square_wave, 0, STACK_SIZE, 0, 0, 0);
    rt_set_oneshot_mode();

    start_rt_timer(0);
    rt_task_make_periodic_relative_ns(&thread, 0, TICK_PERIOD);

    return 0;
}

void cleanup_module(void)
{
    stop_rt_timer();
    rt_task_delete(&thread);
}
```

Listing C.6: RTAI User Space

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <sys/io.h>
#include <rtai_lxrt.h>

#define TICK_PERIOD 1000000
#define LPT 0x378

int stopflag=1;

void stop(int signum){
    printf("\nCaught␣signal␣%d,␣exiting\n", signum);
    stopflag=0;
}

int main(void){
    char byte=0x00;
```

```
    RT_TASK *task;
    RTIME internal_count;

    if (ioperm(LPT,1,1))
    fprintf(stderr, "Couldn't get the port at %x, are you root?\n", LPT), exit(1);

    //On SIGINT (Ctrl-C), call stop()
    signal(SIGINT, stop);

    //Create RT_TASK with name MYTASK and change scheduling policy to SCHED_FIFO
    task = rt_task_init_schmod(nam2num("MYTASK"), 9, 0, 0, SCHED_FIFO, 0xF);

    rt_set_oneshot_mode();
    internal_count = start_rt_timer(nano2count(TICK_PERIOD));

    //lock all memory so we don't have paging (and therefore, no page faults)
    mlockall(MCL_CURRENT | MCL_FUTURE);

    rt_make_hard_real_time();

    rt_task_make_periodic_relative_ns(task,0+0*TICK_PERIOD,TICK_PERIOD);

    while(stopflag){
        rt_task_wait_period();
        outb(byte,LPT);
        byte=~byte;
    }

    rt_make_soft_real_time();
    stop_rt_timer();
    rt_task_delete(task);

    return 0;
}
```

Appendix D

Xenomai Ubuntu Installation Guide

How-to Install Xenomai in Ubuntu 10.04

Jorge Azevedo
jorge.amado.azevedo@gmail.com

v1.0 – October 22, 2011

Abstract

This *how-to* describes one way to install Xenomai 2.5.5.2 on Ubuntu (or *Any*buntu) 10.04.3 (Lucid Lynx) with Linux kernel 2.6.32.

Disclaimer

The author makes no representations or warranties with respect to the contents or use of this document. Use it at your own risk.

1 Introduction

This guide is intended to help installing Xenomai in an Ubuntu based Linux system. As it specifically provides commands to this distribution and version, the process should be quite similar on other Debian based distros. This Ubuntu version comes with 2.6.32.x kernel version, so a 2.6.32 kernel should be used.

This guide is a variant of "How-to Install RTAI in Ubuntu Hardy" by Cristóvão Sousa, who was kind enough to release his original work under a Creative Commons license and send me his source files.

2 Xenomai Libraries

The first step in installing Xenomai is installing its libraries.

- All the necessary packages for compiling, building and installing the Xenomai libraries can be installed via the following commands

```
sudo apt-get install devscripts debhelper dh-kpatches
```

- Got to Downloads folder

```
cd ~/Downloads
```

- Download and untar Xenomai

```
wget -O - http://download.gna.org/xenomai/stable/xenomai-2.5.5.2.tar.bz2 | tar -jxf -
```

- Go to its folder

```
cd xenomai-2.5.5.2
```

- Now we're ready to build the packages. Debian packages need at least our personal information and a note about versioning. So first we define that

```
DEBEMAIL="your@email" DEBFULLNAME="Your Name" debchange -v 2.5.5.2 Release 2.5.5.2
```

- Now everything is in place, and all we have to do is build the packages

```
debuild -uc -us
```

- The resulting packages will be produced in the parent folder, so you can install them by doing

```
sudo dpkg -i ../*.deb
```

- Now we can check which kernel version are supported by our Xenomai version

```
ls -1 /usr/src/kernel-patches/diffs/xenomai
```

- Typically, the output will look something like this

```
adeos-ipipe-2.6.30-arm-1.15-03.patch.gz
adeos-ipipe-2.6.31-arm-1.16-02.patch.gz
adeos-ipipe-2.6.32.20-x86-2.7-03.patch.gz
adeos-ipipe-2.6.33-arm-1.18-00.patch.gz
adeos-ipipe-2.6.34.4-powerpc-2.10-05.patch.gz
adeos-ipipe-2.6.34.5-x86-2.7-04.patch.gz
adeos-ipipe-2.6.35.7-powerpc-2.11-02.patch.gz
adeos-ipipe-2.6.35.7-x86-2.7-04.patch.gz
```

- Since our version of Ubuntu ships with kernel 2.6.32, we're gonna choose version 2.6.32.20. Adapt accordingly.

3 Kernel

- These packages will prepare your system for building a custom kernel package (these will take up almost 300mb of disk space, so consider yourself warned).

```
sudo apt-get build-dep --no-install-recommends linux-image-2.6.32-21-generic
sudo apt-get install libncurses5-dev kernel-package
```

- Linux source code is usually located in `/usr/src`, so let's start by going there

```
cd /usr/src
```

- Download and untar kernel source for version 2.6.32.20

```
sudo wget http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.32.20.tar.bz2
```

```
sudo tar -jxf linux-2.6.32.20.tar.bz2
```

- Enter the source folder

```
cd linux-2.6.32.20
```

- Apply the patch

```
sudo /usr/src/kernel-patches/i386/apply/xenomai
```

- Now our ready for configuration. Instead of starting a configuration from scratch, our approach is to copy the original Ubuntu kernel .config to our folder and start configuration from there. This way we know for sure we have a kernel configuration that boots in our computer.

```
sudo cp ../linux-headers-2.6.32-33-generic/.config .config
```

- Since there's a slight missmatch in versions between the Ubuntu kernel from which we got the .config and our own kernel, we need to account for the possivel difference in configuration options. We do this by running the following. It is usually safe to just press enter and leave the different configurations at their default values.

```
sudo make oldconfig
```

- We're now ready to configure our new Xenomai kernel.

```
sudo make menuconfig
```

- This is where the main difficulty of installing Xenomai resides. The Linux kernel has many different options and some of them are not compatible with Xenomai. Which options are problematic is something that varies between systems. The basic recommended setup is the following

  – `Processor type and features`
    `Processor family` = choose yours
    `[ ] Enable -fstack-protector buffer overflow detection`
  – `Power management and ACPI options`
    `[ ] Cpu Frequency scaling`
  – `- ACPI (Advanced Configuration and Power Interface) Support`
    `- < > Processor`

- Processor family

  Here you should choose your system's processor family. It is not recommended that you leave the default i586 option since, althought it is generic, it may present problems to Xenomai. (ref: xenomai.org)

- Enable -fstack-protector buffer overflow detection

  This is an Ubuntu kernel specificity. Without this option disabled, compilation will fail do to an incompatibility between our options and the Ubuntu default options.

- < > Processor

- Cpu Frequency scaling

  These options try to make the processor run as predicatibily as possible. By deactivating the processor ACPI support, the processor isn't thrown into deep sleep states when inactive. By deactivating frequency scaling support, the clock frequency does not vary as a function of system load.

After all changes have been done, exit and say yes to save the configuration.

At this point the kernel source is ready for compilation. The next command will compile the kernel and generate debian packages. You should adjust concurrency level (number of paralell jobs) to your machine. Some suggest that it should be the number of cores plus one for maximum cpu usage. You should also adjust the –append-to-version parameter to fit your needs. In this case, the generated kernel version will be `2.6.32.20-xenomai-2.5.5.2`.

```
sudo CONCURRENCY_LEVEL=5 CLEAN_SOURCE=no fakeroot make-kpkg --initrd \
--append-to-version -xenomai-2.5.5.2 --revision 1.0 kernel_image kernel_headers
```

This can take from 30 minutes to 3 hours depending on the machine you are using. It can also consume up to 4 GB of disk space. You will see many warning messages, don't worry.

- Two deb packages are generated in `/usr/src/`, kernel image and source headers. Installation is as per usual

```
sudo dpkg -i ../*.deb
```

- Finaly, we need to manually generate an initramfs or otherwise the system won't boot. This is a specificity of Ubuntu 10.04 and beyond. Previous versions of Ubuntu (and Debian) worked in a different manner.

```
sudo update-initramfs -c -k 2.6.32.20-xenomai-2.5.5.2 && sudo update-grub
```

4 Xenomai Test

If everything goes smoothly in the above steps Xenomai is installed. To test it, reboot the computer and choose the new Xenomai kernel. If it boots properly, then execute the latency test:

```
cd /usr/share/libxenomai-dev/examples/native/
sudo make
sudo ./trivial-periodic
```

Press `Ctrl-C` to stop. A typical output from a successfull instalation looks like

```
Time since last turn: 1000.145987ms
Time since last turn: 1000.123452ms
Time since last turn: 999.123545ms
Time since last turn: 1000.123432ms
Time since last turn: 999.12332ms
Time since last turn: 999.87643ms
```

trivial-periodic is a simple application with a period of one second. On each activation, the time elapsed since the last activation is displayed on screen. In essence, what we see is the variation of the application's period. This is a gross estimation of the sheduling jitter, but it's a sufficient test to assert the validity of our installation. If there are wild variations on theses values, something is wrong.

There's a small trick to allow non root user access to Xenomai. During installation, a user group called "xenomai" was added to the system with group id number 125 (you can check it in /etc/group). First we'll add ourselves to that group

```
sudo usermod -a -G xenomai username
```

Where "username" should be changed accordingly.

Now we need to pass the group id as a special boot parameter to the system. This way, all users belonging to the "xenomai" group will have access to real-time performance. The boot parameter is passed through grub, so we edit its default behavior

```
sudo gedit /etc/default/grub
```

And we edit the line that says

```
GRUB_CMDLINE_LINUX_DEFAULT="quiet splash"
```

To

```
GRUB_CMDLINE_LINUX_DEFAULT="quiet splash xeno_nucleus.xenomai_gid=125"
```

The we update grub

```
sudo update-grub
```

On restart, trivial-periodic should now run without sudo.

If your kernel fails to boot or refuses to work in a normal, usefull fashion, please refer to for help. One common workaround is disabling MSI (under Bus Options) and other more advanced options.

5 Conclusion

That's all. Your machine is ready for real time. If some problems arise during the process you can try to solve them by searching the internet for the error.

6 Further reading

Here is a list of helpful documents for further help on Xenomai.

- **Xenomai- Building Debian Packages** (on which this guide was based) contains the same content but under a different lens.
- **Xenomai - Configuring x86 kernels** is the "go to" resource on kernel compilations. Some of the more obscure kernel options are mentioned in this document.
- **Xenomai FAQ** at also contains further information on kernel configuration, under the section "Tips and tricks".

Appendix E

RTAI Ubuntu Installation Guide

How-to Install RTAI in Ubuntu 10.04

Jorge Azevedo
jorge.azevedo@gmail.com

v1.0 – April 21, 2011

Abstract

This *how-to* describes one way to install RTAI 3.8.1 on Ubuntu (or *Any*buntu) 10.04 (Lucid Lynx) with Linux kernel 2.6.32.

Disclaimer

The author makes no representations or warranties with respect to the contents or use of this document. Use it at your own risk.

1 Introduction

This guide is intended to help installing RTAI in an Ubuntu based Linux system. As it specifically provides commands to this distribution and version, the process should be quite similar on other Debian based distros. This Ubuntu version comes with 2.6.32.x kernel version, so a 2.6.32 kernel should be used[1].

This guide is an updated version of "How-to Install RTAI in Ubuntu Hardy" by Cristóvão Sousa, who was kind enough to release his original work under a Creative Commons license and send me his source files.

2 Preparation

The first step in installing RTAI is installing the necessary packages from the repositories. These will prepare your system for building a custom kernel. The second step is downloading the necessary source code.

- All the necessary packages for compiling, configuring and generating a package can be installed via the following commands

```
sudo apt-get build-dep linux-image-2.6.32-21-generic
sudo apt-get install libncurses5-dev kernel-package
```

[1] Please note that the latest 64 bit kernel supported by RTAI is version 2.6.23.

- Linux source code is usually located in `/usr/src`, so let's start by going there

```
cd /usr/src
```

- Download and extract Linux kernel 2.6.32.11

```
wget -O - www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.32.11.tar.bz2 | sudo tar -jxf -
```

- Download and extract RTAI 3.8.1

```
wget -O - https://www.rtai.org/RTAI/rtai-3.8.1.tar.bz2 | sudo tar -jxf -
```

- Rename linux source directory to a more descriptive name (optional):

```
sudo mv linux-2.6.32.11 linux-2.6.32.11-rtai-3.8.1
```

- Create symbolic links to the two new folders (you can choose other names if you want):

```
sudo ln -snf linux-2.6.32.11-rtai-3.8.1 linux
sudo ln -snf rtai-3.8.1 rtai
```

3 Kernel Patching and Configuration

- Patch the kernel source with the correspondent RTAI patch:

```
cd /usr/src/linux
sudo patch -p1 -b < /usr/src/rtai/base/arch/x86/patches/hal-linux-2.6.32.11-x86-2.6-03.patch
```

- In order to make the new kernel configuration the most similar to the already installed one you can do the two next steps:

```
sudo cp /boot/config-`uname -r` ./.config
sudo make oldconfig   # then press Enter on all prompts
```

  Press Enter on all the prompts of the last command.

- Now it's time to configure the new kernel:

```
sudo make menuconfig
```

  In the menu there are many parameters you can change if you know what you are doing. The changes needed by RTAI are:

– [ ] Enable loadable module support
  [ ] Module versioning support
– Processor type and features
  [*] Interrupt pipeline
  Processor family = choose yours
  Symmetric multi-processing support = no/yes

If your machine has a single core processor uncheck this last option. If your machine has multicore you can choose to use all or only one[2].

Because power management is a "latency killer" you should turn it off:

– Power management and ACPI options
  [ ] Power Management support
  [ ] CPU Frequency scaling > CPU Frequency scaling

After all changes exit and say yes to save the configuration.

- Optionally you can make a backup for the new configuration file:

```
cd /usr/src/linux ; sudo cp .config /boot/config-2.6.24-rtai-3.6.1
```

4 Kernel Compilation

- First prepare for a clean compilation

```
sudo make-kpkg clean
```

- At this point the kernel source is ready for compilation. The next command will compile the kernel and generate debian packages. You should adjust concurrency level (number of paralell jobs) to your machine. Some suggest that it should be the number of cores plus one for maximum cpu usage. You should also adjust the –append-to-version parameter to fit your needs. In this case, the generated kernel version will be `2.6.32.11-rtai-3.8.1`.

```
sudo CONCURRENCY_LEVEL=5 CLEAN_SOURCE=no fakeroot make-kpkg --initrd \
--append-to-version -rtai-3.8.1 kernel_image kernel_headers
```

This can take from 30 minutes to 3 hours depending on the machine you are using. It can also consume up to 4 GB of disk space. You will see many warning messages, don't worry. If this command ends with an error you should try to fix it and compile again (the internet is your friend).

- Two deb packages are generated in `/usr/src/`, kernel image and source headers. If you wish you can free the disk space used in the compilation with `sudo make-kpkg clean`.

[2] In case you want to simulate a single core machine.

5 Kernel Installation

- Go to the packages directory:

```
cd /usr/src/
```

- Install the kernel image and source headers packages:

```
sudo dpkg -i linux-headers-2.6.32.11-rtai-3.8.1_2.6.32.11-rtai-3.8.1-10.00.Custom_i386.deb
sudo dpkg -i linux-image-2.6.32.11-rtai-3.8.1_2.6.32.11-rtai-3.8.1-10.00.Custom_i386.deb
```

- Although the package installation creates a new grub entry, it doesn't create a new initramfs image and therefore the kernel is unbootable. Notice that all kernels installed in your system can be listed with

```
ls /lib/modules
```

  To generate a new initramfs and update grub run

```
sudo update-initramfs -c -k 2.6.32.11-rtai-3.8.1 && sudo update-grub
```

- Due to a bug in Grub 2, the system is still unbootable. While removing Grub 2 and installing Grub 1 will solve the issue, a more practical solution is to edit directly the grub.cfg entry for the newly compiled kernel and add 16 to end of the word linux and initrd. So, with your favorite editor you can edit grub.cfg

```
sudo vim /etc/grub/grub.cfg
```

  And change the entry so that instead of

```
linux   /boot/vmlinuz-2.6.32.11-rtai-3.8.1 (...)
initrd  /boot/initrd.img-2.6.32.11-rtai-3.8.1
```

  it reads

```
linux16   /boot/vmlinuz-2.6.32.11-rtai-3.8.1 (...)
initrd16  /boot/initrd.img-2.6.32.11-rtai-3.8.1
```

  You should now force save your changes since grub.cfg is a read only file. Please bear in mind that any call to `update-grub` will generate a fresh grub.cfg. The workaround will have to be applied manually every time.

- Now, you must reboot. Choose the new RTAI patched kernel in Grub. If the process fails, the most likely problem is a faulty kernel configuration. You might be interested in consulting section 9.

6 RTAI Configuration and Installation

- Creating a build tree separated from the source tree of RTAI is advised by RTAI people, so do

```
cd /usr/src/rtai
sudo mkdir build
cd build
```

- Configure and compile RTAI:

```
sudo make -f ../makefile menuconfig
```

  - General > Linux source tree = /usr/src/linux-2.6.32.11-rtai-3.8.1
  - Machine (x86) > Number of CPUs (SMP-only) = the right number (if applicable)

  At the end exit and say yes to save the configuration. RTAI will be compiled.

- If there were no errors, install RTAI:

```
sudo make install
```

- Configure RTAI dynamic libraries to be wide available:

```
sudo -s
echo /usr/realtime/lib/ > /etc/ld.so.conf.d/rtai.conf
exit
sudo ldconfig
```

- The RTAI binaries directory can be added automatically to the $PATH variable. To do that, add the line

  PATH="$PATH:/usr/realtime/bin"

  to the end of ~/.profile (user) and/or /root/.profile (root). It also comes in handy to define an alias for sudo so that it inherits the $PATH variable. To do this, add

  alias sudo="sudo env PATH=$PATH"

  to ~/.bashrc.

7 RTAI Test

If everything goes smoothly in the above steps RTAI is readily installed. To be sure execute the latency test:

```
cd /usr/realtime/testsuite/kern/latency/
sudo ./run
```

Press `Ctrl-C` to stop. A typical output from a successfull instalation looks like

```
RTAI Testsuite - KERNEL latency (all data in nanoseconds)
RTH|    lat min|    ovl min|    lat avg|    lat max|    ovl max|   overruns
RTD|      -1566|      -1566|      -1525|       -995|       -995|          0
RTD|      -1571|      -1571|      -1489|       8257|       8257|          0
RTD|      -1569|      -1571|      -1527|        -89|       8257|          0
RTD|      -1566|      -1571|      -1525|       -481|       8257|          0
RTD|      -1566|      -1571|      -1529|      -1049|       8257|          0
RTD|      -1566|      -1571|      -1529|       -939|       8257|          0
```

8 Conclusion

That's all. Your machine is ready for real time. If some problems arise during the process you can try to solve them by searching the internet for the error sentence.

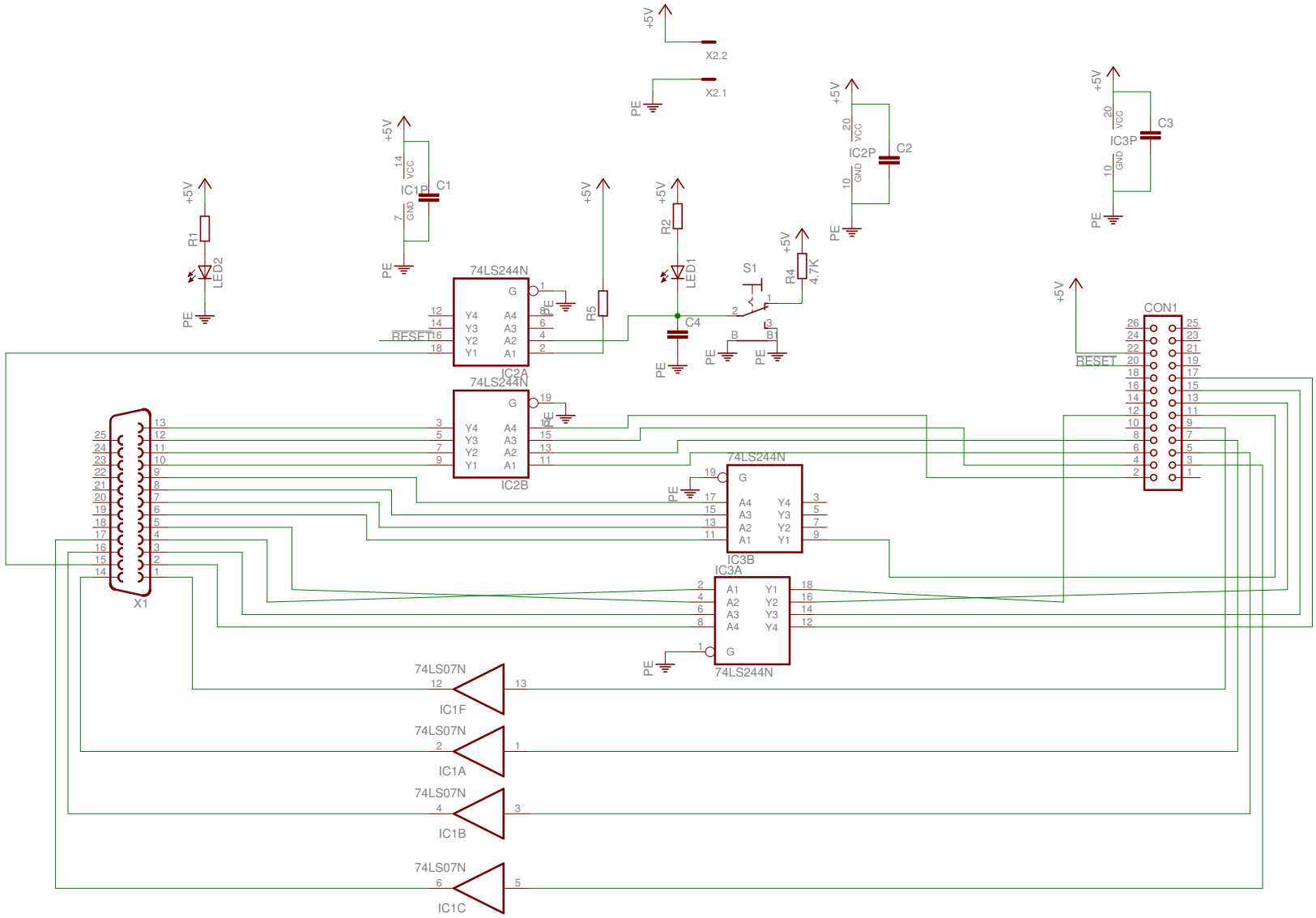
9 Further reading

Here is a list of helpful documents for further help on RTAI.

- **How-to Install RTAI in Ubuntu Hardy** (on which this guide was based) and **RTAI Tutorial** by Cristóvão Sousa contain valuable information about both installation and programming using the RTAI API.
- **RTAI Installation Complete Guide** and **Building Your Way Through RTAI** by João Monteiro approach the same matters as the previous set of documents, but with a more in-depth view of the RTAI API.
- A **Guide to Installing RTAI Linux** by Keith Shortridge is a more extensive look at the installation procedure than the more practical approach developed in this guide.
- **RTAI 3.8 on Ubuntu(9.10)-Linux-kernel : 2.6.31.8** by Manuel Arturo Deza is another installation guide.

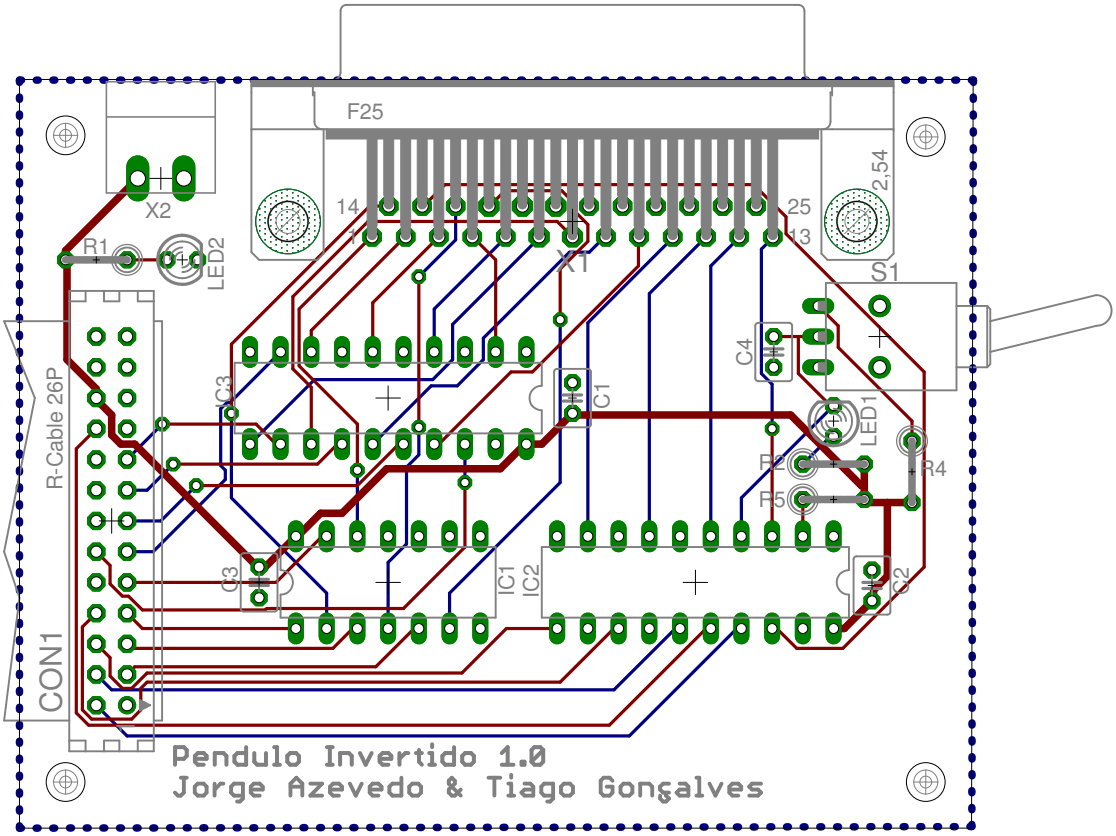
Appendix F

Inverted Pendulum Schematic


+5V
X2.2
X2.1
PE
IC1P
C1
VCC
GND
14
7
IC2P
C2
20
10
IC3P
C3
R1
LED2
R2
LED1
R4
4.7K
R5
S1
C4
74LS244N
IC2A
IC2B
IC3A
IC3B
RESET
CON1
X1
74LS07N
IC1F
IC1A
IC1B
IC1C

Appendix G

Inverted Pendulum Printed Circuit Board


F25
14
25
13
2,54
X2
R1
LED2
XT
S1
C4
C3
C1
LED1
R2
R5
R4
IC1
IC2
C2
R-Cable 26P
CON1
Pendulo Invertido 1.0
Jorge Azevedo & Tiago Gonçalves

Bibliography

[1] K. Ogata, *Modern Control Engineering*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 4th ed., 2001.

[2] E. S. Raymond, *The Cathedral and the Bazaar*. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 1st ed., 1999.

[3] R. Love, *Linux kernel development*. Developer's Library, Addison-Wesley, 2009.

[4] S. J. Vaughan-Nichols, "Linux: It doesn't get any faster." `http://blogs.computerworld.com/linux_it_doesnt_get_any_faster`, June 2009.

[5] J. Niccolai, "Ballmer still searching for an answer to google." `http://www.pcworld.com/businesscenter/article/151568/ballmer_still_searching_for_an_answer_to_google.html`, September 2008.

[6] B. Byfield, "Linux desktop market share: Greater than one percent?." `http://itmanagement.earthweb.com/open-source/Linux-Desktop-Market-Share-Greater-Than-One-Percent-3818696.htm`, May 2009.

[7] J. D'Azzo, C. Houpis, and S. Sheldon, *Linear control system analysis and design with MATLAB*. No. v. 1 in Control engineering, Marcel Dekker, 2003.

[8] K. Ogata, *Discrete-time control systems*. Prentice Hall, 1995.

[9] A. L. L. Antunes, *Algoritmos de Controlo Distribuído em Sistemas Baseados em Microprocessadores*. PhD, Departamento de Electrónica, Telecomunicacoes e Informática – Universidade de Aveiro (DETI/UA), 2008.

[10] D. Gillies, "Real-time computing faq." `http://www.faqs.org/faqs/realtime-computing/faq/`.

[11] K. Yaghmour, *Building embedded Linux systems*. O'Reilly Series, O'Reilly, 2003.

[12] V. Yodaiken *et al.*, "The rtlinux manifesto," in *Proc. of the 5th Linux Expo*, 1999.

[13] K. Koolwal and R. Engineer, "Myths and realities of real-time linux software systems," in *Proc. Real-Time Linux Workshop (RTLWS 2009)*, 2009.

[14] W. River, "Vxworks offical website." `http://www.windriver.com/products/vxworks/`.

[15] Q. S. Systems, "Qnx offical website." `http://www.qnx.com/`.

[16] L. Bic and A. Shaw, *Operating systems principles*. Prentice Hall, 2003.

[17] C. Liu and J. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM (JACM)*, vol. 20, no. 1, pp. 46–61, 1973.

[18] V. J. Yodaiken, "Adding real-time support to general purpose operating systems," 11 1999.

[19] DIAPM, "Rtai official website." `http://www.rtai.org/`.

[20] Xenomai, "Xenomai official website." `http://www.xenomai.org/`.

[21] P. Marquet, É. Piel, J. Soula, J. Dekeyser, *et al.*, "Implementation of artis, an asymmetric real-time extension of smp linux," in *Sixth Real-Time Linux Workshop, Singapore*, 2004.

[22] T. Gleixner and D. Niehaus, "Hrtimers and beyond: Transforming the linux time subsystem," *Proc. Linux Symposium, Ottawa, Ontario, Canada*, July 2006.

[23] T. Gleixner, "hrtimer - high-resolution timer subsystem." `http://lwn.net/Articles/162773/`, December 2005.

[24] T. Gleixner, "ktimer subsystem." `http://lwn.net/Articles/152363/`, September 2005.

[25] T. Gleixner, "A new approach to kernel timers." `http://lwn.net/Articles/152436/`, September 2005.

[26] T. Gleixner, "Clockevents and dyntick." `http://lwn.net/Articles/223185/`, February 2007.

[27] PREEMPTRT, "Preemptrt official website." http://rt.wiki.kernel.org/.

[28] P. McKenney, "A realtime preemption overview." http://lwn.net/Articles/146861/, August 2005.

[29] J. Cobert, "The realtime preemption endgame." http://lwn.net/Articles/345076/, August 2009.

[30] S. Rostedt and D. V. Hart, "Internals of the rt patch," *Proc. Linux Symposium, Ottawa, Ontario, Canada*, June 2007.

[31] M. Barabanov, "A linux-based real-time operating system," Master's thesis, New Mexico Institude of Mining and Technology, June 1997.

[32] P. Cloutier, P. Mantegazza, S. Papacharalambous, I. Soanes, S. Hughes, and K. Yaghmour, "Diapm-rtai position paper," *RTSS 2000 - Real Time Operating Systems Workshop*, 2000.

[33] P. Gerum, "Announce: xenodaptor." http://www.mail-archive.com/rtl@fsmlabs.com/msg01156.html, August 2001.

[34] Xenomai, "Project history." http://www.xenomai.org/index.php/Xenomai:History.

[35] J. Cobert and E. O. Coolbaugh, "Real-time linux is patented." http://lwn.net/2000/0210/, February 2000.

[36] J. Cobert and E. O. Coolbaugh, "The rtlinux patent." http://lwn.net/2001/0215/, February 2001.

[37] J. Cobert and E. O. Coolbaugh, "The other free software war." http://lwn.net/2000/0914/, September 2000.

[38] L. Devices, "Ceo interview: Victor yodaiken, fsmlabs." http://www.linuxfordevices.com/c/a/Linux-For-Devices-Articles/CEO-Interview-Victor-Yodaiken-FSMLabs/, December 2003.

[39] K. Yaghmour, "Adaptive domain environment for operating systems," *Opersys inc*, 2001.

[40] M. Roy, "Interview with philippe gerum." http://www.advogato.org/article/803.html, October 2004.

[41] L. Dozio and P. Mantegazza, "Real time distributed control systems using rtai," in *Object-Oriented Real-Time Distributed Computing, 2003. Sixth IEEE International Symposium on*, pp. 11–18, IEEE, 2003.

[42] P. Mantegazza, "What's ahead." http://article.gmane.org/gmane.linux.real-time.rtai/7754, April 2004.

[43] P. Gerum, "A tour of the native api." http://www.xenomai.org/documentation/xenomai-2.0/pdf/Native-API-Tour.pdf, 2006.

[44] A. Barbalace, A. Luchetta, G. Manduchi, M. Moro, A. Soppelsa, and C. Taliercio, "Performance comparison of vxworks, linux, rtai and xenomai in a hard real-time application," in *Real-Time Conference, 2007 15th IEEE-NPSS*, pp. 1 –5, 29 2007-may 4 2007.

[45] DIAPM, "Rtai 3.4 user manual." https://www.rtai.org/index.php?module=documents&JAS_DocumentManager_op=viewDocument&JAS_Document_id=44, 2006.

[46] M. Piątek, "Real-time application interface and xenomai modified gnu/linux real-time operating systems dedicated to control,"

[47] W. Betz, M. Cereia, and I. Bertolotti, "Experimental evaluation of the linux rt patch for real-time applications," in *Emerging Technologies Factory Automation, 2009. ETFA 2009. IEEE Conference on*, pp. 1 –4, sept. 2009.

[48] M. Chiandone and G. Sulligoi, "Performance assessment of a motion control application based on linux rtai," in *Power Electronics Electrical Drives Automation and Motion (SPEEDAM), 2010 International Symposium on*, pp. 687 –692, june 2010.

[49] J. Blanchette and M. Summerfield, *C++ GUI programming with Qt 4*. Prentice Hall open source software development series, Prentice Hall in association with Trolltech Press, 2008.

[50] P. O. Kristensson, "Ansi c applications settings management." http://pokristensson.com/settings.html.

[51] Boost, "Boost c++ library." http://www.boost.org/.

[52] J. Elson, "Parapin – a parallel port pin programming library for linux." `http://parapin.sourceforge.net/`.

[53] Microchip Technology Inc., *PIC18FXX8 Data Sheet*, 2003.

[54] Microchip Technology Inc., *PICmicro^{TM}Mid-Range MCU Family Reference Manual*, 1997.