Design and Evaluation of a Real-Time Sensor Monitor System on Raspberry Pi using Xenomai

DAVID RENE GONZALEZ
Design and Evaluation of a Real-Time Sensor Monitor System on Raspberry Pi using Xenomai

Master’s Thesis submitted in partial fulfillment of the requirements for the Degree of Master of Science in ICT Innovation - Embedded Systems at KTH Royal Institute of Technology, Stockholm

DAVID RENE GONZALEZ

Master’s Thesis Report
Supervisor: Dr. Stefano Zambon
Examiner: Dr. Carlo Fischione

September 2019
Abstract

A real-time system computes information before a deadline in a deterministic fashion. Xenomai is a real-time framework to port real-time industrial applications to Linux. It has a dual-kernel architecture, with one kernel dedicated to handling real-time tasks exclusively. This co-kernel operates along with the Linux kernel in the same memory space.

MIND Music Labs is a Stockholm-based music technology company that specializes in real-time operating systems (RTOS) for audio applications. Its flagship product, the ELK OS, uses Xenomai to achieve real-time performance.

The purpose of this research project is to design a real-time system with a Raspberry Pi 3 Model B running Xenomai to monitor and control sensors connected to its Serial Peripheral Interface (SPI). To this end, a Xenomai system was configured to run on a Raspberry Pi. Afterward, a Real-Time Driver Model (RTDM) SPI driver was tested. Several software layers were built on top of it so that a large-scale C++ application could use the RTDM SPI driver. An AVR microcontroller was connected to the Raspberry Pi through a General-Purpose Input/Output (GPIO) extension shield to emulate multiple sensor devices. This application is designed to provide flexible user control over these devices, without it being necessary to know the low-level details of the SPI interface or Xenomai. An evaluation application was written to test the system’s response times to external stimuli. These observations were benchmarked against a standard Linux system.

The results observed during this research project suggest that a Raspberry Pi running Xenomai may offer hard real-time guarantees to communicate data over SPI for frequencies of up to 1 kHz, even when the system is under a heavy workload.

**Keywords:** Real-time, Raspberry Pi, Xenomai, Real-time driver model, Serial peripheral interface, General-purpose input/output pins.


Sammanfattning


MIND Music Labs är ett Stockholmsbaserat musikteknologiföretag som specialiserar sig på realtidsoperativsystem (RTOS) för ljudapplikationer. Dess flaggskepsprodukt, ELK OS, använder Xenomai för att uppnå realtidsprestanda.


Resultaten som observerats under detta forskningsprojekt visar att en Raspberry Pi som kör Xenomai kan erbjuda hårda realtidsgarantier för att kommunicera data över SPI för frekvenser upp till 1 kHz, även när systemet har en tung workload.

Acknowledgments

First and foremost, I wish to thank my family for their unconditional support; for making my education a priority, and for always giving me the tools to chase my dreams. I would not be here if it were not for my parents and my brother and all their love. That is why I want to thank you. I also wish to thank the rest of my family, everyone back home and to the ones I met here. It truly is a blessing to have you for a family; to know that I can rely on you, and to find in you someone I can trust.

Following, I would like to thank the team at MIND Music Labs; for giving me the opportunity to join you and carry on my research project within your company. From the very first day I joined MIND, I felt welcomed and knew I was surrounded by capable, brilliant, and driven individuals. What they have accomplished there is a testament to their incredible know-how, and I feel both humbled and fortunate to have had a chance to learn from everyone at MIND. I want to thank in particular my supervisor, Stefano Zambon, for his constant guidance, both in the academic and industrial realms. For believing in me and giving me the autonomy to carry out this project. For sharing his knowledge and experience, and for his valuable review of my work. To Nitin Kulkarni, for helping me navigate the world and intricacies of the Linux kernel and Xenomai. To Sharan Yagneswar, for advising me while writing the software for a real-time application. To Gustav Andersson, for explaining SENSEI to me and for his guidance while working with a large-scale application. To Alberto Gianelli, for helping me to understand the SIKA board, with which I carried most of my tests in this project. To Michele Benincaso, for welcoming me as another member of the MIND team. And to all of the people at MIND Music Labs, for making every day in the office wonderful.

I also wish to thank my examiner, professor Carlo Fischione, for being so supportive from the very first time I contacted him, up until the day I finished this project. His guidance, enthusiasm, and approachability were essential for me to carry on my project.

Additionally, I would like to thank my friend Sam Tabibzadeh, for providing the Swedish translation of my abstract, and for his words of encouragement as I did my research.

To my opponents, Tallulah Gilliard and Pratik Walvekar, for their valuable input on my report; for being both supportive and critical of my work; I included multiple corrections and additions based on their feedback. It was both a pleasure and a privilege to have them review my work and listen to my presentation.

To the European community, for welcoming me into your countries and giving me the means to study here. To the people at the European Institute of Innovation and Technology, Technische Universität Berlin, and Kungliga Tekniska Högskolan, for believing in me.

Finally, I must thank all my friends, both those that are back home and every friend I have made here in Europe. I feel blessed for being surrounded by all these kind, brilliant, beautiful people. I believe my accomplishments are thanks to you.

The work presented in this report could not have been completed if it were not for all the
people mentioned here. For their support and guidance in the academic, industrial, and personal realms, I managed to make it across the finish line. For that, and for everything, I thank you.

Mexico City, September 2019

David Rene Gonzalez
Table of Contents

Abstract

Sammanfattning

Acknowledgments

List of Figures

List of Tables

1 Introduction

1.1 Problem Statement ................................. 2
1.2 Goals .............................................. 2
1.3 Related Work ..................................... 2

2 Background

2.1 Real-time Systems ................................. 3
   2.1.1 Classification .................................. 3
   2.1.2 Nyquist-Shannon sampling theorem ............ 3
   2.1.3 Real-time operating systems ................... 4
   2.1.4 Communication in real-time operating systems ... 5
2.2 Latency and Jitter ................................. 6
   2.2.1 Timing jitter ................................... 6
   2.2.2 Period jitter ................................... 7
   2.2.3 Cycle-to-cycle jitter ......................... 7
2.3 Linux .............................................. 8
   2.3.1 Process scheduler ............................. 8
   2.3.2 External devices ............................... 8
2.4 Xenomai ............................................ 9
   2.4.1 Architecture ................................. 9
   2.4.2 Co-kernel .................................... 9
   2.4.3 Core and nucleus ............................ 10
   2.4.4 Scheduler .................................. 10
   2.4.5 I-pipe ..................................... 10
   2.4.6 Real-time driver model ..................... 12
2.5 Raspberry Pi .................................... 13
   2.5.1 Motivating its usage in this project .......... 13
   2.5.2 Hardware features ........................... 14
   2.5.3 Broadcom BCM2837 chip ..................... 14
   2.5.4 Cortex-A53 processor ....................... 15
   2.5.5 SPI overview ................................ 15
   2.5.6 Comparison with other boards ............... 16
   2.5.7 Real-time capabilities ..................... 16
List of Figures

1 Timing jitter ....................................................... 6
2 Period jitter ....................................................... 7
3 Cycle-to-cycle jitter ........................................... 7
4 Xenomai architecture ........................................... 10
5 Interrupt pipeline ................................................ 11
6 Event sharing in an I-pipe system .............................. 11
7 Xenomai and Linux in an I-pipe system ....................... 12
8 Real-Time Driver Model ......................................... 13
9 SPI master typical usage ....................................... 15
10 High-level overview of the software hardware system ........ 21
11 Front and back-end layers of the SPI system ................ 24
12 SENSEI architecture ............................................ 27
13 Lock-free communication between SENSEI and a real-time task .... 28
14 Real-time task flowchart ........................................ 30
15 I/O latency results for Raspberry Pi running Xenomai ......... 32
16 Latency results .................................................... 33

List of Tables

1 Single-board computer comparison ............................... 16
2 I/O latency summary results for Raspberry Pi running Xenomai .... 32
3 Summary results for Linux and Xenomai latencies ................ 34
4 SPI input data acquisition ....................................... 35
1 Introduction

Since the invention of the transistor, advances in materials science, manufacturing techniques, computer architecture, software engineering, and many other fields have made digital computers increasingly faster. Every year, computers can store and process more data and become more powerful, but as this happens, so they become more complex. A perfect example of this complexity is the operating system: a set of computer programs that coordinate thousands of tasks so that they can all share the underlying hardware of the computer. An operating system provides many abstractions that make it easier to design and implement software on top of hardware, abstractions without which it would be extremely difficult to build a modern personal computer.

Despite the success of the operating system to bring a personal computer to almost half of the world’s population, many areas heavily rely on digital computers that do not require an operating system. Some of these use computers that provide exact timing behaviors: they do things precisely when they have to, not too soon nor too early. These systems are called real-time systems, and unlike an average personal computer, they are meant to show an extremely high level of dependability and predictability. For example, a personal computer application might unexpectedly crash, and take an unknown amount of time to restart. Some real-time systems are designed never to crash and to have well-known response times to events.

Many real-time systems are relatively simple to achieve a high level of predictability. Unlike other computational systems, a real-time one might be designed to do one specific task and run only a few processes. A simple design makes the system easier to analyze, and therefore, to know its response times to different events. However, by deciding to forgo an operating system, system designers must deal with all the difficulties of working directly or almost directly with the computer’s hardware.

In the past twenty years, there has been much ongoing research on operating systems that can exhibit real-time performance, as some industrial areas could greatly benefit from having such a thing; one such area is the one of musical technology. A musical device to record or process audio must sample a signal at a constant rate without losing any data. Furthermore, any device of this kind might have to handle input and output hardware: sensors and actuators, such as buttons, knobs, or LEDs to enable user interaction.

Most music technology companies build these devices without the need for an operating system: by designing deterministic software on top of reliable hardware, they can guarantee that their products will perform in real-time. However, this poses many limitations. First of all, it makes it challenging to write and maintain software since there is, at best, just a simple hardware abstraction layer to interact with the hardware, and, at worst, programmers must write software that directly deals with the intricacies of the hardware. Second of all, each time a company chooses to switch hardware (because it has been discontinued or because there is a new and improved version of it), it must rewrite all of its software.

With an operating system, these problems would almost be non-existent. Among the many
advantages of using an operating system, is that it makes code portable across all platforms that support it. MIND Music Labs, a Stockholm-based music technology company, realized how this fact could be turned into a business opportunity. Its flagship product, ELK, is the world’s first music operating system that offers real-time performance. ELK uses Xenomai, a real-time framework for Linux systems, to ensure that it can guarantee responses to stimuli in real-time.

MIND Music Labs already supports several hardware platforms for professional developers. This year it became interested in bringing ELK to a more affordable computer for hobbyists and other enthusiasts: the Raspberry Pi is a cheap and popular single-board computer that is ideal for building prototypes. Throughout this project, the real-time capabilities of a Raspberry Pi in the context of musical technology were evaluated.

1.1 Problem Statement

General-purpose operating systems do not offer real-time guarantees. This project is set to determine if it is possible to attain real-time performance on a Raspberry Pi running Xenomai. Specifically, can a Raspberry Pi running Xenomai offer hard real-time guarantees to monitor and control general-purpose input/output pins connected to a Serial Peripheral Interface at a frequency of up to 1 kHz.

1.2 Goals

The main goals of this project are to:

• Configure a Raspberry Pi to run the latest version of Xenomai
• Design a system to communicate data over a Serial Peripheral Interface in real-time
• Monitor sensor devices connected to the Raspberry Pi in real-time
• Evaluate the system’s performance in contrast to a pure Linux solution

1.3 Related Work

Two master thesis projects have come out of MIND Music Labs, showing different approaches to real-time systems for music technology. [1] showed that a Xenomai-based system could achieve real-time performance on x86 processors. Another project to come out of KTH Royal Institute of Technology proved that, to a degree, a Raspberry Pi running Xenomai is suitable for certain real-time applications [2].
2 Background

2.1 Real-time Systems

A real-time system is any computational system where the “correctness of it depends not only on the logical result of the computation but also on the time at which the results are produced” [3]. In the scientific literature, most definitions of real-time systems highlight how they are an “integral part of a physical control system”, with the computation being “consistent with changes in the physical inputs and the output data rates required by the system” [4].

It is not enough for a system to be very fast to qualify as real-time if it cannot respond to the environment it controls at a precise rate all the time. There is an essential difference between real-time systems and systems that are very fast. A real-time system will have deterministic responses to stimuli, bounded latencies (not too late, not too soon), and repeatable results, all while optimizing and quantifying the worst-case execution time. Conversely, a system that is just fast will use sophisticated software and hardware designs and techniques, such as pipelines or advanced cache memory accesses, to optimize the average-case execution time [5].

2.1.1 Classification

Real-time systems must perform computations within a certain time frame. A system is classified as a hard real-time system if its failure to meet a deadline results in a total system failure. If it does not meet a deadline, but this only results in a degradation in the system’s performance, then it is considered to be a soft real-time system [6]. Hard real-time systems can be divided into sub-categories. 95% hard systems have deadlines that should be met at least 95% of the time. 100% hard systems must meet their timing requirements all the time. Life-safety hard systems must always meet their deadlines, but failure to do so can result in someone being injured or killed, or substantial property damage may happen [7].

There are additional categories of real-time systems, for example, firm systems. These are systems that can afford to produce results after a deadline without causing any damage, but the results will not be considered useful anymore [8].

2.1.2 Nyquist-Shannon sampling theorem

The Nyquist-Shannon sampling theorem states that “if the spectrum of a signal $x(t)$ contains no frequencies above $f = W$, then $x(t)$ is completely characterized by its samples values uniformly spaced in time with a period $T_s \leq \frac{1}{2W}$” [9] [10]. Therefore, the minimum
sampling frequency, $f_s$, to reconstruct a signal $x(t)$ with maximum frequency $W$ is given by the inequality:

$$f_s \geq 2 \times W$$  \hspace{1cm} (1)

To control a physical system that operates at a maximum frequency $W$, a real-time system must be able to sample its state at a frequency greater or equal than $2 \times W$. Establishing this criterion is essential at the time of designing the computing system and determining if it can satisfy the established real-time requirements.

### 2.1.3 Real-time operating systems

“An Operating System (OS) is a specialized collection of system programs that manage the physical resources of the computer” [11]. An OS has a kernel to provide its core functionality. The kernel is in charge of managing and allocating the computer’s resources. A kernel performs process scheduling, memory management, provides a file system, creates and terminates processes, grants access to devices, handles network messages, provides a system call Application Programming Interface (API), among other things [12].

A general-purpose OS does not provide real-time guarantees [13]; instead, it is designed to maximize the system’s average performance [14]. Conversely, a Real-Time Operating System (RTOS) is “designed to achieve real-time responses” [13] and aims at “upper-bounding the execution time” of the system [14]. To achieve this, an RTOS offers preemptive, priority-based scheduling, predictable task synchronization, and support for deterministic behaviors. Deterministic performance is attained by ensuring that a task’s dispatch time, switch latency, and interrupt latency are all time-bounded and consistent, independent of the total number of tasks [13].

One way to achieve real-time performance on an operating system is by running two kernels simultaneously: one for any general-purpose operation and another one to exclusively handle real-time processes. The disadvantage of this approach is that any real-time application must rely only on the services provided by the real-time kernel. That in turn means that any software written for a non-real-time kernel must be ported to operate in real-time by replacing any system calls to use those of the real-time kernel. Furthermore, certain services from the standard C library (glibc) can call on services provided by the non-real-time kernel, therefore making it necessary to replace these functions as well to ensure real-time performance in a process. Given this, it becomes evident that developing and maintaining real-time applications on a real-time operating system can become complex and expensive if the design does not make a clear separation between real-time and non-real-time activities [15].
2.1.4 Communication in real-time operating systems

In an RTOS environment, it is not unusual to have real and non-real-time threads exchange data. There are several problems associated with this exchange: real-time threads have strict deadlines, while non-real-time threads do not. Real and non-real-time threads may run at significantly different speeds. Furthermore, because a non-real-time thread has an unbounded execution time, it may be impossible to know when exactly it will dispatch data to a real-time thread [16]. All these problems can be solved by implementing a communication scheme that guarantees that real and non-real-time threads can share data without blocking or slowing down each other.

Algorithms for concurrent access to a shared data structure can be either blocking or non-blocking. A blocking algorithm prevents a thread from completing its operation to allow slower threads to access the data structure. In a system with a scheduler that supports preemption (such as the one in this project), such an algorithm results in poor performance and cannot offer real-time guarantees [17]. Conversely, a non-blocking algorithm is designed to “guarantee that if there are one or more active processes trying to perform operations on a shared data structure, some operation will complete within a finite number of steps” [17]. This guarantee is what makes non-blocking algorithms ideal for real-time systems [16]. Another important property of non-blocking algorithms is that if a thread fails or is suspended, it cannot result in the failure or suspension of another thread with which it shares a data structure [18].

Non-blocking algorithms can be further divided into lock-free and wait-free algorithms. A lock-free algorithm offers a weaker property than a wait-free algorithm [16]. An algorithm is classified as lock-free if among all processes accessing a shared data structure, “at least one will succeed to finish its operation” [16]. From this definition, it becomes clear that there are no guarantees that all processes will finish their operation, meaning that one (or many) processes could be starved by the other ones. “A wait-free algorithm is both non-blocking and starvation free: it guarantees that every active process will make progress within a bounded number of steps” [17].

Many solutions to communicate real and non-real-time threads use a First-in, first-out (FIFO) data structure [17]. A FIFO is well-suited for the application in this project since threads should handle events in the same order as they occurred, that is, the first event or interrupt should be handled first, and so on. A non-blocking algorithm for shared FIFO structures supports add (enqueue) and remove (dequeue) operations. It has been shown that restricting concurrency to a single enqueuer and a single dequeuer can result in wait-free behavior [19]. In Section 5.3, this approach will be shown, illustrating how to communicate real and non-real-time threads by sharing two FIFO structures while restricting enqueuing and dequeuing to one thread per FIFO. This is the same as to say that information always flows from producer (enqueuer) to consumer (dequeuer).
2.2 Latency and Jitter

It is essential to measure a system’s latency if it has real-time requirements because latencies must be bounded for it to qualify as real-time [5]. When discussing operating systems, latency is defined as “the delay before a transfer of data begins following an instruction for its transfer” [20]. The following are sources of latency in a kernel [21]:

- Interrupt latency: the time elapsed since an interrupt is issued until it is handled by the interrupt handler.
- Scheduler latency: the time it takes for the scheduler to begin execution after the interrupt handler has completed its execution.
- Scheduling latency: also called task latency [22], it is the time it takes for a task that was in sleep mode to resume execution after a dedicated interrupt has been issued.

Jitter is “the unwanted variation of the release times of a periodic task. It can be characterized in various ways such as an interval around the desired release time, a maximal deviation from the desired time point, or a standard deviation from the mean value” [22]. There are different ways to measure jitter, and it is worth mentioning three distinct and specific ways to quantify it.

2.2.1 Timing jitter

Timing jitter is the difference between the time a signal occurs and the time it was expected to be issued. This ideal time is a multiple of the desired period $T$ [23]. Equation 2 shows the timing jitter.

$$j_t = t_n + (-nT)$$  \hspace{1cm} (2)

Figure 1 illustrates the definition of timing jitter. If the signal arrives too soon, the jitter is negative. If it arrives late, the jitter is positive. If the signal arrives exactly at $nT$, then the jitter is zero.

![Figure 1: Timing jitter](image-url)
2.2.2 Period jitter

Period jitter is “the variation of the period of a signal from the average period” [23]. It is given by Equation 3.

\[ j_p = t_n - t_{n-1} - T \]  \hspace{1cm} (3)

Figure 1 represents the concept of timing jitter. Ideally, the difference \( t_n - t_{n-1} \) would equal \( T \), meaning that there is no jitter.

![Figure 2: Period jitter](image)

2.2.3 Cycle-to-cycle jitter

Cycle-to-cycle jitter is “the variation in the period of a signal from one period to the next” [23]. Equation 4 shows how to compute the cycle-to-cycle jitter. This formula is useful when the ideal period \( T \) is unknown, as it does not depend on it.

\[ j_{cc} = t_n - 2t_{n-1} + t_{n-2} \]  \hspace{1cm} (4)

Figure 3 shows three consecutive signals and the time at which they arrive. With these three time values, it is possible to compute the cycle-to-cycle jitter.

![Figure 3: Cycle-to-cycle jitter](image)

Ideally, for every cycle, the following equation holds, meaning that the jitter is zero:

\[ t_n - t_{n-1} = t_{n-1} - t_{n-2} \]  \hspace{1cm} (5)
2.3 Linux

Colloquially, GNU/Linux operating systems are referred to as Linux. A Linux system is composed of the Linux kernel, “a UNIX-like operating system kernel”; the GNU Project, a set of programs designed for UNIX systems among which are the GNU C Compiler, the bash shell, and GNU Make; and many other software packages \[12\]. Two aspects that are of interest in this thesis are how the Linux kernel schedules processes and how it manages external devices connected to the computer.

2.3.1 Process scheduler

The Linux process scheduler is what allows the “apparent simultaneous execution of multiple processes by switching from one process to another in a very short time frame” \[24\]. It has to achieve multiple objectives, among which are responding fast enough while reconciling the “need of low and high-priority processes” \[24\].

To better understand process priorities, it is necessary to classify processes. Interactive processes are those that continuously interact with their users, for example, through key presses. Batch processes do not interact with users and usually run in the background. An example of this is a programming language compiler. Finally, real-time processes “have very stringent scheduling requirements” \[24\]. Real-time processes will be further discussed in Section 2.1, but suffice to say that these “processes should never be blocked by lower-priority processes” \[24\].

From Linux 2.6.23 onwards, the kernel uses the Completely Fair Scheduler. This “models an ideal, precise multi-tasking CPU on real hardware” \[25\]. It was designed to “provide good interactive performance while maximizing overall CPU utilization” \[26\].

Despite distinguishing between real-time and non-real-time processes and providing data structures and functions to manage their scheduling \[24\], the standard mainline Linux kernel is not meant to be used in a real-time operating system (RTOS), as it is designed to be a general-purpose kernel. Linux has sacrificed how deterministic is the process by which CPU time is allocated to tasks by the scheduler to optimize the system’s average throughput instead \[27\]. However, the Linux priority policies for real-time systems could work well in some embedded RTOS where the number of tasks is minimal, and their behavior is perfectly known \[27\].

2.3.2 External devices

The Linux kernel treats external devices, such as a storage device or serial interface, as a file. The Virtual File System makes these filesystems available to the kernel. Unix systems support five file types: regular files, directories, symbolic links, device files, and pipes \[24\].
Complex devices are usually controlled through a device driver [24]; a piece of software that controls some hardware according to a “well-defined programming interface” [28]. A device driver should “interpret the high-level commands received from the I/O interface and force the device to execute specific actions by sending proper sequences of electrical signals to it” and “convert and properly interpret the electrical signals received from the device and modify (through the I/O interface) the value of the status register” [24].

There are three fundamental types of devices: character devices, block devices, and network interfaces. Character devices will be the subject of interest in this thesis. These devices are “accessed as a stream of bytes, just like a file”, and are represented as a filesystem node, but unlike regular files, most character devices can only be accessed sequentially [28]. A character device’s driver usually implements open, close, read, and write system calls. A network interface sends and receives data packets and is not mapped to a filesystem node. Instead of read and write, a network interface driver “calls functions related to packet transmission” [28]. As it will be seen in section 4, the driver tested in this thesis is a character driver.

2.4 Xenomai

Xenomai is a real-time framework for Linux [29]. It is designed to run industrial applications with strict real-time constraints on a GNU/Linux environment. It does this by executing a co-kernel along Linux that is devoted to real-time tasks [15].

2.4.1 Architecture

The Xenomai architecture splits processes and services, the operating system itself, and device drivers into two categories: real-time and non-time-critical [30]. Figure 4 illustrates this separation.

The Xenomai co-kernel handles Xenomai processes, and sometimes a real-time device driver that controls specific hardware. Similarly, the Linux kernel handles Linux processes and controls non-time-critical devices.

The Hardware Abstraction Layer (HAL) “gathers all the CPU and platform-dependent code needed to implement a particular Xenomai port so that every layer starting from the nucleus and higher is free from machine-dependent code” [15].

2.4.2 Co-kernel

The Xenomai co-kernel (named Cobalt kernel) is a lightweight kernel that runs along Linux to control Xenomai applications when they are in real-time mode. When running a system with a dual-kernel configuration, Linux is referred to as the host kernel. This host provides all non-time-critical services, which is why the co-kernel only has to implement some specific functions (most OS operations are not meant to be real-time safe) [15].
2.4.3 Core and nucleus

The Xenomai core is an abstract RTOS that supplies all the operating system resources required to emulate a traditional RTOS API. The core defines multiple building blocks which in turn compose a single loadable module called the Xenomai nucleus. This nucleus provides a real-time thread object, an interrupt object to connect to the hardware’s Interrupt Request (IRQ) lines, a memory allocator, a timer management service, etcetera [15].

2.4.4 Scheduler

The Xenomai scheduler is preemptive and enforces a fixed-priority scheduling scheme. It supports multiple thread priority levels. Like most RTOS schedulers, it also supports round-robin scheduling [15]. The Xenomai scheduler can schedule Linux tasks that are promoted to a real-time domain. It has its own separate data structures to handle real-time tasks, but it can access certain fields of the Linux scheduler’s data structures [15].

2.4.5 I-pipe

The interrupt pipeline, I-pipe, is a software layer based on Adeos [31] that generates events such as interrupts and system calls for Linux or the co-kernel to handle. I-pipe distinguishes between two domains, Xenomai and Linux, and ensures that Linux will never handle a real-time event before Xenomai processes it [15].
I-pipe modifies the real IRQ mask before this reaches the Linux kernel. This way, processes in both domains can execute without concerns of one kernel affecting the other [15].

Different domains are connected to I-pipe in order of priority. “Incoming events (including interrupts) are pushed to the head of the pipeline” and captured by the highest priority domain “and progress down to its tail” to the lowest priority domain [32]. Figure 6 shows a generic I-pipe system with multiple domains and the root domain at an arbitrary priority level. The root domain is Linux, and a higher priority domain would be a real-time kernel.

If a stage of the pipeline stalls, any new interrupts will not flow down to lower priority domains. However, domains before the stalling stage will still operate normally and handle any incoming interrupts that belong to them. Figure 7 shows an example of an RTOS with a Xenomai co-kernel and Linux connected to I-pipe. Because Xenomai is given a higher
priority, even when Linux stalls, Xenomai will still be able to receive interrupts at any time with no additional delays. This shows how a Linux IRQ cannot block Xenomai in any way.

Figure 7: Xenomai and Linux in an I-pipe system

In Figure 7, an additional domain exists between Xenomai and Linux. The interrupt shield logs any events that are meant to be handled by Linux if Linux is stalled at the time. These events are eventually dispatched by the interrupt shield once Linux is ready to accept interrupts [1].

2.4.6 Real-time driver model

The Real-Time Driver Model (RTDM) is a common framework [15] designed to unify “the interfaces against which real-time device drivers and the applications using them can be developed” [33]. Without it, real-time device driver developers must design non-standard protocols to implement a driver so that an application can communicate with it. Furthermore, without RTDM, there is no common interface for device drivers to request services or resources from the real-time kernel. That means that for each Linux real-time extension, the driver must call a specific API, making it harder to port driver code across different extensions [15].

Figure 8 shows how the RTDM can mediate an application’s request to a hardware driver. Designers can choose to further abstract the usage of the RTDM API with a wrapper library for the application. Similarly, a HAL may be used so that the RTDM communicates to the driver via a protocol common to multiple hardware drivers [33]. This additional abstraction layers should result in greater code reuse and increased portability.

The RTDM supports protocol and named devices. The former are for message-oriented devices and are addressed per the POSIX socket model. The latter are akin to Linux character devices [15]. They can be further subdivided into read/write devices, and those that only work through the ioctl interface [33].

The RTDM provides an API of basic RTOS services to drivers. These include synchronization services, such as spinlocks, mutexes, and semaphores, and utility services to allocate memory in a real-time context and access user-space memory areas [33].
2.5 Raspberry Pi

The Raspberry Pi is a very low-cost Linux-based single-board computer [34] [35]. It was originally designed as a tool for Computer Science students at Cambridge University, but the project evolved to be about providing children with a general-purpose computer with which they could learn about Computer Science. Almost since its inception, it was meant to drive physical computing projects, which is why the Raspberry Pi board features, among many other things, exposed General-purpose input/output (GPIO) pins [35].

2.5.1 Motivating its usage in this project

The Raspberry Pi is designed to run GNU/Linux. It uses a Broadcom BCM2835 System on Chip (SoC), which is based on the ARM processor design. ARM-based devices are ubiquitous in embedded systems due to their low power requirements and Reduced Instruction Set Computer (RISC) architecture [35]. In Sweden, a Raspberry Pi 3 Model B or B+ can be purchased for less than 400 SEK. It is a very appealing board for software and hardware enthusiasts, and since its first release in 2012, it has mesmerized aficionados and professionals alike.

Given the Raspberry Pi’s ability to run embedded Linux systems, its low cost, and its ever-increasing popularity, MIND Music Labs wanted to port the ELK system to this board and evaluate its performance. The Raspberry Pi 3 Model B was used to conduct this research project.
2.5.2 Hardware features

A Raspberry Pi 3 Model B has the following hardware features [36]:

- Quad Core 1.2GHz Broadcom BCM2837 64bit CPU
- 1GB RAM
- BCM43438 wireless LAN and Bluetooth Low Energy (BLE) on board
- 100 Base Ethernet
- 40-pin extended GPIO
- 4 USB 2 ports
- 4 Pole stereo output and composite video port
- Full size HDMI
- CSI camera port for connecting a Raspberry Pi camera
- DSI display port for connecting a Raspberry Pi touchscreen display
- Micro SD port for loading the operating system and storing data
- Upgraded switched Micro USB power source up to 2.5A

2.5.3 Broadcom BCM2837 chip

The Raspberry Pi 3 uses the Broadcom BCM2837 chip, which is similar to the BCM2836 (used in the Raspberry Pi 2) and BCM2835 chips (used in the Raspberry Pi 1). The only difference between these chips are the ARM processors in them. The BCM2835 has an ARM1176JZF-S processor [37], the BCM2836 upgraded to a quad-core Cortex-A7 cluster [38], and the BCM2837 has a newer quad-core ARM Cortex-A53 (ARMv8-A) cluster [39].

These Broadcom chips include the following peripherals which can be accessed by the ARM processor [40]:

- Timers
- Interrupt controller
- GPIO
- USB
- PCM/I2S
- DMA controller
- I2C master
- I2C/SPI slave
- SPI0, SPI1, SPI2
- PWM
- UART0, UART1

2.5.4 Cortex-A53 processor

The Cortex-A53 processor used by the Raspberry Pi 3 implements the ARMv8-A architecture. It has four cores with an an L1 memory system and a single shared L2 cache each. It has an in-order pipeline with symmetric dual-issue of most instructions [41], and it is a superscalar processor [42] as it can issue more than one instruction per clock cycle [43]. This processor supports Single Instruction Multiple Data (SIMD) and floating-point extensions, as well as ARMv8 Cryptography Extensions [41].

2.5.5 SPI overview

The Broadcom 2837 chip has an SPI interface. It features a 3-wire serial protocol, which is the one used during this project, but supports other SPI implementations as well [40]. Figure 9 shows how an SPI master device connects to two slave devices in 3-wire mode [40].

![Figure 9: SPI master typical usage](image)

The serial interface has a 16-word write and a 16-word read First-in, first-out (FIFO) queue, TX FIFO and RX FIFO, respectively. The Broadcom 2835 datasheet defines a word as a 32-bit unit [40]. The SPI device driver used during this project operates by polling two flags. It polls the TXD flag and writes data to the TX FIFO as long as it can accept at
least 1 byte of data. At the same time, it reads the bytes from the RX FIFO, as it polls the RXD flag.

Once it is done polling TXD, it polls the DONE flag. This flag indicated if transfers are complete. Once DONE is set to 1, the driver clears the TA bit, which means that there are no more active transfer operations [40].

The maximum SPI clock rate depends on the maximum APB clock, which runs at 250MHz [44]. The SPI interface’s frequency is given by equation 6:

\[ SPI_{CLK} = \frac{APB_{CLK}}{2 \times (speed\_field + 1)} \]  

The maximum SPI clock speed is given when the speed_field is 0, thus getting a speed of 125 MHz [40].

### 2.5.6 Comparison with other boards

There are many single-board computers available in the market that can be used to control physical systems and power IoT projects. Table 1 compares some of the platforms supported by the ELK Operating System. The performance metric is a rough measure obtained by averaging some relevant benchmarks for signal processing.

<table>
<thead>
<tr>
<th></th>
<th>RPi 3 B</th>
<th>i.MX8</th>
<th>UP Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>Price (USD)</td>
<td>28</td>
<td>65</td>
<td>75</td>
</tr>
<tr>
<td>Performance (normalized)</td>
<td>1</td>
<td>1.53</td>
<td>1.33</td>
</tr>
<tr>
<td>CPU cores</td>
<td>quad-core ARM Cortex A53</td>
<td>quad-core ARM Cortex A53</td>
<td>quad-core Intel Atom x5-Z8350</td>
</tr>
<tr>
<td>Clock speed (GHz)</td>
<td>1.5</td>
<td>1.8</td>
<td>1.44 - 1.92</td>
</tr>
</tbody>
</table>

Table 1: Single-board computer comparison

At a fraction of the price of the i.MX8 or UP Core, the Raspberry Pi can offer similar performance to them. On top of this, the Raspberry Pi features a rich set of communication interfaces that makes it the ideal platform for inexpensive, Linux-based projects.

### 2.5.7 Real-time capabilities

In [35], the creator of the Raspberry Pi, Eben Upton, advised users against using the Raspberry Pi for “true real-time operations”, and to use a dedicated real-time microcontroller instead. However, [1] showed that by using the Xenomai framework and running Linux with a Cobalt co-kernel, it is possible to achieve hard real-time performance on an x86 microprocessor. Furthermore, [2] demonstrated that a Raspberry Pi running a Cobalt co-kernel could be used in firm real-time applications. However, it is worthwhile to highlight
that the deadlines in [2] must be met within 10 µs, whereas the deadlines in this project require to send and receive SPI messages at most every 500 µs. Therefore, it is reasonable to assume that a Raspberry Pi could achieve hard real-time performance, provided that the deadline times are in the order of a few hundred microseconds, and not any shorter.
3 Xenomai setup for Raspberry Pi

The first prerequisite to implement the real-time system is to install Xenomai and a Cobalt co-kernel on a Raspberry Pi. As this project was being carried out, the latest version of Xenomai was 3.0.8, which requires Linux 4.14.85. This section describes the process to build a Linux kernel with Xenomai support. At the time of writing, however, Radboud University Nijmegen [45] released a Raspbian image that supports Xenomai 3.0.8. A reader interested in building Xenomai applications for Raspberry Pi may refer to the university’s website to download this image file. The steps presented here differ from the instructions described by [46], and their approach was not tested in this project.

3.1 Building a Linux Kernel

The first step to get Xenomai to run on a Raspberry is to cross-compile a Linux kernel for Raspbian. [47] describes the general process to build a kernel for Raspberry Pi. A GNU/Linux computer is recommended to carry out these steps; during this project, a machine running Ubuntu 18.04 LTS was used.

Appendix A shows each step necessary to build a Linux kernel. The key steps are to patch the source code back to version 4.14.85 so that it is compatible with Xenomai 3.0.8 (the latest stable release at the time of writing). These steps are based on the work in [1] and [48].

3.2 Applying the I-pipe Patch


It is likely that not all files can be patched automatically; these files can be identified by looking for .rej files in the working directory. It is necessary to manually handle the discrepancies between the source file and the patch. [49] describes how to do this, and [48] provides the fix to two files which will probably reject the patch on the first try.

3.3 Xenomai Kernel Configuration

Before compiling the kernel, it is necessary to disable some features that Xenomai would otherwise warn against [48]. Disable the following options: contiguous memory allocator, allow for memory compaction, CPU frequency scaling (this causes the timing of real-time threads to become unpredictable [50]), and KGDB.
After compiling the kernel, it is necessary to also build some external kernel modules and a device tree blob. A device tree blob is a “database that represents the hardware components on a given board” [51]. The `modules_install` command builds out-of-tree kernel modules [52], and `dtbs_install` allows puts device tree blobs in a standards place [53].

### 3.4 Xenomai Kernel Installation

Once the kernel, kernel modules, and device tree have been built, it is necessary to transfer them to a working Raspberry Pi. Assuming a Raspberry Pi running the Raspbian OS, compress the recently-built kernel and send it to the Raspberry Pi via the network, or copy the files directly to the Raspberry Pi’s SD card.

Appendix B shows how to install the kernel by transferring the files to the appropriate locations inside the Raspberry Pi. To test that the kernel works it is necessary to install and run the Xenomai tools.

### 3.5 Xenomai Libraries Installation

The final step after installing Xenomai is to install the Xenomai libraries which provide tools to test real-time applications. Appendix C shows how to achieve this. After installing the libraries, to verify that the kernel has been configured correctly and the tools work, run the `latency` test which is part of the Xenomai test suite.
4 Real-Time Driver Model SPI System

4.1 Real-Time Requirements

At the beginning of this project, it was decided that the goal was to implement an RTDM-based system that can read and write SPI data every 1000 ms. The system should be able to write at least four to five bytes of data during this period. It has been hypothesized that a Raspberry Pi running Xenomai with a dual-kernel configuration should be able to meet these real-time requirements 100% of the time, whereas Linux would not provide these guarantees.

4.2 High-Level Overview of the System

The final deliverable consists of a user-space application that reads and writes data to the Raspberry Pi’s SPI interface. This interface must be controlled by an RTDM driver; therefore, the operating system must host a Cobalt co-kernel.

Figure 10 shows the different software and hardware layers that compose this system. This system overview is analogous to the RTDM model shown in Figure 8.

The application is SENSEI, a daemon designed by MIND Music Labs to manage GPIO devices such as buttons and LEDs. It has a user front-end to send and receive packages via the Open Sound Control (OSC) protocol. There is an additional front-end to control the SIKA board. This software layer allows SENSEI to perform operations specific to this board.

The SIKA front-end imports a user-space access library to the SPI interface. This library is a wrapper library for the SPI driver, making it simpler for a user-space application to request services from the SPI interface. The library provides a function to initialize a real-time thread which periodically sends and receives data through the SPI hardware.

The following layers operate in kernel space. The Cobalt co-kernel uses the services provided by the RTDM library to interact with the SPI driver. Depending on the services requested by the Xenomai user-space application (in this case, the real-time thread), the Cobalt core may request additional services from Linux.

The BCM2835 SPI driver is the software layer that configures and controls the SPI interface. Finally, the SPI hardware is one of the serial communication interfaces on the Broadcom 2837 chip embedded on the Raspberry Pi.

4.3 Building a Kernel Module

A kernel module is any software that is loaded to the kernel during runtime [28]. Device drivers can be loaded while the operating system is running, making them one example of kernel modules. To build a kernel module, it is necessary to have the Linux source files for
Figure 10: High-level overview of the software hardware system

the target computer. Recall the steps followed in Appendix A; to compile a module, one must provide the make tool with the path to the Linux source files.

Kernel modules are built with GNU Make and the kernel build system. A Makefile should, at the very list, have a line to tell Make to build a module from an object file [28]. For the SPI driver, this line is:

```
$ obj-m += spi-bcm283x-rtdm.o
```

Invoking the following Make command will compile the kernel module, which then can be transferred to the Raspberry Pi:

```
$ make ARCH=arm CROSS_COMPILER=arm-linux-gnueabihf- KERNEL_DIR=/home/ ! linux
```

The `insmod` command is used to load the kernel module during runtime. Once this is done, the module registers itself to serve requests from other processes [28].
4.4 RTDM SPI Driver

An open-source RTDM SPI driver was made available by [54]. This implementation works out of the box with a Xenomai kernel, and makes it ideal to test the real-time capabilities of the Raspberry Pi.

4.4.1 BCM2835 user-space library

The driver in [54] is based on the user-space driver for the Broadcom 2835 chip by [55]. User-space drivers are easy to write and test but can access less operating system and hardware resources than a kernel-space driver. Furthermore, a user-space driver is slower than a kernel one since “a context switch is required to transfer information or actions between the client application and the hardware” [28].

To operate in real-time and use the RTDM framework, the SPI driver must necessarily run in kernel-space, which is why the driver in [54] is ideal for conducting this research project.

4.5 User-space Access Library

The user-space access library is a software abstraction layer between the SPI RTDM driver and the SENSEI application. Its purpose is twofold: to provide an easy-to-use interface to the SPI driver and to enable the communication of data between a real-time and a non-real-time process.

The driver in [54] responds to user-space requests made through Xenomai system calls, such as __cobalt_read and __cobalt_write. These system calls are not made directly accessible to SENSEI; instead, SENSEI (or any other non-real-time process) can include the header file to the access library and call the spi_receive and spi_send functions to read and write data to the SPI hardware.

To communicate data between SENSEI and a real-time process in a transparent fashion, the real-time process pushes any data it reads from the SPI hardware to a FIFO. SENSEI can retrieve this data by performing a dequeue operation on the FIFO, and given the nature of this data structure, the data will always be retrieved in the same order as it was received by the SPI hardware; this is essential for applications that need to preserve the order in which the data is sent and received.

In the same way, whenever SENSEI sends data through the SPI hardware, it calls on spi_send, which internally performs an enqueue operation on the FIFO. The real-time process periodically dequeues this FIFO and sends this data over SPI.
4.5.1 Using Xenomai system calls

A real-time thread should only use Xenomai system calls to guarantee a deterministic behavior. If it requests a service from Linux, it loses real-time guarantees and risks missing a deadline. A thread running in a real-time context is said to be in primary mode, and when it switches to a Linux-controlled context, it is in secondary mode [15].

The Xenomai system calls used in the user-space access library are \_\_cobalt_ioctl, \_\_cobalt_read, \_\_cobalt_write, and \_\_cobalt_clock_nanosleep. To verify that no function is issuing a Linux system call, the Xenomai framework provides a way to monitor mode switches; this is further explained in Section 4.5.5.

4.5.2 pthreads and real-time threads

POSIX threads, or pthreads, are a mechanism that allow an application to run multiple tasks at the same time [12]. Xenomai provides wrapper directives to replace Linux pthreads with Xenomai threads. To give them real-time priority, the pthread_attr_setinheritsched, pthread_attr_setschedpolicy, and pthread_attr_setschedparam functions must be called before pthread_create. The scheduling policy must be set to SCHED_FIFO so that the real-time kernel schedules the thread [56].

To bypass the wrapper directives and use Xenomai service calls directly to configure and run a real-time thread, the \_\_cobalt_ prefix can be added to functions. For example, to create a thread without a wrapper, call the \_\_cobalt_pthread_create function [57].

4.5.3 Front vs. back-end

One important design decision was the separation of tasks between the real-time and non-real-time domain. During this project, it was decided that to reduce the total overhead for a real-time task, this should only act as an intermediary between SENSEI and the SPI interface. Therefore, as soon as it receives incoming SPI data, it should dispatch it to a non-real-time task. There it will be handled, and any additional computing should be done in a non-real-time domain.

From an engineering standpoint, it is necessary to further place certain tasks in the front or back-end of a real-time thread. It was decided that the user-space access library should only expose functions to initialize and close the SPI interface and to send and receive SPI messages. Figure 11 shows how tasks were divided into multiple abstraction layers, from the SIKA front-end to the SPI driver. The SIKA front-end requests to send or receive SPI data by calling the \_\_cobalt_write and \_\_cobalt_read functions provided by the user-space access library. Internally, this library calls another function, \_\_cobalt_wr_rd_device, which is a wrapper for the Xenomai system calls \_\_cobalt_write and \_\_cobalt_read.
4.5.4 Other design considerations

As it was explained in Section 4.5.1, to prevent any mode switches, it is necessary to choose the correct system calls and (perhaps) function wrappers. However, to guarantee real-time performance, it not sufficient to prevent mode switches; it is also necessary to choose functions that have bounded latencies and deterministic behavior, making this is one of the challenges when designing a real-time system [15].

One such design consideration is that of memory allocation. The GNU C Library provides an API for dynamic memory allocation and deallocation through malloc and free. Dynamic memory allocation is rarely used in real-time systems because of its non-deterministic behavior. A dynamic request for memory allocation can be rejected if there is not enough memory available, and if it is, the time it takes to allocate it is not bounded [58]. Given this, programmers typically choose to allocate memory before the execution of a real-time task statically. To keep the system’s behavior as deterministic as possible, all memory used by real-time tasks has been statically allocated.

Another critical design decision is related to exchanging data between real-time and non-real-time tasks. Data fetched by the real-time task from the SPI hardware should be dispatched to a non-real-time task as quickly as possible for it to be handled. One option to communicate real-time and non-real-time tasks would be to run each one in a different process. To communicate data between them, an interface such as a UNIX socket would be necessary, which is slower than the proposed solution via inter-thread communication.

To communicate two threads, they must be part of the same process. Threads of one same process share the same global memory and thus can exchange data in an easy and fast manner [12]. “However, in order to avoid the problems that can occur when multiple threads try to update the same information”, a special synchronization technique must be used [12]. In this case, data was shared between real-time and non-real-time threads through a circular FIFO data structure, typical for lock-free single producer/consumer tasks [28]. The FIFO is based on the C++ implementation by [59].

One advantage of using a lock-free data structure in a real-time context is that a slow or stalled process will not prevent other processes from accessing the data structure [60]. This means that, for the purpose of exchanging data between real and non-real-time tasks, if the non-real-time process is currently sleeping, the real-time task can still read or write...
to the data structure. Once the non-real-time task wakes up, it can respond to whatever data was added to the data structure.

### 4.5.5 Debugging with GDB

To verify that a real-time thread has not unexpectedly switched to secondary (Linux) mode, Xenomai provides a tool to monitor mode switches. The `/proc/xenomai/sched/stat` file contains multiple statistics about an ongoing Xenomai process, and can be viewed with the `cat` command. If any number of mode switches are listed under the MSW column, then there is a way to debug the responsible code.

By adding the following line [61] to a real-time task, it is possible to see if it triggers a mode switch:

```c
pthread_setmode_np(0, PTHREAD_WARNSW, NULL);
```

To catch an error signal, use the GNU Debugger (GDB) with the following commands in the terminal:

```
$ gdb real_time_application
$ catch signal SIGXCPU
$ run
```
5 Integration with SENSEI

SENSEI is a non-real-time background process that runs on ELK operating systems to manage sensors and GPIO pins on a device. It configures these hardware inputs and outputs by reading a JSON file that specifies where each one is connected, its type, and its expected behavior. By doing this, SENSEI makes it possible to reconfigure the hardware connections without having to recompile the software.

5.1 SENSEI Architecture

To make SENSEI hardware-agnostic, each ELK-supported device has its own hardware front-end. These front-ends contain specific details about a device’s input and output pins, and the communication scheme to interface SENSEI with a real-time application. For example, some systems use a microcontroller to handle all real-time tasks. In these cases, the communication between SENSEI and the microcontroller is done through a UNIX socket, and the details about this communication process are specified in the hardware front-end.

Apart from the hardware front-end, SENSEI has a user front-end to handle communication with a product-specific application. This front-end uses OSC or gRPC Remote Procedure Calls (gRPC) to communicate with an application that defines the system’s behavior. It can, for example, map specific input pins’ values to some output pins’ values, or communicate some input data via OSC or gRPC to another application.

For this project, data exchanged between SENSEI and a real-time task is communicated via a circular FIFO to ensure that it is read in the same order as it was generated. Since SENSEI behaves as a bottleneck (as it is not real-time safe), it was decided to minimize the data sent from a real-time task to SENSEI. To do this, a real-time task only adds data to the FIFO when the SPI interface detects a change in the state of a hardware device.

Figure 12 depicts SENSEI’s architecture and communication schemes to interface with other processes. This diagram shows one or many product-specific applications communicating to SENSEI via OSC or gRPC. SENSEI is capable of handling messages from many applications during run-time.

5.2 SENSEI Controllers

A JSON configuration file is used to declare and configure all hardware input and output devices connected to a physical system. The configuration file includes an ID number, a hardware type, the pin locations, and perhaps additional details about the hardware’s behavior. When SENSEI starts running, it parses this file and creates an object for each hardware device. These objects are called controllers and are used to keep track of each device’s state.
A controller object provides functions to get or modify a hardware device’s state in a way that will be transparent to a user. This way, a product developer may easily connect some hardware to a physical device, and specify its desired behavior through a user program that communicates with SENSEI. SENSEI keeps track of the hardware and enforces the behavior by issuing commands through the hardware front-end, which in turn controls the hardware.

The real-time task owns the controller objects and is in charge of receiving commands from SENSEI to configure and change their status. When the transaction is completed, the task sends an acknowledgment back to SENSEI. To exchange these messages, SENSEI has two loops to retrieve and push data to the FIFOs it shares with the real-time task.

### 5.3 Read and Write Loops

Two non-real-time threads are stuck in a continuous loop that read and write controller data. The read loop checks if there are any messages from the real-time thread. If so, then it decodes the message and pushes an answer to a command queue. The write loop takes the commands from the queue and sends them to the real-time task by adding them to the circular FIFO.
Figure 13 illustrates how lock-free communication between SENSEI and a real-time task works. SENSEI gets commands from a queue and pushes them to a circular FIFO. When the real-time task is awake, it will dequeue this command and send one in return through another circular FIFO. SENSEI will interpret the answer and push it to another queue.

5.4 GPIO Communication Protocol

The data exchanged between the real and non-real-time tasks follows a special format and message order per a communication protocol written by MIND Music Labs. This GPIO communication protocol allows tasks to send and receive data to handle SENSEI controllers. Messages or so-called packets have a command, data, time-stamp, and sequence number field. Any message sent from SENSEI should receive an acknowledgment from the real-time task, and SENSEI confirms all messages have been received by checking the sequence number field of the acknowledgments.

When SENSEI starts, it issues a few commands to verify the device is connected and to configure the hardware controllers. SENSEI will first send a system reset command, followed by a request for information on the hardware, then it sets the GPIO pins’ tick rate, and then it stops the GPIO system. Once the system is configured, SENSEI will instantiate the controllers declared on the JSON configuration file, and after they have all been configured, SENSEI will start the system.

5.5 SIKA Front-end

The SIKA is an extension shield for Raspberry Pi. It provides, among many other features, a total of 32 digital input and 32 digital output pins. These pins can be read or written through the SPI interface. The implementation details to control them are coded in a SENSEI hardware front-end called SIKA front-end.
One of the primary duties of the SIKA front-end is to initialize the real-time task that reads and writes data to SPI. Apart from this, the front-end exchanges GPIO protocol commands with the real-time task through the aforementioned circular FIFOs.

5.5.1 Real-time task

The SIKA front-end imports the wrapper library to control the SPI interface shown in Figure 10. The front-end launches the real-time task and then exchanges data with it. Despite wanting to minimize the overhead in the real-time domain (something that was mentioned in Section 4.5.3), there is no easy way around decoding GPIO protocol packets and handling the different possible commands in real-time. Because packets are unpacked in real-time, a command such as adding a new SENSEI controller must also be handled in real-time. Therefore, the memory that will be used by controller objects must be statically allocated before the real-time task starts running.

Figure 14 depicts how the real-time task handles commands from the GPIO communication protocol. After SENSEI creates the real-time thread, this checks if the circular FIFO is empty. In case it is not, it reads the oldest command inserted into it, and depending on the command type, it performs one of many possible operations. Figure 14 only shows some of these operations. Regardless of the type, an acknowledgment (ACK) message must be sent back to SENSEI to notify it the command was read and executed.

If the FIFO is empty, then the real-time task gets the SPI input values and checks if anything changed since the last iteration. If there has been a change in input values, it generates a message and sends the new values to SENSEI. If not, it does not send any messages to SENSEI and goes back to sleep. This scheme is meant to minimize the number of messages sent to SENSEI from the real-time task.
Figure 14: Real-time task flowchart
6 Benchmarking

The following tests were conducted on one or multiple Raspberry Pi 3 Model B V 1.2. The goal of these tests is to show the extent of a Raspberry Pi’s real-time capabilities when running a real-time co-kernel compared to solely running a Linux kernel.

6.1 Metrics

In any real-time context, it is important to know the system’s worst-case execution time. In fact, the average case is usually not as relevant, at least for hard real-time systems, since these systems require to know that the response time will always be small enough to meet all deadlines. The following tests measure Xenomai’s maximum latencies under various circumstances. To put these results in context, the same tests have been performed for Linux-only systems. To understand how these systems would behave in a real-world scenario, an additional load is added in the background as they perform a real-time task. To this end, the stress-ng tool was used to run the BCM2837 chip’s four cores at full capacity.

The loop-back test is meant to show a rough measurement of Xenomai’s scheduling latencies and response times to an SPI input signal. There is, however, a more precise tool to measure Xenomai’s scheduling latencies, with the results being reported in 6.3. Finally, to better understand how the RTDM SPI system measures up against a non-real-time Linux-based system, a test was set up to continually send data to a Raspberry Pi through SPI and count how many data packets are lost.

Because SENSEI is not designed to be real-time safe, no tests were done to measure its performance. The ultimate purpose of these tests is to prove that up to certain frequencies, under a given load, no SPI data will be lost and it can be communicated to SENSEI in real-time. SENSEI’s response time to these inputs is a whole other matter beyond the scope of this project.

6.2 Loop-back Test

To test the jitter displayed by the SPI interface, its Master Output Slave Input (MOSI) and Master Input Slave Output (MISO) pins are connected to each other with a cable forming a loop-back. It is known that the time it should take the data to travel from the SPI output registers, across the cable, and into the SPI input registers is considerably less than 1000 µs.

Let us imagine a real-time thread whose only function is to send data over SPI, sleep 1000 µs, and check if the sent data has already been received. Ideally, this thread would perform this check exactly every 1000 µs, and the purpose of this test is to measure how much this check would deviate from this ideal period. This deviation is the input/output jitter of the SPI interface in a real-time context.
To calculate the jitter, recall the definition of period jitter, given by Equation 3. If the resulting jitter were zero for all samples, it would mean that the system has perfect timing, that is, it does not have any additional latencies on top of the sleeping period. This is not possible, of course, since all RTOS have interrupt, scheduler, and task latencies.

To characterize the system, the mean value of all latencies can be computed. The system’s overall jitter is given by the standard deviation from this mean value [22].

This test was run three times on three different Raspberry Pi and the data was aggregated. Figure 15 shows the frequency distribution for the latency times of all three tests.

![Figure 15: I/O latency results for Raspberry Pi running Xenomai](image)

Table 2 shows the runtime, minimum period, maximum period, mean period, and standard deviation of the observed period times. The standard deviation is the system’s jitter.

<table>
<thead>
<tr>
<th>runtime (hrs)</th>
<th>min (µs)</th>
<th>max (µs)</th>
<th>mean (µs)</th>
<th>SD (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>38.42</td>
<td>15.625</td>
<td>113.854</td>
<td>22.265</td>
<td>3.667</td>
</tr>
</tbody>
</table>

Table 2: I/O latency summary results for Raspberry Pi running Xenomai

Note how the latency was never below zero, showing that the jitter in a Xenomai system is minimal, but they also show that the sleep period never finishes before the 1000 µs mark. The most important metric from this test is the jitter, which is 3.667 µs.
6.3 Latency Test

For Linux systems, there is a tool named cycletest to measure the latency of a real-time task. There is a similar tool for Xenomai called latency. Both launch a real-time task and measure its latency over a period of time. These tools offer a more accurate way to measure the latency of a system than the loop-back test.

Figure 16 shows the latency distributions for a Raspberry Pi running Linux only. Table 3 compares the latency distributions for a Raspberry Pi running Linux only and one with Xenomai and a Cobalt co-kernel. The standard deviation cannot be computed for this test because cycletest and latency only report integer and half values, respectively, and do not report latencies above an unspecified threshold. Therefore, there is no way to calculate the standard deviation accurately. Nevertheless, the maximum latencies already give a valuable insight into the possible worst-case execution times of the different systems under various workloads.

![Figure 16: Latency results](image)
<table>
<thead>
<tr>
<th></th>
<th>stress</th>
<th>runtime (hrs)</th>
<th>min (µs)</th>
<th>max (µs)</th>
<th>mean (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>no</td>
<td>16</td>
<td>5</td>
<td>586</td>
<td>15</td>
</tr>
<tr>
<td>Linux</td>
<td>yes</td>
<td>16</td>
<td>6</td>
<td>3853</td>
<td>12</td>
</tr>
<tr>
<td>Xenomai</td>
<td>no</td>
<td>20.51</td>
<td>-2.397</td>
<td>21.631</td>
<td>3.171</td>
</tr>
<tr>
<td>Xenomai</td>
<td>yes</td>
<td>20.31</td>
<td>-2.01</td>
<td>420.320</td>
<td>4.821</td>
</tr>
</tbody>
</table>

Table 3: Summary results for Linux and Xenomai latencies

### 6.4 SPI Input Test

A 100% hard real-time system that receives data from a remote source must not miss a single data packet. To test the real-time capabilities of a Raspberry Pi running Xenomai and an RTDM SPI driver, a test program was written to capture the values sent by an AVR microcontroller every 4000 µs (250 Hz). Per Nyquist’s theorem, the sampling frequency $f_s$ must satisfy:

$$ f_s \geq 2 \times 250\text{Hz} \tag{7} $$

Table 4 shows how many SPI data packets were lost while running a Raspberry Pi with Linux only (running the default SPI driver, spidev), versus the data lost when running Linux and a Cobalt co-kernel. Xenomai can also sample a 0.25 kHz signal with additional load in the background without losing any data.

To test how far the system can be pushed, a test was run at 25 kHz to sample a 12.5 kHz signal. Unlike previous tests (where the sampling frequency was four times the signal frequency) the sampling frequency is exactly two times the signal frequency, the minimum frequency to satisfy the Nyquist sampling criterion. It was observed that at frequencies equal or higher than 4 kHz, it is not possible to record the sampled data without some of it becoming corrupted. That is, read and write operations on files are not real-time safe at frequencies higher than 4 kHz. The proposed workaround to this problem was to verify in real-time if there has been a data loss. Once the application is ended, it reports the total execution time, the total number of records, and the number of lost packets.
<table>
<thead>
<tr>
<th>device driver</th>
<th>stress</th>
<th>signal freq.</th>
<th>sampling freq.</th>
<th>runtime (hrs)</th>
<th>lost data (packets)</th>
<th>total data (packets)</th>
<th>lost data</th>
</tr>
</thead>
<tbody>
<tr>
<td>spidev</td>
<td>no</td>
<td>0.25 kHz</td>
<td>1 kHz</td>
<td>16.58</td>
<td>306</td>
<td>14911597</td>
<td>0.002%</td>
</tr>
<tr>
<td>spidev</td>
<td>yes</td>
<td>0.25 kHz</td>
<td>1 kHz</td>
<td>2.285</td>
<td>115</td>
<td>2055823</td>
<td>0.006%</td>
</tr>
<tr>
<td>RTDM SPI</td>
<td>no</td>
<td>0.25 kHz</td>
<td>1 kHz</td>
<td>18.11</td>
<td>0</td>
<td>16292216</td>
<td>0</td>
</tr>
<tr>
<td>RTDM SPI</td>
<td>yes</td>
<td>0.25 kHz</td>
<td>1 kHz</td>
<td>14.82</td>
<td>0</td>
<td>13332764</td>
<td>0</td>
</tr>
<tr>
<td>spidev</td>
<td>no</td>
<td>12.5 kHz</td>
<td>25 kHz</td>
<td>0.483</td>
<td>15119049</td>
<td>19840710</td>
<td>76.202%</td>
</tr>
<tr>
<td>spidev</td>
<td>yes</td>
<td>12.5 kHz</td>
<td>25 kHz</td>
<td>3.263</td>
<td>99868129</td>
<td>135152911</td>
<td>73.893%</td>
</tr>
<tr>
<td>RTDM SPI</td>
<td>no</td>
<td>12.5 kHz</td>
<td>25 kHz</td>
<td>1.788</td>
<td>0</td>
<td>80186959</td>
<td>0</td>
</tr>
<tr>
<td>RTDM SPI</td>
<td>yes</td>
<td>12.5 kHz</td>
<td>25 kHz</td>
<td>10.55</td>
<td>916861</td>
<td>472155196</td>
<td>0.194%</td>
</tr>
</tbody>
</table>

Table 4: SPI input data acquisition
7 Discussion

7.1 Analysis

The results in Section 6 show that no matter how lax the requirements, a Linux-only system cannot offer the same real-time guarantees as a Xenomai-based one. The latency test shows that even under ideal circumstances, where there is no additional workload on the system, Linux exhibits very long scheduling latencies. For most applications (those for which a general-purpose OS was designed), this is fine, but for a real-time application, it would be undesirable.

In Section 6.4, a real-case situation was simulated: having multiple inputs connected to the SIKA board changing values at a rate of 0.25 kHz. The purpose of running these tests at a relatively low update rate is to show that even under the most generous circumstances, Linux is not suitable for hard real-time applications. During these tests, there was significant over-sampling and the sampling frequency was relatively low, and yet Linux missed a few deadlines, resulting in a 0.002% data loss. When running additional processes in the background, resulting in all four cores running at full capacity, this loss increased to 0.006%. By contrast, an RTDM SPI system does not miss a single one.

To show how far a Xenomai system can go, the experiments were done again but for a signal that updates every 12.5 kHz. Under ideal circumstances, the system would only have to monitor this signal. In such a case, the results show that no data would be lost. However, if the CPU is busy with other tasks, some of it is bound to be lost along the way. In this case, 0.194% of the deadlines were missed.

7.2 Limitations

One of the limitations of this research project is that it heavily relies on the cycle test and latency tools to measure a system’s latency times. These tools offer very accurate results but do not offer the same level of granularity as (for example) logging each data point as it was done in Section 6.2. The standard deviation that can be computed from the data returned by cycle test and latency would be imprecise. This metric yields valuable information to characterize a real-time system, and not being able to report it limits the analysis that can be done in this research project. It was, however, beyond the scope of this research to write an application to measure a system’s scheduling latencies that could simultaneously be as precise as latency, but return more granular data points.

7.3 Future Research

This research project will hopefully elicit new questions about real-time systems, whether it is about Raspberry Pi, Xenomai, music technology, or something at the intersection of them all. Some of the topics identified along the way that might be of interest for future
research are how to compute a system’s latencies’ standard deviation based on the data returned by the latency tool. Another interesting question is how to make read and write file operations real-time safe for high-speed systems that need to log data from the environment. As it was mentioned in Section 6.4, above 4 kHz, data written to a text file can become corrupted.

Aside from this question, it will be interesting to see how the new Raspberry Pi 4 will perform in terms of meeting real-time deadlines. Its hardware improvements should make it faster, but this might not necessarily mean it will be more predictable. However, before this question is answered, it will be necessary to port Xenomai to the new Raspberry Pi, which in and of itself might be an interesting research topic: to analyze the challenges associated with building a real-time system on Raspberry Pi 4 with Xenomai.

7.4 Conclusions

The established goal at the beginning of this research project was to assess if it is possible to use a Raspberry Pi to achieve real-time performance, specifically to exchange data over SPI to monitor and control GPIO pins. Previous research has shown that it is possible to meet firm and hard real-time requirements on systems running Xenomai. This project demonstrates that a Raspberry Pi that has a Linux and Cobalt co-kernel configuration is suitable for hard real-time applications that require it to operate at a 1 kHz frequency, even under heavy stress. Furthermore, it has been shown that a comparable Linux-only system might exhibit relatively similar minimum latencies to a Xenomai one, yet it is not suitable for real-time applications due to its unpredictability, exemplified by its maximum latency times.

An experiment was set up to show that Xenomai will not lose any SPI data up to a particular frequency and workload. A Linux system, no matter how low the sampling frequency is, will lose some data during the experiment, showing how it can (at best), offer soft real-time guarantees.

Ultimately, the goals of this project have been met. The problem posed in the beginning has been solved: an RTDM SPI system was designed to exchange data in real-time, showing that a Raspberry Pi running Xenomai can, in fact, offer real-time guarantees for systems that must sample data at 1 kHz.
References


Appendix

A Building a Linux Kernel with Xenomai

Install these tools which will be necessary to complete the remaining steps:

```
$ sudo apt-get install make git bison flex libssl-dev libncurses-dev bc
     build-essential autoconf libtool gcc-arm-linux-gnueabihf pkg-config libfuse-dev
```

Download the toolchain to the home directory:

```
$ cd /home
$ git clone https://github.com/raspberrypi/tools
```

Assuming a 64-bit host computer to cross-compile the kernel, issue the following command:

```
$ echo PATH=$PATH:/home/tools/arm-bcm2708/gcc-linaro-arm-linux-gnueabihf-raspbian-x64/bin >> ~/.bashrc
$ source ~/.bashrc
```

Get the kernel source files for version 4.14.y:

```
$ git clone --depth=1 --branch rpi-4.14.y https://github.com/raspberrypi/linux
```

While this project was being done, Linux 4.14.y was on revision 98, making it version 4.14.98. To downgrade to version 4.14.85, create a new git branch and make a directory where to save the Linux patches:

```
$ cd linux
$ git checkout -b ver-4.14.85
$ mkdir patches-linux
$ cd patches-linux
```

Download patches 97-98 through 85-86 with the following command:

```
```

Then, decompress the downloaded files

```
$ unxz *
```

And apply each patch in reverse, starting from patch 97-98 down to 86-85:
Commit these changes:

$ git add .
$ git commit -m "Message about patches here."

Make a new branch for Xenomai:

$ git checkout -b xenomai-rpi-4.14.85
$ mkdir patches-ipipe
$ cd patches-ipipe
$ cd ..
$ mkdir xenomai
$ cd xenomai
$ tar xjf xenomai-3.0.8.tar.bz2
$ git add .
$ git commit -m "Message about the new xenomai branch."
$ git push origin xenomai-rpi-4.14.85

Make a new branch to make your own I-pipe patch:

$ find . -name "*.rej"

For each .rej file, edit the associated .c file.

Create your new patch:


Create a new branch:

$ git checkout -b working-xenomai-rpi-4.14.85
$ git add .
$ git commit -m "Made a new branch where to apply the newly created xenomai patch."
$ git push --all origin

A new patch has been created, which should now work without any errors when running the prepare-kernel.sh script. Apply this new patch:
To start building the sources and device tree files:

```
$ KERNEL=kernel7
$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- bcm2709_defconfig
```

Configure the kernel options:

```
$ export ARCH=arm
$ make menuconfig
```

Inside menuconfig, disable the following kernel features:

- Contiguous Memory Allocator (under Kernel features)
- Allow for memory compaction (under Kernel features)
- CPU Frequency scaling (under CPU Power management)
- KGDB (under Kernel hacking)

Build the sources:

```
$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- zImage modules dtbs
```

Build external kernel modules and device tree blobs with modules_install and dtbs_install:

```
$ export INSTALL_MOD_PATH=/home/rt-kernel/
$ export INSTALL_DTBS_PATH=/home/rt-kernel/
```

Inside the ‘/home/linux‘ directory run the following:

```
$ cd /home/linux
$ make -j4 modules_install
$ make -j4 dtbs_install
```

Make a blob of the data:

```
$ mkdir $INSTALL_MOD_PATH/boot
$ ./scripts/mkknlimg ./arch/arm/boot/zImage $INSTALL_MOD_PATH/boot/ $KERNEL.img
```
B Installing a Linux Kernel with Xenomai

Transfer the Kernel:

```bash
$ cd $INSTALL_MOD_PATH
$ tar czf ../xenomai-kernel.tgz *
```

Transfer the .tgz file to the Raspberry Pi using scp:

```bash
$ cd ..
$ scp xenomai-kernel.tgz pi@<ipaddress>:/tmp
```

Now, type in the Raspberry Pi terminal:

```bash
$ cd /tmp
$ tar xzf xenomai-kernel.tgz
$ sudo cp *.dtb /boot/
$ cd boot
t$ sudo cp -rd * /boot/
t$ cd ../lib
$ sudo cp -dr * /lib/
$ cd ../overlays
$ sudo cp -d * /boot/overlays
$ cd ..
$ sudo cp -d bcm* /boot/
```

Add the following to /boot/config.txt:

```bash
$ kernel=kernel7.img
$ device_tree=bcm2710-rpi-3-b.dtb
```

Add these options to /boot/cmdline.txt:

```bash
$ dwc_otg.fiq_enable=0 dwc_otg.fiq_fsm_enable=0 $ dwc_otg.nak_holdoff=0
```
C Installing the Xenomai Libraries

```bash
$ cd /home/linux/xenomai-3.0.8
$ ./scripts/bootstrap --with-core=cobalt -enable-debug=partial
$ export CROSS_COMPILE=arm-linux-gnueabihf-
$ ./configure CFLAGS="-march=armv7-a -mfpu=vfp3" LDFLAGS="-mtune=cortex-a53" --build=i686-pc-linux-gnu --host=arm-linux-gnueabihf --with-core=cobalt --enable-smp CC=${CROSS_COMPILE}gcc LD=${CROSS_COMPILE}ld
$ make -j6 install DESTDIR=${PWD}/target
$ cd target
$ tar czf ../../xenomai-tools.tgz *
$ cd ../..
$ scp xenomai-tools.tgz pi@<ipaddress>:/tmp
```

To deploy the test suite on a Raspberry Pi, type in the terminal:

```bash
$ cd /tmp
$ sudo tar xzf xenomai-tools.tgz
$ sudo cp dev/* /dev/
$ cd ../
$ scp xenomai-tools.tgz pi@<ipaddress>:~/tmp
```

Test the kernel latency using the test suite:

```bash
$ cd
$ sudo /usr/xenomai/bin/latency
```

To verify that the installation was successful, run xeno-test:

```bash
$ sudo /usr/xenomai/bin/xeno-test
```