UNIVERSIDAD AUTÓNOMA DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

Máster Universitario en Investigación e Innovación en TIC (i2-TIC)

TRABAJO FIN DE MÁSTER

REAL-TIME SOFTWARE TECHNOLOGY AND ITS USE IN EXPERIMENTAL NEUROSCIENCE

Autor: Rodrigo Vicente Amaducci Szwarc
Tutor: Pablo Varona Martínez

Junio 2017

REAL-TIME SOFTWARE TECHNOLOGY AND ITS USE IN EXPERIMENTAL NEUROSCIENCE

Autor: Rodrigo Vicente Amaducci Szwarc
Tutor: Pablo Varona Martínez

Grupo de Neurocomputación Biológica (GNB)
Escuela Politécnica Superior
Universidad Autónoma de Madrid
Junio 2017

Resumen

Resumen

Debido a las complicadas dinámicas no lineales de los sistemas neuronales así como a la incapacidad existente a la hora de observar más de unas pocas de las señales que participan en dichas dinámicas de forma simultánea, el estudio de estos sistemas es muy complejo. Además, el paradigma tradicional de trabajo es el de estímulo-respuesta, en el cual se registra el comportamiento del sistema al responder a ciertos estímulos de entrada y se estudian estos resultados a posteriori, lo que impide caracterizar completamente la dinámica de su funcionamiento. Las tecnologías de ciclo cerrado permiten superar estas adversidades mediante la observación, el control y la interacción bidireccional con estos elementos neuronales. Sin embargo, la implementación de este tipo de tecnologías no es tan sencilla debido a que en muchos casos la detección y estimulación del sistema biológico debe hacerse de acuerdo a ciertas restricciones temporales precisas. Esta capacidad del sistema para ejecutar tareas y responder a eventos externos (síncronos o asíncronos) en una determinada franja de tiempo es lo que se conoce como funcionamiento en "tiempo real".

Los ordenadores personales actuales poseen la suficiente potencia de procesamiento como para cumplir con los requisitos de tiempo real, sin embargo debido al funcionamieno de los planificadores de los sistemas operativos de proposito general (Windows, Linux, MacOS), que no puede ser controlado por el usuario, no existe manera de asegurar que un proceso de tiempo real se ejecutará sin interrupciones y cumpliendo con las restricciones temporales. Por otra parte, las implementaciones en hardware pueden cumplir con dichas restricciones temporales pero son menos programables. Por ello, existen también los llamados sistemas operativos de tiempo real (RTOS). Sin embargo, esta tecnología es a menudo difícil de instalar, configurar y manejar. Estas dificultades relativas a los RTOS provocan que muchos equipos y laboratorios dedicados a la neurociencia no vean viable invertir tiempo y esfuerzo en dominar esta tecnología para realizar experimentos de ciclo cerrado.

En este trabajo se realiza una comparativa cuantitativa de las herramientas para tiempo real RTAI, Xenomai y Preempt-RT, de acuerdo a su rendimiento así como su usabilidad y accesibilidad, en la que se compara sus valores de latencia y la variabilidad (jitter) de estos. La comparativa se lleva a cabo en el contexto del uso de la tecnología de tiempo real en neurociencia experimental. Además se ha desarrollado una librería de modelos neuronales y sinápticos en tiempo real para su uso en circuitos híbridos, con neuronas vivas y modelos artificiales, y experimentos de ciclo cerrado. El correcto funcionamiento de dicha librería ha sido probado mediante su integración en circuitos híbridos, tanto con neuronas vivas como electrónicas, así como con el manejo de un motor de pasos para la estimulación mecánica.

Palabras Clave

Tecnología software en tiempo real, Tecnologías de ciclo cerrado en neurociencia, Circuitos híbridos, Modelos neuronales, RTAI, Xenomai, Preempt-RT

Abstract

Due to the complicated non linear dynamics of neuronal systems, as to the existing inability to observe simultaneously more than a few signals of the ones involved in said dynamics, the study of these systems is quite complex. Moreover, traditionally the working paradigm is the stimulus-response one, where the system behaviour is recorded while it responds to certain input stimuli and the results are studied afterwards, thus preventing the complete characterization of the behavioural dynamics. Closed-loop technologies allow to overcome these difficulties through online observation, control and bidirectional interaction with these neural elements. Nevertheless, implementing this kind of technologies is not an easy task because in many cases the detection and stimulation must be done within some precise temporal boundaries. This ability of the system to complete tasks and respond to external events (synchronous and asynchronous) within a determined time slot is known as "real-time" performance.

Actual computers have enough processing power and speed to comply with real-time requirements, but due to the general purpose operating systems (Windows, Linux, MacOS) schedulers' behaviour, which can not be controlled by the user, there is no way to ensure that a real-time process will be run without interruptions and respecting the temporal restrictions. On the other hand, hardware implementations can fullfil such boundaries, but are also less programmable. For this reason the real-time operating systems (RTOS) exist. However, this technology is often difficult to install, configure and use. This RTOS-related complications provoke that many neuroscience researching teams and laboratories do not consider feasible to spend time and effort to implement this tools for closed-loop experiments.

In this work a quantitative comparison between the real-time solution RTAI, Xenomai and Preempt-RT is carried out, focusing on their performance, usability and accessibility, by comparing their latency values and jitter. The comparison done in the context of real-time software technology usage in experimental neuroscience. Furthermore, a real-time neuron and synapse model library was developed for its use in hybrid circuits and closed-loop experiments. To validate the correct functioning of said library it was used in hybrid circuits, with both electronic and living neurons, and to control a stepper motor for mechanical stimulation.

Key words

Real-time sfotware technology, Closed-loop technology in neuroscience, Hybrid circuits, Neuron model, RTAI, Xenomai, Preempt-RT, Closed-loop technology in neuroscience

Acknowledgements

Para empezar, agradecer al Grupo de Neurocomputación Biológica el que me haya dado la oportunidad de introducirme en el mundo de la investigación y la neuroinformática,. Especialmente a mi tutor Pablo, por la confianza y apoyo brindados todo este tiempo, así como a todos mis compañeros del laboratorio, por toda la ayuda que me han proporcionado, pero sobretodo por hacer más amenas los largos días de trabajo.

Como siempre, dar las gracias a mis padres, que toda mi vida han sido el punto de referencia, que me han animado y apoyado en mis decisiones y que si hoy estoy escribiendo esto es sobretodo gracias a ellos. Y por supuesto, agradecer a Andrea todo el cariño y el ánimo que me ha dado mientras recorriamos juntos (y seguiremos recorriendo) esta nueva etapa de nuestras vidas.

Contents

Figures index

1 Introduction

1.1 Motivation of the project

Neural systems have non linear dynamics, affected by various learning and adaptation mechanisms, and processes information in different spatial and temporal scales. Moreover, the nervous system is only partially observable, meaning that usually few signals of the ones concerned in the neural dynamics can be accessed at the same time. Along this limited spatial and temporal resolution, most of the experimental neuroscience research is based in traditional stimulus-response paradigms, recording the behaviour of the system under different stimuli and then analysing the collected data offline, so the highly complex non-stationary neural activity, which is influenced by the context and previous events feedback, can not be completely understood. A way to overcome the difficulties of studying these transient dynamics is to interact online with the system through closed-loop technologies, which are able of producing precise stimulus according to the input information and can provide truly valuable insights on the neural processes along with allowing more flexibility in the experiment, and also favour its automation and control of neural dynamics [Varona et al., 2016] [Chamorro et al., 2012].

Nevertheless, closed-loop technology, which allows online observation, control and interaction with the neural elements, presents some difficulties in its implementation. Both the information acquisition and the stimulation of biological components during the experiments often require of the compliance of certain precise temporal restrictions, in the scale of milliseconds or lower. To this effect the system must run in what is known as real-time, which can be defined as the capacity of the system to carry its tasks and respond to asynchronous external events in a limited time slot [Furht et al., 1991]. Hence, the use of real-time technology ensures the accomplishment of the operations within some established temporal boundaries, either tolerating some faults under a determined threshold or none, which is usually the case in the biomedical field.

Delivering a hardware-based real-time implementation is relatively easy, since electronic components are able to fullfil without complications the speed and precision requirements specified in most cases [Robinson and Kawai, 1993] [Le Masson et al., 1995] [Broccard et al., 2017]. However, their main disadvantage is that they are poorly manageable and programmable, in contrast to software solutions, which offer systems and programming languages far more user-friendly and workable. Furthermore, at the beginning of this century personal computers had already achieved a processing power and speed that let them comply with very strict real-time restrictions in hardware capacity terms, despite of some latency generated in the data busses. The problem is that typical modern operating systems are *general purpose operating systems* (GPOS), such as Windows, MacOS or Linux, which are multitask environments with schedulers that are in charge of assigning the computer resources to the different running tasks due to some specific policies, and cannot be handled by the user. Because of this, there is no way of ensuring that a specific task will be run without interruptions and therefore real-time cannot be assured. A special kind of operating systems, the *real-time operating systems* (RTOS), are needed for a task to run within the established time margins and hence to perform software-based real-time experiments.

Numerous RTOS implementations exist, each of them with a different purpose and architecture, and they all present some advantages and disadvantages regarding to the others in various aspects, such as performance, usability and accessibility. This heterogeneity in the RTOS solutions, along with the difficulties they may lead to, causes many neuroscience research teams and laboratories to overlook real-time closed-loop experiments. Therefore, an analysis of the available tools and platforms and how can they be applied to experimental neuroscience could be of great use for such teams and laboratories.

1.2 Project goals

The main goal of this work is to provide an analysis for a future development of a standardized approach for real-time software technology aimed at closed-loop interaction in different aspects of experimental neuroscience, such as electrophysiology or Brain-Computer Interfaces (BCI). It can be divided in the following subtasks:

- Test and comparison of three open-source and free real-time solutions for a GPOS, Linux, analysing their behaviour in different situations and with various system configurations: RTAI, Xenomai and Preempt-RT.
- Study of the viability of the tested RTOS in order to design and develop real-time software aimed at its use in experimental neuroscience, accordingly both to their capacity to fullfil the temporal restrictions (latency, jitter, etc) with speed and precision, as to their accessibility and usability.
- Implementation of a real-time neuron model library in order to use it in hybrid circuits experiments. The paradigms are the Izhikevich, Hindmarsh-Rose and Rulkov Map models. Two types of synapse models are also implemented: electrical and slow and fast gradual chemical synapses.
- Adaptation of the library to communicate the models with National Instruments' data acquisition devices, which act as interfaces between the computer and the real neurons, using the open-source Comedi drivers library.
- Implementation of the library as a stand-alone command-line interface program and performance of validation tests with electronic, living neurons and mechanical stimulation devices.

1.3 Organization of the report

This document is divided in the following chapters:

- **State of the art:** in this chapter the current context of the techniques and tools which will be used in the development of this work will be studied.
- **Real-time solutions comparison:** comparison among the different real-time solutions introduced and described in Chapter 2, focusing on their performance, usability and accessibility.
- **Neuron model library implementation:** in this chapter the design and implementation of the real-time neuron model library will be detailed, also describing the differences between the versions developed for each RTOS.
- **Validation tests:** this chapter will explain the carried out tests in order to validate the correct functioning of the library described in the previous one, and their results will be presented.
- **Conclusions and future work:** in this final chapter the results of the project will be summarized, and the fulfilment of the goals discussed. Additionally, other aspects and ideas will be introduced as possible future work.

2 State of the art

In this chapter the state of the art of the technologies, tools and techniques approached along this project will be studied.

2.1 Real-time in neuroscience

Electrophysiology is the study of electrical properties of biological cells and tissues, such as membrane potential or current changes [Covey and Carter, 2015]. In the field of neuroscience, electrophysiology techniques had been around for almost 200 years, since Galvani discovered that electrical activity was connected to the functioning of the nervous system [Scanziani and Hausser, 2009].

Nevertheless, it was not until sixty years ago when a major scientific progress was made, and became a basic technique for real-time control in electrophysiology [Bauer et al., 2014]. First, a neuron model to describe ions movement through the cell membrane to generate the action potential was designed, based on the demonstration of such potential influence in membrane's ions permeability [Cole and Curtis, 1939] and that during this event the membrane potential surpasses 0 mV [Hodgkin and Huxley, 1939]. In order to better study these phenomena, the *voltage-clamp* technique was developed by Cole and Marmont, and used in Hodgkin and Huxley experiments [Hodgkin et al., 1952]. It uses an intracellular electrode to measure membrane potential, which goes through an operational amplifier which fits the measured value to a fixed control one and, lastly, it is reintroduced as a current back into the neuron using a second electrode, thus forming a closed-loop circuit. Numerous alternative techniques, such as *space-clamp*, *patch-clamp* and *single-electrode voltage-clamp*, have been derived from the original voltage-clamp.

Voltage-clamp allows the user to control membrane potential, keeping it fixed at a specific value. At the 90's a new variant appeared, the *dynamic-clamp* [Sharp et al., 1993] [Destexhe and Bal, 2009], which reads and writes the signal as the standard voltage-clamp, but also performs some operations over it in the middle. This way more complex studies can be carried on than the ones done by fixing the potential, like creating hybrid circuits between real neurons and computer-simulated neuron models connected bidirectionally. However, dynamic-clamp needs a computer able to receive the data, perform the appropriate operations and send it back to the *data acquisition* device (DAQ) with a proper frequency, established by the DAQ, neither faster nor slower.

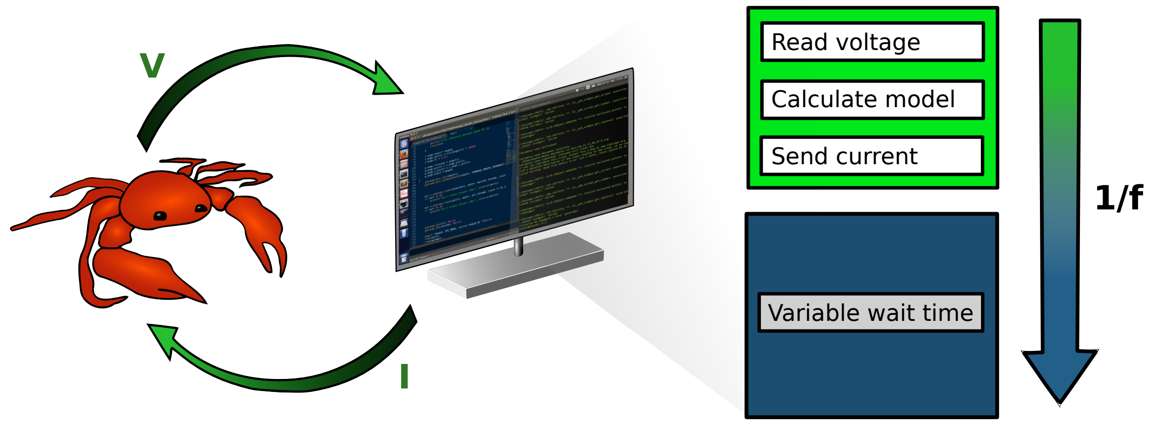


Figure 2.1: Diagram of an hybrid circuit experiment. Membrane potential (V) of the real neuron is obtained using and electrode and sent to a computer, which uses it as input for a computational neuron model and sends back the resulting current (I), which is introduced in the real cell through another electrode. All this process has to be completed within an specific period, due to the neuron activity frequency (f).

This kind of closed-loop experiments may need hard real-time. Nevertheless, not all closed-loop implementations require this technology, as for example in models simulations [Elices and Varona, 2017]. In most cases the researcher wants them to run faster than a realistic situation or maybe the model is so computationally heavy that it cannot run as fast a biological system, but these simulation conditions there are not temporal boundaries. Other scenarios that may just need soft real-time, as the one discussed later in Chapter 5. Even if hard real-time is required, not always an RTOS is necessary, since hardware-based real-time solutions can be available (despite of the mentioned drawbacks) [Franke et al., 2012][Müller et al., 2013][Tessadori et al., 2012].

Although this project focuses on the uses of real-time technology in electrophysiology, other neuroscience aspects can also benefit from it. But first, is important to emphasize that the "real-time" term has different meanings in neuroscience literature. In most cases it refers to an online recording, feedback or control, like for example the explained in [Rana et al., 2016], about *functional magnetic resonance imaging* (fMRI), a technique that allows to record the brain activity with high spatial resolution and in a non invasive way (but has low temporal resolution), where closed-loop experiments can be used to observe the brain response to different stimuli online and change them dynamically. More examples can be found in *optogenetic* [Prsa et al., 2017] or *electroencephalography* (EEG) experiments, like the ones that combine BCIs with virtual reality [Arrouët et al., 2005]. All these stimulation and detection techniques can also use strict hard real-time technology to improve their performance, as are used for different activity-dependent stimulation experiments, like examples which use simultaneous electrophysiological and video tracking [Muñiz et al., 2011], electric signalling [Lareo et al., 2016] or drug microinjection [Chamorro et al., 2009]. In this work we always refer to strict hard real-time.

2.2 Real-Time Operating Systems

A real-time system can be described as one which "controls an environment by receiving data, processing them, and returning the results sufficiently quickly to affect the environment at that time" [Martin, 1965]. Contrary to what most people would likely think when they hear this term, real-time is not a matter of speed (which is of course important, but also not a problem for actual computers in most cases) but of precision: the results must be delivered at the appropriate time, neither sooner nor later. This is precisely the kind of behaviour that many neurons have: slow activity (less than 1 kHz) but precise spiking coding. Therefore a real-time computer is needed in order to work with the desire precision within the temporal bounds set.

There is always some latency in these operations, and some variation (jitter) among these latencies, which depend on the computer and software where the process is running on. Not all the tasks are equally sensitive to latency or data loss, therefore existing two categories of real-time [Shin and Ramanathan, 1994]: *soft/firm real-time*, when some deadlines can be missed and the system still works, but performance will degrade if too many are lost (for example, an online music streaming application can lose a few packages and the service goes on unaffected), and *hard real-time*, when every deadline must be hit or the system fails (the computer systems of a nuclear plant or satellite communication systems, for example).

There are different ways of implementing an RTOS, but all of them have in common that they rely on their scheduling algorithms (many of them are preemptive, which means that it can interrupt a task running on the processor without its permission) and the way they handle hardware interrupts. There are full real-time oriented RTOS and others that are based on GPOS, transforming real-time environments by patching their kernel, which can also be classified in dual-kernel (use a microkernel with the real-time functionalities along the standard one) and single-kernel solution (the standard kernel is transformed to be real-time capable itself).

In the next sections several non-commercial approaches to implement RTOS are described. Commercial solutions are usually discarded in research oriented applications due to funding restrictions and dissemination goals.

2.2.1 RTAI

Real-Time Application Interface, mostly known as RTAI, is one of the first and most extended open-source solutions to get a Linux-based RTOS. Developed since 1996 by Paolo Mantegazza [Mantegazza et al., 2000], professor at the Dipartimento di Ingenieria Aerospaziale of Politecnico di Milano, this dual-kernel patch became one of the most reliable and widely used, and was the cornerstone for later RTOS projects as Xenomai.

Architecture

RTAI functioning is based on a *Hardware Abstraction Layer* (HAL), over which run both Linux standard kernel and a microkernel with the real-time capabilities. Interruptions requests (IRQ) generated by hardware components are first intercepted by RTAI kernel [Barbalace et al., 2008] and, in case they do not alter real-time performance, sent to the HAL, that resend them to Linux. In this structure, Linux is treated as a low priority process managed by RTAI real-time kernel, which handles the interruptions and can preempt the other operating system tasks when considers it necessary.

At first, the HAL implementation used was *RTHAL*, which collects all the pointers to functions and data structures relevant for time critical operations in a single structure, so it is easier to replace all these standard kernel functionalities with RTAI ones when hard real-time is needed. This idea was similar to the used by other RTOS project active at the late 90's, RTLinux [Yodaiken, 1999], and there was dispute for the patent, so to avoid further problems this implementation was replaced in RTAI 3.0 by an open-source one, *ADEOS*, which offered slightly greater maximum latencies but works just as fine [Zhang et al., 2006].

ADEOS handles the different operating systems that are mounted over it as domains with priorities, allowing and managing the shared usage of hardware resources among them [Neelakandan et al., 2005]. Highest priority domain receives the IRQs first and, if it is not interested, sends them to the pipeline, which propagates them to the next highest domain, and so on.

How it works

Four different schedulers are included in RTAI [Dozio and Mantegazza, 2003], with preemptive implementations of policies FIFO (First In First Out, the process does not leave the processor until it finishes its task or is preempted), Round-Robin (same as FIFO, but with limited time slots) and EDF (Early Deadline First, the user assigns estimated execution times to each process, and the smaller this time is, the higher the priority). The four schedulers are Uniprocessor (UP), optimized for one processor computers; Symmetric Multi Processor (SMP), able of running tasks in any CPU or fix it to one particularly; Multi Uniprocessor (MUP), which forces the task execution into a specific CPU in return for a better use of its resources and performance; and NEWLXRT, that unifies the previous three and is able of handling both RTAI and Linux processes.

RTAI is implemented as kernel modules that are loaded when they are necessary, except for the basic *rtai* module that always must be loaded. Therefore, by default real-time programs must work in kernel space and be loaded as modules, calling the function *rt_ task_ init* to specify that they want to work under RTAI scheduler and use its services. Kernel programming is quite limited, since only functions and libraries loaded in kernel can be used, so an RTAI extension is included to allow the use of real-time services in user space. This extension is called LXRT and when the real-time initialization function is called from user space it creates a task in the microkernel, linked to the Linux process, that provides the real-time utilities.

However, while working in hard real-time mode in user space no system calls or kernel services can be used, since this would migrate the task execution back to Linux scheduler control, until the standard kernel operation is completed, which prevents ensuring that the time restrictions will be respected and might cause unexpected behaviours. Hence this practice is not recommended unless there is complete awareness of which calls and services the program is going to use [Team, 2006].

Most of the modules that RTAI incorporates are related to Inter-Process Communication (IPC), i.e. mechanisms to communicate processes running over the same domain or different ones. These services include FIFO queues (*rtai_fifo*), shared memory (*rtai_ shm*), semaphores (*rtai_ sem*), mailboxes (*rtai_ mbx*), remote procedure calls (*rtai_ netrpc*), event flags (*rtai_ bits*) and message queues (*rtai_ mq*). In addition to these, it also includes its own malloc allocation implementation (*rtai_ malloc*), tasklets for periodic or event-triggered functions (*rtai_ tasklets*) or POSIX-threads support (*rtai_ pthread*).

Overall

RTAI presents the following advantages

- Very low latencies values, as will be quantified later in thsi report.
- Widely used.

and disadvantages

- Documentation is quite old (last user manual is for version 3.4, from 2006).

- RTAI installation involves patching a vanilla Linux kernel, which is not an easy task for non-specialized users.
- While running in user space is difficult to ensure real-time restrictions compliance.
- Long and not stable support and maintenance (before last release, in May 2017, the newest change submitted was in 2013).

2.2.2 Xenomai

Xenomai is a dual-kernel solution to provide hard real-time support over a Linux kernel, trying to be as integrated with it as possible [Yaghmour, 2003]. Born in 2001, and merged with the RTAI project in 2003, Xenomai started a path on its own in 2005, becoming the open-source dual-kernel preferred tool for obtaining a RTOS.

In order to provide the most comfortable user experience it emulates the system calls and structure from other architectures and RTOS, such as POSIX [Xenomai Team, 2016a], VxWorks, pSOS+, VRTX, ulTRON and RTAI, naming these libraries *skins*. It also includes a new API, called Native, that allows developing real-time applications without previous knowledge of any of the other APIs, and the *Real-Time Driver Model* (RTDM), which provides a development interface for real-time devices drivers.

Architecture

One of the most important elements of its architecture is an evolution of RTAI ADEOS pipeline, called in this case *I-Pipe* or *Interrupt Pipeline*, which Xenomai uses as a virtual pipeline between hardware components, the standard Linux kernel and its own real-time microkernel. I-Pipe organizes the system in domains that share a common address space, which allows a process to use both microkernel and Linux kernel resources if needed, where Xenomai real-time domain has the highest priority and treats Linux kernel as a lowest priority process. When an event arrives to the pipeline, Xenomai handlers manage it first and decide which domain will take care of it. I-Pipe architecture is easily exportable to other CPUs, but has to be specifically adapted to the Linux version that is going to be used.

The different APIs emulations provide the basic operating system resources that Xenomai needs, since having a distinct kernel for the real-time capabilities isolate its processes from the standard one services. The emulators are build combining and specialising the different basic blocks provided by the module *nucleus*. This module main elements are the following:

- A real-time threads object controlled by Xenomai scheduler. It is preemptive and able of managing multiple priority levels, as well as scheduling types as FIFO and Round-Robin. All the skins thread management (priority management, preemption, thread suspend, etc) is based on this abstract object.
- An IRQ handling object to connect to hardware handlers, using a simple mechanism that fits in the complex solutions that the different emulations implement.
- A memory allocator object that is bounded by predictable latencies.
- A synchronization object that implements the thread lock mechanisms for resources access (by priority, FIFO, etc) on which mutexes, queues, semaphores and mailbox are based.
- A time management object, setting its own timebase, with support to nanoseconds and clock ticks, so the different skins timers can be used separately but concurrently.

How it works

The dual-kernel approach carries certain problems to use Linux kernel utilities in real-time programs, since they are not bound to latency restrictions and may cause unexpected behaviour on the real-time side, and because both kernels work independently. Xenomai deals with this issue by creating its real-time threads from standard POSIX threads, keeping all their functionalities while running in non-critical-time mode, and using the Real-Time Shadow extension, that allows the Linux-like task to be handled by Xenomai scheduler, when time restrictions are needed.

Each skin also implements a new set of system calls, that are included in libraries that replace the standard Glibc in the microkernel. As said before, both kernels work separately, which can cause troubles in different situations, for example, if a Linux task is in the middle of a critical section and is preempted by a Xenomai one, that also makes changes on that section, causing the first one to fail when it is woke up. To prevent this from happening two running modes exist: the primary one, controlled by the microkernel, and the secondary one, controlled by the standard kernel. Each kernel system calls can only be done from the proper mode and Xenomai puts each thread in the correct one depending on the system calls invoked. This scheduler change is called domain migration.

Overall

Xenomai presents the following advantages

- Xenomai does not depend on Linux development cycle, so it is protected against the problems that changes in the mainline may cause.
- Non real-time threads misbehaviour does not affect to real-time ones.
- In case of wrong latencies measures, the causes will be in the microkernel, much more lighter and easy to debug that the standard.
- Linux kernel is not affected by real-time mechanisms.
- Skins make easy to migrate code from other RTOS to Xenomai, and also provide a wide range of tools to new developers.

and disadvantages

- Linux drivers and libraries are not accesible (in real-time mode) from the microkernel.
- Xenomai's installation involves patching a vanilla Linux kernel, which is not an easy task for non-specialized users.
- The documentation [Xenomai Team, 2016b] is many times incomplete or unclear, and some functions does not work as described on it.
- Xenomai uses its own version of the Comedi tools, called Analogy, but again, the documentation is not very complete.
- Since Xenomai is completely independent from Linux development, its microkernel does not benefit from its support and maintenance, and also has to be adapted to new hardware and software architectures independently (with no guarantees of how long will this be done).

2.2.3 Preempt-RT

Since the apparition of the RTLinux project mentioned before, numerous efforts have been made in order to make Linux kernel fully preemptable [Dietrich and Walker, 2005], and therefore turn it into an RTOS by itself, without the need of microkernels. Nowadays, all the changes and improvements made in this regard can be found in the Preempt-RT patch, which offers a viable alternative to dual-kernel systems.

This project was started and directed for many years by Ingo Molnár, and since then it has been mainly supported and funded by the Open Source Automation Development Lab (OSADL), until 2015, when it was transferred to the Linux Foundation, meaning more funding, developers and support. Currently, 80% of the Preempt-RT patch is already included in the mainline Linux kernel, due to the fact that part of the improvements solve some general problems that were found in the kernel when it was made preemptive, mainly related with locks and race conditions, and help increasing the standard kernel stability and performance.

How it works

The modifications made by this patch to the vanilla kernel in order to make it fully preemptive are the following:

- Reimplementation of the mutexes to make kernel spinlocks preemptive. An spinlock is a software lock over a resource where the locking process checks constantly if the resource is available (active waiting) so the scheduler thinks it is active even if it not really doing any task. They are useful for short and important locks, as the kernel ones.
- Critical sections protected by *spinlock_t* and *rwlock_t* are now preemptive. In kernel space no preemptive critical sections can be created with *raw_spinlock_t*.
- Implementation of *priority inheritance* [Sha et al., 1990] for kernel spinlocks and semaphores to solve priority inversion. Priority inversion is a situation that happens when a process prevents another one with higher priority of completing its task. This may happen, for example, if a low priority process (1) creates a lock over a resource and then is preempted by a higher priority one (2), but then a process with priority higher than both of them (3) wants access to the locked resource, unsuccessfully until the second process release the processor and lets the first one complete its task and release the lock. This is a problem since there is no way of calculating the time that the higher priority process will be waiting.

  Priority inheritance is a mechanism to solve this issue, consisting in giving to a process that is holding a lock over a resource the priority of the highest priority process that is waiting at that moment for that same resource until it releases the lock. By doing so, at the previous example, process 1 would have the same priority as process 3, thus process 2 would not be able of preempting it before releasing the resource.
- Interrupt handlers are run as preemptive kernel threads [Henriques, 2009]. In this manner, these threads have an specific priority and cannot preempt another process running at that moment with a higher one. In any other aspect, these threads imitate the behaviour of the standard handlers, like having CPU affinity, for example.
- Implementation of a new set of high resolution timers. Linux kernel measures time in *jiffies*, a variable time unit defined by kernel's constant *HZ*. Alarms storage is implemented through a *timer wheel*, an structure divided in buckets. The first wheel's layer represents the next 256 jiffies in the future, one per each of the 256 buckets that it contains. The next

layers also contain 256 buckets, but each one of them represent 256 jiffies. For example, if an alarmis programmed for 20 jiffies in the future, it will be saved at the 20th bucket of the first layer, but if it is for 276 jiffies in the future it will be placed in the second layer. When the time represented by the first layer passes, the elements stored in the first bucket of the second layer have to be rehashed to the unitary buckets of the first one, an operation that have an O(n) cost, where *n* is the number of elements moved. Moreover, the computational cost of adding or removing a timer from the wheel is O(1), being a really cheap operation in contrast to rehashing.

While trying to find an improvement to this structure, Thomas Gleixner [Gleixner and Niehaus, 2006] realised that there were two kind of timers being stored in the timer wheels: the *action timers* and the *timeout timers*. The former are those used by processes to be notified of an event and are removed after that, thus if they are programmed for a far future moment they will be rehashed many times, but added or removed just a few, so using a timer wheel for these timers is quite inefficient (lots of O(n) operations against few O(1)). On the other hand, timeout timers are triggered when an event does not occur (for example, a network package that does not arrive). These kind of timers stay a short time at the timers wheel (few O(n) costs) but are added and deleted frequently (many O(1) costs), so they do benefit from the timers wheel paradigm.

With this in mind, Gleixner designed the following solution: he implemented a new structure, called *hrtimers*, that would store the action timers using red/black trees, instead of hash tables, having these an O(log n) cost for adding or removing elements (O(1) for the first element), but no cost for rehashing since the tree is already sorted. Additionally, hrtimers work in nanoseconds instead of jiffies, so its performance depends on the hardware clock and not a software constant. This improvement was added to the mainline Linux kernel 2.6.

Overall

Preempt-RT presents the following advantages

- On some operating systems, such as Debian, it can be easily installed with an *apt install* command.
- There is no need of learning new programming APIs because standard POSIX functions and structures are capable of ensuring real-time.
- Being supported by the Linux Foundation, and included in the mainline kernel, it is more likely that the project will be maintained and continued in the future.
- Being supported by the Linux Foundation, and included in the mainline kernel, it is more likely that the project will be maintained and continued in the future.
- The same code can be used in real-time and standard executions.

and disadvantages

- Latency measures maximums are greater than the ones obtained by dual-kernel options.
- POSIX libraries are not designed specifically for real-time applications, which is seen by some users as a possible source of problems.
- Any bug on the Linux kernel is also a bug on the RTOS.

2.2.4 Other RTOS solutions

All three previously explained RTOS implementations correspond to open-source, free and Linux-based solutions. Nevertheless, there exist multiple commercial options, which being hard real-time specifically oriented are many times considered as more robust than the others. On the other hand, they have some inconveniences of their own, like their price to begin with, which is usually one of the main drawbacks for many research groups and laboratories, that cannot afford it. Moreover, the usage in embedded systems is the main context for real-time technology one, so the commercial RTOS are mostly aimed to it and they are not GPOS.

One of the most widely used commercial RTOS is *VxWorks*, created by Wind River Systems in 1987 and nowadays used at BOEING-777 airplanes and NASA's Curiosity Rover (sent to Mars), among others. In 1982, the company Quantum Software Systems released *QNX Neutrino*, one of the first real-time kernels that reached the market and used in a wide range of embedded systems, from mobile phones to cars. Of course Microsoft also developed a real-time version of its Windows OS in 1996, called *Windows Compact Embedded*, which they use for their Windows Phones, for example. Other proprietary software-based real-time solutions are *RTOSWin*, that runs paravirtualized real-time software next to Windows; *xPC*, a MATLAB's toolbox that runs the real-time functionalities in another device; or National Instruments' *LabVIEW Real-Time*, that works in a similar way than xPC. Finally, it is important to emphasize that there are numerous works and papers comparing different RTOS [Hambarde et al., 2014][Aroca and Caurin, 2009] and they prove that performance of open-source solutions is not worse than commercial ones, on the contrary, it is similar or even better in some cases.

2.2.5 Real-time software for closed-loop neuroscience

Nowadays there exist numerous open-source software applications designed to perform closed-loop experiments, even some of them are specific for dynamic-clamp. These tools are quite diverse, both in functionality as in running platforms and architectures. Some of them use proprietary software, which is usually expensive and many laboratories and research groups cannot afford it , while others run over GPOS, which is more user-friendly but cannot ensure hard real-time behaviour, and others do work over free tools and RTOS, but may be more difficult to use for non-specialised in computer science users. In figure 2.2 some of this tools are listed and described (references [Linaro et al., 2014], [Biró and Giugliano, 2015], [Nowotny et al., 2006], [Pinto et al., 2001], [Dorval et al., 2001], [Lin et al., 2010], [Ortega et al., 2014], [Muñiz et al., 2009]).

2.3 Neuron models

In order to understand how does the brain works it is necessary to support the experimental research with computational simulations of its behaviour. The first step towards reaching a complete brain simulation is designing neuron models that represent the behaviour of single neurons, and that these can be combined to form neural networks and other more complex structures. Neuron models are sets of mathematical equations which describe properties of the neurons and their performance.

However, real neurons dynamics are quite complex and diverse, hence the models must represent a simplified version of them, but still reproducing their biological qualities. Back in 1907, Lapicque designed the integrate-and-fire model, a very simple and light one, but also little realistic [Gerstner et al., 2014]. The most known realistic model is the one developed by Hodgkin and Huxley in 1952 [Hodgkin and Huxley, 1952a] [Hodgkin and Huxley, 1952b] [Hodgkin and

Name	Application	Platform	Characteristics	References
LCG	Cellular dynamic clamp	UNIX-based OS	Command-line program Performs simple tasks that can be connected No real-time	[Linaro et al., 2014]
PC NEURON Simulink	Cellullar electrophysiology	Windows	Uses xPC Matlab and Simulink (propietary) Real-time	[Biró and Giugliano, 2015]
StdpC	Cellular dynamic clamp	Windows	Easy to use and install Works on a propietary platform No real-time	[Nowotny et al., 2006]
DYNCLAMP4	Cellular dynamic clamp	Windows	Can perform dynamic clamp control over 4 neurons together Works on a propietary platform No real-time	[Pinto et al., 2001]
RTXI	Cellular, network, cardiac dynamic clamp Deep brain stimulation	RTAI Xenomai	Only works on RTAI and Xenomai Has different modules for different tasks New modules can be programmed Real-time	[Dorval et al., 2001] [Lin et al., 2010] [Ortega et al., 2014]
RTBiomanager	Cellular dynamic clamp Behavioural studies EEG-based BCI	RTAI	Only works on RTAI Only works for old Linux kernels (compatible with Qt3) Real-time	[Muñiz et al., 2009]

Figure 2.2: Different real-time applications for closed-loop experiments.

Huxley, 1952c], for which they received the Nobel Prize in Physiology-Medicine in 1963, which represents very accurately the live biophysical dynamics, at the expense of a high computational cost.

Many other neuron models have been proposed over the years as more simple, and therefore less computationally expensive, but able of reproducing some aspects of the biological behaviour (but not at the level of Hodgkin-Huxley). Each one of these models have advantages and disadvantages and are more useful in specific situations. A deeper insight of three of them is provided below.

2.3.1 Izhikevich neuron model

In 2000, Eugene M. Izhikevich presented a model of neural spiking and bursting, light as an integrate-and-fire model but able of replicating biophysical behaviour of certain cortical neurons as realistically as Hodgkin-Huxley model [Izhikevich, 2003]. Based in the bifurcations of dynamical systems he designed a two ordinary differential equations system(2.1) (2.2) with an auxiliary condition (2.3).

$$v' = 0.04v^2 + 5v + 140 - u + I \tag{2.1}$$

$$u' = a(bv - u) \tag{2.2}$$

$$\text{if } v \geq 30mV, \text{ then} \begin{cases} v \leftarrow c \\ wu \leftarrow u + d \end{cases} \tag{2.3}$$

Variable v corresponds to the neuron's membrane potential, while variable u models the activation of the potassium ionic channels and inactivation of the sodium ones. The auxiliary condition resets the variables to some specific values after an spike, v due to the fast K^+ conductances and u due to Na^+ and K^+ slow conductances. Parameter a adjust the recovering time of u (the smaller a is, the slower the recovery); b describes the coupling between v and u (the coupling is higher as b is), which makes u less or more sensitive to subthreshold oscillations of v; c is the value used at condition 2.3 to reset v; and the last parameter d models the reset of variable u. Parameter I is used to introduce synaptic or dc-currents in the equations.

All variables and parameters are dimensionless, but equation 2.1 is fitted so the potential has scale mV and the time, scale ms. Different configurations of these parameters lead to distinct types of neural behaviour, for example imitating those of different types of cortical neurons, as shown in figure 2.3.

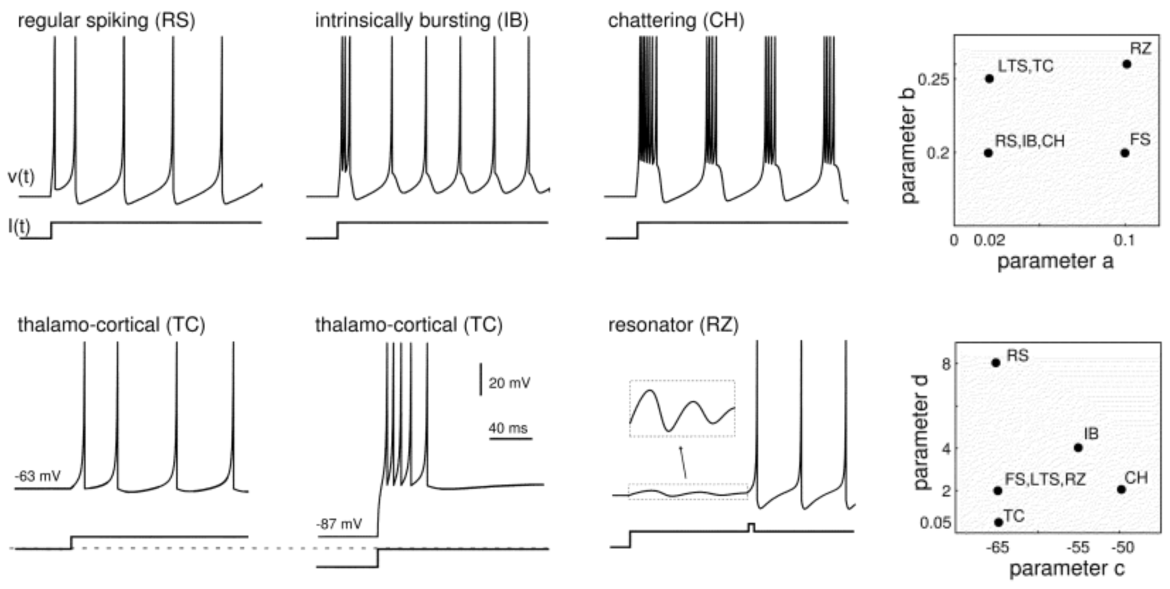


Figure 2.3: Different dynamics generated by Izhikevich model and parameters diagram (from [Izhikevich, 2003]).

2.3.2 Hindmarsh-Rose neuron model

In 1984, Hindmarsh and Rose published a spiking and bursting dynamics of neurons model, quite simple but capable of describing many behavioural patterns (including chaotic dynamics), which is an extension of their two-variable model presented the previous year [Hindmarsh and Rose, 1982], which at the same time is a generalisation of Fitzhugh model [Fitzhugh, 1961] (a second order model designed as a simplification of Hodgkin-Huxley). Their previous model fired indefinitely, therefore they added a third differential equation to be able of include a slow current to provoke the slow wave after-hyperpolarisation behaviour [Hindmarsh and Rose, 1984].

To study the performance of the model and its spiking dynamics, the system is represented in the phase space and its nullclines, limit cycles and critical points are analysed. When the neuron enters a limit cycle it get caught in there, going over the values in it periodically, thus giving place to the spiking behaviour. The third equation is needed to have a time-modelled variable which allows the system to leave the limit cycle.

$$\begin{cases} \dot{x} = y - ax^3 + bx^2 + I - z \\ \dot{y} = c - dx^2 - y \\ \dot{z} = r(s(x - x_1) - z) \end{cases} \tag{2.4}$$

All parameters and variables in the ordinary differential equations of system 2.4 are dimensionless. Variable x represents membrane potential, while y, also called spiking variable, symbolises Na^+ and K^+ fast ionic channels, and z, known as bursting variable, models the other slow ionic channels. In the original paper the constants have fixed values $a = 1$, $b = 3$, $c = 1$ and $d = 5$ to ease the phase space study of the system, while I represents the external current. Fixing constants r, s and x_1 to different values, diverse performances of the neuron are obtained. Due to the value of constant r, much smaller than the other parameters, of the order of 10^{-3}, z variation over time is slower than x and y ones, hence it can be considered as a slow variable and get rid of the third equation to work in a two-dimensional phase space (see figure 2.4).

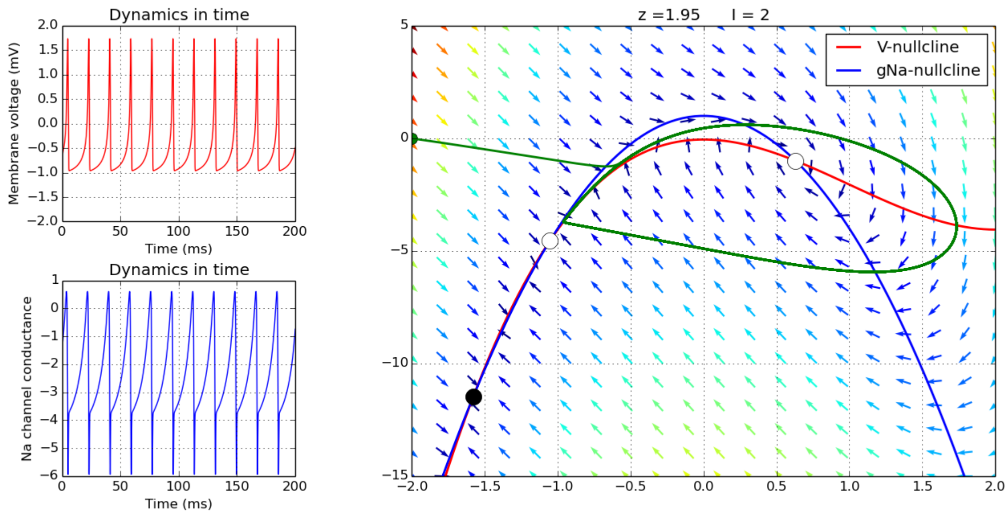


Figure 2.4: Hindmarsh-Rose neuron model dynamics and phase space for the model with fixed values z = 1.95 and I = 2.

2.3.3 Rulkov map neuron model

A high dimension and strictly non linear dynamic system is difficult to study, and with this kind of equations is very complicated or even impossible to analyze the mechanisms that act in the restructuring collective behaviour of neurons. Furthermore, some neural conducts require of an elevate number of dimensions in the system to be described, as the chaotic bursting which needs three equations. Rulkov wanted to provide a method to represent the complex neural dynamics in a simpler manner, so he propose a two dimensional map able to describe generation of tonic spikes, irregular spiking, and both regular and irregular bursting, which is simple enough to study its underlying dynamics [Rulkov, 2002].

$$x_{n+1} = f(x_n, y_n + \beta_n) \tag{2.5}$$

$$y_{n+1} = y_n - \mu(x_n + 1) + \mu\sigma_n \tag{2.6}$$

$$f(x,y) = \begin{cases} \alpha/(1-x)+y) & x \leq 0 \\ \alpha + y & 0 < x < \alpha + y \\ -1 & x \geq \alpha + y \end{cases} \tag{2.7}$$

To describe both fast spiking with slow oscillation it is necessary to have a fast and a slow dynamic. The fast one is modeled by equation 2.5, while the slow is shown in equation 2.6, because of the small value $\mu = 0.001$. β is the variable which represents the injection of current, either from external current (I_{dc}) or synaptic inputs (I_{syn}) and σ is a control parameter which also symbolizes external influence. To model the individual spikes dynamics during bursting the non-linear equation 2.7 was included, with α being another control parameter.

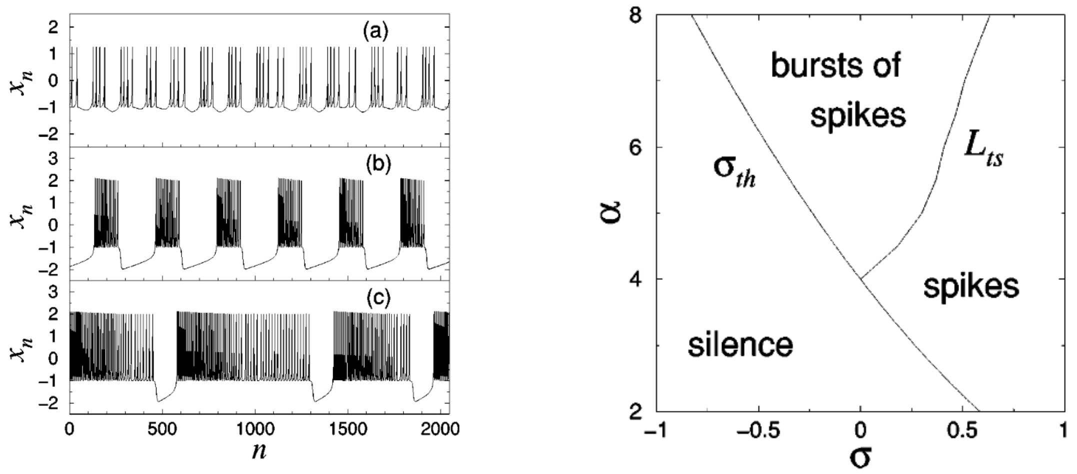


Figure 2.5: (a) Spiking-bursting behaviour generated by Rulkov Map with different σ and α values. (b) Bifurcation diagram on the parameter plane (σ,α). Both figures taken from [Rulkov, 2002].

2.3.4 Synapse models

Synapses are the mechanism that neurons use to communicate between them. There are basically two kinds of synapses: electrical and chemical. They have inhibitory or excitatory effects on the postsynaptic neuron, that may last for just milliseconds to minutes. Electric ones send a current proportional to the voltage difference of the two cells connected , sending simple and fast signals, while chemical synapses can provoke more complex changes on the receiver by sending highly nonlinear current modulated on time [Kandel, 2013]. This is reflected in the equations that model them and have been used in this library, being the electrical synapse just the difference between the post and presynaptic potentials, multiplied by the synaptic conductance, as in

$$elec\_syn = g * (V_{post} - V_{pre}) \tag{2.8}$$

while chemical synapses typically include different dynamics (fast and slow) and numerous parameters [Golowasch et al., 1999], as for example in a graded synapse

$$chem\_syn = fast\_chem + slow\_chem \tag{2.9}$$

$$fast\_chem = \frac{g_{fast} * (Vpost - E_{syn})}{1 + exp[s_{fast}(V_{fast} - v_{pre})]} \tag{2.10}$$

$$slow\_chem = g_{slow} * m_{slow} * (Vpost - E_{syn}) \tag{2.11}$$

$$\frac{dm_{slow}}{dt} = \frac{k_1(1 - m_{slow})}{1 + exp[s_{slow}(V_{slow} - V_{pre})]} - k_2 m_{slow} \tag{2.12}$$

3 Real-time solutions comparison

The aim of this chapter is to offer a detailed comparison among the different real-time solutions introduced and described in section 2.2. Since the main goal of this work is to provide an analysis for a future development of a standarized approach in neuroscience research, available for any researcher who may need it without economic or technical boundaries, the tools that will be used are free open-source extensions for the also free, open and general purpose operating system, Linux. Three of them will be tested, RTAI, Xenomai and Preempt-RT, considering both their real-time performance and their usability and user-friendliness for any kind of user.

3.1 Structure of the test

Neuron model implementations are periodic functions which generate points at a specific rate. In order to test the real-time performance of the mentioned RTOS a very simple periodic code in C language has been implemented. This program sleeps until a determined time, does something and then sleeps again, imitating the models behaviour. A snippet of the pseudocode of this simple program is shown in algorithm 3.1, and will be the cornerstone for further functions implemented in this work.

Algorithm 3.1: Periodic sleeping function.

```
input: int time, int period
begin
    target_time = period

    for (i = 0; i < time; i++)
        sleep_until(target_time)
        do_something()
        target_time = target_time + period
    end
end
```

When a process is idle (sleeping, for example) the scheduler takes the resources that it was using, including the processor, and gives them to other tasks until the idle process awakes again. In a non real-time OS, if this process has higher priority than the ones running at the expected awaking time, it will have preference. However, that does not mean that the scheduler will give it the resources back immediately: it may, or may not, according to its policy. On the other hand, a RTOS will return the resources to the higher priority task as soon as possible.

This latency between the expected awaking time and the real one is going to be taken as a measure of the real-time capabilities of a system, as well as the jitter. With this test also interruptions are taken into account, since an interruption longer than the period will cause a high latency value, but a shorter one will not affect the execution. Algorithm 3.2 include these additions to the code.

Algorithm 3.2: Periodic sleeping latency function.

```
input: int time, int period
begin
   start_time = get_time()
   target_time = start_time + period

   for (i = 0; i < time; i++)
      sleep_until(target_time)

      awake_time = get_time()
      latency = awake_time − target_time
      print(latency)

      do_something()

      target_time = target_time + period
   end
end
```

For each platform a different implementation of this code will be required, with different advantages and drawbacks, which will be discussed in further sections. Two more tools will be also used for these tests, which are described below.

Sysbench

Sysbench is an open-source benchmarking tool suite designed to easily test some aspects of a computer, as its file I/O performance, the scheduler performance, memory allocation and transfer speed, threads implementation performance or database server performance [Kopytov, 2009]. The most basic functionality it has is to execute a thread which calculates prime numbers, introducing a huge workload in a processor and making it work at full capacity. This can be run as many times as needed until the percentage of use of all the cores of every processor in the machine is at 100%. In this scenario, the real-time task should not have any problem to preempt any of the threads and take control of the processor every time it wakes up. An example of a Sysbench execution is showed in listing 3.3.

Listing 3.3: Sysbench command line execution.

```
sysbench --test=cpu --time=5600 --cpu-max-prime=20000 run
```

Core isolation

Modern computers usually have more than one processor or, more typically, a multicore processor (a single component with two or more independent processing unit). With this in mind, a logical question raises: can a processor unit be reserved to run only the tasks specified by the user, and therefore allow a real-time task to do its job unmolested without the need of an RTOS?

This situation will be tested, both on the RTOS and the non real-time operating system. Isolating a core in a Linux OS requires just a few simple steps:

- Open the file */etc/default/grub* and modify the line *GRUB_CMDLINE_LINUX_DEFAULT ="quiet splash"* to *GRUB_CMDLINE_LINUX_DEFAULT="quiet splash isolcpus=n"*, where *n* is the ID number of the core or cores that will be isolated (separated by commas).
- Run *update-grub*.
- Restart the computer.
- Now the *isolcpus=n* should appear in */proc/cmdline*.

Once a specific core is isolated, a process can be bound to it by using the function *sched setaffinity* (or *pthread_setaffinity_np* for threads), so the task will supposedly run alone in the isolated core without sharing it with other processes.

3.2 Results

The latency tests consisted in two separated 20 minutes executions of the program described in algorithm 3.2, adapted in each case to the particularities of the operating system, from which the worst case is the one included here. During all of them the processor was working at full capacity, with Sysbench-created workload. Since the aim of the project was to develop a neuron model library, the *do_something()* part of the code consisted of an implementation of the Izhikevich neuron model, which delivered a point of the signal with a frequency of 10 kHz. Further explanation of the implemented models will be discussed in Chapter 4.

Some issues that affected to the real-time behaviour, like printing the results, and the ways they have been dealt with will also be addressed in the next chapter.

3.2.1 Non real-time Linux

The first trials were run over a standard non real-time Linux kernel, just to check which were the baseline latencies and that they were not predictable. The computer used had the following characteristics:

- **Processor:** Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz, 8 hyperthreaded cores (4 physical).
- **RAM memory:** 16 GB.
- **Operating system:** Debian GNU/Linux 9.0 (stretch), kernel 4.9.0-2-amd64.

Numerous high-level languages can operate in real-time [Burns and Wellings, 2009][Bollella and Gosling, 2000], however C was chosen to implement the program because POSIX time-management tools are good enough by themselves (if not disturbed by the OS) and the APIs and libraries used by the studied RTOS are designed for this language. The functions and data structures used to implement the periodic loop are:

- **struct timespec:** data structure used to store times, with two fields: *tv_sec* for the seconds and *tv_nsec* for nanoseconds. By adding both the full time is obtained.
- **clock_gettime:** gets the time of one of the system clocks. In this case, the clock used was *CLOCK_MONOTONIC*, which represents the absolute elapsed wall-clock time since some unspecified fixed point in the past.

- **clock_nanosleep:** function that makes the process sleep for a determined time period or until an specified moment, according to the same clocks as the previous function. For our program the second option was chosen, so as to not add the latencies delay in each iteration. There are other POSIX functions designed for the same operation, but they are not precise enough.
- **pthread_setschedparam:** function used to set the priority of a thread to a determined value, in this case, to the highest possible (99).
- **mlockall:** function that locks all the calling process' virtual address space into RAM, so that memory can not be paged to the swap area, avoiding the unpredictable latencies that this may cause.

The latencies obtained, after loading the system with Sysbench workload (see figure 3.1a), in the non real-time kernel are showed in figure 3.3. The same test was repeated with an isolated core (actually cores 0 and 1, since the processor is hyperthreaded and every pair of logical processors are just a physical one, and therefore share some resources), with the same workload (see figure 3.1b) and the process with affinity to core 0, as shown in figure 3.4.

The model had to provide a value with a frequency of 10 kHz, which means one point every 100 μs, hence any latency superior to this time would be a wrong-sent point and a fault in the real-time behaviour. In both figures can be verified that the deadline is overcame by far, reaching values greater than 1000 μs, even in the case of the isolated core. The reason of the bad performance of the isolated core scenario is that only user tasks are prevented from using it, but not root ones (see figure 3.2), so there is still some work running on it and interfering with the real-time task. Anyway, even if the maximum latency obtained in that case is high, in modern computers and without overloading the processor, this configuration can be still be used to achieve soft real-time performances.

Notice that if the acquisition frequency was 20 kHz, which is also a normal case, then any latency superior to 50 μs would be a fault.

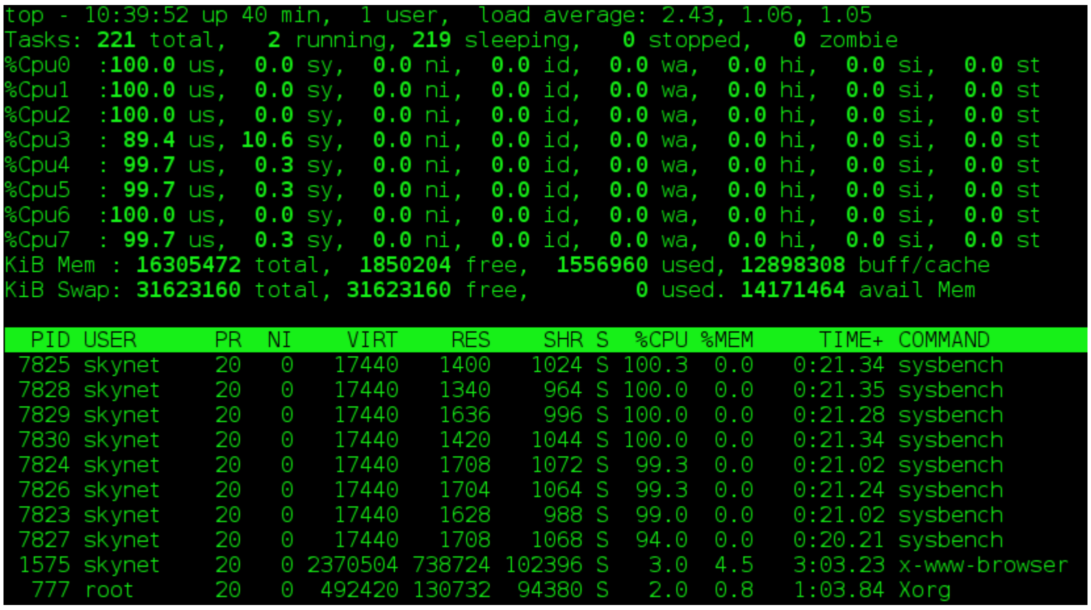
3.2.2 RTAI

At first, the RTAI tests were conducted on an old uni-core processor computer, which has been used for some years in eletrophysiology experiments at GNB laboratory. Nevertheless, there was some unexpected behaviour both in the time measurement as in the results of the model, in addition to crashes and failures of the computer itself, which led to re-run the tests in a more modern computer. The specifications of the first machine are:

- **Processor:** Intel(R) Pentium(R) 4 CPU 3.20GHz, 1 core.
- **RAM memory:** 2 GB.
- **Operating system:** Ubuntu 10.04, kernel 2.6.35.7-rtai.

And the second one:

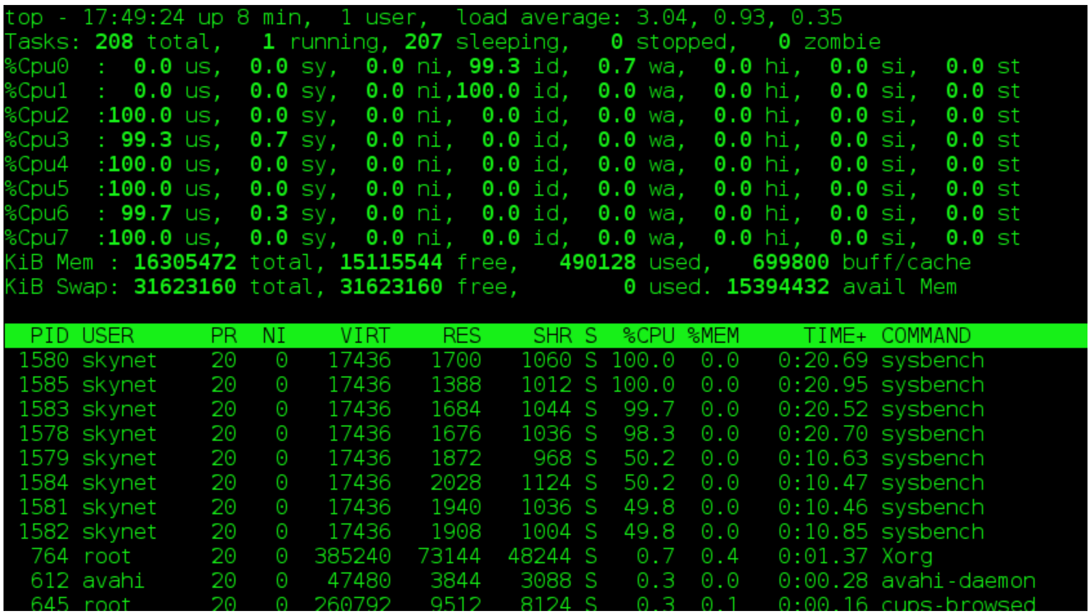
- **Processor:** Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz, 4 physical cores.
- **RAM memory:** 4 GB.
- **Operating system:** Ubuntu 10.04, kernel 2.6.34.5-rtai.



```
top - 10:39:52 up 40 min,  1 user,  load average: 2.43, 1.06, 1.05
Tasks: 221 total,   2 running, 219 sleeping,   0 stopped,   0 zombie
%Cpu0  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu1  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu2  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu3  : 89.4 us, 10.6 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu4  : 99.7 us,  0.3 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu5  : 99.7 us,  0.3 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu6  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu7  : 99.7 us,  0.3 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem : 16305472 total,  1850204 free,  1556960 used, 12898308 buff/cache
KiB Swap: 31623160 total, 31623160 free,        0 used. 14171464 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 7825 skynet    20   0   17440   1400   1024 S 100.3  0.0   0:21.34 sysbench
 7828 skynet    20   0   17440   1340    964 S 100.0  0.0   0:21.35 sysbench
 7829 skynet    20   0   17440   1636    996 S 100.0  0.0   0:21.28 sysbench
 7830 skynet    20   0   17440   1420   1044 S 100.0  0.0   0:21.34 sysbench
 7824 skynet    20   0   17440   1708   1072 S  99.3  0.0   0:21.02 sysbench
 7826 skynet    20   0   17440   1704   1064 S  99.3  0.0   0:21.24 sysbench
 7823 skynet    20   0   17440   1628    988 S  99.0  0.0   0:21.02 sysbench
 7827 skynet    20   0   17440   1708   1068 S  94.0  0.0   0:20.21 sysbench
 1575 skynet    20   0 2370504 738724 102396 S   3.0  4.5   3:03.23 x-www-browser
  777 root      20   0  492420 130732  94380 S   2.0  0.8   1:03.84 Xorg
```

(a) Without core isolation.

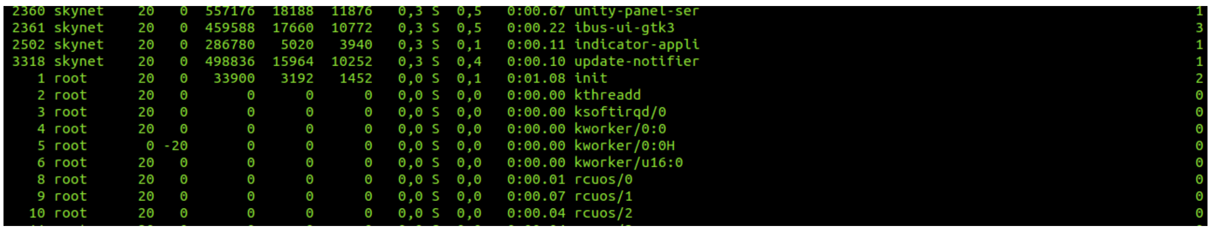


```
top - 17:49:24 up 8 min,  1 user,  load average: 3.04, 0.93, 0.35
Tasks: 208 total,   1 running, 207 sleeping,   0 stopped,   0 zombie
%Cpu0  :  0.0 us,  0.0 sy,  0.0 ni, 99.3 id,  0.7 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu1  :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu2  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu3  : 99.3 us,  0.7 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu4  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu5  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu6  : 99.7 us,  0.3 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu7  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem : 16305472 total, 15115544 free,   490128 used,   699800 buff/cache
KiB Swap: 31623160 total, 31623160 free,        0 used. 15394432 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 1580 skynet    20   0   17436   1700   1060 S 100.0  0.0   0:20.69 sysbench
 1585 skynet    20   0   17436   1388   1012 S 100.0  0.0   0:20.95 sysbench
 1583 skynet    20   0   17436   1684   1044 S  99.7  0.0   0:20.52 sysbench
 1578 skynet    20   0   17436   1676   1036 S  98.3  0.0   0:20.70 sysbench
 1579 skynet    20   0   17436   1872    968 S  50.2  0.0   0:10.63 sysbench
 1584 skynet    20   0   17436   2028   1124 S  50.2  0.0   0:10.47 sysbench
 1581 skynet    20   0   17436   1940   1036 S  49.8  0.0   0:10.46 sysbench
 1582 skynet    20   0   17436   1908   1004 S  49.8  0.0   0:10.85 sysbench
  764 root      20   0  385240  73144  48244 S   0.7  0.4   0:01.37 Xorg
  612 avahi     20   0   47480   3844   3088 S   0.3  0.0   0:00.28 avahi-daemon
  645 root      20   0  260792   9512   8124 S   0.3  0.1   0:00.16 cups-browsed
```

(b) With cores 0 and 1 isolated.

Figure 3.1: *top* command execution in a non real-time system, with the processors running at full capacity. Intel I7 processor used.



```
 2360 skynet    20   0  557176  18188  11876   0,3 S  0,5   0:00.67 unity-panel-ser      1
 2361 skynet    20   0  459588  17660  10772   0,3 S  0,5   0:00.22 ibus-ui-gtk3         3
 2502 skynet    20   0  286780   5020   3940   0,3 S  0,1   0:00.11 indicator-appli      1
 3318 skynet    20   0  498836  15964  10252   0,3 S  0,4   0:00.10 update-notifier      1
    1 root      20   0   33900   3192   1452   0,0 S  0,1   0:01.08 init                 2
    2 root      20   0       0      0      0   0,0 S  0,0   0:00.00 kthreadd             0
    3 root      20   0       0      0      0   0,0 S  0,0   0:00.00 ksoftirqd/0          0
    4 root      20   0       0      0      0   0,0 S  0,0   0:00.00 kworker/0:0          0
    5 root       0 -20       0      0      0   0,0 S  0,0   0:00.00 kworker/0:0H         0
    6 root      20   0       0      0      0   0,0 S  0,0   0:00.00 kworker/u16:0        0
    8 root      20   0       0      0      0   0,0 S  0,0   0:00.01 rcuos/0              0
    9 root      20   0       0      0      0   0,0 S  0,0   0:00.07 rcuos/1              0
   10 root      20   0       0      0      0   0,0 S  0,0   0:00.04 rcuos/2              0
```

Figure 3.2: *top* command execution in an environment with core 0 isolated. The right-most column indicates the core over the process is running, showing that root processes are using the isolated core. Intel I7 processor used.

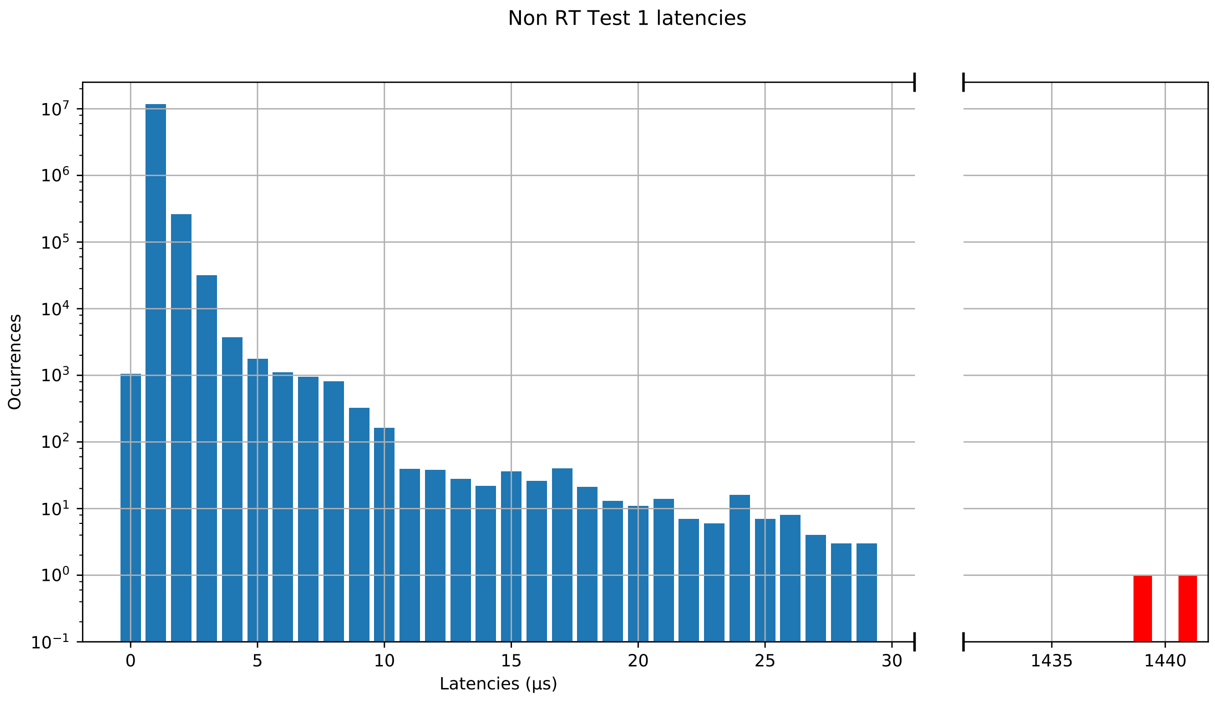


Figure 3.3: Latency values (in microseconds) occurrences with a non real-time operating system. Intel I7 processor used.

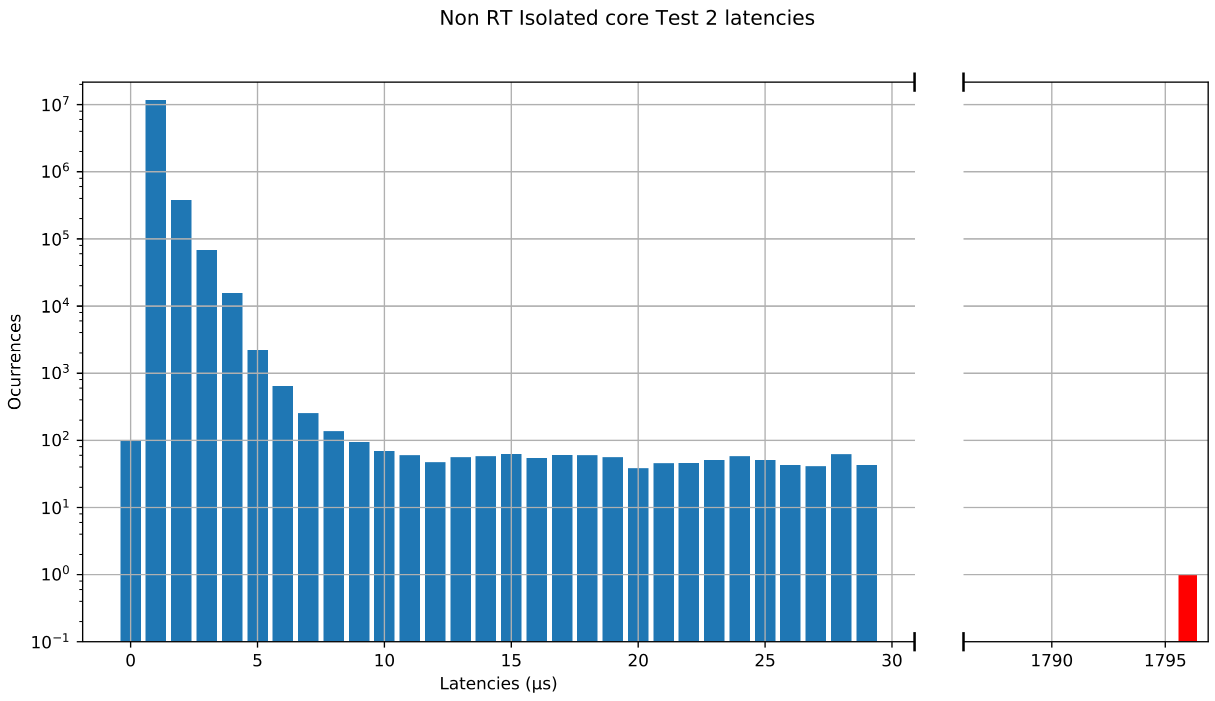


Figure 3.4: Latency values (in microseconds) occurrences with a non real-time operating system and an isolated core. Intel I7 processor used.

Despite of the differences on hardware, both computers run the same operating system, same kernel and same version of the RTAI patch. Installing RTAI involves patching a vanilla Linux kernel, compiling it and then installing it, which can prove quite challenging, even for experienced computer users. In Appendix B we include a manual to install RTAI, but even following it the user may encounter different problems, as missing libraries or configuration errors. In order to use the RTAI functions, its modules have to be manually loaded before.

As mentioned before, the program was implemented in C language, but using RTAI APIs instead of POSIX functions. The most reliable way of ensuring real-time with RTAI is implementing the real-time depending functions as a kernel module, that is also trickier since only loaded in kernel functions can be used in this case, which is a huge limitation. On the other hand, the LXRT module provides real-time performance in user space, simplifying enormously the programming of the code, but is also less stable. In both cases, the functions and data types used to make the task periodic are the following:

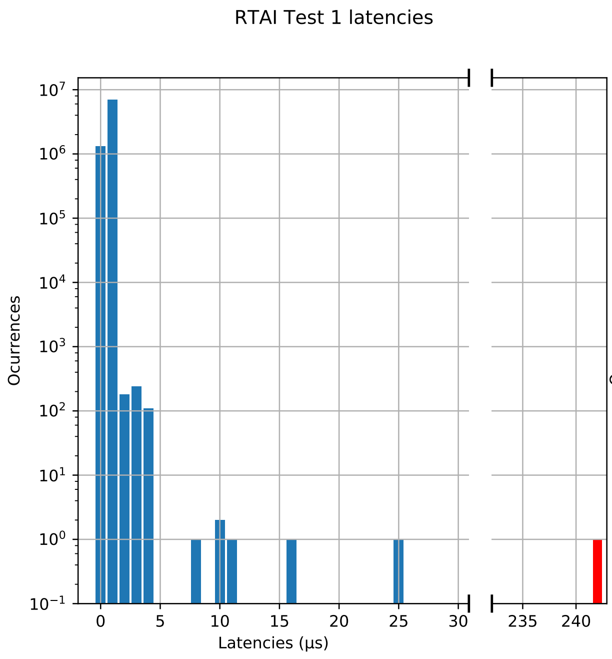
- **RTIME:** data type used by RTAI to store times. It uses *counts* as a unit, which are related to the CPU clock frequency. To convert nanoseconds to counts the function *nano2count* is used.
- **rt_get_time:** gets the time elapsed in real time clock ticks since the moment defined by RTAI internal value RT_TIMER. To convert counts to nanoseconds the function *count2nano* is used.
- **rt_set_periodic_mode:** function that sets the timer in a periodic timing mode, with period defined in *start_rt_timer*.
- **rt_task_make_periodic:** function that makes the process periodic.
- **rt_task_wait_period:** function that makes the process sleep until the next period is reached, which was specified when the timer was started.
- **rt_make_hard_real_time:** function used only by LXRT, which allows a user space process to run in hard real-time.

Due to the architecture explained in section 2.2.1, isolating a core in the Linux kernel does not have any impact on the real-time microkernel, and therefore there was no point in testing the real-time capabilities of the system in that scenario. In the same section was also stated that RTAI programs based on the LXRT module may have difficulties ensuring the temporal boundaries, so the implementation as a kernel module, even if more troublesome, was preferred for the tests. Moreover, the same program was executed in the two machines mentioned before, obtaining quite distinct results. In both cases the processor was full-loaded with Sysbench tasks.

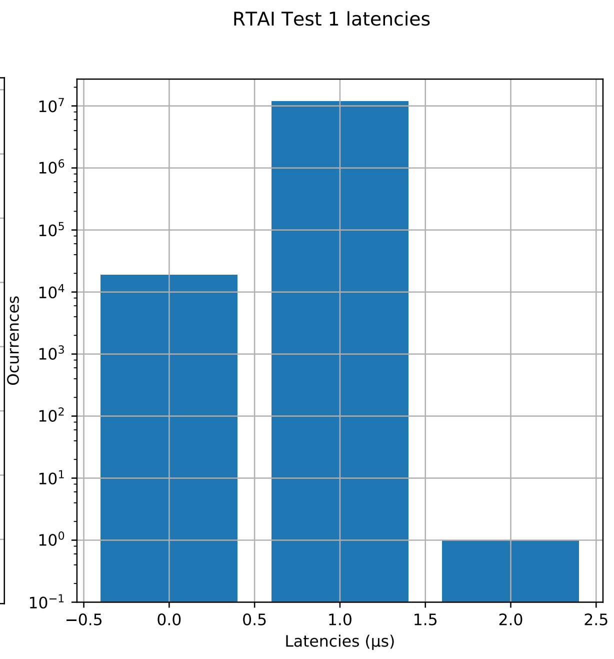
Latencies obtained with the single-core Pentium 4 processor computer are shown in figure 3.5a, and even if most of them are quite low, some high latencies also occur, reaching even 242 μs, which is an abnormal behaviour for RTAI according to the existing literature [Barbalace et al., 2008]. On the other hand, with the four-core processor the results are more reasonable, as can be seen in figure 3.5b, with a maximum latency of 2 μs. The reasons behind this performance are not known, but since the software used was exactly the same in both cases, it is clear that they have to be hardware-related. Also notice that the figures represent absolute latency values, but with RTAI most of them are negative, which means that the process is woken up before it was specified, since RTAI foresees that waking the process may take some extra time and hence, wakes it up earlier. Of course, a negative latency is as bad as a positive one (as said before, precision is needed, not just speed), so they are all represented as absolute values.

3.2.3 Xenomai

For the Xenomai trials only one machined was used, with the following set up:



(a) Pentium 4 processor.



(b) I7 processor.

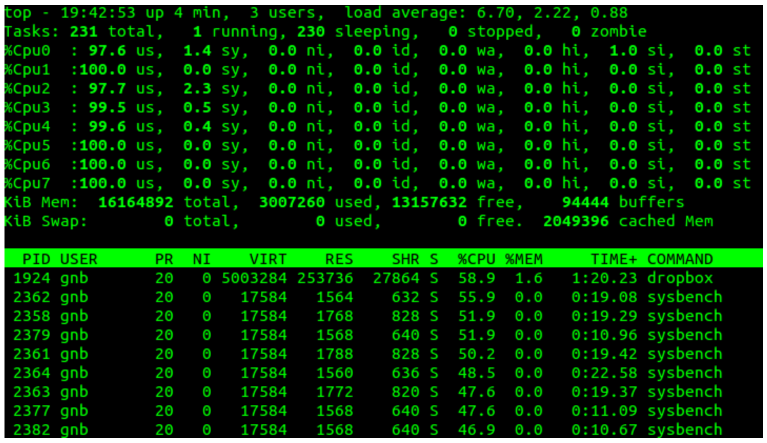
Figure 3.5: Latency values (in microseconds) occurrences with RTAI.

- **Processor:** Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz, 8 hyperthreaded cores (4 physical).
- **RAM memory:** 16 GB.
- **Operating system:** Ubuntu 14.04, kernel Linux 3.8.13-xenomai-2.6.4.

The way of installing Xenomai is similar to RTAI: patching a vanilla kernel is needed and therefore the same complications may appear. Fortunately, the developers of RTXI [Lin, 2011] also offer an Ubuntu live CD, which includes an Ubuntu 14.04 already patched with Xenomai 2, that has been used for these tests. Since Xenomai inherited its architecture from RTAI, isolating a core has the same results as in the previous case, so has not been tested.

Furthermore, C language was also used to implement the program, but this time also one of Xenomai skins needed to be chosen. *Native skin* was the first choice, but due to some problems that will be discussed in Chapter 4 was discarded in favour of *POSIX skin*, selected because the same functions had been already used for the non real-time case. Therefore the functions utilized are again *clock_gettime*, *clock_nanosleep* and *mlockall*; and the structure *struct timespec*. The only difference with an standard POSIX implementation is that there is no need of setting the task priority manually, since it will already run in the real-time microkernel.

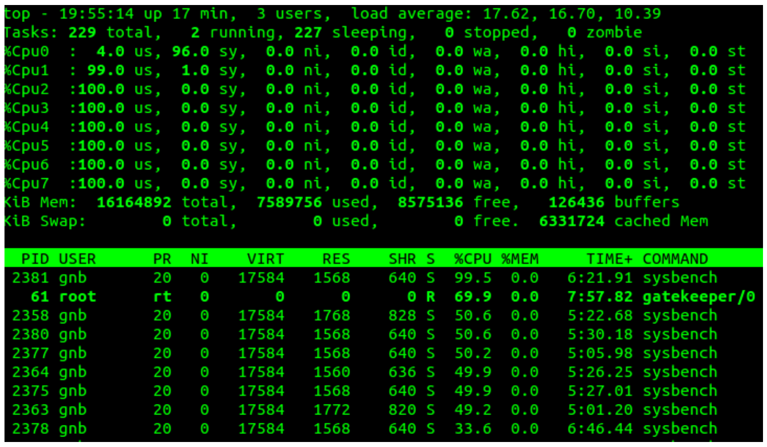
Once the Sysbench workload is running, the real-time program is executed. Notice that this program appears among the list of processes as *gatekeeper/0*, instead of the name given to it (see figure 3.6b), which is a Xenomai service that manages the relations among domains. The results of the test are shown in figure 3.7, and these are quite similar to the obtained with RTAI: latency values are low and may also be negative, for the same reasons (the range actually was from -2 μs to 6 μs).



```
top - 19:42:53 up 4 min,  3 users,  load average: 6.70, 2.22, 0.88
Tasks: 231 total,   1 running, 230 sleeping,   0 stopped,   0 zombie
%Cpu0  : 97.6 us,  1.4 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  1.0 si,  0.0 st
%Cpu1  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu2  : 97.7 us,  2.3 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu3  : 99.5 us,  0.5 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu4  : 99.6 us,  0.4 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu5  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu6  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu7  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem:  16164892 total,  3007260 used, 13157632 free,    94444 buffers
KiB Swap:        0 total,        0 used,        0 free.  2049396 cached Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 1924 gnb       20   0 5003284 253736  27864 S  58.9  1.6   1:20.23 dropbox
 2362 gnb       20   0   17584   1564    632 S  55.9  0.0   0:19.08 sysbench
 2358 gnb       20   0   17584   1768    828 S  51.9  0.0   0:19.29 sysbench
 2379 gnb       20   0   17584   1568    640 S  51.9  0.0   0:10.96 sysbench
 2361 gnb       20   0   17584   1788    828 S  50.2  0.0   0:19.42 sysbench
 2364 gnb       20   0   17584   1560    636 S  48.5  0.0   0:22.58 sysbench
 2363 gnb       20   0   17584   1772    820 S  47.6  0.0   0:19.37 sysbench
 2377 gnb       20   0   17584   1568    640 S  47.6  0.0   0:11.09 sysbench
 2382 gnb       20   0   17584   1568    640 S  46.9  0.0   0:10.67 sysbench
```

(a) Without real-time task.



```
top - 19:55:14 up 17 min,  3 users,  load average: 17.62, 16.70, 10.39
Tasks: 229 total,   2 running, 227 sleeping,   0 stopped,   0 zombie
%Cpu0  :  4.0 us, 96.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu1  : 99.0 us,  1.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu2  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu3  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu4  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu5  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu6  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu7  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem:  16164892 total,  7589756 used,  8575136 free,   126436 buffers
KiB Swap:        0 total,        0 used,        0 free.  6331724 cached Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 2381 gnb       20   0   17584   1568    640 S  99.5  0.0   6:21.91 sysbench
   61 root      rt   0       0      0      0 R  69.9  0.0   7:57.82 gatekeeper/0
 2358 gnb       20   0   17584   1768    828 S  50.6  0.0   5:22.68 sysbench
 2380 gnb       20   0   17584   1568    640 S  50.6  0.0   5:30.18 sysbench
 2377 gnb       20   0   17584   1568    640 S  50.2  0.0   5:05.98 sysbench
 2364 gnb       20   0   17584   1560    636 S  49.9  0.0   5:26.25 sysbench
 2375 gnb       20   0   17584   1568    640 S  49.9  0.0   5:27.01 sysbench
 2363 gnb       20   0   17584   1772    820 S  49.2  0.0   5:01.20 sysbench
 2378 gnb       20   0   17584   1568    640 S  33.6  0.0   6:46.44 sysbench
```

(b) With real-time task.

Figure 3.6: *top* command execution in Xenomai, with the processors running at full capacity. Intel I7 processor used.

3.2.4 Preempt-RT

Finally, the Preempt-RT tests were conducted over the same computer and operating system as the ones for the non real-time case, but this time the corresponding patch was applied:

- **Processor:** Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz, 8 hyperthreaded cores (4 physical).
- **RAM memory:** 16 GB.
- **Operating system:** Debian GNU/Linux 9.0 (stretch), kernel 4.9.0-2-rt-amd64 PREEMPT RT.

As the other RTOS previously analysed, the installation of Preempt-RT needs patching a vanilla Linux kernel. Nevertheless, there also exist an *apt-get* package for Debian distributions, making the transformation into a RTOS as simple as typing *sudo apt-get install linux-headers-«kernel version»-rt-amd64* (for 64 bits architectures) in a shell console.

The code used is also the same as in the first trial, with no changes, but since now the scheduler is a real-time one the POSIX functions will work unmolested and respecting the temporal restrictions, as can be seen in figure 3.8. Core isolation was also tested, with same configuration as in the non real-time scenario, and the results can be seen in figure 3.9. In both

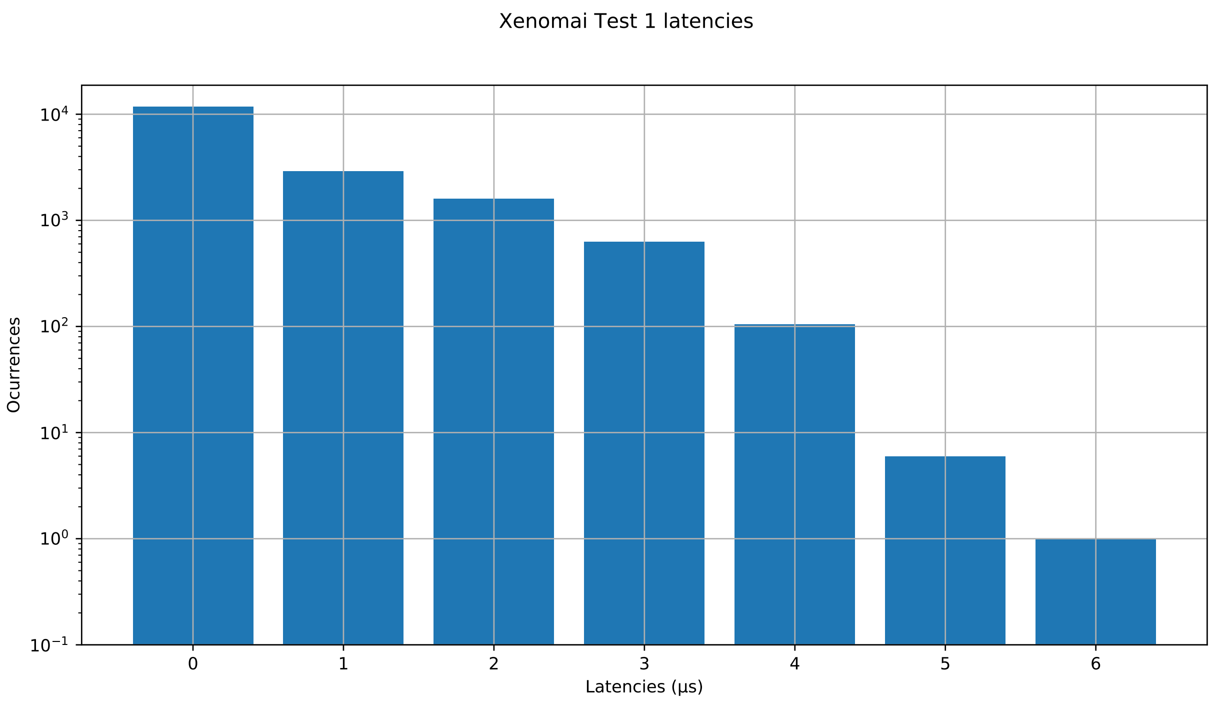


Figure 3.7: Latency values (in microseconds) occurrences with Xenomai. Intel I7 processor used.

cases the latency values obtained are almost similar, and worse than in the dual kernel solutions, but still within the real-time boundaries. All the latencies are positive, because all the processes are woken up after the determined time.

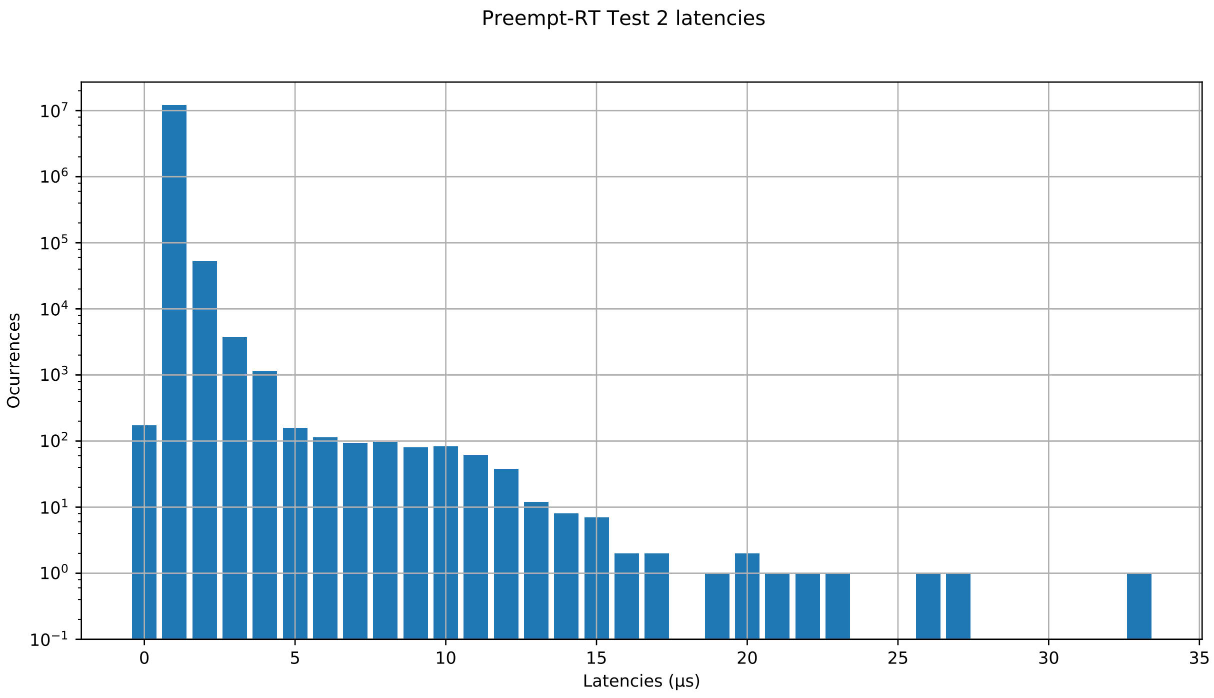


Figure 3.8: Latency values (in microseconds) occurrences with Preempt-RT. Intel I7 processor used.

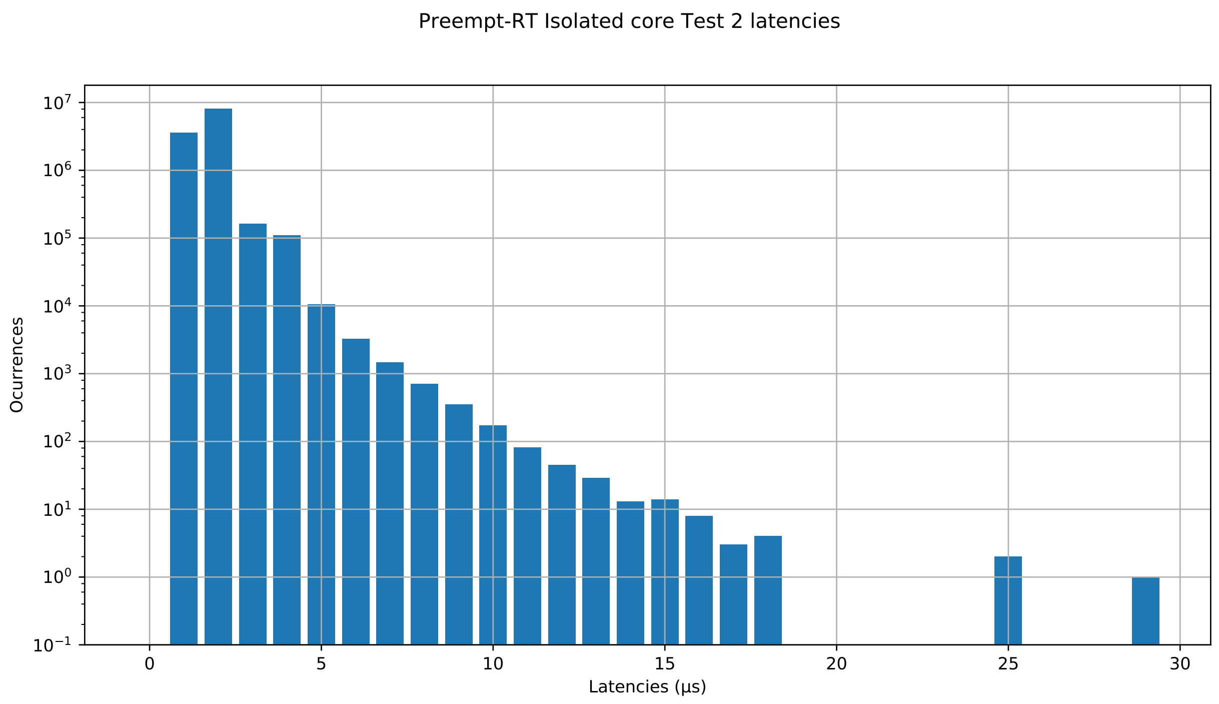


Figure 3.9: Latency values (in microseconds) occurrences Preempt-RT and an isolated core. Intel I7 processor used.

3.3 Comparison

In this final section of the chapter a quantitative comparison between the different RTOS tested will be carried on, regarding to the results obtained and the characteristics observed, and will be addressed from different aspects of the solutions.

Installation

The official installation method of all the real-time patches is the same: download a vanilla Linux kernel, apply the appropriate patch and compile it. As said before, this is not a trivial task, even for users with knowledge in Computer Science, which may find numerous difficulties through the process. However, there are alternatives for specific distributions: an Ubuntu 14.04 patched with Xenomai 2 provided by the RTXI team, and Debian *apt* packages of Preempt-RT provided by Pengutronix. Despite of being the oldest one, and probably the most used, there is no easy way of installing RTAI.

Implementation

The test program was developed from algorithm 3.2 and there are just a few differences among the RTOS for which is was implemented, that are collected in table 3.1. Even so, finding the appropriate functions and how to use them for each solution was not equally easy.

The easiest one was Preempt-RT, since it uses normal POSIX code and the user just needs to be careful of using functions with enough precision for the times and operate with them correctly. Using the POSIX skin for Xenomai is of course as simple, but in the case of Preempt-RT, the sleeping period has to be handle manually. The Native skin was also quite easy to use, but the documentation is not very detailed and there are not many examples (also mention that at the moment of writing this work, Xenomai's webpage, including the documentation, has been offline without explanation for almost a month). Native functions handle the sleeping period on their own, as well as RTAI ones. There are more examples of RTAI than Xenomai, but still the documentation is scarce, and many times confusing, with contradictory examples.

Table 3.1: Time functions and data types used by the different real-time solutions

	POSIX	**RTAI**	**Xenomai (POSIX skin)**
Real-time functions	pthread_setschedparam mlockall	rt_set_periodic_mode start_rt_timer rt_task_make_periodic	mlockall
Wait functions	clock_nanosleep	rt_task_wait_period	clock_nanosleep
Get time functions	clock_gettime	rt_get_time	clock_gettime
Time data type	struct timespec	RTIME	struct timespec

Performance

RTAI may be the most difficult RTOS to install and use, but, in modern multicore architectures, it is also the one with better real-time performance: its higher latency is 2 μs, three times smaller than Xenomai's one, the second better, even when the computer had worse specifications. Preempt-RT offers worst results than both dual-kernel solutions, but still respects the real-time boundaries. Also mention that core isolation does not seem to have any impact on real-time behaviour. All these results are shown in figure 3.10.

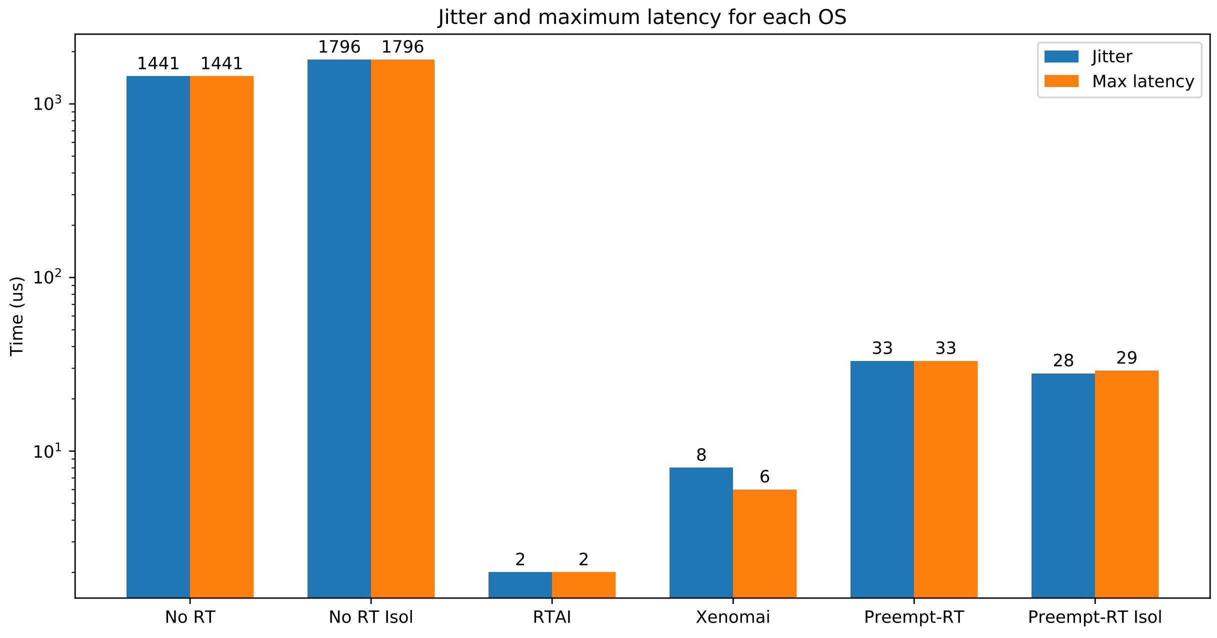


Figure 3.10: Jitter and maximum latency comparison among the different real-time solutions tested.

Conclusions

If a real-time environment with very low latencies is needed, the best choice would be a dual-kernel solution, with RTAI having slightly better performance. But since the real-time restrictions are not that strict for most closed-loop experiments, and user-friendliness is a very relevant aspect to consider in this case, Preempt-RT seems to be the RTOS which better fulfil our needs, because even though its latency values are significantly greater than with the other tools, they are still within the required temporal boundaries.

4 Neuron model library implementation

This chapter describes the implementation of the neuron and synapse model library. Following the results of the previous chapter, the main development environment has been Preempt-RT, and then the library was ported to RTAI and Xenomai. The real-time software was designed in a modular and generic way, making it easier to adapt to the different RTOS and to add models to the library. In the coming sections each of these modules will be detailed. As mentioned before, since RTAI and Xenomai programming APIs are designed for it, and because POSIX time-management functions can ensure real-time performance if the scheduler allows it, the chosen coding language was C.

4.1 Neuron models

All the models explained in section 2.3, both of neuron and synapses, have been implemented in file *model_library.c*, following the same pattern and structure so using them is transparent for the user and there is no need of knowing how the model works, since there is no difference in the callings for any of them. This favours maximum flexibility to change from one given model to another in the experimental setup. In table 4.1 are described the functions used to implement the models, as well as the parameters they receive as input. The return value of all of them is *void* and all the input arguments are pointers, so the data can be updated and returned inside them.

Izhikevich and Hindmarsh-Rose models have differential equations that have to be solved numerically, so the 6(5) Runge-Kutta method was implemented [Hull et al., 1972], again in a generic way, so it can be used indistinctly by any model: it receives the same parameters as the general model function (dim, dt, vars, params and syn) but also a pointer to the corresponding equations function. In figures 4.1 and 4.2 the behaviour of Izhikevich and Hindmarsh-Rose models can be seen, respectively, along to the figures from the original papers that are reproduced.

But Rulkov's model is not described with differential equations but with a two-dimensional map, meaning that it is computationally cheaper and faster than a model implemented with differential equations, but has also a drawback: it lacks temporal resolution. Therefore for every point generated by the model, a number of points between it and the previous one must be calculated. The method to generate new data between two points that was used is called

Table 4.1: Functions used to implement the neuron models

	Description	**Parameters**
ini_«model»	Initialize the variables of the model	**vars:** variables of the model **min:** threshold value over which the model is in spiking behaviour **minABS:** lowest value reached by the model **max:** greatest value reached by the model
«model»_f	Equations of the model	**vars:** variables of the model **ret:** return values **params:** parameters of the model **syn:** synapse value
«model_name»	Logic of the model (derivation of the equations, auxiliar conditions, etc)	**dim:** number of variables of the model **dt:** integration step of the model **vars:** variables of the model **params:** parameters of the model **syn:** synapse value
«name»_syn	Main synapse function	**v_post:** postsynaptic potential **v_pre:** presynaptic potential **g:** conductances **ret:** return values **aux:** auxiliary data

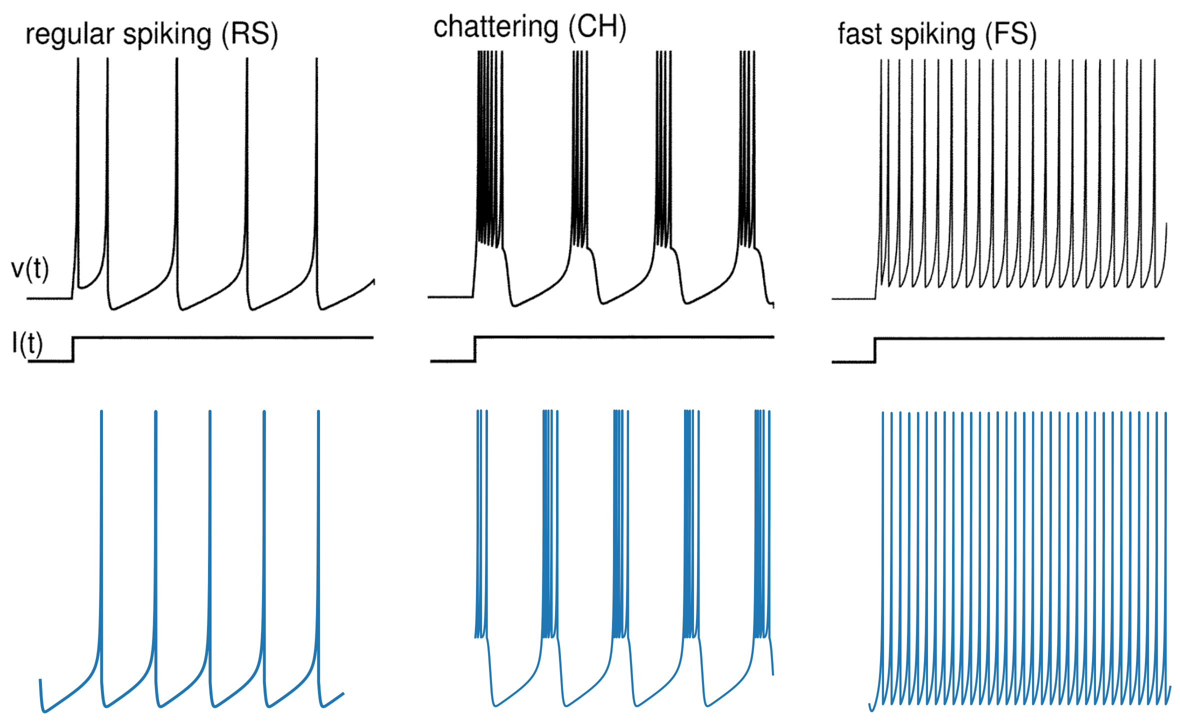


Figure 4.1: Comparison of Izhikevich model original paper and implemented model. The parameters for each behaviour can be seen in figure 2.3.

Linear Interpolation [Hazewinkel, 2002]. In figure 4.3 there is a comparison among the model implemented and the figures from the original paper.

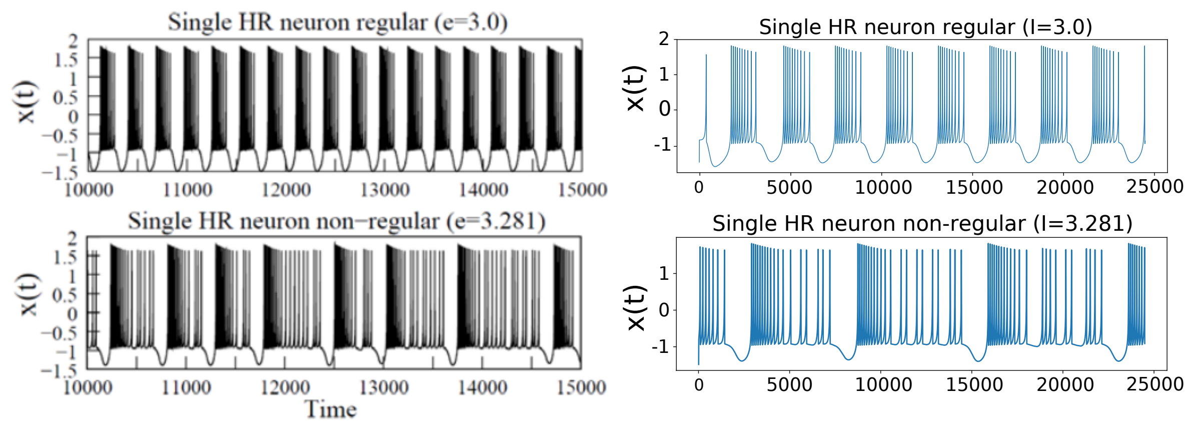


Figure 4.2: Comparison of Hindmarsh-Rose model original paper and implemented model. The parameters are: $e = 3.0$ (a), $e = 3.281$ (b), $a = 1.0$, $b = 3.0$, $c = 1.0$, $d = 5.0$.

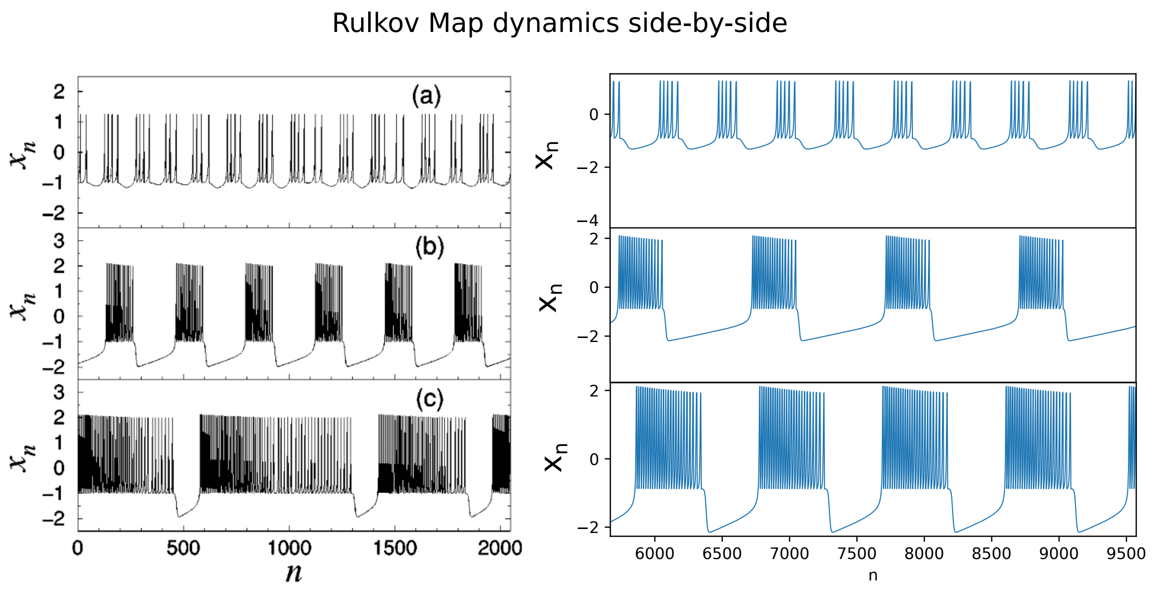


Figure 4.3: Comparison of Rulkov Map model original paper and implemented model. The parameters are: $\alpha = 4.5$, $\sigma = 0.14$ (a), $\alpha = 6.0$, $\sigma = -0.1$ (b), $\alpha = 6.0$, $\sigma = 0.386$ (c). The rest of the parameters are not specified.

4.2 DAQ communication

In order to connect the virtual neuron model to a real or electronic neuron, an intermediate device is needed to transform the digital signals to analog ones, and the other way around. These devices are known as *data acquisition devices (DAQ)* and can be connected to the computer through its I/O ports or special cards. As any other hardware, the appropriate drivers have to be installed in the computer which is going to use them, to act as an interface between the user and the device.

Comedi is a free, open-source set of drivers for various DAQ cards of different brands, available for Linux distributions. It offers an API to perform synchronous and asynchronous read and write operations over the device [Schleef et al., 2012]. Comedi installation, even if a bit long, is extensively explained in its documentation and should not represent a difficult task.

Files *comedi_functions.c* and *.h* contain the structures and functions developed to handle the Comedi drivers:

- **Comedi_session:** data structure which stores the information of a Comedi session, being this a connection to a determined device through specific input and output subdevices.
- **open_device_comedi:** given a device name (e.g. "/dev/comedi0") opens a connection with it and returns a pointer to a *comedi_t* structure, used as identifier.
- **close_device_comedi:** given a comedi_t pointer, closes the connection to the associated device.
- **create_session_comedi:** given a comedi_t pointer, a Comedi session is created.
- **get_range_info_comedi:** auxiliary function to obtain the *range_info* value for a channel, needed by the read/write functions.
- **get_maxdata_comedi:** auxiliary function to obtain the *maxdata* value for a channel, needed by the read/write functions.
- **read_comedi:** given a session and a list of channels, reads the data from those channels and return them in the *ret* variable. Calls to the function *read_single_data_comedi.*
- **write_comedi:** given a session, a list of channels and a set of values, writes each value to the corresponding channel. Calls to the function *write_single_data_comedi.*
- **read_single_data_comedi:** given a session, the range_info and maxdata values and a channel, reads a single value from the channel and returns it in variable ret.
- **write_single_data_comedi:** given a session, the range_info and maxdata values, a channel and a data, writes the single value into the channel.

This library can be used by any program that runs in an environment that has the Comedi drivers and libraries installed, it does not matter if it is RTAI, Xenomai, Preempt-RT or a non real-time operating system. In our case the DAQ used was a National Instruments' PCI NI BNC-2090A device, but any other National Instruments card would work as well.

4.3 Real-time functions

Once the models and DAQ communication modules are ready, it is time to implement the real-time periodic functions, following the scheme presented in algorithm 3.2. Probably the most troublesome issue faced was the fact that printing any data to a file or the terminal caused unpredictable high latencies, unacceptable for a hard real-time execution. Due to this problem, a new way of storing the desired data in a file was needed.

The solution designed was the following: if a real-time process can not handle a call to a print function within the temporal boundaries, then another process should take care of this task. IPC mechanisms provide the necessary tools to send the information from the real-time process to the writing one. In this case, FIFO message queues were used, so the first task could write the data in the queue and it would wait there until the second read it and printed it to the file, when he could. Therefore, two threads were created, *rt_thread* and *writer_thread*, one running in real-time and the other not. Two files are created in each execution of the program, and a line is written in them per iteration of the periodic loop. The names of the files are the hour, minute and second when the program was run, followed by the type of the file, 1 or 2, (e.g. 19h_23m_7s_1.txt), and they are stored in a directory with the current date as name (e.g. data/2017y_5m_28d). The first line of type 1 files have two values, the number of input channels and output channels, and each row of the rest of the file represents:

- **t_mon:** time in milliseconds since the set of the computer's monotonic clock.
- **t_exec:** time in milliseconds since the beginning of the execution.
- **i:** number of points sent.
- **lat:** latency between the real and the expected awakening time of the loop sleep function, in nanoseconds.
- **v_model:** voltage generated by the computational model.
- **v_model_scaled:** voltage generated by the computational model scaled to the range of the external neuron.
- **c_model:** current generated by the computational model.
- **c_real:** current generated by the external neuron.
- **data_in:** variable number of columns, defined by the number of input channels, which represent the data that entered through each input channel of the DAQ used.
- **data_out:** variable number of columns, defined by the number of output channels, which represent the data that was sent to each output channel of the DAQ used.

The second type of file saves information related to the calibration of different parameters of the model, and also contains a header row, which this time indicates the number of conductance values that are stored (e.g. one for the electrical synapse, two for the chemical), while the following rows have this structure:

- **ecm:** mean squared error between the voltages of the real neuron and the model.
- **extra:** auxiliary parameter.
- **g_real_to_virtual:** variable number of columns, defined by the number of conductances, which represent the value of those from the real neuron to the model.
- **g_virtual_to_real:** variable number of columns, defined by the number of conductances, which represent the value of those from the model to the real neuron.

There exists a third file in the same directory, named *summary.txt*, which stores some useful data from each execution of the program, as the model and synapse type used, the sample frequency in kHz, the duration of the run, if the neurons behaviours were in antiphase, the calibration mode, the model jump points (this will be explained shortly) or the bursts duration. The writer_thread functions receives as an argument a pointer to *writer_args*, an structure which contains all the information it needs, such as the file names or the number of input and output channels. First, it creates the necessary files and adds the execution information to the summary, then it enters the loop to read the data sent by the real-time thread in the queue (which will block until there is some data to read) and prints it row by row in the files. The pseudocode for this function can be found in algorithm 4.1.

Algorithm 4.1: writer_thread

```
input: writer_args
begin

    filename_1 = writer_args.filename + _1.txt
    filename_2 = writer_args.filename + _2.txt
    filename_3 = writer_args.path + /summary.txt

    f1 = file_open(filename_1, w)
    f2 = file_open(filename_2, w)
    f3 = file_open(filename_3, a)

    print_to_file(f3, data_to_f3)
    file_close (f3)

    for (i = 0; i < (5 * writer_args.freq + writer_args.points) * writer_args.s_points; i++)
        if (i % writer_args.s_points == 0)
            message = receive_from_queue(writer_args.msqid)

            if (i == 0)
                print_to_file(f1, message.n_in_chan, message.n_out_chan)
                print_to_file(f2, message.n_g)
            end

            print_to_file(f1, data_to_f1)
            print_to_file(f2, data_to_f2)

            for (j = 0; j < message.n_in_chan; j++)
                print_to_file(f1, message.data_in[j]
            end

            for (j = 0; j < message.n_out_chan; j++)
                print_to_file(f1, message.data_out[j]
            end

            for (j = 0; j < message.n_g; j++)
                print_to_file(f2, message.g_real_to_virtual[j]);
                print_to_file(f2, message.g_virtual_to_real[j]);
            end
        end
    end

    file_close (f1)
    file_close (f2)
end
```

On the other hand, rt_thread handles the logic of the program, executes the periodic loop with the model that represents the virtual neuron and connects it to the real one. Its input argument is a pointer to *rt_ args*, a structure that stores the basic information for the execution (points to calculate, input and output channels, sample frequency, etc) but also the essential model data, such as arrays with its variables and parameters, the dimension (number of variables), the integration step to be used and pointers to the appropriate functions for the model (as described in table 4.1). Since the model functions are declared as pointers to function and they all share a common structure, their use is transparent and in the context of the real-time thread there is no difference in using one or another.

Another relevant aspect of the code is the *s_points* variable. Some computational neuron models, as the Izhikevich or Hindmarsh-Rose ones, generate more points than real neurons (depending on the integration step) during the data acquisition and response cycle. To deal with this, the solution is to not send all the points generated by the model, since it calculates more points than necessary due to its high integration precision, but only the needed to match the real neuron bursting duration and frequency. Therefore *s_points* represents how many points of the model should be skipped and is use in a module operation: if *iteration_ counter mod s_points* is equal to zero, the point have to be sent 4.4. Is in this case when

1. The task sleeps until the target time is reached.
2. The latency between the expected and real waking up times is calculated.
3. All the data that needs to be put in the queue is inserted in the *message* structure.
4. The synaptic current that is going to be injected in the real neuron is calculated.
5. This current and the model voltage are stored in the array to be written into the DAQ (it has been fixed that the current goes through the first output channel available, and the voltage through the second).
6. The previous data is written into the DAQ using the Comedi functions.
7. The message is sent to the queue. This function is non-blocking, meaning that if the data can not be inserted in the queue the process will go on without halting.
8. The target time is incremented another period.
9. Data from the DAQ is read using the Comedi functions (it has been fixed that the current of the external neuron comes through the first available channel).

The synaptic current from the real neuron to the model is calculated and the model calculates a point in any case, regardless the point is going to be used or not. It is also desirable to record the isolated behaviour of both neurons before connecting them, so the loop is repeated twice, but the first time it just runs for five seconds and the injecting currents are keep at 0.

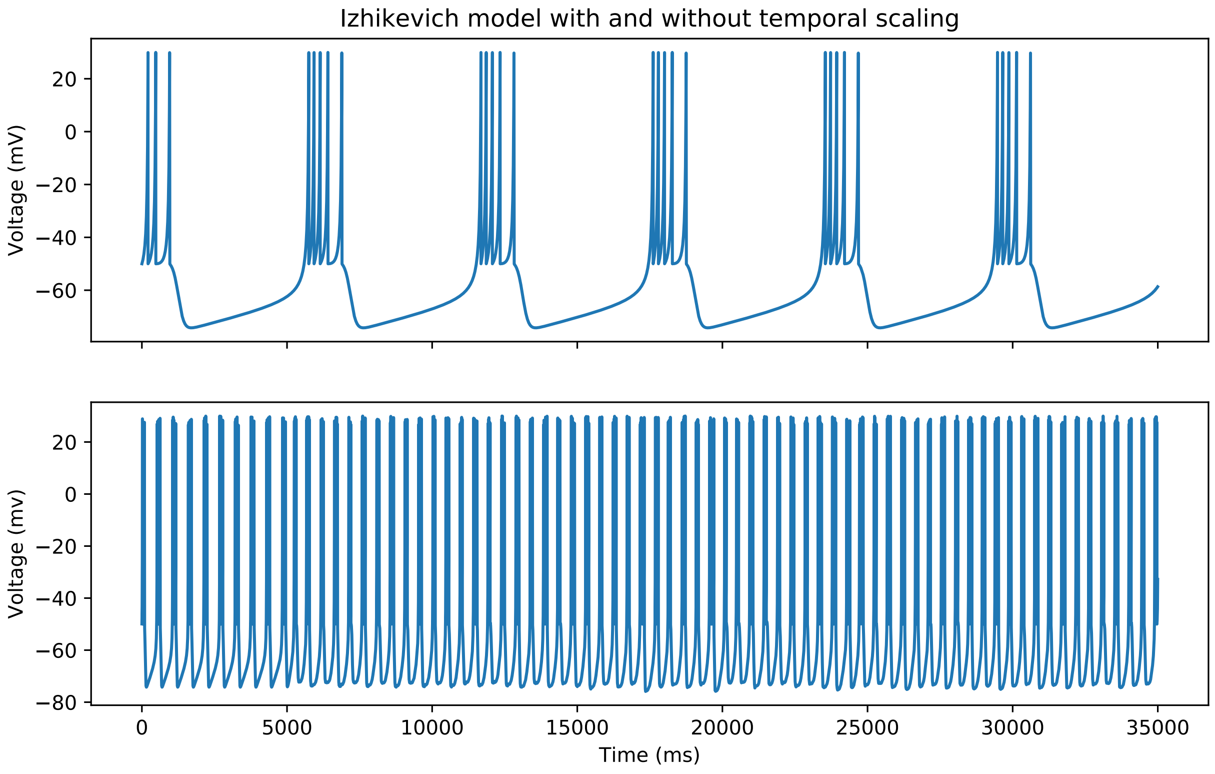


Figure 4.4: Comparison between an Izhikevich model sending every point generated (and sleeping after that) (a), and another skipping 11 points to have the same bursting behaviour as an electronic neuron (b).

Note that two neurons or models do not have to be necessarily in the same range, either because they work in a different order of magnitude or even because these values are dimensionless.

The time scales can also be very different. In any case, the signals must be scaled and adjusted to each other to have a proper neural interaction. As *s_points* is used to perform some temporal adjustment, other variables are included in the code to do the same task over the model values and parameters. All the calibration work included in the code, and used in the validation tests that will be described in Chapter 5, is part of the project "Algoritmos para el establecimiento de conexiones neuronales híbridas" [Reyes-Sanchez, 2017], and thus have not been described in this work.

Algorithm 4.2: rt_thread

```
input: rt_args
begin
   message.n_in_chan = rt_args.n_in_chan;
   message.n_out_chan = rt_args.n_out_chan;

   d = open_device_comedi(dev_name)
   session = create_session_comedi(d)

   prepare_real_time()

   start_time = get_time()
   target_time = start_time + period

   for (i = 0; i < 5 * rt_args->freq * rt_args.s_points; i++)
      if (i % rt_args.s_points == 0)
         sleep_until(target_time)
         awake_time = get_time()

         message.id = 1
         message.i = i
         message.v_model = rt_args.vars[0]
         message.v_model_scaled = rt_args.vars[0] * scale
         message.lat = awake_time − target_time
         message.t_absol = awake_time − start_time
         message.t_unix = awake_time
         message.g_virtual_to_real = g_virtual_to_real
         message.g_real_to_virtual = g_real_to_virtual

         c_model = 0
         out_val[0] = c_model
         out_val[1] = v_model_scaled
         message.in_data = ret_val
         message.out_data = out_val

         write_comedi(session, rt_args−>n_out_chan, rt_args−>out_channels, out_val)

         send_to_queue(rt_args.msqid, message)

         target_time = target_time + period

         ret_val = read_comedi(session, args.n_in_chan, args.in_channels)
      end

      v_real_scaled = ret_val[0] * scale
      c_real = 0
      message.c_real = c_real;

      rt_args.func(rt_args.dim, rt_args.dt, rt_args.vars, rt_args.params, c_real)
   end

   for (i = 0; i < rt_args.points * rt_args.s_points; i++)
      if (i % rt_args.s_points == 0)
         sleep_until(target_time)
         awake_time = get_time()

         message.id = 1
         message.i = i
         message.v_model = rt_args.vars[0]
         message.v_model_scaled = rt_args.vars[0] * scale
         message.lat = awake_time − target_time
```

```
                message.t_absol = awake_time − start_time
                message.t_unix = awake_time
                message.g_virtual_to_real = g_virtual_to_real
                message.g_real_to_virtual = g_real_to_virtual

                c_model = rt_args.syn(v_model_scaled, v_real)
                out_val[0] = c_model
                out_val[1] = v_model_scaled
                message.in_data = ret_val
                message.out_data = out_val

                write_comedi(session, rt_args−>n_out_chan, rt_args−>out_channels, out_val)

                send_to_queue(rt_args.msqid, message)

                target_time = target_time + period

                ret_val = read_comedi(session, args.n_in_chan, args.in_channels)
            end

            v_real_scaled = ret_val[0] ∗ scale
            c_real = rt_args.syn(v_real_scaled, v_model)
            message.c_real = c_real;

            rt_args.func(rt_args.dim, rt_args.dt, rt_args.vars, rt_args.params, c_real)
        end

        close_device_comedi(d)
    end
```

Even with all the work carried out during an iteration of the loop (calibration, reading and writing from DAQ, sending to the queue, etc) and the calculation of the model itself, most of the time is spent sleeping, as can be seen in figure 4.5. More than a half of it, the process is waiting, and the model takes less than a twentieth part of the period to complete its job (this may variate depending on the model used). Nevertheless, this execution of the program was using a frequency of 10 kHz, so implementations of more consuming time models or operations with higher frequencies may lead to more severe artifacts.

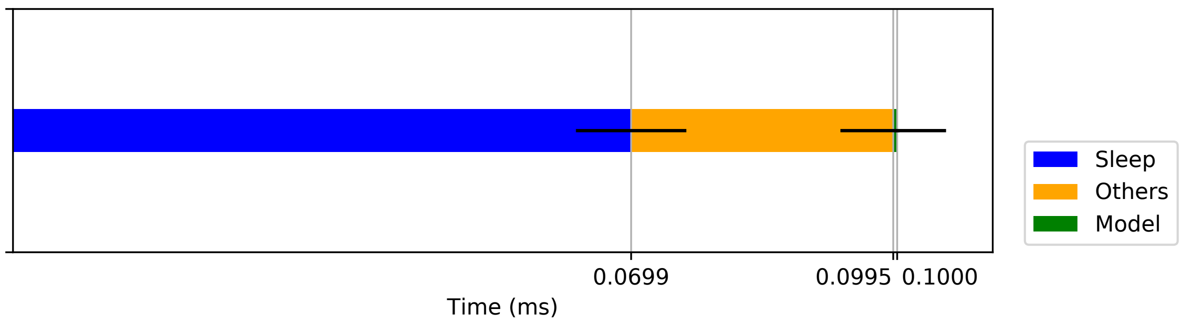


Figure 4.5: Mean time (and standard deviation) spent by the sleeping function, the calculation of the model (Izhikevich) and the rest of the operations during a 0.1 ms iteration of the loop. The values were calculated after a two minutes execution.

Algorithm 4.2 shows the pseudocode for the real-time periodic function. Implementation of this code in the different platforms (Preempt-RT, RTAI and Xenomai) is almost the same, just existing differences in the time related functions and structures (specified in table 3.1) and the IPC functions used. In order to make the rt_thread as generic as possible, the queue-related functions (*open_queue*, *close_queue*, *send_to_queue* and *receive_from_queue*) are included in a separate file, *queue_functions.c*, which implements the specific IPC methods for each solution, while in the periodic function the calls are transparent and the same for any of them. RTAI uses their own implementation of real-time queues in rt_thread, reading from it like from a file in the writer_thread. Xenomai POSIX skin [Xenomai Team, 2016a] includes its message queues services, which has been used, with the drawback of having to set the queue itself as blocking or not, but not the specific calls to the send and receive functions. Because of this, in the Preempt-RT code the *System V message queues* have been used, as they offer the same functionality but with that problem solved. Table 4.2 summarises these functions for each environment.

Table 4.2: IPC functions used by the different real-time solutions

	Preempt-RT	**RTAI**	**Xenomai (POSIX skin)**
Open	msgget	rtf_create	mq_open
Close	msgctl	close	mq_close
Send	msgsnd	rtf_put	mq_send
Receive	msgrcv	read	mq_receive

4.4 Main program

Finally, the last piece of code is the *main.c* file, the entry point to the program. It can receive the input arguments listed below:

- **-f, –frequency:** sample frequency (in kHz). Default value is 10.
- **-t, –time:** simulation time (in seconds). Default value is 0.
- **-m, –model:** neuron model (0 = Izhikevich, 1 = Hindmarsh-Rose, 2 = Rulkov Map). Default value is 0.
- **-s, –synapse:** synapse type (0 = electrical, 1 = chemical). Default value is 0.
- **-i, –input_channels:** input channels, separated by commas (e.g. 0,2,3,7). By default is empty, which means no input.
- **-o, –output_channels:** output channels, separated by commas (e.g. 0,2,3,7). By default is empty, which means no output.
- **-a, –antiphase:** turn on antiphase in electrical synapse (phase in the chemical). Default value is 0.
- **-c, –calibration:** automatic calibration process. Default value is 0.
- **-h, –help:** print help.

After receiving and parsing the input arguments, the main function initialises the *rt_ args* and *writer_ args* structures with the appropriate values for each model, creates the corresponding message queue and threads and waits until both of them are done to finish the execution of the program. One more latency test was run in order to check the impact that the addition of all this new code, and specially the Comedi input/output functions, implies to the real-time performance of the algorithm. The version tested was the one for Preempt-RT, in the same computer as in Chapter 3 (Intel I7 processor), with similar outcome (see figure 4.6).

As stated in previous chapters, an RTAI program can work in two different environments: the kernel space and the user space. User space real-time with RTAI is achieved by using the LXRT

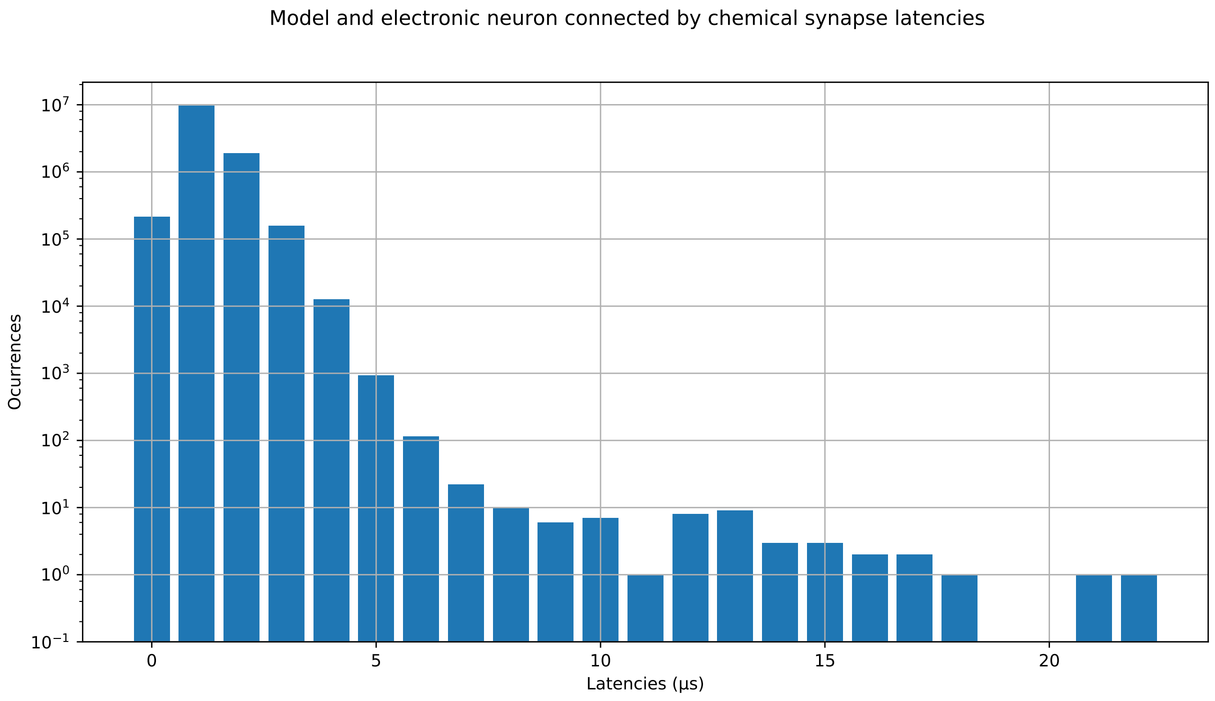


Figure 4.6: Latencies during a 20 minutes two-neuron interaction between an Izhikevich model and an electronic neuron, connected by a chemical synapse. Sysbench workload was running on a Intel I7 processor.

module, but is also quite easy in this mode to do something that makes the task to not respect the temporal restrictions strictly, and even in the documentation its use is not recommended but for seasoned RTAI programmers. Thus, a kernel space execution is preferred to ensure the hard real-time boundaries, and therefore the program must be implemented as a kernel module. Nevertheless, kernel programming have its own drawbacks [Salzman et al., 2007], as it can not use functions or structures than are not loaded into kernel themselves, like for example *printf* or *fprintf*, so communication with the kernel is done via the */proc* files [Jones, 2006]. Special directives, like *EXPORT_SYMBOL*, make the selected elements of a module to be available for others when it is loaded into kernel. The load operation is done through the *insmod* command (e.g. sudo insmod ./RTTask.ko), and it can be removed from kernel with the instruction *rmmod* (e.g. sudo rmmod RTTask).

Every module implementation has an *init* function, called when the module is loaded, that performs the first instructions of the program (in our case, starting a real-time thread, for example), and an *exit* function, called when *rmmod* is executed. These characteristics of kernel programming cause the RTAI kernel module to be a bit different from the versions for Preempt-RT and Xenomai: in the main function, the real-time thread creation should be replaced by a system call to *insmod*, to load the kernel module which will create that thread, and also all the auxiliary files (*comedi_functions*, *model_library*, etc) have to be loaded into kernel, with the necessary symbols exported. The *writer_thread* can be created as always, and when the execution is finished another system call should be performed to remove the modules from kernel.

5
Validation tests

In this chapter all the work developed in the previous pages will be tested and validated in various scenarios. The different setups, tools and technologies that will be used are detailed and explained, as well as the results obtained. Three experiments will be carried out:

- **Stepper motor experiment:** to validate the need of real-time technology on systems with low frequency activities in a device used for mechanical stimulation in neuroscience experiments.
- **Electronic neuron experiments:** to perform preliminary trials with an external neuron before implementing an hybrid circuit with living neurons.
- **Real neuron hybrid circuit experiments:** to validate the need of real-time technology on systems with high frequency activities, and therefore high precision requirements.

Clarify that in all following figures the blue signal represents the voltage generated by the model or the current injected to the biological neuron, while the orange one represents the opposite direction.

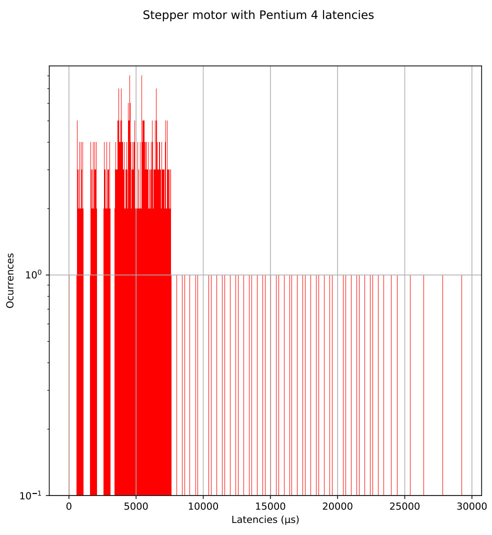
5.1 Stepper motor experiment

5.1.1 Setup

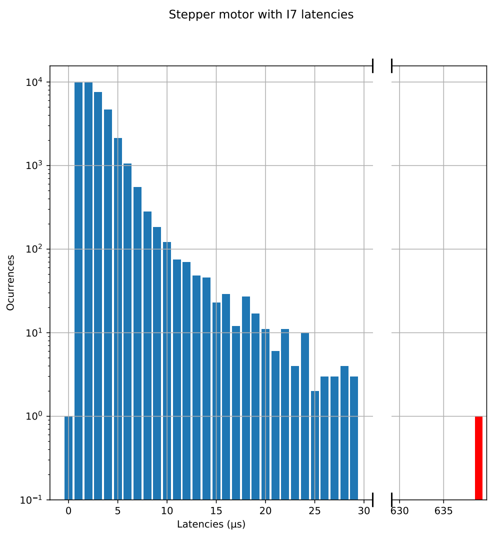
As mentioned in section 2.1, electrophysiology is not the only recording technique in neuroscience that can benefit from real-time software. In order to check the impact of this kind of technology over a different target, this time a mechanical stimulation device, part of the content of [Muñiz et al., 2008] was reproduced. The aim of that work is to use a stepper motor to perform mechanical stimulation in neuroscience experiments, and to have precise control over the device they use a real-time program, working over RTAI and a Pentium 4 with 3.2 GHz uniprocessor. For this trial their original code has been ported to a POSIX/Preempt-RT version, and proved in a computer with the same specifications, as well as the one detailed in section 3.2.4. The stepped motor used was a Mclennan 2400 stepper motor P535L482U-G17L82 (Mclennan Servo Supplies Ltd., Surrey, UK), a unipolar-geared stepper motor with 2400 steps per revolution which can receive 200 commands per second at most.

5.1.2 Results

Both their RTAI code and the Preempt-RT one worked perfectly in real-time environments. With no real-time, in the Pentium 4 computer the average latency was 4720.84 μs and the latency values distribution is shown in figure 5.1a, so in this case the use of real-time technology to control the mechanism with precision is necessary, as stated in the paper. But when the same test was repeated in the newer computer, the mechanism worked just fine even without ensuring the compliance of the temporal boundaries. The average latency obtained with the standard Debian OS (having overloaded the processor using Sysbench) was 2.88 μs, and their distribution can be seen in figure 5.1b.



(a) Pentium 4 latencies.



(b) I7 latencies.

Figure 5.1: Latency values distribution during the stepper motor experiment without real time.

It seems clear that some experiments with low frequency performance, even if not so many years ago required precise real-time technology, now can work just fine with an standard system [Fry et al., 2008]. The reasons of this improvement in the OS capabilities are numerous, like the enhancement of the hardware technology (more GHz and cores, for example) but also of the software, since, as mentioned in section 2.2, nowadays the 80% of the kernel changes that allow Linux to become an RTOS with the Preempt-RT patch are already included in the mainline kernel. It is important to emphasize that other stimulation devices, frequently used in closed-loop with electrophysiological recordings whose detection events require real-time restrictions, may in any case require the use of RTOS solutions.

5.2 Electronic neuron experiments

5.2.1 Setup

As mentioned in the first chapters of this report, implementing an analog hardware-based model that works in real-time is easier than a computational version, but it is less programmable and manageable. Even so, an analog device is very useful for preliminary tests, since they are small, simple, inexpensive and easy to connect to other devices [Pinto et al., 2000]. For this experiment an electronic implementation of a three-dimensional Hindmarsh-Rose model, which can have its I parameter (external input current) adjusted, has been used. The model frequency for all the trials was 10 kHz. The computer used for these experiments had the following characteristics:

- **Processor:** Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz, 8 hyperthreaded cores (4 physical).
- **RAM memory:** 16 GB.
- **Operating system:** Debian GNU/Linux 9.0 (stretch), kernel 4.9.0-3-rt-amd64 PREEMPT RT.
- **DAQ:** National Instruments' PCI NI BNC-2090A.

5.2.2 Results

As has been mentioned earlier, Hindmarsh-Rose model is not very flexible, i.e. the length of its bursts can be modified, but the slow wave after them remains immovable. Therefore, it is difficult to provoke big changes in its subthreshold behaviour, but it is still a good scenario to test the basic functioning of both electrical and chemical synapse. Figure 5.2 shows an example of an electrical synapse, between a computational Izhikevich model and an external electronic neuron, with conductances in both ways having value g = 0.3, and each current having opposite signs to cause the antiphase behaviour. In figure 5.3 an example of chemical synapse can be seen, again between a computational Izhikevich model and the electronic neuron. The model sends current to the external one using a fast synapse with conductance g = 0.2, while the other way round is a slow synapse with the same conductance. The chemical synapse implemented is an inhibitory one (hence always negative) and if two neurons inhibit themselves mutually using this connection an antiphase behaviour is also obtained. The temporal precision achieved by the model results in free of artefacts signals.

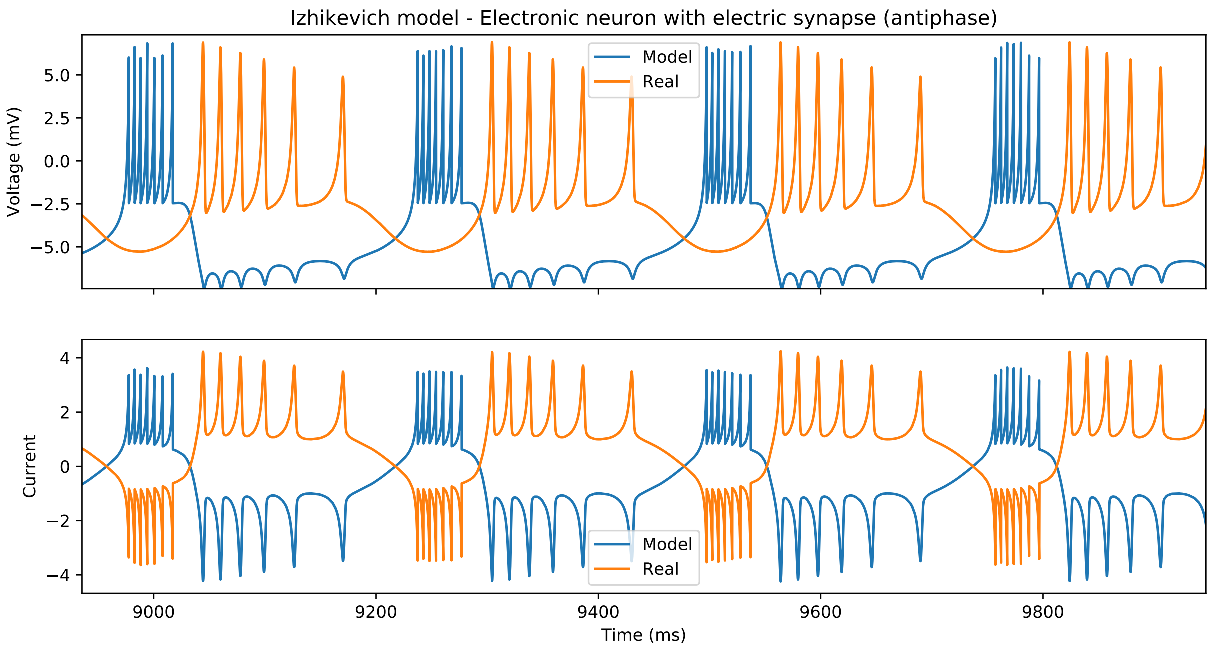


Figure 5.2: A computational Izhikevich model connected to an electronic neuron through an electric synapse.

5.3 Real neuron hybrid circuit experiments

With the performance of the real-time software tested in Chapter 3, the reliability of the neuron models implemented proven in Chapter 4 and the correct connection between neurons through

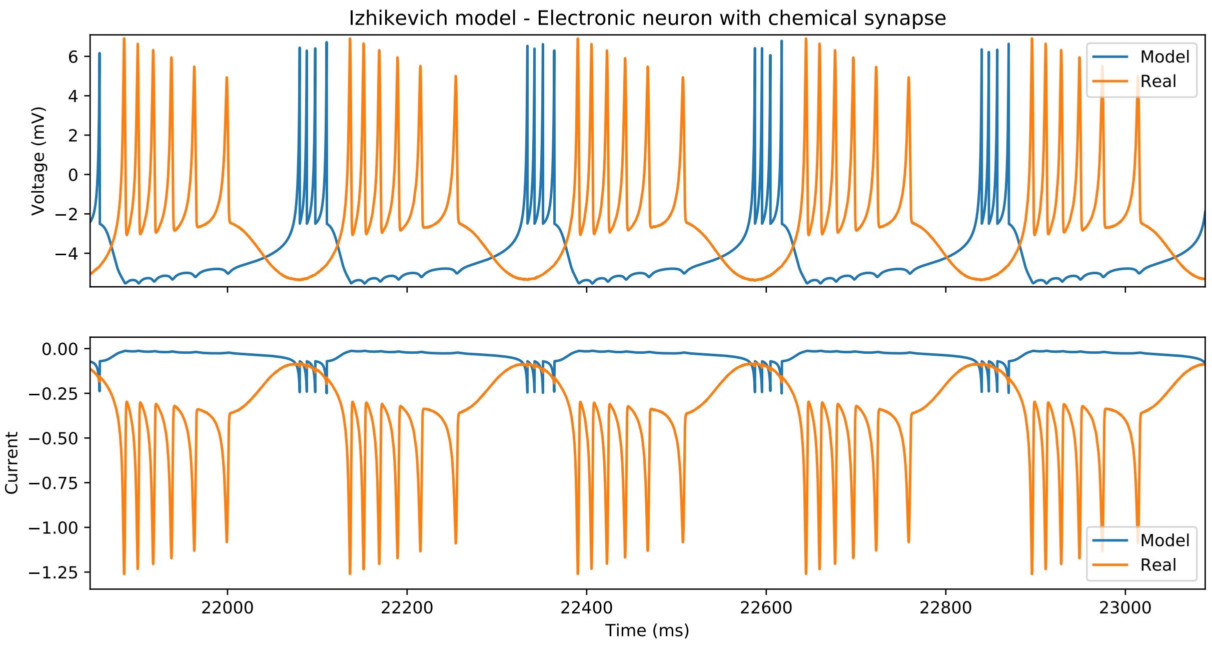


Figure 5.3: A computational Izhikevich model connected to an electronic neuron through a chemical synapse.

different kinds of synapses just confirmed in section 5.2 it is time to test the neuron models library in a hybrid circuit implementation with a real neuron.

5.3.1 Setup

Central pattern generators, commonly known as CPGs, are neural networks which control certain motor functions (such as walking, swimming, breathing, etc) through the generation of rhythmic patterns [Marder and Eisen, 1984] [Grillner, 2003]. They typically possess what is called a non-open topology, i.e. every neuron has at least one synaptic input from another neuron of the circuit [Huerta et al., 2001]. At the same time this architecture is based in oscillatory circuits with pairs of neurons that inhibit themselves mutually [Miller and Selverston, 1982] [Sakurai et al., 2014]. Despite of this networks being formed by neurons with irregular dynamics, the reciprocal inhibition between them causes them to regularize their behaviours when they are connected to other neurons of the circuit [Elices and Varona, 2015]. This rhythm negotiation in terms of the inputs, which derives in regular rhythmic patterns, is the reason why CPGs can coordinate motor functions [Selverston et al., 2000].

The biological component of the experiment is obtained using in vitro electrophysiology from a *Carcinus maenas*, commonly known as *shore crab*, which are bought in a local fish store and are kept in artificial sea water. The neural circuit that will be used for the validation tests are the pyloric and gastric CPGs from the stomatogastric ganglion. Before the procedure, the crab is anesthetized by introducing it in the freezer for 20/30 minutes, and after this its stomach is extracted. The ganglion is attached to a Petri with Sylgard and cold saline dissolution (13-15ºC kept by a microcontroller and always perfused) using pins. This saline dissolution has the following composition (in mmol/l): NaCl 433, KCl 12, $CaCl_2.2H_2O$ 12, $MgCl_2.6H_2O$ 20, Hepes 10. To adjust the pH to 7.4-7.6, NaOH 0.1 M is added. To record the intracellular activity 10MΩ crystal electrodes are used, filled with KCl 2 M.

The following tests have been carried out using one of the neurons of a pyloric CPG, specifically the one named LP, as the one shown in figure 5.5. Being a real neuron, its behaviour it is

(a) Carcinus maenas.

(b) Stomatogastric ganglion.

Figure 5.4: Pictures of the crab used in the experiments and its stomatogastric ganglion seen through the microscope.

not always perfect and can be affected by the environment conditions or the experiments that had taken place before, so it variate a little during the trials. The model frequency for all the trials was 10 kHz. The computer used was the same as in section 5.2.

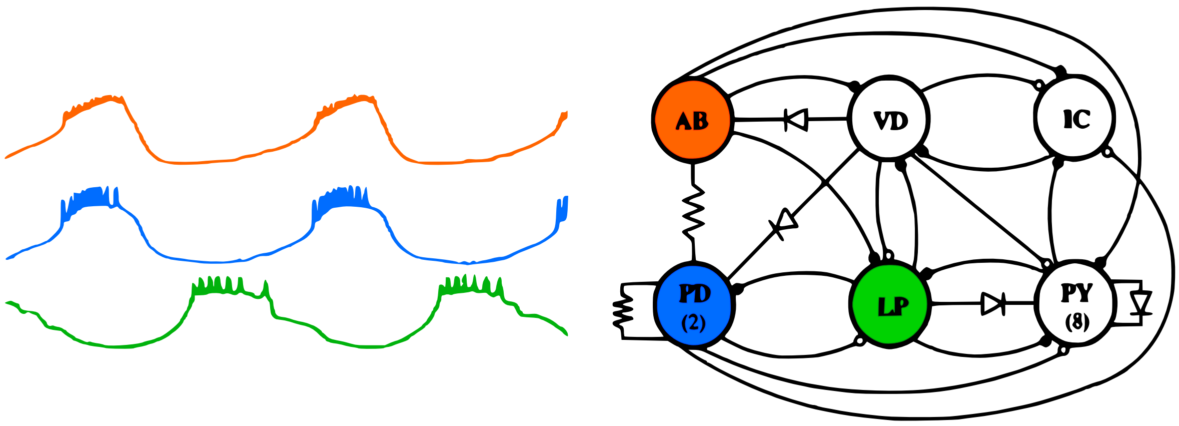


Figure 5.5: Pyloric CPG and representation of some of its neuronal behaviour.

5.3.2 Results

The first model tested was Izhikevich, using different configurations. Figure 5.6 shows the interaction between the model and the LP neuron using an electric synapse, with conductances g = 0.2 from the model to the real neuron and g = 0.3 the other way around, no antiphase behaviour caused. Figure 5.7 a represents similar case but provoking antiphase rhythm, this time using g = 0.2 conductances in both ways. The last trial carried out was with a chemical synapse, specifically a fast synapse from the real neuron to the virtual with conductance g = 0.2, and a slow one with g = 0.02 in the other direction, as can be seen in figure 5.8. It can be seen that the expected rhythm negotiation of the biological circuits is reproduced in these hybrid circuits, i.e. phase and antiphase coupling can be achieved with electrical synapse, and a regular rhythmic pattern is obtained when chemical synapse is used.

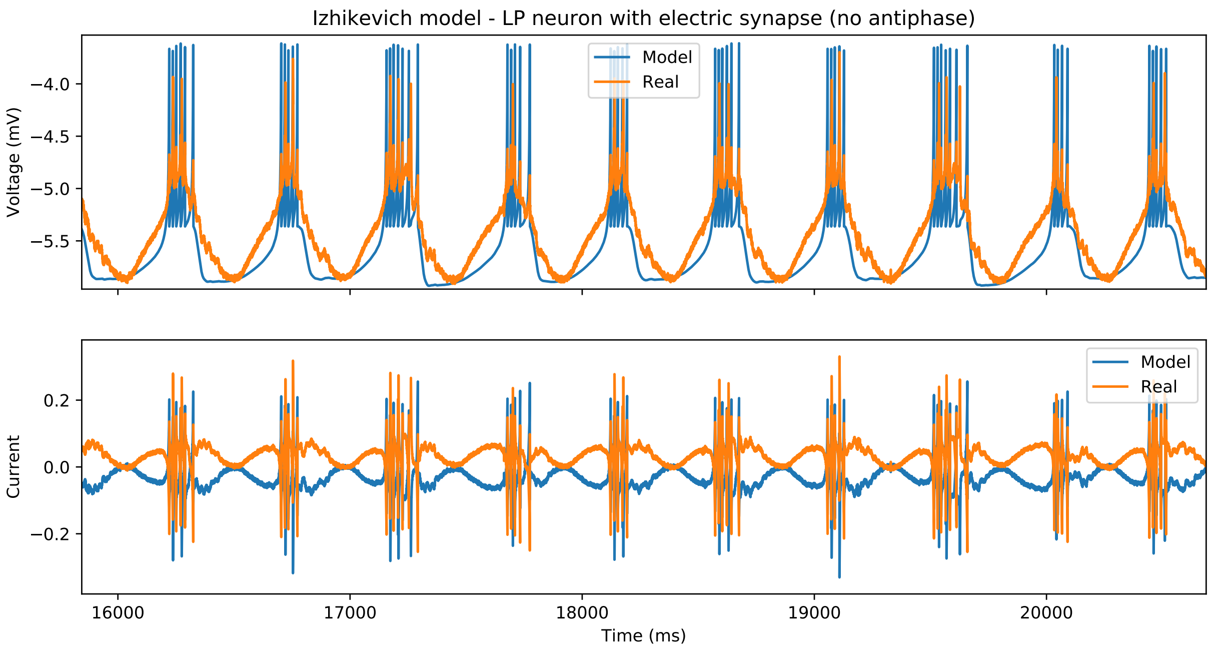


Figure 5.6: A computational Izhikevich model connected to a LP neuron through an electric synapse. Conductances g = 0.2 from the model to the neuron and g = 0.3 in the other way.

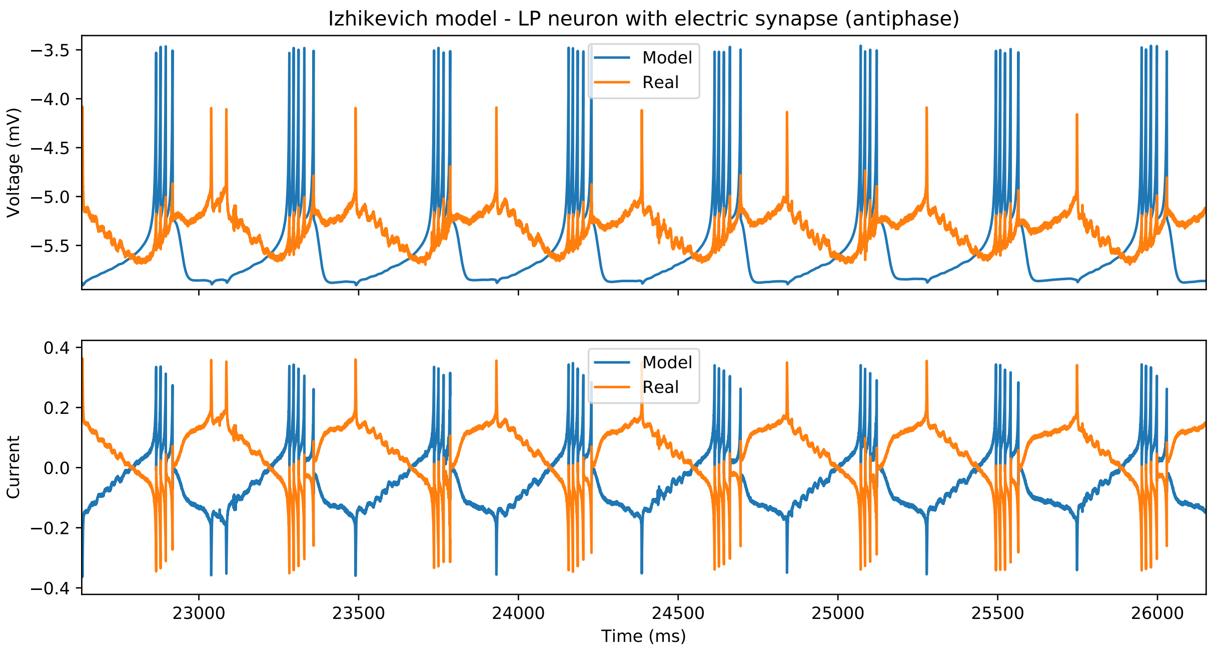


Figure 5.7: A computational Izhikevich model connected to a LP neuron through an electric synapse. Conductances g = 0.2 from the model to the neuron and g = 0.2 also in the other way, but one synapse is positive and the other negative to cause the antiphase behaviour.

The next one was Hindmarsh-Rose model, and the same scenarios were studied. In figure 5.9 the electrical connection without antiphase, with conductances g = 0.2 from the model to the LP and g = 0.3 in the other way. The case with antiphase can be seen in figure 5.10, this time with g = 0.2 in both directions. Lastly, the chemical synapse experiment is shown in figure 5.11, with a fast connection from the real neuron to the virtual with g = 0.3 and a slow synapse in the other direction with g = 0.1.

Finally, the last model tested was Rulkov Map, and the same strategy as with the other was

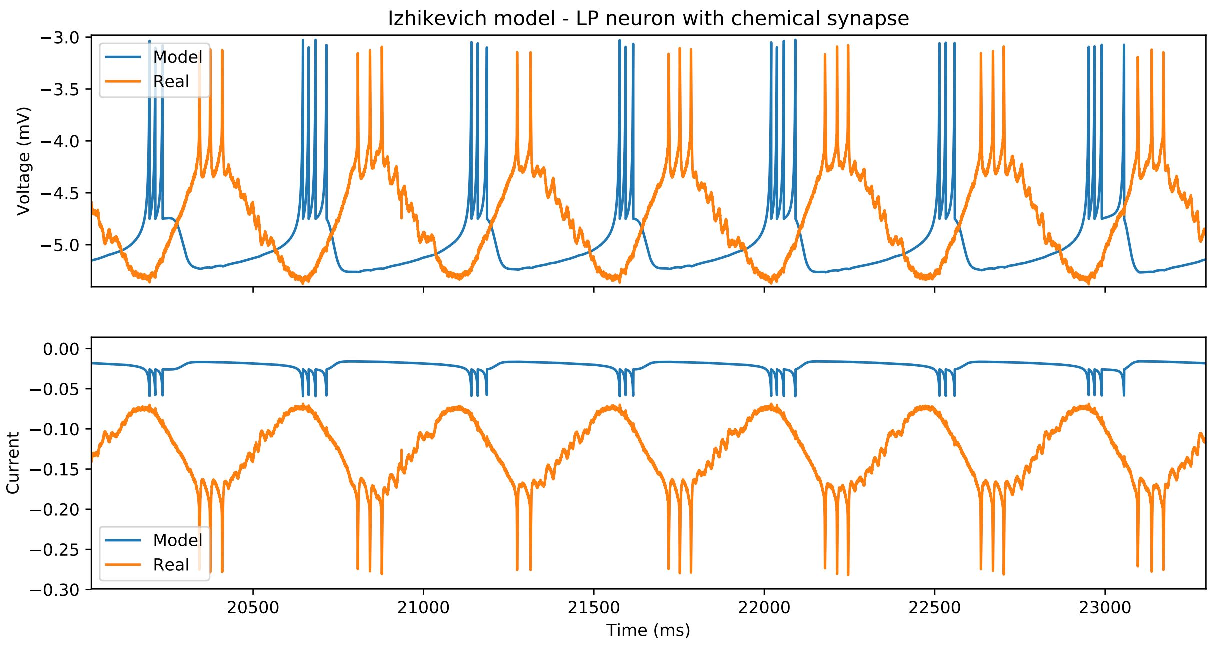


Figure 5.8: A computational Izhikevich model connected to a LP neuron through an chemical synapse. From the real neuron to the model there is a fast synapse with conductance g = 0.2, while from the virtual to the real there is a slow synapse with g = 0.02.

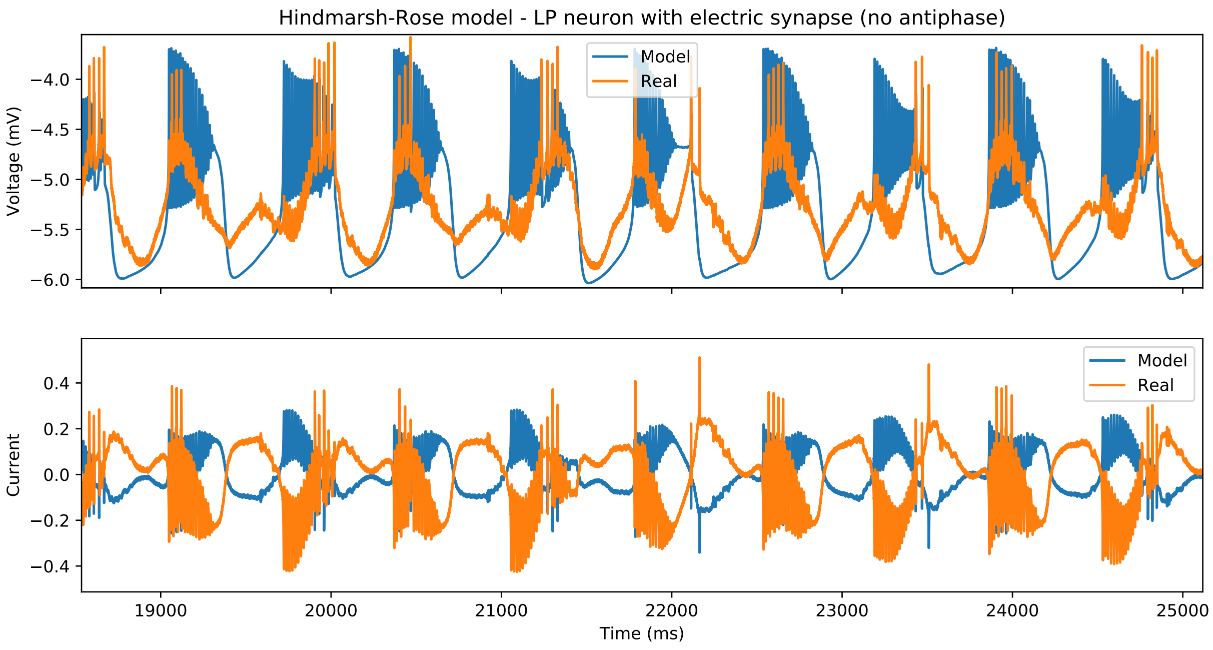


Figure 5.9: A computational Hindmarsh-Rose model connected to a LP neuron through an electric synapse. Conductances g = 0.2 from the model to the neuron and g = 0.3 in the other way.

followed. The first experiment consisted of connecting the LP neuron with the model through an electric synapse, with no antiphase, with similar g = 0.2 conductances in both direction, as is shown in figure 5.12. For the antiphase case the conductances were quite asymmetric, with the one from the real neuron to the virtual being g = 0.2 and the other g = 0.02, giving place to the behaviour seen in figure 5.13. Last of all, the chemical synapse trial was done with the same values as with Hindmarsh-Rose: from real to virtual, fast synapse with g = 0.3 and from

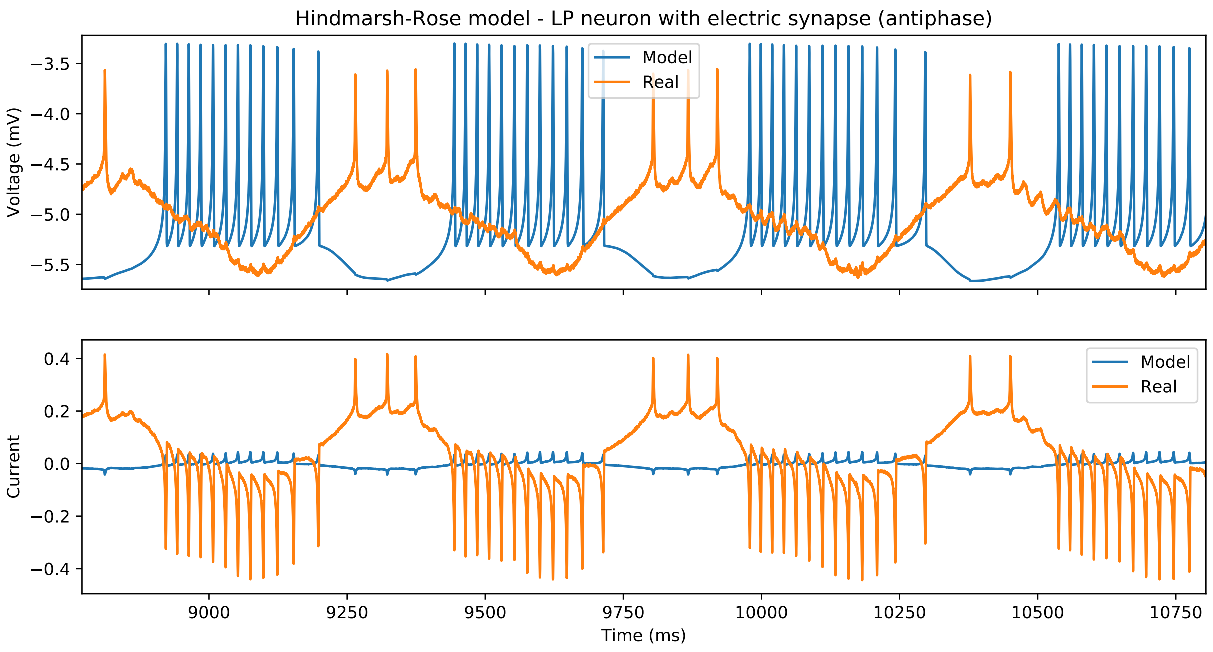


Figure 5.10: A computational Hindmarsh-Rose model connected to a LP neuron through an electric synapse. Conductances in both ways have value g = 0.2, but one synapse is positive and the other negative to cause the antiphase behaviour.

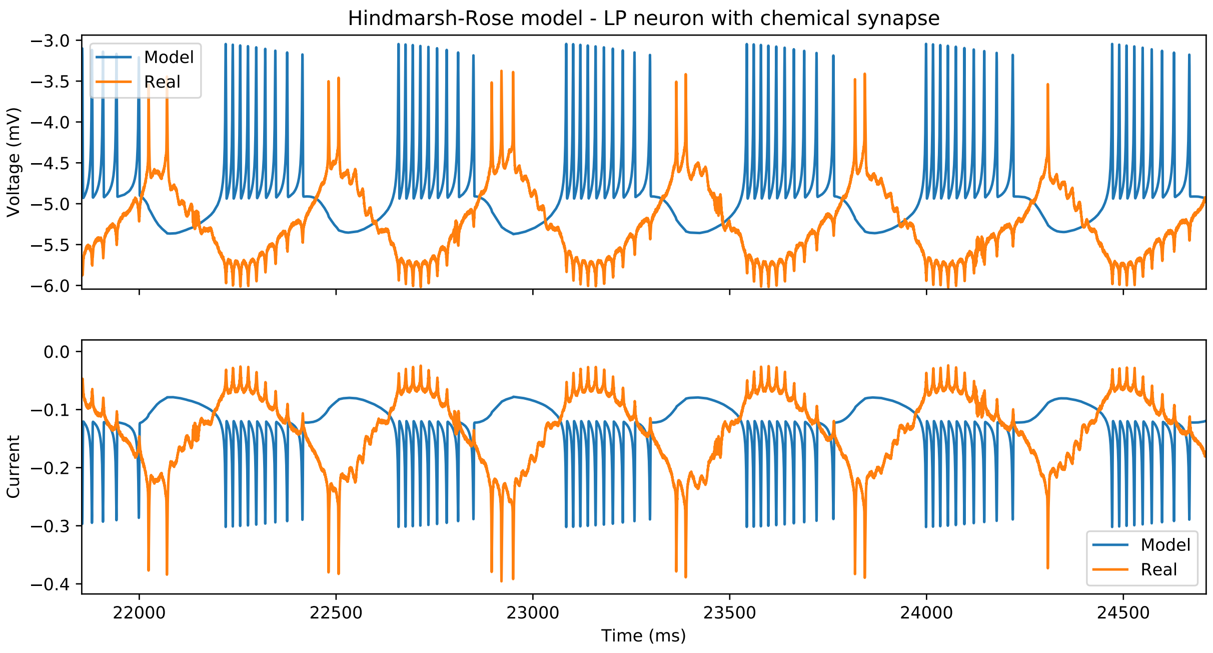


Figure 5.11: A computational Hindmarsh-Rose model connected to a LP neuron through an chemical synapse. From the real neuron to the model there is a fast synapse with conductance g = 0.3, while from the virtual to the real there is a slow synapse with g = 0.1.

virtual to real, slow synapse with g = 0.1. Figure 5.14 shows the results.

During all these tests the real-time restrictions were respected, with latency values never surpassing the 100 μs threshold. Just to illustrate this affirmation, the latency histograms corresponding with figures 5.8, 5.7, 5.9, and 5.12 are shown in figure 5.15.

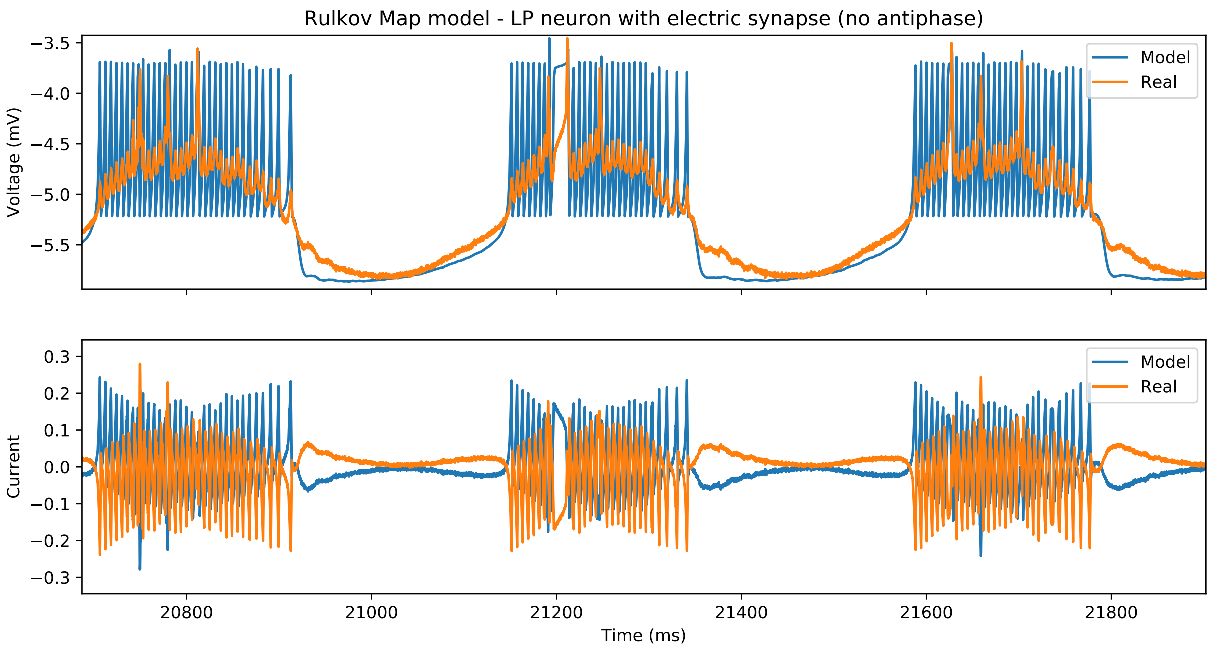


Figure 5.12: A computational Rulkov Map model connected to a LP neuron through an electric synapse. Conductances g = 0.2 from the model to the neuron and g = 0.3 in the other way.

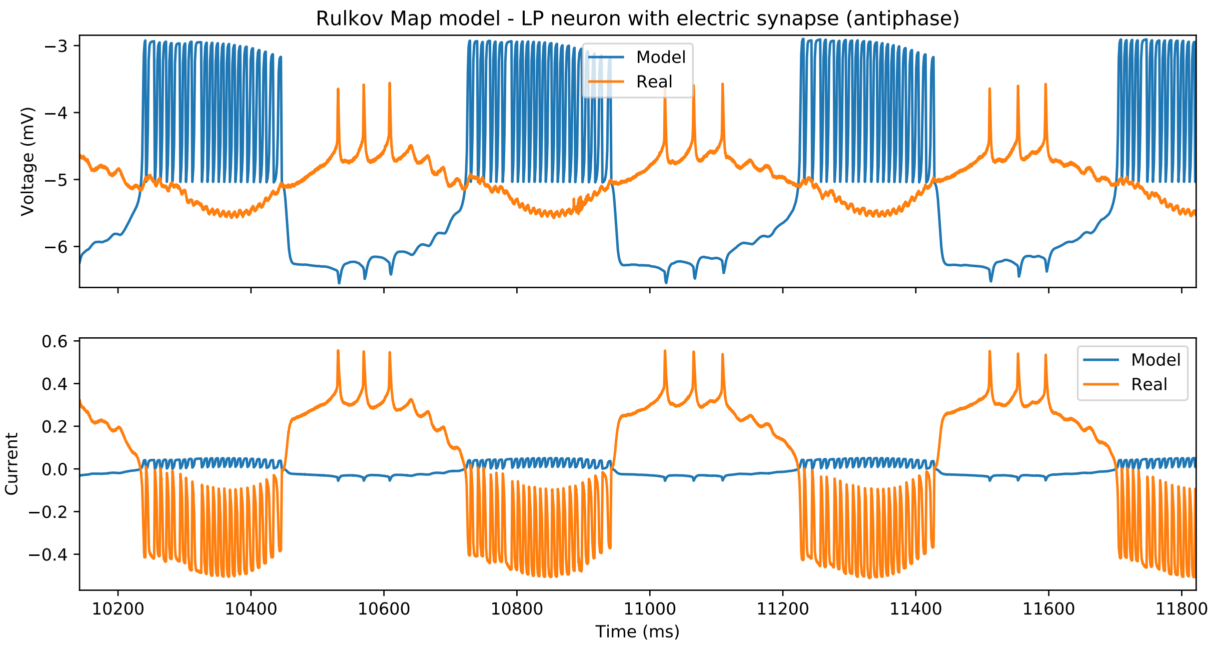


Figure 5.13: A computational Rulkov Map model connected to a LP neuron through an electric synapse. Conductances g = 0.02 from the model to the neuron and g = 0.2 in the other way.

Circuits without real-time

In order to test the real impact of not using a real-time platform during the electrophysiology experiments, one more trial was conducted: a computational Izhikevich model was connected with the previously used electronic neuron, and the model's voltage was also sent to a second computer, which recorded this signal. If everything work properly, both original and received signal should be almost identical (but for the connection noise), both in shape and temporal aspects. But if real-time is not ensured, and therefore high latency values appear, then some

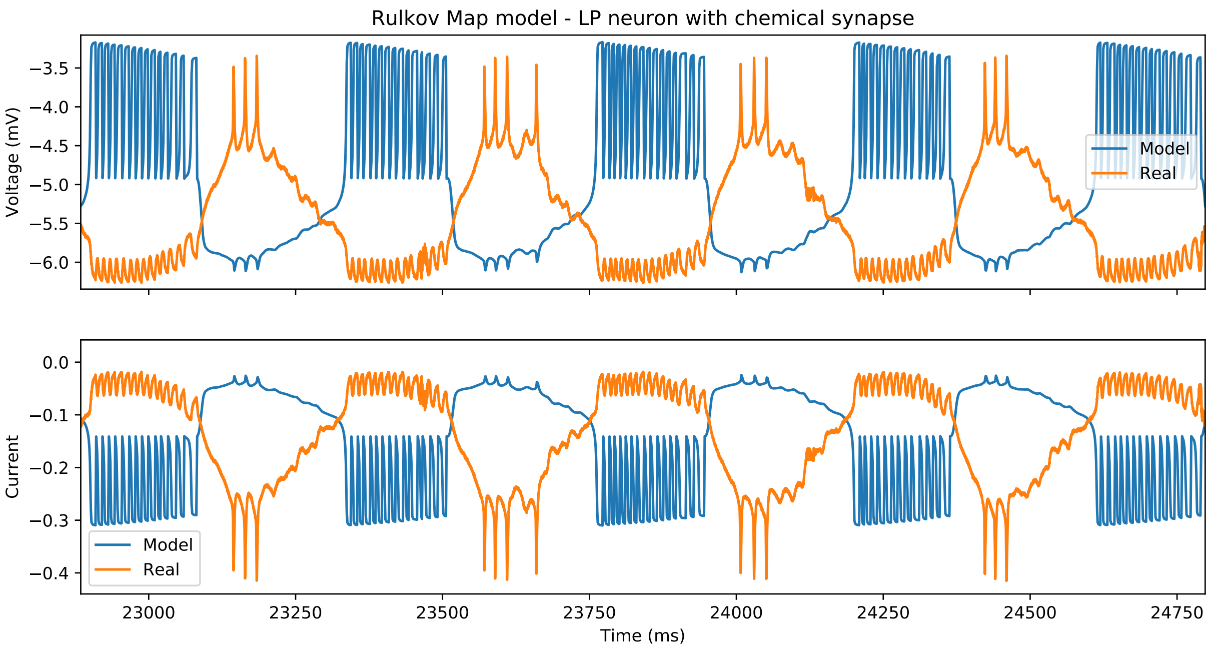
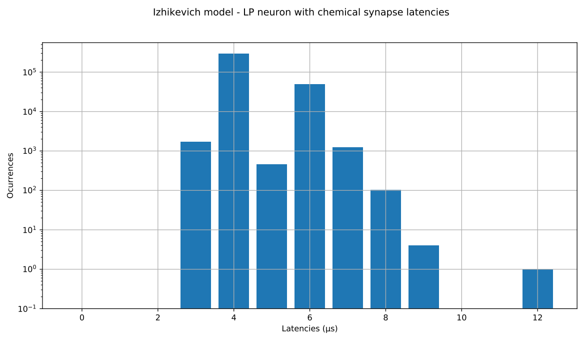
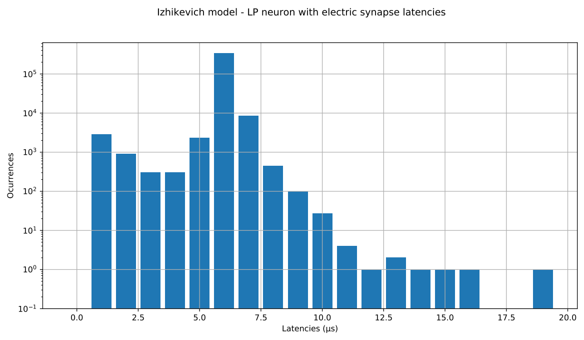


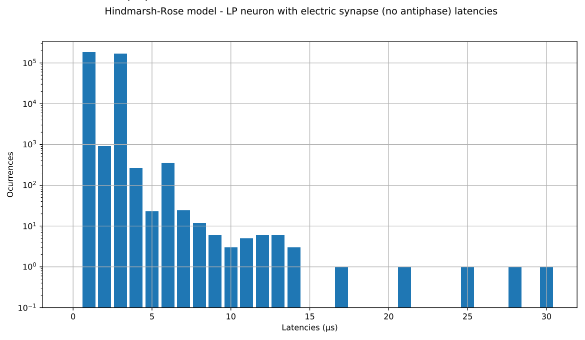
Figure 5.14: A computational Rulkov Map model connected to a LP neuron through an chemical synapse. From the real neuron to the model there is a fast synapse with conductance g = 0.3, while from the virtual to the real there is a slow synapse with g = 0.1.



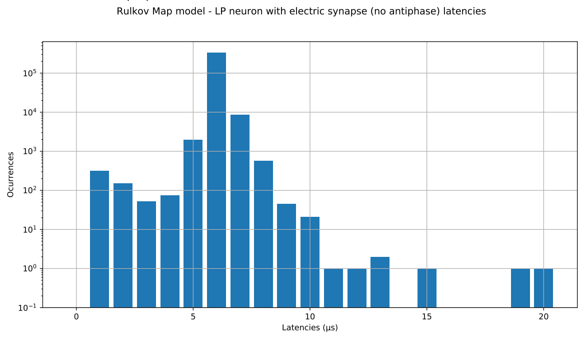
(a) Corresponds to figure 5.8.



(b) Corresponds to figure 5.7.



(c) Corresponds to figure 5.9.



(d) Corresponds to figure 5.12.

Figure 5.15: Latencies for various of the hybrid circuits experiments performed.

points can be sent late, and the receiving computer will record the same point for some iterations, until the new value arrives and the signal continues. This temporal delay will not be recovered in the remaining execution. In figure 5.16 can be seen how this event occurs around 8320 ms, and from this point the received signal have always a 6 milliseconds delay. The latency values histogram is represented in figure 5.17.

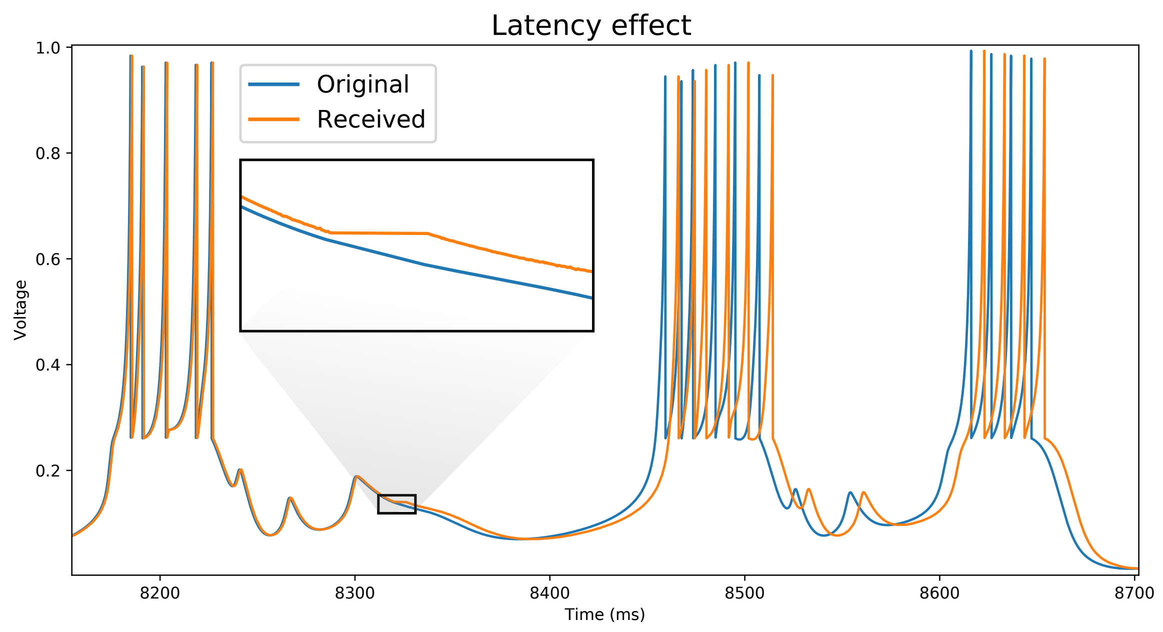


Figure 5.16: Signal delay caused by high latency values. The blue signal was recorded in the sending computer, while the orange one was the received in another one. After the real-time error occurred at 8320 ms, the received signal has a delay of 6 ms for the rest of the execution.

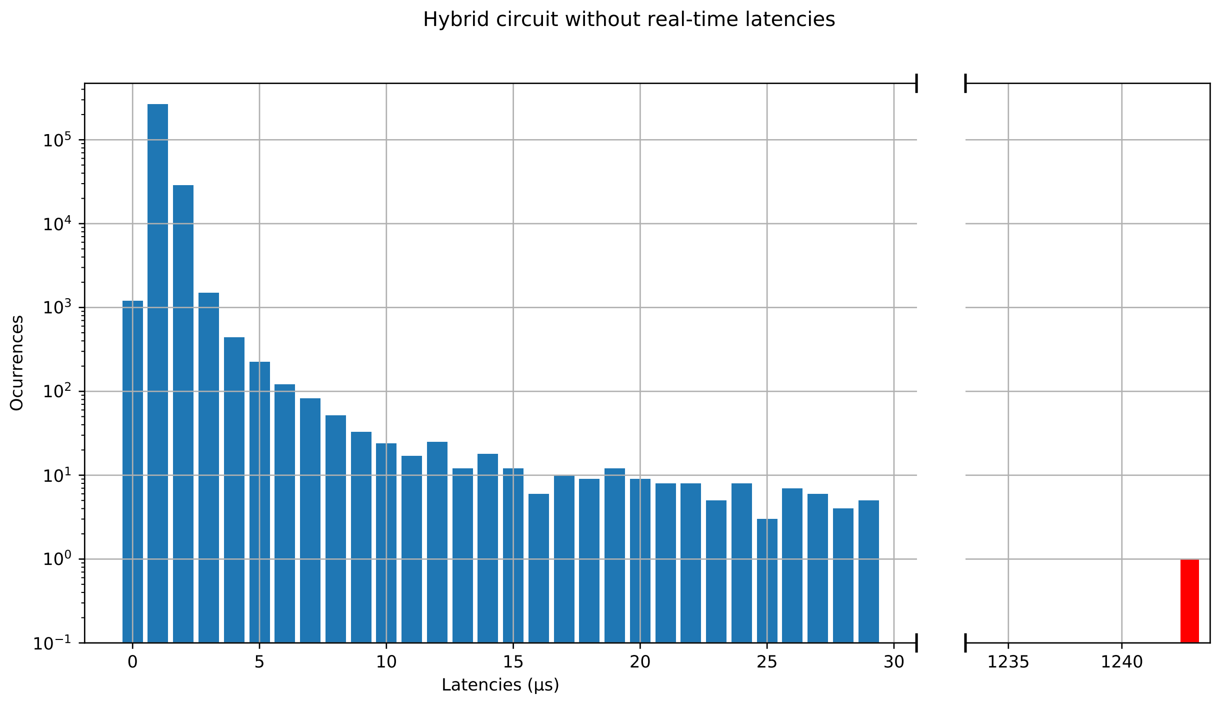


Figure 5.17: Latency values during the hybrid circuit experiment without real-time.

6 Conclusions and future work

6.1 Conclusions

The first goal of this work was to compare different RTOS solutions and list their advantages and disadvantages. The three tools analyzed have been RTAI, Xenomai and Preempt-RT, since they are free and open-source and work over Linux, which also share this characteristics. If the capacity of complying the temporal restrictions is studied, the best performance is given by the dual-kernel implementations, especially RTAI (under 5 μs), followed by Xenomai (under 10 μs). Preempt-RT obtains higher latency values (around 30 μs, can reach 80 μs) than the other alternatives, but still could be a reasonable choice for most neuroscience closed-loop experiments. In terms of usability and accessibility, Preempt-RT wins by far, since in usage aspects is just a standard Linux OS and its installation for some distributions is really easy, while Xenomai and RTAI both are quite difficult to install, even having previous experience in Computer Science and following detailed instructions, and only work through their own special programming APIs. Another strong point in favour of Preempt-RT is that it nowadays is a Linux Foundation project, meaning that the maintenance and supporting is greater than the ones for RTAI and Xenomai. This work has also shown that the same code can be used in Linux with Preempt-RT and without real-time scenarios, while RTAI and Xenomai codes only work on those platforms and in real-time. This versatility favours the spread of protocols in different kinds of applications, which can be run by any UNIX user, with real-time if they need it, or not.

It is important to mention that isolating a core and setting the affinity of a process to it does not improve its temporal performance in real-time platforms (when possible). In a standard OS without real-time it does not prevent root processes of using the isolated core, thus the latency values are not predictable, but at least user tasks will not interfere so in some machines it can help to achieve soft real-time.

Once the real-time platforms had been compared, the next objective was to development an stand-alone command-line neuron models library which used this technology. The main version of it was implemented for Preempt-RT for the reasons just mentioned, and then ported to RTAI and Xenomai. Rulkov Map, Izhikevich and Hindmarsh-Rose models were implemented, as well as the electrical and gradual chemical synapses, using in all cases C programming language, since it fits the temporal requirements quite well and is also the one used by the dual-kernel solutions APIs. In this version, also the automatic calibration work from [Reyes-Sanchez, 2017] was implemented. To connect the computer to other neurons a DAQ device is needed, and in this case the one used was a National Instruments' PCI NI BNC-2090A, controlled through the open-source Comedi drivers. The singularities of RTAI and Xenomai APIs (in addition to the lack of documentation an examples in many cases) makes it more difficult to program for this environments. In particular, the best way to ensure the compliance of the temporal boundaries in RTAI is to develop the program as a kernel module, thus having several limitations. The program also record relevant information of each experiments, as the values of the signals or the parameters used, which is really important in closed-loop.

Finally, different trials were carried out in order to validate the correct behaviour of the real-time library developed. From the results of those tests the proper functioning of the program was checked, for every neuron and synapse model, building a hybrid circuit both with an electronic neuron and a biological one. Another observation that can be drawn from these experiments is that not all the closed-loop neuroscience experiments require an RTOS implementation to its proper execution with the actual computer performance, as the stepper motor case, simulations (which can be faster or slower depending the case), or the ones whose real-time performance rely on hardware components. Others will require stricter temporal boundaries, as electrophysiology ones, hence standardized real-time software technology for neuroscience closed-loop experiments would come in handy for many laboratories and researchers that nowadays find it difficult to deal with these tools.

6.2 Future work

This work was developed as a first approach to a project that will be continued in a PhD thesis, and hence there is plenty future work to be done:

- First of all, optimize the neuron model library, as well as improve some aspects of its usage.
- Implement all the features of the program also in RTAI and Xenomai, including the calibration functions.
- Add more models, both of neurons and synapses, to the library.
- Add the model library to the RTBiomanager software developed by the GNB in [Muñiz et al., 2009], which nowadays works on RTAI, so it should also be ported to Xenomai and Preempt-RT. most closed-loop protocols would benefit from graphic tools for the design and visualization of its performance.
- Dissemination of the technology and use of other recording techniques: optogenetics, video, BCI, behaviour experiments, etc.

In summary, in the future more work in order to design and implement standardized real-time software technology, easy to install and use for any kind of user is needed. If this is achieved, many researching groups and laboratories would see feasible to perform closed-loop experiments for online control and interaction in different fields, from mechanical or visual stimulation, BCI interactions or, of course, electrophysiology.

Bibliography

Rafael V Aroca and Glauco Caurin. A Real Time Operating Systems (RTOS) Comparison. 2009. URL `http://home.iitj.ac.in/{~}saurabh.heda/Papers/Survey/RTOSPerformanceComparison-2008.pdf`.

Cédric Arrouët, Marco Congedo, Jean-Eudes Marvie, Fabrice Lamarche, Anatole Lécuyer, and Bruno Arnaldi. Open-ViBE: A Three Dimensional Platform for Real-Time Neuroscience. *Journal of Neurotherapy*, 9(1):3–25, jul 2005. doi: 10.1300/J184v09n01_02. URL `http://www.isnr-jnt.org/article/view/16837`.

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. *IEEE Transactions on Nuclear Science*, 55(1):435–439, 2008. ISSN 0018-9499. doi: 10.1109/TNS.2007.905231. URL `http://ieeexplore.ieee.org/document/4448543/`.

Jennifer A. Bauer, Katherine M. Lambert, and John A. White. The past, present, and future of real-time control in cellular electrophysiology. *IEEE Transactions on Biomedical Engineering*, 61(5):1448–1456, 2014. ISSN 15582531. doi: 10.1109/TBME.2014.2314619.

István Biró and Michele Giugliano. A reconfigurable visual-programming library for real-time closed-loop cellular electrophysiology. *Frontiers in neuroinformatics*, 9:17, 2015. doi: 10.3389/fninf.2015.00017. URL `http://www.ncbi.nlm.nih.gov/pubmed/26157385http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=PMC4477165`.

G. Bollella and J. Gosling. The real-time specification for Java. *Computer*, 33(6):47–54, jun 2000. ISSN 00189162. doi: 10.1109/2.846318. URL `http://ieeexplore.ieee.org/document/846318/`.

Frédéric D Broccard, Siddharth Joshi, Jun Wang, and Gert Cauwenberghs. Neuromorphic neural interfaces: from neurophysiological inspiration to biohybrid coupling with nervous systems. *Journal of Neural Engineering*, 2017. URL `http://iopscience.iop.org/1741-2552/14/4/041002`.

Alan Burns and Andrew J. Wellings. *Real-time systems and programming languages : Ada, Real-Time Java and C/Real-Time POSIX*. Addison-Wesley, 2009. ISBN 9780321417459. URL `https://www.cs.york.ac.uk/rts/books/RTSBookFourthEdition.html`.

Pablo Chamorro, Rafael Levi, Francisco B Rodríguez, Reynaldo D Pinto, and Pablo Varona. Real-time activity-dependent drug microinjection. *BMC Neuroscience*, 10(Suppl 1):P296, 2009. ISSN 1471-2202. doi: 10.1186/1471-2202-10-S1-P296. URL `http://bmcneurosci.biomedcentral.com/articles/10.1186/1471-2202-10-S1-P296`.

Pablo Chamorro, Carlos Muñiz, Rafael Levi, David Arroyo, Francisco B. Rodriguez, and Pablo Varona. Generalization of the Dynamic Clamp Concept in Neurophysiology and Behavior. *PLoS ONE*, 7(7):e40887, jul 2012. doi: 10.1371/journal.pone.0040887. URL `http://dx.plos.org/10.1371/journal.pone.0040887`.

Kenneth S. Cole and Howard J. Curtis. ELECTRIC IMPEDANCE OF THE SQUID GIANT AXON DURING ACTIVITY. *The Journal of General Physiology*, 22(5), 1939. URL `http://jgp.rupress.org/content/22/5/649`.

Ellen Covey and Matt Carter. *Basic electrophysiological methods.* 2015. ISBN 9780199939862. URL `http://www.worldcat.org/title/basic-electrophysiological-methods/oclc/900633113`.

Alain Destexhe and Thierry. Bal. *Dynamic-clamp : from principles to applications.* Springer, 2009. ISBN 9780387892795. URL `https://books.google.es/books?hl=es{&}lr={&}id=MpMesDViOOOC{&}oi=fnd{&}pg=PR4{&}dq=destexhe+dynamic+clamp{&}ots=j2diXUpJcO{&}sig=HpJizP5Jt{_}BbCg043Ak3fYBPxh4{#}v=onepage{&}q=destexhedynamicclamp{&}f=false`.

Sven-Thorsten Dietrich and Daniel Walker. The Evolution of Real-Time Linux. 2005. URL `http://linuxdevices.linuxgizmos.com/ldfiles/rtlws-2005/SvenThorstenDietrich.pdf`.

Alan D Dorval, David J Christini, and John A White. Real-Time Linux Dynamic Clamp: A Fast and Flexible Way to Construct Virtual Ion Channels in Living Cells. 2001. doi: 10.1114/1.1408929ÍŤ. URL `https://link.springer.com/content/pdf/10.1114{%}2F1.1408929.pdf`.

Lorenzo Dozio and Paolo Mantegazza. Linux Real Time Application Interface (RTAI) in low cost high performance motion control. 2003.

Irene Elices and Pablo Varona. Closed-loop control of a minimal central pattern generator network. *Neurocomputing*, 170:55–62, dec 2015. ISSN 09252312. doi: 10.1016/j.neucom.2015.04.097. URL `http://linkinghub.elsevier.com/retrieve/pii/S0925231215008796`.

Irene Elices and Pablo Varona. Asymmetry Factors Shaping Regular and Irregular Bursting Rhythms in Central Pattern Generators. *Frontiers in computational neuroscience*, 11:9, 2017. doi: 10.3389/fncom.2017.00009. URL `http://www.ncbi.nlm.nih.gov/pubmed/28261081http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=PMC5311053`.

R Fitzhugh. Impulses and Physiological States in Theoretical Models of Nerve Membrane. *Biophysical journal*, 1(6):445–66, jul 1961. ISSN 0006-3495. URL `http://www.ncbi.nlm.nih.gov/pubmed/19431309http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=PMC1366333`.

Felix Franke, David Jäckel, Jelena Dragas, Jan Müller, Milos Radivojevic, Douglas Bakkum, and Andreas Hierlemann. High-density microelectrode array recordings and real-time spike sorting for closed-loop experiments: an emerging technology to study neural plasticity. *Frontiers in Neural Circuits*, 6:105, 2012. ISSN 1662-5110. doi: 10.3389/fncir.2012.00105. URL `http://journal.frontiersin.org/article/10.3389/fncir.2012.00105/abstract`.

Steven N. Fry, Nicola Rohrseitz, Andrew D. Straw, and Michael H. Dickinson. TrackFly: Virtual reality for a behavioral system analysis in free-flying fruit flies. *Journal of Neuroscience Methods*, 171(1):110–117, 2008. ISSN 01650270. doi: 10.1016/j.jneumeth.2008.02.016. URL `http://www.sciencedirect.com/science/article/pii/S0165027008001210`.

Borko Furht, Dan Grostick, David Gluch, Guy Rabbat, John Parker, and Meg McRoberts. Introduction to Real-Time Computing. pages 1–35. Springer US, 1991. doi: 10.1007/978-1-4615-3978-0_1. URL `http://link.springer.com/10.1007/978-1-4615-3978-0{_}1`.

Wulfram. Gerstner, Werner M. Kistler, Richard. Naud, and Liam. Paninski. *Neuronal dynamics : from single neurons to networks and models of cognition.* 2014. ISBN 1107060834. URL `https://books.google.es/books/about/Neuronal{_}Dynamics.html?id=D4j2AwAAQBAJ{&}redir{_}esc=y`.

Thomas Gleixner and Douglas Niehaus. Hrtimers and Beyond: Transforming the Linux Time Subsystems. 2006. URL `https://www.landley.net/kdocs/ols/2006/ols2006v1-pages-333-346.pdf`.

Jorge Golowasch, Michael Casey, L F Abbott, and Eve Marder. Network Stability from Activity-Dependent Regulation of Neuronal Conductances. 1999. URL `http://neurotheory.columbia.edu/Larry/GolowaschNeuralComp99.pdf`.

Sten Grillner. The motor infrastructure: from ion channels to neuronal networks. *Nature Reviews Neuroscience*, 4(7):573–586, jul 2003. ISSN 1471003X. doi: 10.1038/nrn1137. URL `http://www.nature.com/doifinder/10.1038/nrn1137`.

Prasanna Hambarde, Rachit Varma, and Shivani Jha. The Survey of Real Time Operating System: RTOS. In *2014 International Conference on Electronic Systems, Signal Processing and Computing Technologies*, pages 34–39. IEEE, jan 2014. ISBN 978-1-4799-2102-7. doi: 10.1109/ICESC.2014.15. URL `http://ieeexplore.ieee.org/document/6745342/`.

Michiel. Hazewinkel. *Encyclopaedia of mathematics.* Springer-Verlag, 2002. ISBN 1402006098. URL `https://www.encyclopediaofmath.org/index.php/Linear{_}interpolation`.

Luís Henriques. Threaded IRQs on Linux PREEMPT-RT Linux PREEMPT-RT. 2009. URL `http://www.artist-embedded.org/docs/Events/2009/OSPERT/OSPERT09-Henriques.pdf`.

J. L. Hindmarsh and R. M. Rose. A model of the nerve impulse using two first-order differential equations. *Nature*, 296(5853):162–164, mar 1982. ISSN 0028-0836. doi: 10.1038/296162a0. URL `http://www.nature.com/doifinder/10.1038/296162a0`.

J L Hindmarsh and R M Rose. A model of neuronal bursting using three coupled first order differential equations. *Proceedings of the Royal Society of London. Series B, Biological sciences*, 221(1222):87–102, mar 1984. ISSN 0950-1193. URL `http://www.ncbi.nlm.nih.gov/pubmed/6144106`.

Hodgkin and AF Huxley. Action Potentials record from inside a nerve fiber. *Nature (Lond)*, 144, 1939. URL `http://www.citeulike.org/group/448/article/3332664`.

A. L. Hodgkin and A. F. Huxley. The components of membrane conductance in the giant axon of Loligo. *The Journal of Physiology*, 116(4):473–496, apr 1952a. doi: 10.1113/jphysiol.1952.sp004718. URL `http://doi.wiley.com/10.1113/jphysiol.1952.sp004718`.

A. L. Hodgkin and A. F. Huxley. Currents carried by sodium and potassium ions through the membrane of the giant axon of Loligo. *The Journal of Physiology*, 116(4):449–472, apr 1952b. doi: 10.1113/jphysiol.1952.sp004717. URL `http://doi.wiley.com/10.1113/jphysiol.1952.sp004717`.

A. L. Hodgkin and A. F. Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of physiology*, 117(4):500–44, aug 1952c. URL `http://www.ncbi.nlm.nih.gov/pubmed/12991237http://www.ncbi.nlm.nih.gov/pubmed/12991237`.

A. L. Hodgkin, A. F. Huxley, and B. Katz. Measurement of current-voltage relations in the membrane of the giant axon of Loligo. *The Journal of Physiology*, 116(4):424–448, apr 1952. doi: 10.1113/jphysiol.1952.sp004716. URL `http://doi.wiley.com/10.1113/jphysiol.1952.sp004716`.

R. Huerta, P. Varona, M. I. Rabinovich, and Henry D. I. Abarbanel. Topology selection by chaotic neurons of a pyloric central pattern generator. *Biological Cybernetics*, 84(1):L1–L8, jan 2001. ISSN 0340-1200. doi: 10.1007/PL00007976. URL `http://link.springer.com/10.1007/PL00007976`.

T E Hull, W H Enright, B M Fellen, A E Sedgwick, and A E Sedgwickt. Comparing Numerical Methods for Ordinary Differential Equations. *Source: SIAM Journal on Numerical Analysis SIAM J. NUMER. ANAL*, 9(4):603–637, 1972. URL `http://www.jstor.org/stable/2156215`.

Eugene M Izhikevich. Simple Model of Spiking Neurons. *IEEE TRANSACTIONS ON NEURAL NETWORKS*, 14(6), 2003. doi: 10.1109/TNN.2003.820440.

M Tim Jones. Access the Linux kernel using the /proc filesystem This virtual filesystem opens a window of communication between the kernel and user space. 2006. URL `http://www.athena.nitc.ac.in/kousthub{_}b130733cs/CollegeBooks/S5/OperatingSystems/Notes/lastlectureppt/l-proc-pdf.pdf`.

Eric R. Kandel. *Principles of neural science*. 2013. ISBN 0071390111. URL `https://books.google.es/books/about/Principles{_}of{_}Neural{_}Science{_}Fifth{_}Editi.html?id=s64z-LdAIsEC{&}redir{_}esc=y`.

Alexey Kopytov. SysBench manual, 2009. URL `http://imysql.com/wp-content/uploads/2014/10/sysbench-manual.pdf`.

Angel Lareo, Caroline G. Forlim, Reynaldo D. Pinto, Pablo Varona, and Francisco de Borja Rodriguez. Temporal code-driven stimulation: Definition and application to electric fish signaling. *Frontiers in Neuroinformatics*, 10:41, 2016. ISSN 1662-5196. doi: 10.3389/fninf.2016.00041. URL `http://journal.frontiersin.org/article/10.3389/fninf.2016.00041`.

G Le Masson, S Le Masson, and M Moulins. From conductances to neural network properties: analysis of simple circuits using the hybrid network method. *Progress in biophysics and molecular biology*, 64(2-3):201–20, 1995. ISSN 0079-6107. URL `http://www.ncbi.nlm.nih.gov/pubmed/8987384`.

R J Lin, J Bettencourt, J A White, D J Christini, and R J Butera. Real-time Experiment Interface for biological control applications. In *2010 Annual International Conference of the IEEE Engineering in Medicine and Biology*, pages 4160–4163. IEEE, aug 2010. ISBN 978-1-4244-4123-5. doi: 10.1109/IEMBS.2010.5627397. URL `http://ieeexplore.ieee.org/document/5627397/`.

Risa Lin. Real-time eXperiment Interface (RTXI) User Guide. 2011.

Daniele Linaro, João Couto, and Michele Giugliano. Command-line cellular electrophysiology for conventional and real-time closed-loop experiments. *Journal of Neuroscience Methods*, 230:5–19, 2014. ISSN 01650270. doi: 10.1016/j.jneumeth.2014.04.003. URL `http://www.sciencedirect.com/science/article/pii/S0165027014001198`.

P. Mantegazza, E. L. Dozio, and S. Papacharalambous. RTAI: Real-Time Application Interface. *Linux Journal*, 2000(72es):10, 2000. URL `http://dl.acm.org/citation.cfm?id=348564`.

E Marder and J S Eisen. Transmitter identification of pyloric neurons: electrically coupled neurons use different transmitters. *Journal of neurophysiology*, 51(6):1345–61, jun 1984. ISSN 0022-3077. URL `http://www.ncbi.nlm.nih.gov/pubmed/6145757`.

James Thomas Martin. *Programming real-time computer systems.* Englewood Cliffs, N.J. : Prentice-Hall, 1965.

J. P. Miller and A. I. Selverston. Mechanisms underlying pattern generation in lobster stomatogastric ganglion as determined by selective inactivation of identified neurons. IV. Network properties of pyloric system. *Journal of Neurophysiology*, 48(6), 1982. URL `http://jn.physiology.org/content/48/6/1416.long`.

Jan Müller, Douglas J. Bakkum, and Andreas Hierlemann. Sub-millisecond closed-loop feedback stimulation between arbitrary sets of individual neurons. *Frontiers in Neural Circuits*, 6:121, 2013. doi: 10.3389/fncir.2012.00121. URL `http://journal.frontiersin.org/article/10.3389/fncir.2012.00121/abstract`.

Carlos Muñiz, Rafael Levi, Meriem Benkrid, Francisco B. Rodríguez, and Pablo Varona. Real-time control of stepper motors for mechano-sensory stimulation. *Journal of Neuroscience Methods*, 172(1):105–111, 2008. ISSN 01650270. doi: 10.1016/j.jneumeth.2008.04.017. URL `http://www.sciencedirect.com/science/article/pii/S0165027008002446`.

Carlos Muñiz, Francisco De Borja Rodríguez, and Pablo Varona. RT-Biomanager: a software platform to expand the applications of real-time technology in neuroscience. 2009. doi: 10.1186/1471-2202-10-S1-P49. URL `http://download.springer.com/static/pdf/154/art{%}253A10.1186{%}252F1471-2202-10-S1-P49.pdf?originUrl=http{%}3A{%}2F{%}2Fbmcneurosci.biomedcentral.com{%}2Farticle{%}2F10.1186{%}2F1471-2202-10-S1-P49{&}token2=exp=1490780066{~}acl={%}2Fstatic{%}2Fpdf{%}2F154{%}2Fart{%}25253A10.1186{%}2525`.

Carlos Muñiz, Caroline G Forlim, Rafael T Guariento, Reynaldo D Pinto, Francisco B Rodriguez, and Pablo Varona. Online video tracking for activity-dependent stimulation in neuroethology. *BMC Neuroscience*, 12(Suppl 1):P358, 2011. doi: 10.1186/1471-2202-12-S1-P358. URL `http://bmcneurosci.biomedcentral.com/articles/10.1186/1471-2202-12-S1-P358`.

Sriram Neelakandan, P Raghavan, and Amol Lad. EMBEDDED LINUX SYSTEM DESIGN AND DEVELOPMENT. 2005. URL `https://pixhawk.ethz.ch/{_}media/dev/literature/embedded{_}linux{_}system{_}design{_}and{_}development.pdf`.

Thomas Nowotny, Attila Szűcs, Reynaldo D. Pinto, and Allen I. Selverston. StdpC: A modern dynamic clamp. *Journal of Neuroscience Methods*, 158(2):287–299, dec 2006. ISSN 01650270. doi: 10.1016/j.jneumeth.2006.05.034. URL `http://linkinghub.elsevier.com/retrieve/pii/S0165027006002810`.

Francis A. Ortega, Robert J. Butera, David J. Christini, John A. White, and Alan D. Dorval. Dynamic Clamp in Cardiac and Neuronal Systems Using RTXI. pages 327–354. 2014. doi: 10.1007/978-1-4939-1096-0_21. URL `http://link.springer.com/10.1007/978-1-4939-1096-0{_}21`.

R. D. Pinto, P. Varona, A. R. Volkovskii, A. Szücs, Henry D. I. Abarbanel, and M. I. Rabinovich. Synchronous behavior of two coupled electronic neurons. *Physical Review E*, 62(2):2644–2656, aug 2000. ISSN 1063-651X. doi: 10.1103/PhysRevE.62.2644. URL `https://link.aps.org/doi/10.1103/PhysRevE.62.2644`.

R.D. Pinto, R.C. Elson, A. Szücs, M.I. Rabinovich, A.I. Selverston, and H.D.I. Abarbanel. Extended dynamic clamp: controlling up to four neurons using a single desktop computer and interface. *Journal of Neuroscience Methods*, 108(1):39–48, 2001. ISSN 01650270. doi: 10.1016/S0165-0270(01)00368-5. URL `http://www.sciencedirect.com/science/article/pii/S0165027001003685`.

Mario Prsa, Gregorio L. Galiñanes, and Daniel Huber. Rapid Integration of Artificial Sensory Feedback during Operant Conditioning of Motor Cortex Neurons. *Neuron*, 93(4):929–939.e6, 2017. ISSN 08966273. doi: 10.1016/j.neuron.2017.01.023. URL `http://www.sciencedirect.com/science/article/pii/S0896627317300478`.

Mohit Rana, Andrew Q Varan, Anis Davoudi, Ronald A Cohen, Ranganatha Sitaram, and Natalie C Ebner. Real-Time fMRI in Neuroscience Research and Its Use in Studying the Aging Brain. *Frontiers in aging neuroscience*, 8:239, 2016. doi: 10.3389/fnagi.2016.00239. URL `http://www.ncbi.nlm.nih.gov/pubmed/27803662http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=PMC5067937`.

Manuel Reyes-Sanchez. *Algoritmos para el establecimiento de conexiones neuronales híbridas*. PhD thesis, Universidad Autónoma de Madrid, 2017.

H P Robinson and N Kawai. Injection of digitally synthesized synaptic conductance transients to measure the integrative properties of neurons. *Journal of neuroscience methods*, 49(3):157–65, sep 1993. ISSN 0165-0270. URL `http://www.ncbi.nlm.nih.gov/pubmed/7903728`.

Nikolai F. Rulkov. Modeling of spiking-bursting neural behavior using two-dimensional map. *Physical Review E*, 65(4):041922, apr 2002. ISSN 1063-651X. doi: 10.1103/PhysRevE.65.041922. URL `http://link.aps.org/doi/10.1103/PhysRevE.65.041922`.

Akira Sakurai, Charuni A. Gunaratne, and Paul S. Katz. Two interconnected kernels of reciprocally inhibitory interneurons underlie alternating left-right swim motor pattern generation in the mollusk Melibe leonina. *Journal of Neurophysiology*, 112(6), 2014. URL `http://jn.physiology.org/content/112/6/1317.long`.

Peter Jay Salzman, Michael Burian, and Ori Pomerantz. The Linux Kernel Module Programming Guide, 2007. URL `http://www.tldp.org/LDP/lkmpg/2.6/html/lkmpg.html`.

Massimo Scanziani and Michael Hausser. Electrophysiology in the age of light. *Nature*, 461(7266):930–939, oct 2009. doi: 10.1038/nature08540. URL `http://www.nature.com/doifinder/10.1038/nature08540`.

David Schleef, Frank Mori, Bruyninckx, Bernd Porr, Ian Abbott, and Éric Piel. Comedi Documentation, 2012. URL `http://www.comedi.org/doc/`.

Allen I. Selverston, Mikhail I. Rabinovich, Henry D.I. Abarbanel, Robert Elson, Attila Szücs, Reynaldo D. Pinto, Ramón Huerta, and Pablo Varona. Reliable circuits from irregular neurons: A dynamical approach to understanding central pattern generators. *Journal of Physiology-Paris*, 94(5):357–374, 2000. ISSN 09284257. doi: 10.1016/S0928-4257(00)01101-3. URL `http://www.sciencedirect.com/science/article/pii/S0928425700011013`.

L. Sha, R. Rajkumar, and J.P. Lehoczky. Priority inheritance protocols: an approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990. ISSN 00189340. doi: 10.1109/12.57058. URL `http://ieeexplore.ieee.org/document/57058/`.

A A Sharp, M B ONeil, L F Abbott, and E Marder. Dynamic clamp: computer-generated conductances in real neurons. *Journal of Neurophysiology*, 69(3):992–995, 1993. ISSN 0022-3077. URL `http://jn.physiology.org/content/69/3/992`.

K.G. Shin and P. Ramanathan. Real-time computing: a new discipline of computer science and engineering. *Proceedings of the IEEE*, 82(1):6–24, 1994. ISSN 00189219. doi: 10.1109/5.259423. URL `http://ieeexplore.ieee.org/document/259423/`.

RTAI Team. RTAI 3.4 User Manual. *October*, 3(October):1–29, 2006.

Jacopo Tessadori, Marta Bisio, Sergio Martinoia, and Michela Chiappalone. Modular Neuronal Assemblies Embodied in a Closed-Loop Environment: Toward Future Integration of Brains and Machines. *Frontiers in Neural Circuits*, 6:99, 2012. ISSN 1662-5110. doi: 10.3389/fncir.2012.00099. URL `http://journal.frontiersin.org/article/10.3389/fncir.2012.00099/abstract`.

P. Varona, D. Arroyo, F.B. Rodríguez, and T. Nowotny. Online Event Detection Requirements in Closed-Loop Neuroscience. *Closed Loop Neuroscience*, pages 81–91, 2016. doi: 10.1016/B978-0-12-802452-2.00006-8. URL `http://linkinghub.elsevier.com/retrieve/pii/B9780128024522000068`.

Xenomai Team. Xenomai POSIX skin API. 2016a. URL `http://xenomai.org/documentation/xenomai-2.6/pdf/posix-api.pdf`.

Xenomai Team. Xenomai API, 2016b. URL `http://www.xenomai.org/documentation/trunk/html/api/index.html`.

Karim Yaghmour. *Building embedded Linux systems.* O'Reilly, 2003. ISBN 0596550480. URL `https://books.google.es/books/about/Building{_}Embedded{_}Linux{_}Systems.html?id=xnFdWfJAK9wC{&}redir{_}esc=y`.

Victor Yodaiken. The RTLinux Manifesto. In *5th Linux Expo*, 1999. URL `http://www.rtlinux.org`.

Guoyin Zhang, Luyuan Chen, and Aihong Yao. Study and Comparison of the RTHAL-Based and ADEOS-Based RTAI Real-time Solutions for Linux. In *First International Multi-Symposiums on Computer and Computational Sciences (IMSCCS'06)*, pages 771–775. IEEE, jun 2006. doi: 10.1109/IMSCCS.2006.272. URL `http://ieeexplore.ieee.org/document/4673801/`.

A

Appendix A: Implemented code repositories

In the following links the implementations for each RTOS of the neuron model library discussed in this work can be found.

Listing A.1: Preempt-RT neuron model library repository.

```
https://github.com/manurs/clamp-cli.git
```

Listing A.2: Xenomai neuron model library repository.

```
https://github.com/RoyVII/clamp-cli_xenomai.git
```

Listing A.3: RTAI neuron model library repository.

```
https://github.com/RoyVII/clamp-cli_rtai.git
```

B

Appendix B: RTOS Installation Manuals

In this appendix is included an installation manual for RTAI 3.9 over a 2.6 kernel, with the Comedi drivers, as the one used in this work. At the same time, the script for installing Preempt-RT is included.

B.1 RTAI

Listing B.1: Installation of necessary items for the compilation and next installations.

```
sudo apt-get update
echo Y|sudo apt-get install cvs subversion build-essential
echo Y|sudo apt-get install kernel-package linux-source libncurses5-dev
echo Y|sudo apt-get install libtool automake
echo Y|sudo apt-get install bison flex
echo Y|sudo apt-get install libboost-dev libboost-program-options-dev libgsl0-dev
echo Y|sudo apt-get install gfortran sablotron tcl8.5-dev tk8.5-dev xaw3dg-dev
    libpvm3 pvm-dev libgtkhtml2-dev libzvt-dev libvte-dev
echo Y|sudo apt-get install qt3-apps-dev
echo Y|sudo apt-get install qt3-assistant
echo Y|sudo apt-get install qt3-designer
echo Y|sudo apt-get install qt3-dev-tools
echo Y|sudo apt-get install qt3-tools
echo Y|sudo apt-get install libqt4-dev libqwt5-qt4-dev
echo Y|sudo apt-get install cmake
echo Y|sudo apt-get install fakeroot
echo Y|sudo apt-get install git-core
echo Y|sudo apt-get install gfortran
echo Y|sudo apt-get install tcl-devel tk-devel libvte
sudo reboot
```

Listing B.2: Download of the vanilla kernel.

```
cd /usr/src
sudo wget http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.35.7tar.bz2
sudo tar xjvf linux-2.6.35.7tar.bz2
sudo ln -s linux-2.6.35.7 linux
echo "Kernel extraction complete"
sleep 5
```

Listing B.3: Download of RTAI.

```
cd /opt
sudo wget --no-check-certificate https://www.rtai.org/RTAI/rtai-3.9-test2.tar.bz2
sudo tar xjvf rtai-3.9-test2.tar.bz2
sudo ln -s rtai-3.9-test2 rtai
echo "Extraction of RTAI complete"
sleep 5
```

Listing B.4: Download of Comedi.

```
cd /opt
git clone git://comedi.org/git/comedi/comedi.git
git clone git://comedi.org/git/comedi/comedilib.git
git clone git://comedi.org/git/comedi/comedi_calibrate.git
git clone git://comedi.org/git/comedi/comedi-nonfree-firmware.git
echo "Download of comedilib complete "
sleep 5
```

Listing B.5: Download of Scilab and Qrtailab. Prepare to patch.

```
cd /opt
sudo wget http://www.scilab.org/download/4.1.2/scilab-4.1.2-src.tar.gz
sudo wget http://www.scilab.org/download/4.1.2/man-eng-scilab-4.1.2.zip
echo "Download of scilab complete "
sleep 5

cd /opt
sudo wget http://downloads.sourceforge.net/qrtailab/QRtaiLab-0.1.6.tar.gz
sudo tar xvzf QRtaiLab-0.1.6.tar.gz
echo "Extraction of QrtaiLab complete "
sleep 5

cd /usr/src/linux
echo "Lets patch 1"
sudo su
```

Listing B.6: Patching the kernel.

```
cd /usr/src/linux
echo "Lets patch 2"
sleep 5
patch -p1 < /opt/rtai/base/arch/x86/patches/hal-linux-2.6.35.7-x86-2.8-01.patch
echo "Patch the complete kernel "
sleep 5

cd /usr/src/linux
cp /boot/config-2.6.32-38-generic .config
sudo cp .config .config.old

cd /usr/src/linux
sudo make clean
sudo make mrproper
sudo make menuconfig
echo "Completed menuconfig number 1 "
sleep 5
```

The following configuration options must be checked (deactivate all the Kernel Hacking options):

- Processor type and features -> Symmetric multi-processing support (multiprocessor)
- Processor type and features-> Processor family
- Enable loadable module support > Module versioning support = no
- Processor type and features > HPET = no
- Processor type and features > Interrupt pipeline = yes
- Power management options > Power Management support = no
- Power management options > CPU Frequency scaling > CPU Frequency scaling = no”
- Device Drivers -> Network device support -> Ethernet (XXXX Mbit) = yes
- Device Drivers -> Network device support -> Wlan =yes
- Device Drivers -> Serial ATA and Parallel ATA driver -> AHCI SATA support
- Device Drivers -> Graphics support -> /dev/agpgart (AGP Support) =yes
- Device Drivers -> Sound card support -> Advanced Linux Sound Architecture -> PCI sound devices=yes

Listing B.7: Patch and install the kernel.

```
cd /usr/src/linux
sudo make-kpkg clean
sudo CONCURRENCY_LEVEL=4 fakeroot make-kpkg -initrd -append-to-version=-rtai
    kernel_image kernel_headers
echo "Compilation of kernel complete "
sleep 5

cd /usr/src
sudo dpkg -i linux-headers-2.6.35.7-rtai_2.6.35.7-rtai-10.00.Custom_i386.deb
sudo dpkg -i linux-image-2.6.35.7-rtai_2.6.35.7-rtai-10.00.Custom_i386.deb
echo "Installation of kernel complete "
sleep 5
sudo reboot
```

In the next menu the following options must be set:

- Installation: /usr/realtime
- Kernel source: /usr/src/linux
- Machine, choose the number of CPUs
- With Kernel 2.26.3 it was necessary to unmark “In-Kernel C++ Support” en “Add-Ons”

Listing B.8: Configuration of RTAI.

```
cd /opt/rtai
sudo mv base/include/asm base/include/asm.old
sudo ln -s base/include/asm.old base/include/asm
sudo make menuconfig
echo "Menu config complete number 2 "
sleep 5

sudo make
sudo make install
sudo sed -i 's/\(PATH=\"\)/\1\/usr\/realtime\/bin:/' /etc/environment
export PATH=/usr/realtime/bin:\$PATH
echo "New PATH added "
sleep 5

cd /opt/comedi
sudo sh autogen.sh
sudo ./configure --with-linuxdir=/usr/src/linux --with-rtaidir=/usr/realtime
sudo make
sudo make install
sudo make dev
echo "Installation of RTAI complete "
sleep 5

sudo su
```

Listing B.9: Comedi configuration and installation.

```
echo 'options comedi comedi_num_legacy_minors=4' > /etc/modprobe.d/comedi
##############################
cd /opt/comedilib
sudo sh autogen.sh
sudo ./configure
sudo make
sudo make install
sudo mkdir /usr/local/include/linux
echo "Comedilib complete "
sleep 5
##############################
cd /opt/comedi_calibrate
sudo autoreconf -i -B m4
sudo ./configure
sudo make
sudo make install
echo "Comedi calibrate complete "
sleep 5

sudo cp /opt/comedi/include/linux/comedi.h /usr/local/include/
sudo cp /opt/comedi/include/linux/comedilib.h /usr/local/include/
sudo ln -s /usr/local/include/comedi.h /usr/local/include/linux/comedi.h
sudo ln -s /usr/local/include/comedilib.h /usr/local/include/linux/comedilib.h
cd /opt/rtai
sudo make menuconfig
echo "Menu config number 3 complete "
sleep 5
```

Listing B.10: Last steps

```
sudo make
sudo make install

cd /usr/local/include
sudo mv comedi.h comedi.h.old
sudo mv comedilib.h comedilib.h.old
sudo cp /opt/comedilib/include/comedilib.h .
sudo cp /opt/comedilib/include/comedi.h .
##########LIBRERIA MATEMATICA######################
cd /opt/rtai/base/math
sudo make
sudo make install
#########COMEDI LIBRARIES#######################
cd /usr/local
sudo mkdir comedi
cd  comedi
sudo mkdir include
cd include
sudo mkdir linux
cd linux
sudo ln -sf /opt/comedi/include/linux/comedi.h
sudo ln -sf /opt/comedi/include/linux/comedilib.h
sudo ldconfig
```

B.2 Preempt-RT

The following script was used to install Preempt-RT and Comedi in Debian 9:

Listing B.11: Preempt-RT with Comedi installation

```
sudo apt-get install git build-essential dkms
sudo apt-get install linux-headers-4.9.0-1-rt-amd64 #Check kernel version (uname -r)
echo "Preempt-RT installed"

git clone https://github.com/Linux-Comedi/comedi.git
cd comedi/
sudo apt install automake
./autogen.sh
cd ..
sudo dkms add ./comedi #Check version (e.g 0.7.76.1+20150923git-1.nodist)
sudo dkms install comedi/0.7.76.1+20150923git-1.nodist
sudo depmod -a
sudo apt install libcomedi-dev libcomedi0
echo "Comedi installed"
```