DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERING, SECOND CYCLE, 30 CREDITS

STOCKHOLM, *SWEDEN 2018*

Real-Time Linux Testbench on Raspberry Pi 3 using Xenomai

GUSTAV JOHANSSON

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Abstract

Test benches are commonly used to simulate events to an embedded system for validation purposes. Microcontrollers can be used for making test benches and can be programmed with a bare-metal style, i.e. without an Operating System (OS), for simple cases. If the test bench would be too complex for a microcontroller, then a Real-Time Operating System (RTOS) could be used instead of a more complex hardware. A RTOS has limited functionalities to guarantee high predictability. A General-Purpose Operating System (GPOS) has a vast number of functionalities but has low predictability.

The literature study looks therefore into approaches to improve the real-time predictability of Linux. The result of the literature study finds an approach called Xenomai Cobalt to be the optimal solution, considering the target use-case and project resources.

The Xenomai Cobalt approach was evaluated on a Raspberry Pi (RPi) 3 using its General-Purpose Input/Output (GPIO) pins and a latency test. An application was written using Xenomai's Application Programming Interface (API). The application used the GPIO pins to read from a function generator and to write to an oscilloscope. The measurements from the oscilloscope were then compared to the measurements done by the application.

The result showed the measured differences between the RPi 3 and the oscilloscope. The result of the measurements showed that reading varied 66.20 µs, and writing varied 56.20 µs. The latency test was executed with a stress test and the worst measured latency was 82 µs.

The resulting measured differences were too high for the project requirements. However, the majority of the measurements were much smaller than the worst cases with 23.52 µs for reading and 34.05 µs for writing. This means the system could be used better as a firm real-time system instead of a hard real-time system.

Keywords

Automatic Testing; General-Purpose Operating System; Raspberry Pi; Real-time Linux; Xenomai Cobalt

Sammanfattning

Testbänkar används ofta för att simulera händelser till ett inbyggt system för validering. Till simpla testbänkar kan mikrokontroller användas. För mer avancerade testbänkar kan RTOS användas på mer komplex hårdvara. RTOS har begränsad funktionalitet för att garantera en hög förutsägbarhet. GPOS har stora mängder funktionaliteter men har istället en låg förutsägbarhet.

Litteraturstudien undersökte därför möjligheterna till att få Linux att hantera realtid. Resultatet av litteraturstudien fann ett tillvägagångssätt vid namn Xenomai Cobalt att vara den optimala lösningen för att få Linux till Real-Time Linux.

Xenomai Cobalt utvärderades på en RPi 3 med hjälp av dess GPIO-pinnar och ett fördröjningstest. En applikation skrevs med Xenomai's API. Applikationen använde GPIO-pinnarna till att läsa från en funktionsgenerator och till att skriva till ett oskilloskop. Mätningarna från oskilloskopet jämfördes sen med applikationens mätningar.

Resultatet visade mätskillnaderna mellan RPi 3 och oskilloskopet med systemet i viloläge. Resultatet av mätningarna visade att läsningen varierade med 66.20 µs och skrivandet med 56.20 µs. Fördröjningstestet utfördes med stresstestning och visade den värsta uppmätta fördröjningen, resultatet blev 82 µs.

De resulterande mätskillnaderna blev dock för höga för projektets krav. Majoriteten av mätningarna var mycket mindre än de värsta fallen med 23.52 µs för läsning och 34.05 µs för skrivning. Detta innebar att systemet kan användas med bättre precision som ett fast realtidssystem istället för ett hårt realtidssystem.

Nyckeord

Automatiserad testning; Generellt operativsystem; Raspberry Pi; Realtids-Linux; Xenomai

Acknowledgements

I would like to thank my KTH supervisor Tage Mohammadat, who was very helpful answering all my emails with questions, and giving me continuous feedback and suggestions on the report.

I would also like to thank my supervisors as Saab Dynamics AB, Björn Johansson, Håkan Pettersson, and Mattias Helsing, for helping me and providing me with all the necessary tools.

Finally I would like to thank Greg Gallagher at Xenomai for answering all my emails, and helping me debug the Xenomai setup.

Stockholm, 23rd of June, 2018
Gustav Johansson

Table of Contents

Abstract **i**

Sammanfattning **ii**

Acknowledgements **iii**

Table of Contents **iv**

List of Figures **vi**

List of Tables **vii**

List of Acronyms **viii**

1 Introduction **1**
1.1 Project Description 1
1.2 Requirements 2
1.3 Problem Statement 3
1.4 Purpose 3
1.5 Goals 3
1.6 Method 4
1.7 Thesis Structure 5

2 Background **6**
2.1 Real-Time Systems 6
2.2 Linux 8
2.3 Real-Time Linux 10
2.4 Xenomai's Architecture 13
2.5 Raspberry Pi 16
2.6 Related Work 19
2.7 Summary 20

3 Method **22**
3.1 Xenomai Cobalt 22
3.2 Application 23
3.3 Data Collection 24
3.4 Summary 29

4 Xenomai Installation & Application **30**
4.1 Xenomai Installation 30
4.2 Application 36

4.3 Summary 42

5 Experimental Setup **43**
5.1 Automation 43
5.2 Data Collection 47
5.3 Encountered Problems 47
5.4 Summary 48

6 Results & Analysis **49**
6.1 Types of Measurements 49
6.2 Results 50
6.3 Analysis 53
6.4 Summary 54

7 Conclusions & Future Work **55**
7.1 Conclusions 55
7.2 Future Work 57
7.3 Summary 57

References **58**

Appendices **62**

A Xenomai Setup (Unsuccessful Attempt Using RPi Kernel) **62**

B GPIO Template **67**
B.1 Makefile 67
B.2 Code 68

List of Figures

2.1 Interrupt Latency 9
2.2 Linux Architecture 10
2.3 Topology of the two approaches for real-time Linux 10
2.4 Xenomai's Architecture 14
2.5 The interrupt pipeline, inspired by [26, Figure 13-2] 14
2.6 Xenomai's Cobalt core configuration, taken from [27, Figure 1] . 15
2.7 Xenomai's Mercury core configuration, taken from [27, Figure 2] 16
2.8 Cache topology for Raspberry Pi 3 19

3.1 The flow of the application 24
3.2 Simplistic structure of the measurement setup 25
3.3 Latency model 26
3.4 A measurement example for writing signals 27
3.5 A measurement example for reading signals 27
3.6 Standard deviation per number of samples for reading 28
3.7 Standard deviation per number of samples for writing 28

4.1 Kernel configuration: System Type 33
4.2 Kernel configuration: Xenomai GPIO 33
4.3 Kernel configuration: Xenomai 34
4.4 Gantt chart over the execution order 38

5.1 The flow of the automation process 43
5.2 Oscilloscope waveform conversion into CSV 45
5.3 A test case example displayed on the oscilloscope 46
5.4 Stress attempt with a stress controller 46

6.1 Histogram of reading and writing for the Regular type 51
6.2 Box plot of writing measurements 52
6.3 Box plot of reading measurements 52
6.4 Box plot of writing differences comparing with and without reading 54

List of Tables

2.1 Raspberry Pi hardware specifications 17
2.2 Cache information 18

3.1 Example showing time difference from application and oscilloscope 27

6.1 Latency result 50

7.1 Final results 55

List of Acronyms

ADEOS	Adaptive Domain Environment for Operating Systems
API	Application Programming Interface
ARM	Advanced RISC Machine
CPU	Central Processing Unit
CSV	Comma Separated Values
DTBS	Device Tree Blobs
EDF	Earliest Deadline First
FIFO	First In First Out
FPU	Floating Point Unit
FSMLabs	Finite State Machine Labs, Inc
GPIB	General-Purpose Interface Bus
GPIO	General-Purpose Input/Output
GPL	GNU General Public License
GPOS	General-Purpose Operating System
HAL	Hardware Abstraction Layer
I-Pipe	Interrupt Pipeline
I/O	Input/Output
IoT	Internet of Things
IRQ	Interrupt Request
ISA	Instruction Set Architecture
LITMUS$^{\mathbf{rt}}$	Linux Test bed for Multiprocessor Scheduling in Real-Time
MOESI	Modified Owned Exclusive Shared Invalid
OS	Operating System
POSIX	Portable Operating System Interface
pSOS	Portable Software On Silicon
RPi	Raspberry Pi

RTAI	Real-Time Application Interface
RTDM	Real-Time Driver Model
RTOS	Real-Time Operating System
SBC	Single-Board Computer
SoC	System on a Chip
SSH	Secure Shell
TSC	Time Stamp Counter
USB	Universal Serial Bus
VISA	Virtual Instrument Software Architecture

1 Introduction

A common approach in building test benches is to use microcontrollers such as Arduino. The microcontrollers are then used to simulate trigger signals towards a target and capturing signals using Input/Output (I/O) pins. A microcontroller without an OS can be developed to support timely measurement of events and I/O trigger signals at the lowest overhead possible, i.e. high efficiency. This, however comes at the expense of lower extensibility and higher development time. Moreover, microcontrollers efficiency have limited memory and computation power which limits the use-cases, e.g. inability to use them to log large amount of data.

RPis are a series of inexpensive platforms with a large and active community. The RPi models have similar physical size and features of an Arduino but have higher computational resources and larger memories that make them capable of running complex OS. Instead of having a fixed memory size as Arduino, it uses SD cards as secondary memory which gives the possibility to use much more storage for applications and logging. However, the RPi sacrifices time predictability for complexity and features when using a Linux OS.

The problem with Linux and many other GPOSs is that it was not originally designed with real-time properties in mind, but the focus was instead on desktop and server environments [1]. The result is a very user-friendly environment but highly unpredictable regarding timely executions.

A solution for unpredictable GPOSs is to use a RTOS instead. The RTOSs have many features to help the application developers to be more efficient while providing real-time guarantees [2, PP. 79-80]. When comparing an RTOS with a GPOS: the RTOS is more limited in regards to features as the RTOS only provides with functionalities which guarantee real-time executions. Many developers would want the features provided by a GPOS but they are aware of the compromises done by the RTOSs.

1.1 Project Description

The purpose is to investigate if real-time performance can be achieved on RPi 3 using Real-Time Linux. The timing accuracy of the RPis GPIO pins needs to be measured as well in order to know if RPi 3 is suitable for test benching purposes.

1.1.1 Problem

A common approach in test systems is to use a microcontroller to simulate trigger signals towards a target and capture output signals using I/O pins. A microcontroller without an OS has real-time properties in terms of time measurements of captured events and timing of out-signals. However, microcontrollers have limited memory and thus cannot log much data during executions. The size of the program is also limited because of the small memory of the microcontrollers.

1.1.2 Motivation

The RPi is a series of low-cost platforms and has a large and active community. It can be equipped with a large SD card and has thus capabilities to store lots of sampled data. The Linux kernel offers several useful functions such as communication and various hardware services. If real-time performance is achieved RPi's would be a powerful replacement to the microcontrollers used for integration testing. The RPi also has support for high-level languages such as Python and could thus make it easier to develop the test-code.

1.1.3 Task

The task is to examine the RPi and the theory behind RTOSs to find out if the RPi can be used for real-time triggering and measuring. The goal is to output signals with a time precision within $\pm 10\,\mu s$, and also timestamp measured changes of input signals with the same precision. These measurements should not be delayed by general OS operations.

1.2 Requirements

From the background represented so far and with the project description, requirements have been established on the system and the implementation.

- The approach chosen need to support the hardware of RPi 3.
- The GPOS preferred is Raspbian.
- The approach needs hard real-time support.
- The system needs to handle real-time tasks which won't be interrupted or degraded while using generic tasks from GPOS.
- The system needs to be able to set up communication between real-time tasks and general-purpose tasks.

- The implementation of the system needs to be able to read and write GPIO pins with a time precision of $\pm 10\,\mu s$, and include timestamps within the same precision.
- The implementation of the system needs to be able to monitor and log data, either during or after the real-time measurements have been performed.

1.3 Problem Statement

Previous attempts have been done into combining the features of GPOSs and the predictability of RTOSs. The question which remains is: how does such an approach perform in terms of time predictability, especially on a RPi 3 for test bench purposes?

1.4 Purpose

The purpose of the thesis is to provide an analysis of how deterministic a real-time Linux OS can perform on RPi 3, a popular and low-cost Internet of Things (IoT) hardware device. Details on how the platform can be set up for future projects which require real-time properties will also be provided.

1.5 Goals

While a GPOS provides a vast amount of complex use cases, it does not provide real-time properties. The RPi is one of the most popular computer systems used for IoT projects. The most popular OS executed on the RPi called Raspbian. Raspbian is, however, a GPOS and thus is limited in regards to real-time properties. The main goals of the thesis are:

1. Modify the GPOS called Raspbian for RPi 3 so it can handle real-time executions.
2. Evaluate the real-time capabilities of the system to make timely digital triggers/stimuli using the GPIO pins accessible on the RPi 3.

1.5.1 Benefits, Ethics and Sustainability

Before the project, a microcontroller was used for the test bench. The microcontroller needed, however, to be connected with a computer which had complete control over it. The computer could then get the logged data and re-flash the

microcontroller whenever needed. With a successful project, the microcontroller and computer could be replaced by a single RPi. The benefit of this is a simpler solution where only one hardware device is needed. Another benefit is the power consumption, RPi 3 uses at most 12.75 W [3] while an average desktop computer uses at average $60 - 250$ W [4]. Therefore, if a RPi 3 would be able to replace a desktop for a test bench, a more sustainable approach could be demonstrated.

The procedure of the thesis will not include personal information of any kind. It will also not give any information about the company which the thesis is conducted at. It is therefore decided that no further details of ethics are needed.

1.6 Method

Acquiring real-time properties for Linux have been in focus for a time. Mainly two approaches have been attempted [1, P. 432]. The first approach is called the virtualization, interrupt abstraction or dual-kernel. The second approach modifies the Linux kernel to decrease latencies and introduce real-time features. More details can be read in section 2.3.

The first real-time extension which was introduced to Linux was called RTLinux [5] (not to be confused with PREEMPT_RT [6] which is often called real-time Linux) which used the dual-kernel approach. In a dual-kernel approach, a microkernel is introduced, which controls everything on the system. The microkernel is responsible for providing the real-time executions. The microkernel executes the Linux kernel as a thread with the lowest priority. After RTLinux, multiple attempts have been done with the same approach, two of them are called Real-Time Application Interface (RTAI) [7] and Xenomai [8].

The most popular approach of kernel modification is called PREEMPT_RT [6]. Its focus was to make the Linux kernel more deterministic and predictable by making the Linux kernel as preemptible as possible to limit latencies and jitter. Another kernel modification approach was called SCHED_DEADLINE [9] or SCHED_EDF which is a Central Processing Unit (CPU) scheduler. As the name suggests, the scheduling algorithm is based on Earliest Deadline First (EDF). SCHED_DEADLINE is now included in the Linux kernel since version 3.14 [10].

Xenomai was chosen to be used for the thesis as it provides everything necessary to fulfill the requirements set in section 1.2. Details on Xenomai can be read in section 2.4.

The objectives in this thesis can be seen in the list below:

1. Research on the current approaches for acquiring real-time properties for Linux (section 2.3).

2. Evaluate the possibility of using chosen approach for RPi 3 (section 1.2).
3. Setup the approach for a RPi 3 running the Raspbian OS (section 4.1).
4. Implement an application which uses the GPIO pins of the RPi 3 in order to characterize measure its performance (section 4.2).
5. Setup an automated experimental setup for data collection (chapter 5).
6. Analyze the result from the data collection (chapter 6).

1.7 Thesis Structure

The thesis is divided into six following chapters. Each chapter is given a short description below.

Chapter 2: Background, describes the theoretical background for the thesis. It describes the important insights in the current literature. Insights such as real-time, real-time Linux approaches, Xenomai, and RPi hardware are listed . The chapter ends with a project description and conclusions based on the literature insight and project requirements.

Chapter 3: Method, describes the method used to be able to fulfill the goals set for the project. The method of installing Xenomai Cobalt onto RPi 3 was first described. Secondly, the measuring application described. Thirdly was the data collection described, which was going to use an automated process to gather all the necessary data. The data collection also described how the analysis was going to be executed as well as determining reasonable sample sizes for the data collection.

Chapter 4: Xenomai Installation & Application, describes the method on how the real-time characteristics are introduced to RPi 3. Afterward, the measuring application needed for the project is described in detail, with the motivation behind every step.

Chapter 5: Experimental Setup, describes in full detail how the experimental setup was done. It describes the automation process which was implemented and used. Data collection is then described in detail, i.e. how the automation process gathered all necessary data for the next chapter.

Chapter 6: Results & Analysis, describes which types of measurements were done and displays the results. The results are then compared and analyzed in detail.

Chapter 7: Conclusions & Future Work, describes the conclusions of the project based on the results derived from the data measurements, and the overall experience. The project requirements which was made at the beginning of the project are then compared with the end result. Lastly examples of future work are described and motivated.

2 Background

In order to progress further into the thesis an insight into the current literature is necessary. This chapter goes through the topics and areas needed to be known to understand the further chapters of the thesis.

2.1 Real-Time Systems

Many computational systems rely on computing tasks with a need for precise execution time. These systems are called real-time systems. In these systems, it is most important that the tasks the system withholds are executed when they should. The result from tasks executing too late or too early can be considered useless or even dangerous depending on the use and responsibility of such a system [1, P. 1].

Today there are real-time systems almost everywhere in our society as most computer systems are in fact real-time systems. Applications, where real-time systems are crucial, are for example power plants, robotics, and flight control.

2.1.1 Classifications

The tasks in real-time systems are often put in three different categories depending on what the result would be of a missed deadline. The categories are hard, firm and soft [1, P. 9].

Hard Tasks considered hard must never miss their deadline as the consequences can be catastrophic for either the system or its environment.

Firm Tasks considered firm can miss their deadlines, but the computed result of the tasks would be considered useless. However, the result is not damaging the system or its environment.

Soft When tasks, considered soft, miss their deadlines, the result would still be useful but not as good as if it would have been, had the result not missed the deadline.

2.1.2 Predictable Systems

In order for real-time systems to be able to satisfy their strict requirements of timing, they need to be predictable. Predictable system design is however very

difficult with the increasing complexity of the computer system architecture design development [11]. The focus on computer system architecture design is often to improve the performance for general-purpose use, which does not include a strict requirement of timing. Because of this, it is not unusual for improvements to affect the predictability in a negative way.

Microarchitecture

Computer system architecture contain something called microarchitecture, which defines how the Instruction Set Architecture (ISA) are processed in the processing unit. This is not identical on all hardware, instead, ISAs may have several ways to be processed on different hardware. This causes different execution times and thus affect the predictability. The main features of typical microarchitectures are described below with their effect on predictability.

CPU Pipeline
: Pipelines increase the performance of the system by allowing different instructions to be overlapped when executed. Pipelines complicate timing analysis however as single instructions can no longer be analyzed separately in isolation [11]. The instructions have to be analyzed collectedly in order to obtain the timing bounds. More complicated pipelines such as superscalar and out-of-order increases the search space, possible interleavings, and thus the complexity of the timing analysis increases.

Cache
: Caches serve as a middle point between the processor and the main memory thanks to their low latencies compared to the main memory. The caches are however very limited in size to keep latencies and costs down. Cache hierarchies and different replacement policies exist to balance the cache's limited sizes. Having caches in the microarchitecture increases the unpredictability, as the latencies vary depending on the temporal and spatial data access patterns with respect to cache architecture. This, consequently, makes it difficult to control and to know access time, and hence overall computation, to a cycle-accurate degree of precision.

Multi-threading
: Hardware multi-threading provides the system with the ability to execute multiple processes or threads concurrently. While multi-threading improves the performance, it increases the complexity of the timing analysis because, among others, of resources being shared [11].

Multi-Core

Systems with multi-core architecture have significant increase in performance in relation to single-core systems, at the expense of increasing complexity and indirectly reducing predictability. The cores on a multi-core architecture usually

each have a private cache but share caches on higher levels of the cache hierarchy. The CPU cores are not isolated from each other because of the shared resources in the system. This causes the timings and performance of each core being dependent on the workload on the other cores. For example, the shared cache can be invalidated by other cores which increases the number of cache misses for a certain other core, thus decreases the predictability [11].

2.1.3 Real-Time Operating System

A RTOS is an OS which provides reliable and predictable functionalities in order to achieve real-time guarantees [2, PP. 79-80]. This means that functionality which cannot be predictable for time requirements does not exist on RTOSs. A RTOS take care of the resources of the hardware so that the application developers do not need to know every detail of the hardware, and instead can focus on the functionality of the application. RTOSs have usually a low overhead compared to a GPOS in order to achieve a lower system cost and as mentioned, predictability.

Most application developers would want to have the functionality provided with a GPOS but accepts the compromises done for the lower overhead and predictability.

2.2 Linux

Linux is a free and open-source GPOS with many different variants called distributions. It was originally developed on Intel x86 architecture in the year 1991 [12] but has since then been ported to many other platforms. Linux was originally designed to be used in desktops and servers [1, P. 432]. Linux is today the most ported OS and is also the most common OS on servers and mainframe computers. The Android OS is based on a modified version of the Linux kernel. Android has a market share of over 73% overall mobile phone devices (late 2017) [13].

2.2.1 Development

The main difference between Linux and most other OSs is that, Linux has always been developed as open-source and free software. The most common open-source license used in Linux is GNU General Public License (GPL) which is a copyleft, meaning that anything taken from GPL must be used under the same license [14].

2.2.2 Linux Kernel

The Linux kernel can be considered as either the heart or brain of the Linux OS. The kernel is the first program to be loaded on the system when the system boots. The kernel is loaded in a protected space in the memory called kernel space [15]. Everything the average user does is done on a different memory space called userspace. The kernel manages everything on the system: e.g. processes, memory, files, and I/O. Whenever a user process needs to access certain areas, for example, hardware details, the kernel takes control and retrieve the information needed to the user process. This process is called system call or syscall for short.

The properties of the kernel vary on different OSs but the kernel usually includes a scheduler. The scheduler determines how all the processes in the system shall be handled, in what order and priority.

The Linux kernel, as many OSs is driven by interrupts. For example, the scheduler is controlled by timer interrupts of a clock. The scheduler is awoken by the interrupt and then reschedules whatever necessary. Other hardware can also generate interrupts for the scheduler for a fast handling of hardware.

An example of an interrupt sequence can be seen in Figure 2.1 below. A task is waiting to be executed with the help of an interrupt. It can, for example, be a task waiting just to be rescheduled when a timer interrupt occurs.

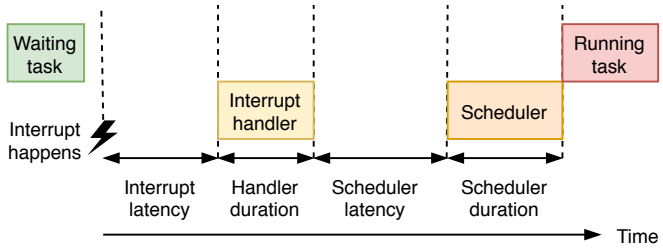


Figure 2.1: Interrupt Latency

The architecture topology of Linux can be viewed below in Figure 2.2. The architecture shows a Hardware Abstraction Layer (HAL) sitting beneath the kernel and above the hardware. The HAL abstracts the hardware with software to achieve a less hardware dependent architecture, so the kernel and processes on top do not need to be changed for each different type of hardware. For a user process to gain access to a device driver, it needs to go through the kernel processes with a system call.

The Linux kernel is in continuous development and can be acquired through its kernel source tree Git repository [16].

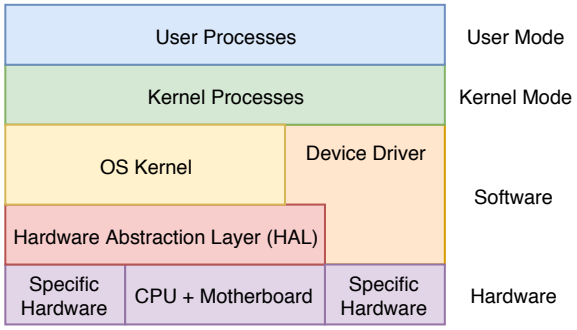
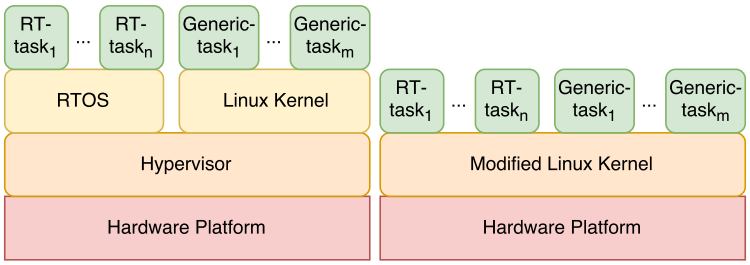


Figure 2.2: Linux Architecture

2.3 Real-Time Linux

Linux is as mentioned a GPOS which was designed for desktops and servers. Because of this, Linux is not suitable for real-time as it can cause high and uncontrolled latencies. But because of the popularity of Linux, and of being open-source, a few approaches have been undertaken in order to make Linux more suitable for real-time computing. There have been mainly two different approaches on achieving real-time properties on Linux. The first approach is commonly called dual-kernel, where interrupt abstraction or virtualisation are applicable. The second approach is by modifying the Linux kernel directly [1]. When comparing the two approaches, it is common to compare them with dual-kernel and single-kernel. For a visual description of the two approaches, see Figure 2.3 below.



(a) Dual-kernel (b) Single-kernel

Figure 2.3: Topology of the two approaches for real-time Linux

virtualization-based

virtualization-based solutions are being done by having a very small kernel (also often called hypervisor) which redirects all hardware interrupts. The hypervisor assign resources such as processor time, memory space, peripherals and more. This is a virtualization step. With this solution,

the hypervisor can prioritize real-time processes over the Linux kernel processes. The Linux kernel is running as a thread with idle priority, meaning that it will only execute when all the real-time tasks have finished executing. Because of this architecture, the dual-kernel approach can achieve hard real-time [17, P. 4]. This approach significantly lowers latencies compared to single-kernel approach [18].

Kernel modification
: This approach focuses instead on modifying the Linux kernel to make its performance more predictable. It is common to reference this approach as the single-kernel approach to easily distinguish it from the dual-kernel approach.

2.3.1 RTLinux

RTLinux [5] was developed by Finite State Machine Labs, Inc (FSMLabs) and was covered by a US patent (5885745), which was not valid outside of USA [18]. RTLinux used a very small kernel called microkernel to handle the hardware and therefore is a dual-kernel approach. The Linux OS and its kernel is running on a thread with the lowest priority. When the microkernel receives an interrupt it checks first if the interrupt is related to the real-time tasks running or not. If the interrupt is real-time related, then the correct real-time task will be notified. If the interrupt is not related to real-time then the interrupt will be flagged and later used when the Linux OS is allowed to execute. The system achieves low latencies for real-time tasks but suffers from some drawbacks. One is that device drivers will often need to be rewritten in order to work for real-time tasks [1, P. 433].

A company called Wind River System acquired FSMLabs in the year 2007 and made a renamed version Real-Time Core available for their Linux distribution. In the year 2011, however, was the Real-time Core discontinued in development [19].

2.3.2 Real-Time Application Interface

RTAI [7] is an open-source community project and started as a modification from RTLinux. Thanks to the project being open-source, RTAI support more architectures than RTLinux [18]. Because of the patent in RTLinux, the community behind RTAI developed a new kernel (nanokernel) called Adaptive Domain Environment for Operating Systems (ADEOS) to replace its relation to RTLinux.

2.3.3 Xenomai

Both RTLinux and RTAI had the same drawback: real-time tasks executed on the same level as the Linux kernel code and there was no memory protection between them [18]. An error in the real-time tasks (e.g. segmentation fault) could cause the entire system to crash. This problem often occurred during development and the developer often needed to reboot the entire system to continue.

This was where Xenomai [8] came in; it used the ADEOS nanokernel but also allowed real-time tasks to execute in Linux userspace. The real-time tasks can execute in two domains. The real-time tasks start initially in the primary domain, controlled by the RTOS, the other domain is controlled by the Linux scheduler. When a real-time task needs to use a function accessed only in Linux API, the task will be transferred temporarily into the Linux domain until the function has finished executing. This allows the developer to take advantage of Linux but increases the unpredictability when real-time tasks are inside the Linux domain [18].

2.3.4 PREEMPT_RT

A project called the real-time Linux collaborative project was publicly announced in 2015. The project has been working continuously on a kernel patch called PREEMPT_RT which modifies the kernel to be more preemptive. Paul McKenney describes what the aim of PREEMPT_RT is, in this quote below [20].

> "The key point of the PREEMPT_RT patch is to minimize the amount of kernel code that is non-preemptible"

The project is using knowledge from existing RTOSs and has been releasing stable versions since kernel version v2.6.11. The PREEMPT_RT is used for real-time tasks to reach lower latencies and jitter. This is achieved by modifying the Linux kernel to be as preemptive as possible, some of the approaches are described below.

Spinlocks
: The kernel uses spinlocks to ensure that only one thread at a time has access to a certain section. They are chosen instead of mutexes because of being simpler and faster [21]. But spinlocks were regarded as a performance bottleneck and PREEMPT_RT, therefore, converted a majority of spinlocks to rt_mutexes instead.

rt_mutex
: PREEMPT_RT replaces all mutexes with rt_mutexes instead. The key difference is that rt_mutex has implemented priority inheritance in order to avoid priority inversion [22].

2.3.5 SCHED_DEADLINE

SCHED_DEADLINE, as the name suggests, is a Linux scheduler with deadline-oriented scheduling [9]. It was first developed because Linux schedulers did not have timing constraints in mind. SCHED_DEADLINE is an implementation of the EDF, and therefore was not a priority-fixed scheduler SCHED_DEADLINE was the first of its kind on Linux [10]. SCHED_DEADLINE has been included to the Linux kernel since the version 3.14 release in the year 2014 [10].

2.3.6 LITMUS^{rt}

The main goal and focus of Linux Test bed for Multiprocessor Scheduling in Real-Time (LITMUS^{rt}) [23], [24] is to provide an experimental platform for real-time system research. LITMUS^{rt} provides with abstractions and interfaces within the kernel, which simplifies further modifications on the kernel compared to an unmodified kernel. LITMUS^{rt} has been modifying the kernel to support their sporadic task model, modular scheduler plugins, and reservation-based scheduling. LITMUS^{rt} has also implemented support for clustered, partitioned, global schedulers, and semi-portioned scheduling.

LITMUS^{rt} has been intended to serve as a proof of concept for the predictability of multiprocessor scheduling on existing hardware. LITMUS^{rt} provides with an API but is not considered to be stable, meaning that implementations can change between releases without any warnings [25].

2.4 Xenomai's Architecture

Xenomai is today a free real-time framework for Linux. Xenomai is providing various emulated RTOS APIs to be used on Linux distributions. Xenomai mimics the functionality of the emulated RTOS APIs, thus preserving the real-time guarantees provided from the APIs. Xenomai can be used in two configurations called Cobalt core and Mercury core. Depending on the purpose for real-time one is preferred over the other. Xenomai's architecture can be viewed below in Figure 2.4.

2.4.1 Interrupt Pipeline

In order for Xenomai to keep latencies predictable the Linux kernel must be blocked from directly handling interrupts. The interrupt must instead be redirected to go first through Xenomai and then the Linux kernel. This is achieved by having a microkernel between the hardware, Linux, and Xenomai. The microkernel acts as a virtual programmable interrupt controller, separating interrupt masks between Linux and Xenomai. This microkernel is called Interrupt

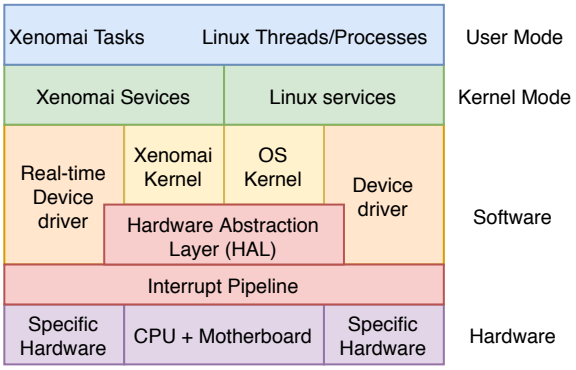


Figure 2.4: Xenomai's Architecture

Pipeline (I-Pipe). I-Pipe is based on the ADEOS microkernel described in section 2.3.2, but is stripped down to the bone to be as simple as possible, and to only handle interrupts. I-Pipe organizes the system into two domains, where the Xenomai domain has higher priority than the Linux domain. The two domains share an address space which allows threads from Xenomai to invoke services in the Linux kernel domain. Interrupts are being dispatched by I-Pipe in domain priority order. Xenomai is set as the highest prioritized domain and will receive the interrupts first. The I-Pipe is illustrated below in Figure 2.5.

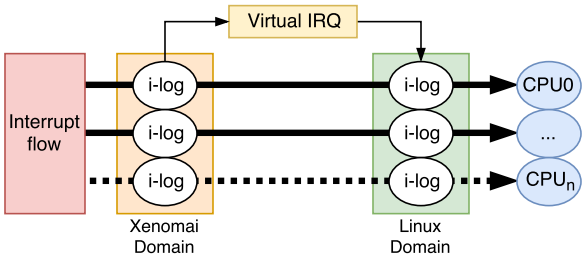


Figure 2.5: The interrupt pipeline, inspired by [26, Figure 13-2]

Figure 2.5 shows a virtual Interrupt Request (IRQ) which has the ability to lock out certain interrupts for other domains when needed to. I-Pipe replaces the hardware interrupt masks with a multiple of virtual masks. The virtual masks are then used such that domain's that use the same interrupt mask will not be affected by another domains action. I-Pipe uses an architecture-neutral API. The API has been ported to a variety of CPUs which simplifies the Xenomai porting, as much less architecture dependent code needs to be implemented.

2.4.2 Cores

The Xenomai core supplies all the necessary resources that the Xenomai skins (section 2.4.4) requires in order to mimic existing RTOS API. Real-time processes on the system need only to use the Xenomai services in order to keep its necessary predictabilities [26, P. 371].

Xenomai is now divided into two different approaches: Cobalt core and Mercury core. The cores have significant differences and are used depending on what the developer wants and need.

Cobalt Core

The Cobalt core is the dual-kernel approach and requires I-Pipe. Together with I-Pipe is Cobalt built into the system to handle all the time-critical parts of the system, such as interrupt handling and the scheduling of real-time tasks. A visual overview of Cobalt's configuration can be viewed below in Figure 2.6.

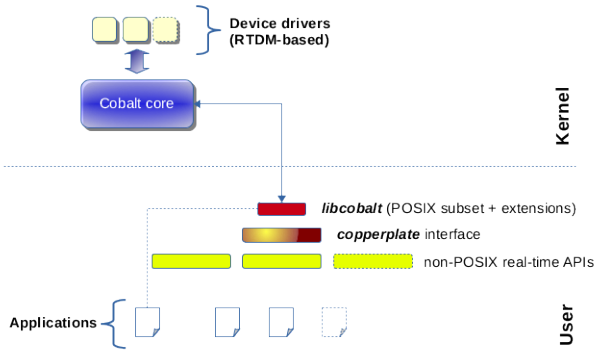


Figure 2.6: Xenomai's Cobalt core configuration, taken from [27, Figure 1]

The Cobalt core is only recommended to be used when the system is using a maximum of four CPUs for real-time tasks. If the system is going to use more CPUs however, it is recommended to use the Mercury core instead [27]. The Cobalt is dependent on being supported by the hardware as all the drivers need to be modified in order for Cobalt to be functioning.

Mercury Core

The Mercury core is a single-kernel approach, which basically is the Xenomai API, running on a GPOS without modifying the kernel. This configuration can be used when the Cobalt core is not supported by the hardware. It is recommended however to install PREEMPT_RT on the kernel when Mercury

core is chosen as PREEMPT_RT improves the latencies. The configuration of a Mercury core can be viewed below in Figure 2.7.

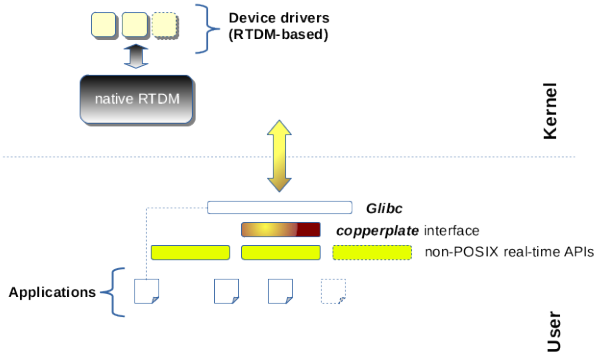


Figure 2.7: Xenomai's Mercury core configuration, taken from [27, Figure 2]

2.4.3 Real-Time Driver Model

Xenomai has been using Real-Time Driver Model (RTDM) early within its development. RTDM is a common framework for developing device drivers for real-time. RTDM started being developed when dual-kernel approaches such as RTLinux and RTAI became known, as the approaches required new drivers for real-time [28]. When a real-time application needs to access a certain driver, RTDM acts as a mediator between the application and the device driver.

2.4.4 Skins

Xenomai provide a multiple of skins. A skin is an impersonation of an existing RTOS's API, such as VxWorks [29], Portable Software On Silicon (pSOS), and more. The skin's purpose is to allow the application developer to choose an impersonation which they are most comfortable with. Skins such as Alchemy or Portable Operating System Interface (POSIX) are also available, but are instead combinations of various of traditional RTOS APIs.

2.5 Raspberry Pi

RPi [30] is a Single-Board Computer (SBC) series, which are a popular choice for embedded projects. They are being developed by the RPi Foundation. The first RPi, model B was released in the year 2012 and became a success. Since then,

several different models of RPi have been developed and released. A comparison of all the latest models can be seen below in section 2.5.1.

2.5.1 Hardware Description

The current RPi models have different specifications, but some are shared, overall details can be seen in Table 2.1 below. All RPi models use an Advanced RISC Machine (ARM) architecture. The System on a Chip (SoC) and the architecture of the later chosen RPi will be of importance.

Table 2.1: Raspberry Pi hardware specifications

Model	SoC	Architecture	CPU
RPi Zero	Broadcom BCM2835	ARMv6Z	1 GHz 1-core ARM 1176JZF-S
RPi Zero W	Broadcom BCM2835	ARMv6Z	1 GHz 1-core ARM 1176JZF-S
RPi 1 A+	Broadcom BCM2835	ARMv6Z	700 MHz 1-core ARM 1176JZF-S
RPi 1 B+	Broadcom BCM2835	ARMv6Z	700 MHz 1-core ARM 1176JZF-S
RPi 2 B	Broadcom BCM2836	ARMv7-a	900 MHz 4-core ARM Cortex-A7
RPi 3 B	Broadcom BCM2837	ARMv8-a	1.2 GHz 64-bit 4-core ARM Cortex-A53

General-Purpose Input/Output

All of the RPi models have GPIO pins which allow sensors, actuators, and much more to be communicated with the RPis.

2.5.2 Raspberry Pi 3: Hardware Specification

The RPi which will be used is RPi 3 and the rest of hardware specifications will therefore only consider RPi 3. As shown earlier in Table 2.1, RPi 3 uses the SoC BCM2837. The SoC BCM2837 in RPi 3 is nearly identical to the BCM2836 which was used in the original RPi 2. The most important difference is that RPi 3 uses a quad-core ARM Cortex-A53 processor which replaced the quad-core Cortex-A7 on RPi 2. The cores run at maximum 1.2GHz without overclocking, which is approximately 50% faster than RPi 2. The ARM Cortex-A53 processor is considered a mid-range, and low-power processor with ARMv8-A architecture [31].

ARM Architecture

The architecture on Cortex-A53 is an implementation of armv8-A. ARMv8-A has several modes of operation. An operation mode is also called a processor mode or a processor state. The states are:

AArch64 Cortex-A53 supports the AArch64, which is the ARM 64-bit execution state. It also supports the A64 ISA (also called armv8-a) needed for AArch64.

AArch32 Cortex-A53 also supports AArch32 which is the corresponding 32-bit execution state, and includes its A32 ISA (also called armv7-a). A32 was previously called ARM ISA

Thumb instruction set Now called T32. T32 is a subset of the A32 instruction set where each instruction is instead of length 16-bit. T32 has also, for every 16-bit instruction a corresponding 32-bit instruction.

Exception levels Cortex-A53 has support for exception level EL0 to EL3. The number corresponds to the software execution privilege where 0 is the lowest (unprivileged) and 3 is the highest. EL2 provides support for processor virtualization and El3 provides support for security states. The exception levels are supported in AArch64 and AArch32.

Pipeline

Each core on the Cortex-A53 processor uses an in-order pipeline. In-order pipelines are executing the instructions it receives in the same order as they were written, as opposed to out-of-order pipelines. This increases the predictability on the microarchitecture level [11]. The pipelines are also dual-issued for the majority of instructions, meaning that the pipelines can execute certain instructions simultaneously in pairs.

Cache

Each core has its own level 1 cache with size 32KB, and share a level 2 cache with size 512KB [32]. According to the ARM Cortex-A53s datasheet [31], the instruction and data are in separated caches. It is therefore uncertain if [32] are referring to level 1 cache of 32KB being the sum of both instruction cache and data cache or not. A pessimistic assumption would be that the instruction cache and data cache is of size 16KB each. See Figure 2.8 below for a visual description of the cache topology. See Table 2.2 below for gathered specifications for the caches.

Table 2.2: Cache information

Cache Level	Data/Instruction	Line size	Set associative	Replacement policy	Size
1	Instruction	64 bytes	2-way	Pseudo-random	16KB
1	Data	64 bytes	4-way	Pseudo-random	16KB
2	Both	64 bytes	16-way	Not found	512KB

As ARM Cortex-A53 has multiple cores, it needs a data cache coherence protocol in order to avoid data corruption when cores share data. The protocol used is

called Modified Owned Exclusive Shared Invalid (MOESI) [31], where each word in the name describes the state a shareable cache line can be in.

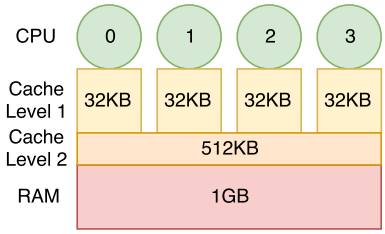


Figure 2.8: Cache topology for Raspberry Pi 3

2.5.3 Raspbian OS

The OS that the RPi Foundation officially support is Raspbian [33]. Raspbian is based on the Debian Linux distribution but is modified to run as smoothly as possible for the RPi devices. Raspbian is currently only executing in 32-bit, even on RPi 3 which has a 64-bit CPU. This limits the available ISAs on RPi 3 as armv8-a cannot be executed in a 32-bit environment [31]. Successful attempts at running Raspbian in 64-bit have been made but is not yet considered stable enough for an official release [34].

The RPi kernel has it own kernel Git repository forked from the mainline kernel. The repository provides with e.g. heavy modified Universal Serial Bus (USB) drivers and other modifications especially for the hardware on the RPis [35].

2.6 Related Work

Related work has been researched. They have different purposes and different approaches, however, their area has been related in terms of either real-time, GPOS and Linux.

2.6.1 Linux and Real-Time: Current Approaches and Future opportunities

The authors Claudio Scordino and Giuseppe Lipari have provided an article describing current approaches and future opportunities regarding real-time for Linux [18]. Even though the article can be considered outdated today, it introduced the area around the approaches such as interrupt abstraction and kernel modifications which are still used today.

2.6.2 An Embedded Multi-Core Platform for Mixed-Criticality Systems

The author Youssef Zaki has written a master thesis about his study and analysis of virtualization techniques for mixed-criticality systems [36]. Mixed-criticality systems are systems which combine tasks of different criticality/importance on the same computing platform. It is most important that the lower critical tasks would not interrupt or degrade the performance of the higher critical tasks. This is achieved by system isolation and virtualization with Multiple OSs. Although virtualization was not a sought out solution, the report provided a good insight on approaches to virtualization.

2.6.3 Real-time Audio Processing for an Embedded Linux System Using a Dual-Kernel Approach

Nitin Kulkarni was working on audio processing with soft real-time requirements using a dual-kernel approach [37]. The dual-kernel approach was by using Xenomai with Cobalt on an Intel Atom board [38, P. 41]. The conclusions from the analysis done by the author state that the overall responsiveness of the system was improved and that Xenomai can indeed be used for hard real-time applications [37, P. 64].

2.6.4 Finding Strategies for Executing Ada-Code in Real-Time on Linux Using an Embedded Computer

The author Adam Lundström has written a study on approaches for executing Ada-code in Real-Time Linux [39]. The author chose PREEMPT_RT as the most optimal approach for the purpose but describes that RTAI and Xenomai could also be promising solutions. Xenomai and RTAI provided with better performance in terms of latencies but lacked support for Ada [39]. The lack of Ada support for Xenomai and RTAI was one of the main argument to be using PREEMPT_RT instead. Adam Lundström also states through [40] that PREEMPT_RT is not able to guarantee hard real-time because of its architecture [39].

2.7 Summary

The conclusions based on the research in the background and the project description (Section 1.1) deemed that a system using Xenomai Cobalt could be an appropriate solution. The system will be running on a RPi 3 with the Raspbian GPOS together with Xenomai Cobalt and the implementation will be using Xenomai's API.

Measuring the predictability of a real-time system is a very complicated process, and with an entire GPOS, sharing resources with the real-time parts complicate the process further. Because of complexity and time constraints, the thesis will analyze the predictability of the system on response jitter to external stimuli, recognized via means of interrupts.

This chapter has been given details on the terms, topics, and areas for which the thesis is based on. Based on the topics and areas in the literature, combined with the project description has a conclusion been made on how the rest of the thesis should be continued. The final conclusion was to use Xenomai Cobalt on RPi 3 in order to provide hard real-time characteristics on the Raspbian OS.

3 Method

This chapter describes the research method in more details. The Xenomai section describes how the installation process will is, and what is achieved by it. The application section describes what needs to be implemented. The data collection describes how the measurements will be done and why for the coming analysis.

3.1 Xenomai Cobalt

Xenomai with Cobalt core is used as it has potential of providing real-time properties without removing the features available in the Linux domain.

3.1.1 Installation Process

The installation process follows the official installation guide for Xenomai 3.x [41]. RPi 3 is supported according to the hardware section of the webpage. RPi 3 is shown in section: "Dual kernel Configuration → ARM → Supported Evaluation Boards". The installation process might still encounter problems because of the lack of information on the webpage. Some developers have attempted this installation process, and luckily, they provided guides on how they succeeded [42] [43].

3.1.2 Backup Plan

The installation process could encounter problems so severe that it could hinder the development of this thesis. A backup plan would be needed, should that happen. The backup plan is to use an image provided by [44]. The image contains the Raspbian OS with an already installed Xenomai for RPi model zero, 1, 2, and 3.

3.1.3 Xenomai API

As mentioned before, Xenomai offers an API which contains everything needed in order for the application to be implemented. The API is well documented [45] and many examples can be viewed online, for example, [44].

3.2 Application

In this section the overall planned structure of the application is described, with motivations of why it was implemented as such.

Xenomai's API is used as much as possible during the implementation, as using the generic API and libraries available in Linux can create mode switching, meaning that; the application may switch from the primary domain to the secondary domain and vice versa. This may, consequently, cause the application temporary to be executed in the Linux domain, and thus causing unpredictable latencies.

3.2.1 Description

The application needs to read and write using the GPIO pins on the RPi 3 with hard real-time requirements. This means that tasks must never miss their deadlines. The aim is to determine the rate which describes how accurate the GPIO pins can be written and read. For example, if a rate of 10 kHz is found, then the application will always be able to write and read a pin within 100 µs.

The application is developed to read a Comma Separated Values (CSV) file which describes when the application should write to the GPIO pins and how long each high signal should be. Reading files causes a switch to the secondary domain, meaning that files will be read and stored before the application requires real-time execution.

Subsequently, the application will handle a limited amount of data and thus can avoid dynamic allocations. The application will initialize everything needed before any real-time requirements are set. This way, there will not be any background tasks running within the application, and it can instead focus on the real-time measurements. A simple flowchart for the application is illustrated in Figure 3.1 below.

When the application seems to be working as described, a trial and error method will be used to increase the rate. Ways, such as creating CPU sets and isolating CPUs, are considered. The CPU set is a way of having a hierarchy describing which resources processes are allowed to use, for example restricting processes from using certain CPU cores. Isolating CPUs refers to the kernel boot option isolcpus. Isolcpus remove specified CPU cores from the kernel scheduler.

3.2.2 Testing

The application will be tested through the whole implementation using an oscilloscope. Measurements from the oscilloscope are compared with the configuration in the application. The test is considered to be successful if the measure-

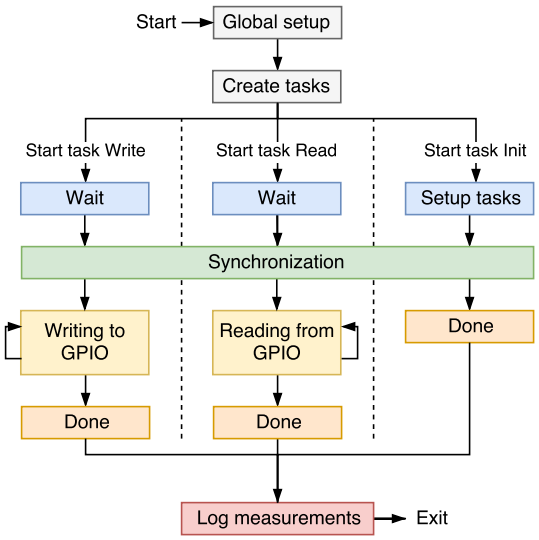


Figure 3.1: The flow of the application

ment differences are lower than the stated rate. During the tests, the system will be under different kind of loads using stress tests (see section 3.3.4). This is to ensure that the real-time requirement will still be met independently from the situation of the rest of the system. Scripts will also be written in order to automate the testing procedure, and thus reducing test time.

3.3 Data Collection

This section describes how the data collection procedure will be done, what it requires and finally how the data will be analyzed.

3.3.1 Measurement Equipment

Two hardware instruments are needed for the data collection:

Oscilloscope: Tektronix MSO2000B series
The oscilloscope has a bandwidth up to 200 MHz and a sampling rate of 1GS/s. It is used as a reference point for the application as it is highly accurate and reliable compared to the RPi 3.

Function generator: Agilent 33120A
The function generator can output frequencies up to 15 MHz, which is more than enough, as the requirement is set to 10 kHz.

3.3.2 Automation

As mentioned earlier in section 3.2.2, attempts on making the test and measurement automatic are made. It is common that instruments such as oscilloscopes or function generators use the standard Virtual Instrument Software Architecture (VISA) [46]. VISA is a standard for configuring, programming or troubleshooting instruments through communication interfaces, for example, General-Purpose Interface Bus (GPIB), Serial, Ethernet, and USB interfaces.

The VISA standard has been implemented as a Python package called PyVISA [47], meaning that communication can be done with Python. The Python community is vast, open and offers a huge amount of different utilities, making the automation process easier, hence Python was chosen for the automation process.

In order to understand the coming automation steps easier, see Figure 3.2 below for an illustrated description of the measurement setup.

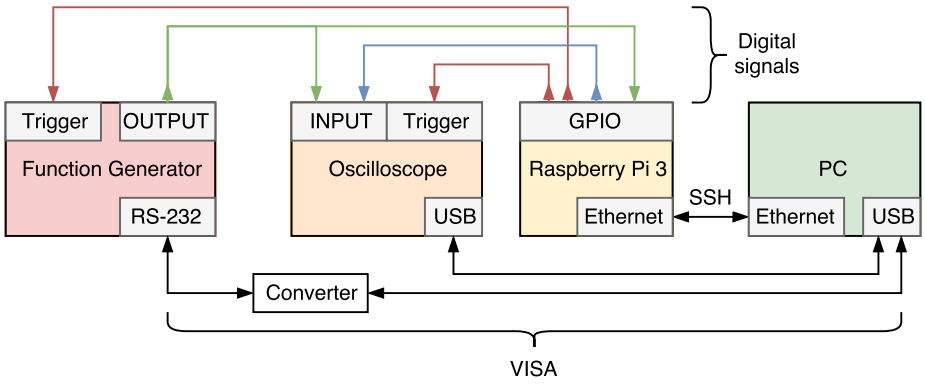


Figure 3.2: Simplistic structure of the measurement setup

As seen in Figure 3.2, a PC is used to communicate with the function generator and the oscilloscope using the VISA protocol. The PC communicates with the RPi through Secure Shell (SSH). The RPi is sending triggers to the function generator and the oscilloscope using its GPIO pins. The function generator and the RPis GPIO pins are connected to the oscilloscopes input channels.

The steps which need to be automated are to:

1. setup the oscilloscope: Configure the channels, the window and make the oscilloscope waits for a trigger.
2. setup the function generator: Configure the function, its amplitude, offset and more. Make the function generator wait for a trigger.
3. generate CSV file for application: The CSV file will be generated with value range chosen pseudo-randomly.

4. execute the application with a specified CSV file. The oscilloscope will be triggered when it receives a high signal from the application.
5. extract data from the oscilloscope and compare it to the application's data.

3.3.3 Latency Test

Latency test is a measurement tool common to use as a reference for how predictable a system can be. The latency test has a periodic task. As soon as the task is awakened, it compares the system time with when it was supposed to be awoken. The difference in these values is the latency. The latency depends on many things, such as hardware, OS, priorities, the architecture, and much more. The latency can also be described as scheduling latency. An interrupt latency has been described before, see Figure 2.1.

Figure 3.3 below illustrates the latency described as release time. The release time varies depending on the scheduler interrupt latency. The response time denotes how long time it takes for the task to finish after being released. Xenomai's API comes with a latency test application by default.

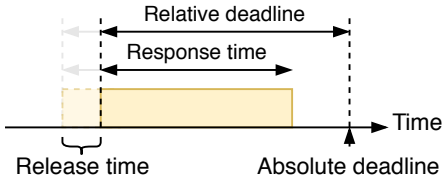


Figure 3.3: Latency model

3.3.4 Stress Test

During each measurement the system will be under stress by applications not considered as real-time applications. This is done to simulate different loads that the system might be having while responding to the real-time stimuli. Stress-NG [48] was chosen as a stress application. The motivation was that Stress-NG has many features such as: cache-thrashing, I/O-syncs, context-switching, and more.

3.3.5 Analysis

The application writes to a GPIO pin in specified points in time. The oscilloscope will then measure the signals. The measurement data from the oscillo-

scope will be compared with when the application was supposed to write to the GPIO pin. See Figure 3.4 below for an illustrated example.

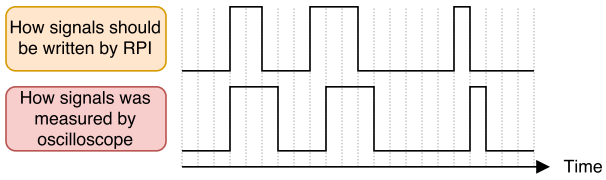


Figure 3.4: A measurement example for writing signals

The application also measures reading signals from a GPIO pin. The signals are sent from the function generator which is also connected to the oscilloscope. The measurement data from the oscilloscope will be compared with the applications measurement data. See Figure 3.5 below for an illustrated example.

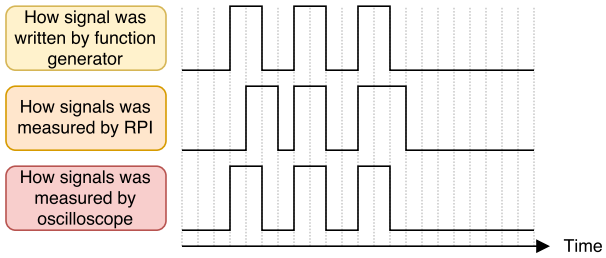


Figure 3.5: A measurement example for reading signals

The comparison of the data mentioned will show the minimum, average, and maximum latencies measured. The result represents how accurate the RPi 3 can write and read signals using the GPIO. See Table 3.1 below for a simplistic example on resulting data.

Table 3.1: Example showing time difference from application and oscilloscope

#Samples	Min	Average	Max
5.00×10^3	24.00 ns	7.46 µs	42.90 µs

If the data is rarely close to either the minimum or maximum data, it can be useful to plot the data using box plots. The box plot will tell where the majority of data will end up.

Sample Size

A sample is in this case defined as the measured difference between RPi and the oscilloscope. The samples will most probably vary for each test, therefore

it is necessary to decide on how many samples which will be needed in order to have reliable results. This decision is dependent on how the standard deviation changes with the number of samples. If the standard deviation reaches a constant value at a certain number of samples, then it is reasonable to limit the number of samples to that number. The number might get lowered further, however, because of time constraints.

The standard deviation for reading and writing GPIO pins are shown below in Figure 3.6 and Figure 3.7.

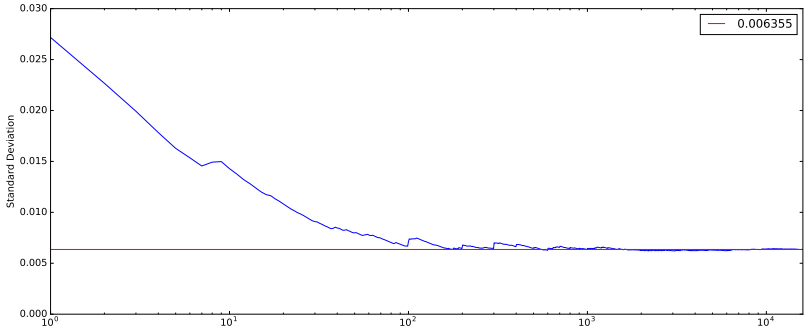


Figure 3.6: Standard deviation per number of samples for reading

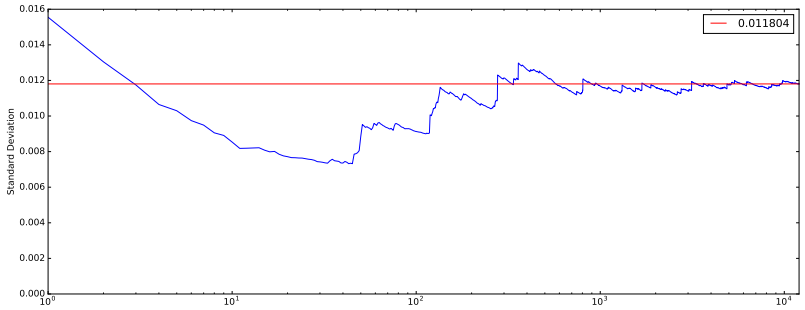


Figure 3.7: Standard deviation per number of samples for writing

What can be seen from Figure 3.6 and 3.7 is that the standard deviation relatively high for the first 100 samples. The standard deviation then stabilizes after 1.00×10^3 samples. A reasonable decision on the sample size is therefore between 1.00×10^3 and 10.00×10^3 samples. During the data collection, the standard deviation will always be in consideration, as the result may vary depending on changes in the system.

3.4 Summary

The planned Xenomai installation process has been described, as well as the backup plan in case of an unsuccessful installation.

The application implementation for the RPi has been described, including methods to possibly improve its rate. The data collection process has been described to be automated, with a description of acceptable sample sizes.

4 Xenomai Installation & Application

This chapter explains first the Xenomai installation on RPi 3. The chapter continues with the application implementation needed to fulfill the requirements in section 1.2.

4.1 Xenomai Installation

In order to install Xenomai Cobalt on the RPi 3, a few steps were needed.. A kernel was needed to be decided. There exist a few options for RPi 3. The RPi foundation has developed its own kernel using a forked version of the mainline kernel (Torvalds Linux kernel). They have modified their kernel especially for the RPi models, which could be a good argument on why the RPi kernel should be chosen. The problem with the RPi kernel is that Xenomai does not focus their development on separated kernels but instead only on the mainline kernel.

Xenomai has released support for the RPi kernel for kernel version 4.1 on a forked repository. The last commit on that repository, however, was 14th February 2017, and it is now considered outdated.

Xenomai has support for the mainline kernel up to version 4.9 and is also more up to date compared to the other options. This is one of the reasons why the mainline kernel was decided to be used. The other option has been attempted but without success. Details on the unsuccessful attempt for Xenomai on the RPi kernel can be seen in Appendix A.

The kernel version was needed to be decided as well. It was important that the kernel version matches the I-Pipe version, otherwise the differences could cause the kernel to be not functioning as intended, or simply cannot be built. I-Pipe was going to be acquired as a patch, which simplifies the merge with the kernel.

A patch is a file created using git diff, which essentially shows all the differences between latest commit and unstaged files. The patches can then be applied on different repositories.

Both the kernel and the Xenomai API was needed to be built, which was done by using a cross-compiler. A cross-compiler is essentially an application which translates the source code into machine code for a different architecture. Cross-compilers are usually used for embedded systems in order to reduce compilation time. The host computer which was used for the entire Xenomai installation was running Ubuntu 16.04 with a 64-bit OS.

4.1.1 Preparation

The parts needed were: the kernel source, a cross-compiler, the Xenomai source, and the I-Pipe patch.

Acquiring I-Pipe

As mentioned earlier, the kernel and the I-Pipe patch need to have the same version or at least be as close as possible. The available releases of I-Pipe can be acquired from the Xenomai's download page [49]. The latest release for ARM architecture was 4.9.51. The I-Pipe patch was downloaded with the command:

```
~/$ wget http://xenomai.org/downloads/ipipe/v4.x/arm/ipipe-core
    ↪ -4.9.51-arm-4.patch
```

Acquiring Kernel

The mainline kernel source was then acquired by using the command:

```
~/$ git clone https://git.kernel.org/pub/scm/linux/kernel/git/
    ↪ stable/linux-stable.git ~/linux
```

The latest version on the kernel source tree was different from the sought out version. It was therefore needed to change it into 4.9.51. It was done by first finding the kernel version using "git tag". The kernel version was then changed to 4.9.51 by using the command:

```
~/$ git checkout v.4.9.51
```

Acquiring Cross-Compiler

The chosen cross-compiler was from RPi own kernel building guide [50]. The cross-compiler was built especially for RPi models. It was downloaded by the command:

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

To make it easier to specify the cross-compiler, the $PATH environment was updated using the commands below:

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

If a host computer would be using 32-bit OS instead, a 32-bit cross-compiler would be needed. A 32-bit cross-compiler was found located in the same repository in path:

```
~/tools/arm-bcm2708/gcc-linaro-arm-linux-gnueabihf-raspbian/bin
```

Acquiring Xenomai Source

The Xenomai source could be found on Xenomai's git repository webpage [51]. The latest Xenomai version was 3.x and acquired with the command:

```
~/$ git clone http://git.xenomai.org/xenomai-3.git/ ~/xenomai
```

4.1.2 Installing Cobalt Core

Next step was to continue with the Xenomai installation. The steps for Cobalt core and Mercury core are similar but the Cobalt core has slightly more configurations because the kernel needs to be modified.

Applying I-Pipe and Xenomai to the Kernel

The next step was to apply I-Pipe and Xenomai to the kernel. I-Pipe could be applied to the kernel separately or with a script included in the Xenomai repository. A "dry-run" could be applied when patching the I-Pipe, in order to make sure that it will be applied as intended. This was done by the command:

```
~/linux$ patch -p1 --dry-run < ~/ipipe-core-4.9.51-arm-4.patch
```

Everything patched successfully, meaning that the Xenomai script could be used without any problems.

```
~/xenomai$ ./scripts/prepere-kernel.sh --linux=~/linux --arch=arm
    ↪  --ipipe=~/ipipe-core-4.9.51-arm-4.patch
```

Configuring The Kernel

The kernel source tree needs a configuration file specifying the hardware and what features that should be included. A configuration could be acquired from an already running RPi provided that the kernel it uses is the same version or close enough. However, the already running RPi was using the RPi kernel and not the mainline, so a different approach was chosen instead. A configuration could be chosen from one of the default configurations available in the kernel source tree. The configuration file for RPi 3 was called "multi_v7_defconfig".

Before specifying the kernel configuration, the architecture and cross-compiler must be specified for the kernel source tree. This was done by the commands:

```
~/linux$ export ARCH=arm
~/linux$ export CROSS_COMPILE=arm-linux-gnueabihf-
```

The configuration file was then set by the command:

```
~\linux$ make multi_v7_defconfig
```

Editing the configuration file was done by using menuconfig:

```
~/linux$ make menuconfig
```

The default configuration was made to fit multiple of different systems. This condition was avoided by deselecting all systems which were not for RPi 3, as shown in Figure 4.1 below.

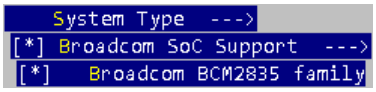


Figure 4.1: Kernel configuration: System Type

Then the GPIO device driver was added to Xenomai as a loadable module, see Figure 4.2 below.

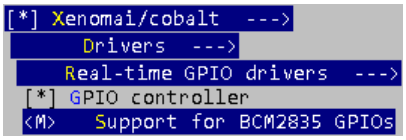


Figure 4.2: Kernel configuration: Xenomai GPIO

Some features were needed to be edited in order for Xenomai to work properly.

CONFIG_CPU_FREQ This option allows the CPU frequency to be modulated depending on the workload. This feature adds unpredictability and was therefore disabled.

CONFIG_CPU_IDLE This feature allows the CPU to enter sleep states. As it takes time for the CPU to be awoken, it adds latencies and unpredictability. The feature could also cause timers for Xenomai to stop working. It was therefore decided to disable this feature.

CONFIG_KGDB Is a kernel debugger which could only be enabled for X86 architectures. As RPi 3 is an ARM architecture, it was necessary to disable it.

CONFIG_CONTEXT_TRACKING_FORCE This option is automatically disabled with the I-Pipe patch.

These configurations can be viewed below in Figure 4.3.

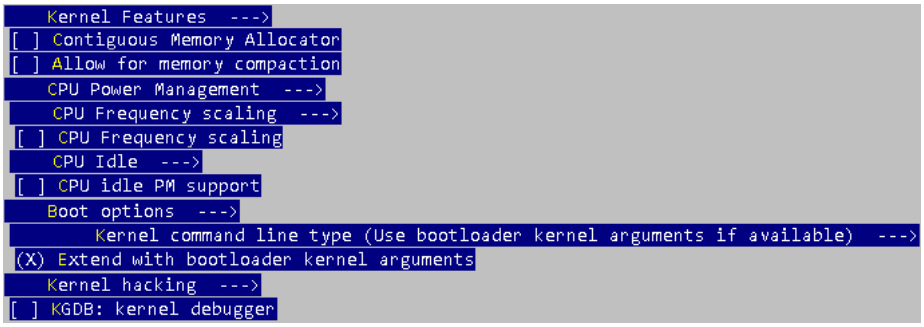


Figure 4.3: Kernel configuration: Xenomai

Installing the Kernel

When the configurations were done, the kernel could finally be built. This is done by the command:

```
~/linux$ make zImage modules dtbs -j12
```

The next step was to install the kernel in a temporary location. The Raspbian OS was already installed on the SD card and was mounted in path "/mnt/rpi. The mount contained two partitions: boot and rootfs. The boot partition handles only the boot sequence (which selects the kernel and its arguments). Rootfs is the other partition which contains the rest of the entire OS. A temporary location was created with the same structure as the SD card. Everything was going to be installed to that location, and when finished, the temporary files would finally be copied to the SD card.

The parts needed to be installed were the kernel image, modules, and Device Tree Blobs (DTBS). These were installed with the commands:

```
~/linux$ export INSTALL_MOD_PATH=~/tmp/rootfs
~/linux$ make modules_install
~/linux$ cp arch/arm/boot/dts/bcm*.dtb ~/tmp/boot
~/linux$ cp arch/arm/boot/zImage ~/tmp/boot/kernel7.img
```

Then the last step regarding the kernel was to set which device tree to be used. The device tree was given by Xenomai, which was called "bcm-2837-rpi-b-cobalt.dtb". It was added to "config.txt" inside the SD card boot partition. The device tree was set by writing "device_tree=bcm2837-rpi-b-cobalt.dtb".

The kernel was then finally installed with Xenomai Cobalt. The next step was to install the Xenomai API so that real-time applications could be developed and used.

4.1.3 Installing Xenomai API

The Xenomai source was acquired through Git, and because of this, a configuration script was needed to be executed in order to generate the necessary Makefiles. The automatic configuration script was executed by the command:

```
~/xenomai$ ./scripts/bootstrap
```

The next step required a build directory and a staging directory. The build was needed to store the files that the succeeding command was going to generate. The staging directory would then store the installed files temporary before moving them to the final location. The chosen directory was ~/tmp/rootfs from the previous section.

It was required to generate an installation configuration for the specific chosen platform. This was done by the commands:

```
~/$ export CFLAGS="-march=armv7-a␣-mfloat-abi=hard␣-mfpu=neon␣-
    ↪ ffast-math"
~/$ export LDFLAGS="-march=armv7-a␣-mfloat-abi=hard␣-mfpu=neon␣-
    ↪ ffast-math"
~/$ export DESTDIR=~/tmp/rootfs
~/xenomai/build$ ../configure --enable-smp --host=arm-linux-
    ↪ gnueabihf --with-core=cobalt
```

The arguments in the scripts describe the architecture on the platform. The OS used was Raspbian, which is a 32-bit OS, meaning that ISA armv7-a or below is necessary. The floating-type hardware is of type neon. The floating-point convention was chosen to be hard, meaning that the code will be transformed into instructions specific to the Floating Point Unit (FPU) RPi 3 uses. Finally, math algorithms were optimized for speed by stating the last command.

When the configuration step was done, the installation of the API could finally start. The installation was done by using the command:

```
~/xenomai/build$ make install
```

The last step for the Xenomai installation was then to copy everything to the SD card. This was done with the command:

```
~/tmp$ sudo cp -r * /mnt/rpi/ --verbose
```

The Xenomai installation was complete. The final step was to run it on the RPi 3. The path to the Xenomai API was needed to be added to ldconfig, this was done by creating a file called /etc/ld.so.conf.d/xenomai.conf which contained the path to the Xenomai library. When that was done, ldconfig could set up all library paths with the command:

```
RPI:~/$ sudo ldconfig --verbose
```

4.1.4 Xenomai Installation Summary

Various problems were encountered during the Xenomai installation, such as patching issues, compilation errors, etc. Fixing the issues required time without any guarantees of success. As the problems could not be solved within the planned time frame, it was decided to use the backup plan mentioned in section 3.1.2. There were some downsides with the backup, however, as further configurations in the kernel could not be done. The purpose of further configurations was to optimize the performance of the implementation. The backup did, however, provide with everything necessary to continue further.

Later in the project, when there was extra time, the Xenomai installation was attempted again, with success. The backup was still used in the rest of the project as the Xenomai installation succeeded so late in the project. The successful attempt is described in this chapter while an unsuccessful attempt can be seen in Appendix A. The GPIO device driver worked as well with RTDM. The pins shown by the device driver was however incorrect, as they started from 400+ instead of starting from 0. Two successful workarounds were attempted.

Increment pin number
: The simplest workaround was simply by adding the lowest pin number shown with the pin wanted when using the GPIO pin. Example: Pin 16 was acquired by specifying pin number $400 + 16 = 416$.

Modifying device driver code
: The more complex workaround was by modifying the device driver source code. The source code which was modified by Xenomai was being compared with the source code from the RPi kernel source tree. The most significant differences were merged and the kernel was recompiled. The device driver then showed the correct pin numbers.

4.2 Application

The application is described in detail in this section. The application was developed using C++ as it is easy to continue with the development for future projects. The skins used by Xenomai's API were RTDM and Alchemy. RTDM was necessary as a real-time driver was needed when using the GPIO pins. A template on how to use the RTDM GPIO device driver can be seen in Appendix B.

4.2.1 Structure

The structure of the application is divided into five parts: Main, Init, Write, Read, and Logging. Init, Write, and Read is real-time tasks, while Main and

Logging are only parts of the execution process. For an illustration of the overall structure of the application, see Figure 3.1 in section 3.2.1.

Main

Main is the first part to be executed when starting the application. It started as a regular Linux process without any real-time characteristics. Main does four things: creates the real-time tasks in a specified order, waits for the real-time tasks to finish, logs the acquired data, and finally writes low to all used GPIO pins.

Init Task

The purpose of Init is to set up the necessary variables and data structures which are used by task Read and Write. In addition, Init provides synchronization between the tasks by using a counting-priority-based semaphore. For details on how this is done, see section 4.2.2. Afterward, Init waits for a specific time to write to a specific GPIO pin which is used by the Function Generator as a trigger. The time is specified by the user when starting the application.

Write Task

The purpose of Write task is to write to a given GPIO pin at specified points in time. When the task first executes after being released by Init task, it reads the current system time and stores it for future reference. The task receives an array containing when each signal should be written and how long they are supposed to be. The system time was then used with the array to determine when each signal should be written. Task Write idles between each writing to the GPIO pin. Setting a task to idle allows other tasks to be executed until the idling task is awoken. However, the task will not be awoken by the exact time specified because of latencies, described in section 3.3.3.

Task Write also synchronizes the oscilloscope with RPi by writing to an additional GPIO pin which the oscilloscope uses as a trigger. The trigger pin is written to directly before reading the system time.

Read Task

The purpose of the Read task is only to read a specified GPIO pin and then stores the system time in an array when the signal is received. The reading GPIO pin is set up as an IRQ with both rising and falling edge detection. This means that whenever an edge is detected by Xenomai IRQ handler, the handler calls the callback function. Whenever task Read calls the read function, it gets

blocked until a signal edge is detected. The current time could then be stored directly after the read function.

Logging

When both Write and Read tasks are finished, the logging begins. The logging reads the array filled by task Read and converts its values from nanoseconds to milliseconds. The logging then writes the data to a CSV file. The system calls used by Logging would cause a mode switch. However, this will not be a problem as the Logging only starts when the real-time tasks finish.

4.2.2 Execution

In order for every part to work together, a specific execution order is needed. Figure 4.4 below illustrates how the whole application is executed.

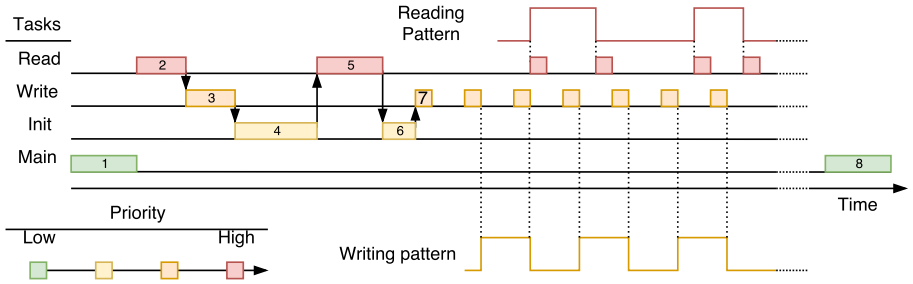


Figure 4.4: Gantt chart over the execution order

Description of every number in Figure 4.4:

1. Main starts, handles arguments, and then creates the real-time tasks.
2. Read is the first real-time task to start and then gets suspended when pending on a locked semaphore.
3. Write is the next real-time task to start and then gets suspended the same way as task Read.
4. Init is the last real-time task. When Init initialized everything necessary for task Read, it signals the semaphore. Init then gets preempted by task Read because of Init's lower priority.
5. Read then waits for GPIO IRQs.
6. Init continues and initializes everything for task Write. When Init is done, it signals the semaphore again, which allows task Write to continue.

7. Write timestamps the current time and starts with the writing process by waiting for when the first GPIO signal should be written.
8. When task Read and task Write finish, Main saves the data acquired by task Read to a file. The Application is then done.

4.2.3 Details

In this subsection, some details of the application are described, for example, implementation decisions based on the hardware of RPi 3, and how they would affect the system's predictability.

Cores

The RPi 3 has four cores, which gives the possibility to setup the application in such way that the real-time tasks are executing concurrently on different cores. This might, however, increase the unpredictability if for example data would be unknowingly be shared between cores. This would cause the data cache coherence protocol to move the data back and forth between the cores. The real-time tasks were therefore chosen to be placed on only one core.

Scheduling

When real-time tasks are created and started with Xenomai's API, they are scheduled with a First In First Out (FIFO) queue and start in the Xenomai domain (also called primary). When a real-time task is executing in the primary mode it has higher priority than the Linux kernel. When a real-time task is scheduled to start, Xenomai preempts the Linux kernel with all conflicting running processes on the specified core. When a real-time task is making a syscall owned by the Linux kernel, it changes the domain to the secondary, owned by the Linux kernel and then gets scheduled by the Linux kernel itself. This is not the case for task Read, Write or Init as they are only using syscall owned by Xenomai.

Xenomai's scheduler uses a system timer to determine when rescheduling should occur. Xenomai does not set the system timer to be periodic but instead programs the system timer in a one-shot mode with the time of the closest coming event scheduled in the timeline. One-shot mode simply means that the system timer will only interrupt once when the time is due and then needs to be reprogrammed if another event should be scheduled.

Cache

As shown in Table 2.2, the cache line size is 64 bytes. Knowing the line size can be beneficial as it is then possible to align data to 64. When data is aligned, the data starts at a memory address which is a multiple of 64. When the data is placed on the cache, the entire line will be used, provided that the data type is big enough.

It might, however, be problematic to use caches because of the increased unpredictability that caches introduce. The unpredictability is caused by delay differences between a cache hit and cache miss. The unpredictability may be increased further because the application will compete with many other applications executing in Linux.

It is possible to reduce the number of applications executing on the same CPU by using CPU sets and isolcpus (described in section 3.2.1). This can decrease the number of cache misses because fewer applications will interfere with the same cache as the real-time application uses.

It is also possible to enable or disable the level 2 cache in attempts to get better performance. But this would cause other applications to interfere as all applications have access to the level 2 cache.

RTDM

When using the GPIO pins, RTDM is needed. If a regular Linux driver would be used to control the GPIO pins the calling task would change its primary domain to secondary. That would change the scheduler to the Linux kernel instead of Xenomai, which would not guarantee timed executions. The RTDM drivers provided with the Xenomai Cobalt have the devices located in /dev/rtdm, not to be confused with regular Linux drivers within /dev. Each GPIO pin has its own file within /dev/rtdm/pinctrl-bcm2835 and can be accessed separately.

In order for Xenomai to be able to use drivers for real-time, all drivers need their interrupts redirected to I-Pipe instead of the generic interrupt handler which the Linux kernel uses. This was done in section 4.1 either within the I-Pipe patch or the backup.

Memory Locking

Linux uses an on-demand paging scheme, meaning that memory pages will first be loaded on the primary memory when the first use occurs. This increases unpredictability as loading the page (because of a page fault) onto primary memory would cause additional latencies. A task executing in the primary domain would be forced to the secondary domain if a page fault happens.

Fortunately, calling mlockall can be used to solve the problem. Mlockall forces the entire process to be allocated on primary memory. This includes but not limited to userspace code, data, stack, shared libraries, shared memory, and memory-mapped files. The pages are guaranteed to stay in the primary memory until the memory is later unlocked.

Xenomai's API since version 2.6.3 automatically calls mlockall when starting the application. Since the Xenomai API version 3.0.X was being used, mlockall was not needed to be included manually in the application code.

Measuring

Accurate measurements must be taken whenever a signal is read and written by the application. As described before, the task Read goes into sleep whenever it is waiting for a signal to be read and then is awoken by the I-Pipe interrupt handler, which detects an edge change in the specified GPIO pin. The Accuracy of task Read is therefore dependent on how quick I-Pipe and Xenomai can schedule task Read.

As described for task Write, the task goes to sleep when waiting for the correct time to write a signal. The signal sent from task Write is then dependent on how accurately task Write can be awoken by Xenomai's scheduler. Because of task Read having higher priority than task Write and uses a GPIO interrupt, it will preempt task Write whenever a GPIO change occurs. This causes task Write's accuracy to be also dependent on how quick task Read will be.

I-Pipe uses a Time Stamp Counter (TSC) as a clock source, meaning that the TSC is used by Xenomai when time stamping. On RPi 3 the TSC is a register which increments its value every clock cycle, and each core on the hardware has its own. When the system boots, I-Pipe automatically calculates the TSC resolution. On RPi 3 the resolution was 19.20 MHz, meaning that the time stamping accuracy will not be better than 52 ns.

4.2.4 Encountered Problems

Problems were encountered during the implementation of the application. Below is a list of some encountered problems and how they were solved.

Cores Task Read and task Write were planned to be on separate cores in the beginning. Whenever a signal would be detected by the interrupt handler, task Read was awakened and could potentially delay task Write. However, forcing the tasks to be executed on separate cores by changing the CPU affinity caused the system to halt and a forced reboot was necessary. A solution to the problem was not found, and it was decided to use only one core.

System halts During testing various errors were encountered which caused the system to halt itself and required a hard reboot. Implementation mistakes such as letting a real-time task get stuck without any sleep would take up all execution time, and such application could not be canceled. The problems were considered easy to solve but sometimes were time-consuming.

4.3 Summary

This chapter described details on how the Xenomai installation was done. Complications with the Xenomai installation caused a backup plan being used. The backup was to use a distributed OS image with Xenomai version 3.0.5 already installed. The application, which was implemented, was detailed in every aspect deemed important.

5 Experimental Setup

When Xenomai was installed and the application was implemented, the focus on the experimental setup was carried out. The experimental setup focused first on how the automation part of measuring the performance of the application could be done. Secondly, the experimental part focused on gathering the data in order for the analysis to be possible.

5.1 Automation

In this section, the automation process is described in detail. It consists of the function generator, the oscilloscope, how the data was measured and gathered, the test cases, and finally how stress testing was made. The flow of the automation can be seen below in Figure 5.1.

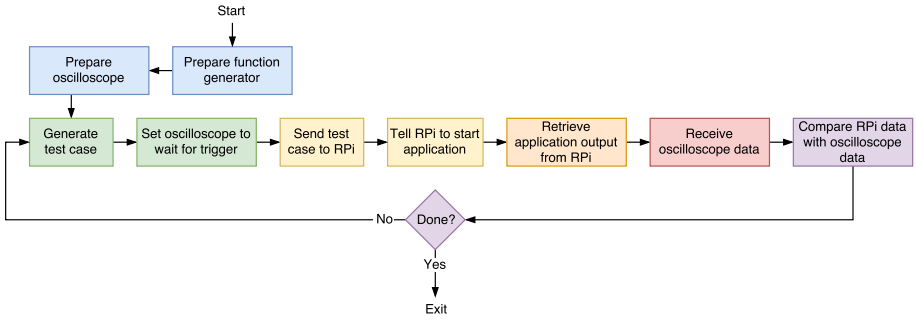


Figure 5.1: The flow of the automation process

5.1.1 Function Generator

Before the automated testing started, it was necessary to set up the function generator, how it should interact with RPi and the properties of the signals being sent. The function generator is capable of sending signals with a much higher voltage than what RPi is capable of handling, meaning a wrong setup could potentially break the RPi.

The RPi used its GPIO pins at a voltage of 3.30 V, meaning that the function generator needed to send its signal at 3.30 V as well. The function generator was set up to send signals as square waves as only digital signals were considered.

The GPIO on the RPi was using direct current, and that was set up on the function generator as well.

The square waves of the function generator have an asymmetry of $1\% + 5\,\text{ns}$. It was however not important to have a highly accurate function generator, because the measurements will be compared by the oscilloscope and RPi.

The function generator then was set up to accept a trigger signal from the RPi. When the function generator received the trigger, it would output a specified number of signals which both the RPi and oscilloscope would read. The frequency on the function generator was decided to be 10 kHz as it was required by the project requirements.

5.1.2 Oscilloscope

It was important to set up the oscilloscope correctly as it was the only device which was used as a comparison with the RPi.

Preparation

The oscilloscope had many properties which needed to be set up before the automated testing. Each channel of the oscilloscope was set up with same properties. The horizontal position on the screen of the oscilloscope was also set up. A few examples on set properties can be seen below in the list.

Coupling Was set to DC as the signals are direct current and not alternating current.

Bandwidth Was set to full, the options are either full or 20 MHz. On this particular oscilloscope was the full option 70 MHz.

Probe gain The probe gain was set up to $1X$.

The trigger of the oscilloscope was directly connected to the RPi on a separate channel and GPIO pin. The trigger was set up to detect the rising edge of the signal.

The width of the window on the oscilloscope was set at 20 ms. The window width was an important step to set up as a smaller width gave a higher accuracy while a larger width gave lower accuracy. The horizontal position sets the trigger point location. The horizontal position was changed so that it would be in the leftmost position of the window. With a window width of 20 ms, the leftmost position is $-10\,\text{ms}$ as the center is 0 ms. This was done to display as much as possible on the window without wasting any width.

The record length of the oscilloscope was also set. The options were either $125k$ or $10M$ where the $10M$ was chosen. A higher record length results in more accurate data because each measured point was closer to each other.

Receiving and Converting Data

The oscilloscope was set up to wait for a trigger before the application would start. When the application finished, data from the oscilloscope was received and converted into CSV files.

The data received from the oscilloscope was the output of the RPi and output from the function generator. The data was first converted into two arrays containing the voltage for each measured point and the time since the trigger. The data from the two arrays were then used to describe when each signal started and what lengths they had. The signal information was stored in a CSV file with the same structure as before. See Figure 5.2 below for an illustration on the conversion.

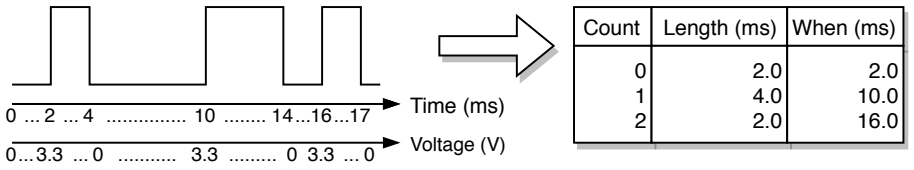


Figure 5.2: Oscilloscope waveform conversion into CSV

The largest width between each measured point was measured to be 16 ns, which is much smaller compared with the estimated accuracy of RPi. It was therefore not needed to discuss oscilloscope inaccuracies in chapter 6.

5.1.3 Test Case

A test case was a CSV file which the application will read and output to the oscilloscope as mentioned before. Each test case was pseudo-randomly generated with a number of signals, length of signals, and when signals should be sent. The number of readings the application should have done was included in each test case, as well as a pseudo-randomly chosen delay for when the trigger to the function generator should be sent. The test cases were made pseudo-randomly in order to have a larger various set of tests.

A typical test case was extracted from the oscilloscope and can be viewed in Figure 5.3 below.

Figure 5.3 shows an example of a test case on the oscilloscope. The channel 1 (yellow) to channel 4 (green) were being used for each test. Channel 1 shows the output of the RPi. Channel 2 shows the output from the function generator, which was also read by the RPi. Channel 3 was used as a trigger for the oscilloscope and channel 4 was used as a trigger for the function generator.

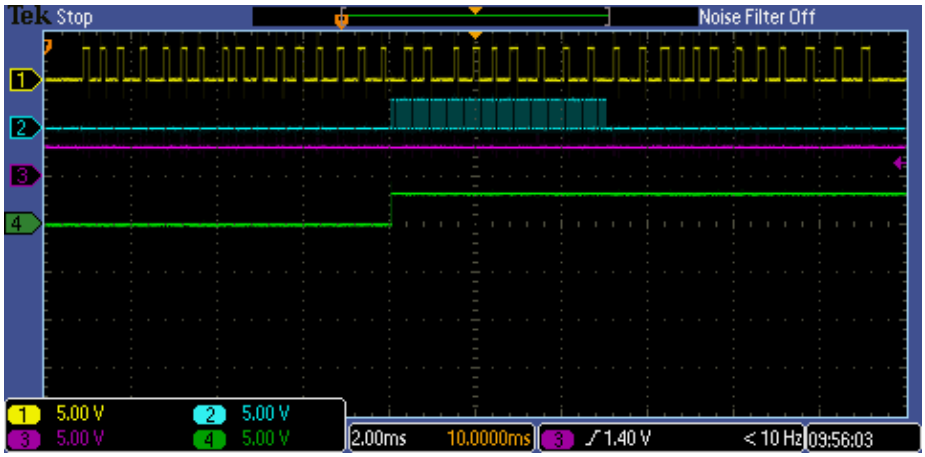


Figure 5.3: A test case example displayed on the oscilloscope

5.1.4 Stress

Applying stress to the RPi while running test cases was not done without difficulties. When the stress application (Stress-NG) was executing, communication with the RPi through SSH was not possible. The SSH process on the RPi was delayed for so long that when trying to communicate with it, a timeout was initiated.

The issue was addressed by having a second real-time application (stress-controller) which starts the stress application and then after a few seconds starts the main application. When the main application was finished, the stress controller closed the stress application. Figure 5.4 below illustrates the stress controller. This solution is not ideal however because of at least three reasons:

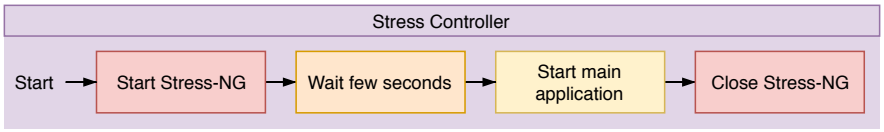


Figure 5.4: Stress attempt with a stress controller

- Whenever the stress controller made a syscall to start an application, it entered the secondary domain and got delayed for a while because of the stress application, and was scheduled by the Linux domain.
- The stress application started every time just a few seconds before the main application, which limits the variation of the stress application.
- The solution increased each test case time with a few seconds and thus became the bottleneck in the automation process.

5.2 Data Collection

The automation process created a lot of measurement data. This section describes how the data was used to make it easier to analyze.

5.2.1 Comparing Test Case

With each test case two readings from the oscilloscope were done: Reading the output of the function generator and reading the output from the RPi.

The readings from the oscilloscope and the RPi were compared with the CSV file which the RPi used for writing. The comparison between the two files showed how accurate RPi wrote. The result was a CSV file which showed the differences between each signal. A negative value meant that the RPi wrote later than specified. A positive value meant that the RPi wrote earlier.

The readings from the oscilloscope of the function generator were being compared with the CSV file which the RPi application generated when reading the function generator. The comparison showed how accurate the RPi could read. The resulting CSV was the same as from the previous comparison. A negative value meant that the RPi thought it read earlier than the oscilloscope, most probably because of the inaccuracy of the time stamp. A positive value meant that the RPi read later than the oscilloscope.

5.2.2 Comparing Every Test Case

All the generated data was read when a certain number of test cases were executed. The standard deviation of the data was then displayed to see if enough test cases have been made. If the standard deviation was stabilized enough then the data was plotted into box plots, showing the time difference result for writing and reading.

5.3 Encountered Problems

A few problems were being encountered during the implementation of the automation process, as well as during the data collection. Below is a list of some of the problems which occurred and how they were handled.

Stress
: Difficulties of adding stress to the system during the automation process were described earlier in section 5.1.4. Because of the difficulties, it was not clear how efficient the stress application was within the limited time frame in which the stress application was being executed. Because of the

complications of the stress and its unknown efficiency, it was decided to not use stress testing at all. This causes the data measurement to not being able to measure a worst-case execution time but instead focused on an ideal case.

Oscilloscope

During the automated measuring process, the oscilloscope sometimes disappeared from detection by the PC. No further communication with the oscilloscope could be done and the automated measuring then was canceled. This problem was fixed by manually changing the USB settings on the oscilloscope after each occurrence. Because of the problem, the measurements could stop at any point in time and thus needed supervision. The root cause for the problem was not identified.

5.4 Summary

This chapter gave insights and details on how the automation process was done, how the data collection was achieved, and what complications occurred. Because of the complications, the stress testing was not used. The automation process needed supervision as the oscilloscope could stop responding at any time.

6 Results & Analysis

This chapter gives a detailed description of the results from the data collection. The data distribution is going be shown first as a histogram, then with box plots to clearly show the data. A few different measurement types had been done and these differences are compared, evaluated, and briefly expounded. The differences may provide insights relevant when using the system for future purposes.

6.1 Types of Measurements

Different system configurations had been used on the RPi 3 in order to try to improve the results. The descriptions for each system configuration are described in the below subsections.

6.1.1 Isolated Core

As mentioned throughout the thesis, the application was executed on a single core together by other programs running in the GPOS. It was therefore interesting to see if the data could be improved when a single core is isolated from the Linux scheduler, thus avoiding regular programs to be executed on that specified core. Since each core had its own level 1 cache, an isolated core would not contain data or instructions from multiple programs in the level 1 cache. The number of cache misses and cache hits should then be different from a not isolated core.

6.1.2 Enabled Level 2 Cache

The level 2 cache was disabled by default on a RPi 3 with Raspbian. Measurements are therefore also done with the level 2 cache enabled. The application was also executed on a core together with regular programs. Enabling the level 2 cache should change the number of cache misses and cache hits during the execution of the application.

6.1.3 Regular

The regular data was done without using the isolated core method and with the level 2 Cache disabled.

6.2 Results

In this section, the results of the data are viewed and explained. First, the data distribution is shown and described. Afterward, each data type is shown in box plots, then compared, and finally evaluated.

6.2.1 Latency Test

The latency test (described in section 3.3.3) was executed overnight together with Stress-NG. The result from the latency test can be viewed in Table 6.1 below. The table shows the best, average, and worst encountered latency.

Table 6.1: Latency result

Best	Average	Worst
−5.98 µs	6.23 µs	82.00 µs

The reason why the best latency value is negative is that the timer interrupts are triggered earlier than expected. This is because Xenomai tried to improve the latencies by setting the timer trigger earlier. The result shows a latency range of 87.96 µs.

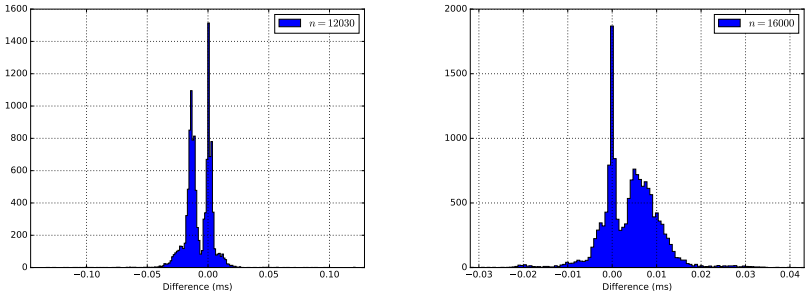
6.2.2 Distribution

It is interesting to see how the data was being distributed. The data distribution is illustrated as a histogram below in Figure 6.1. Only the data distribution of data type Regular is being shown because the data from other measurement types were distributed alike.

The Subfigures 6.1a and 6.1b clearly shows which data occurred the most. An interesting thing to notice is how the data distribution was almost split in two in both of the Sub figures. It is especially noticeable in Sub figure 6.1a.

6.2.3 Box Plots

The results of each measurement type can be seen in Figure 6.2 and Figure 6.3 below. Only the minimum and maximum fliers are shown in the box plots to



(a) Writing (b) Reading

Figure 6.1: Histogram of reading and writing for the Regular type

avoid unnecessary clutter.

What can easily be seen from Figure 6.2 is that the most of the data contained in the box and the whiskers are almost identical. The fliers showing minimal and maximal differs. The measurement type with the smallest range is L2Cache, with 255 µs. L2Cache is the measurement type of an enabled level 2 cache.

The data from Figure 6.3 was showing approximately ten times smaller data than Figure 6.2. In this case, data type Isol0 had the smallest range with 66.20 µs, which is almost four times smaller than the result from writing.

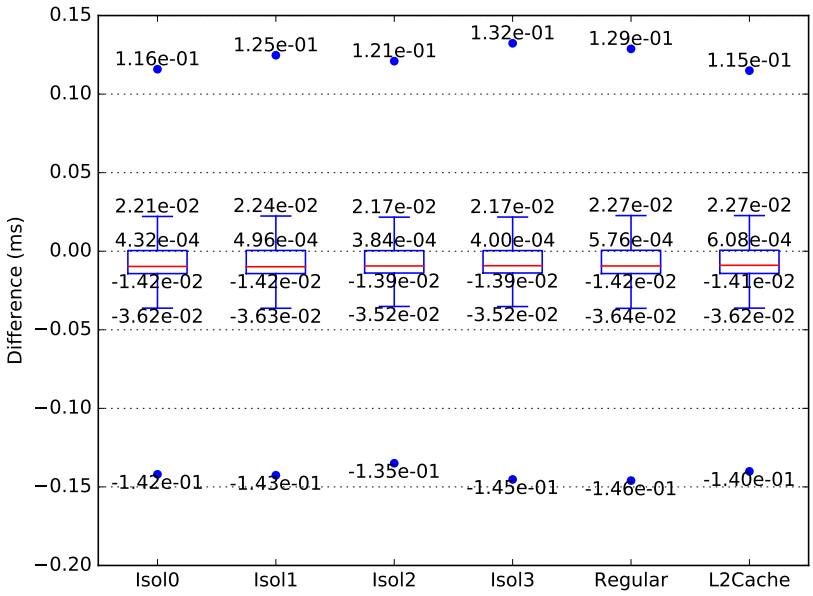


Figure 6.2: Box plot of writing measurements

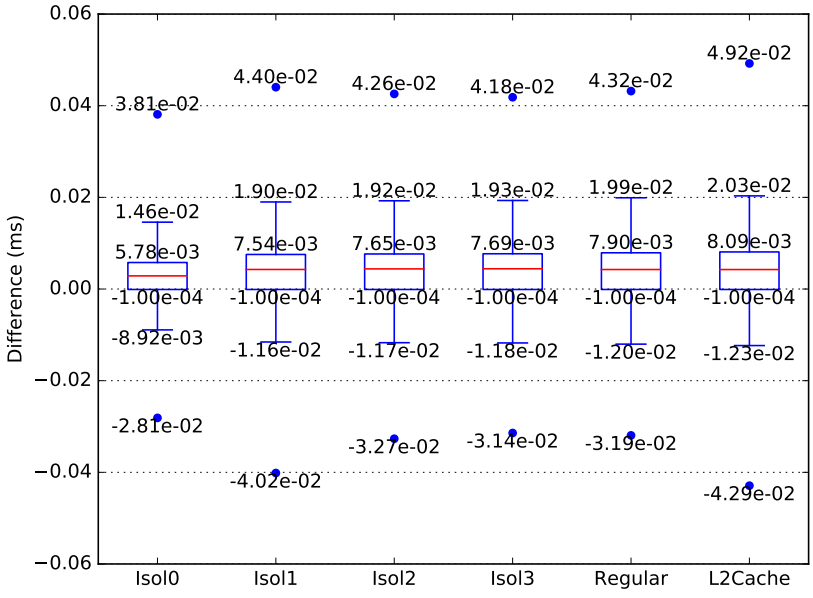


Figure 6.3: Box plot of reading measurements

6.3 Analysis

In this section, the data is analyzed. Each section below describes each part of the data of interest.

6.3.1 Box plot fliers

The fliers show the worst occurred execution time. A possible explanation for the fliers being so far away from the rest of the data could be that a cache miss occurred when calling the read timer function, causing a later timer reading than usual.

6.3.2 Reading Data

The time differences show the interrupt latency from when the GPIO interrupt occurred when the reading task received the timer value. Figure 2.1 shows a representation to the interrupt latency.

6.3.3 Writing Data

The time differences were the result of the timer interrupt latency by the scheduler. An additional latency could be caused by task Read, as it had higher priority than task Write. This means that task Write could be preempted by task Read or delayed before it could write. Another data was done without task Read enabled to demonstrate the differences, see Figure 6.4 below.

The Figure 6.4 clearly shows an improvement when task Read does not interrupt task Write, thus proving that additional latency was caused by task Read. The latency which remained was the scheduling latency. When comparing the result from Figure 6.4 and Table 6.1, it can be seen that the range from the Figure (51.20 µs) was close to the worst (82.00 µs) from the table.

6.3.4 Isol0 performance

The data type of Isol0 has better performance with reading compared to the other data types. This is very likely based on where the interrupts were handled. Almost all interrupts were handled by core 0, the exceptions were for example rescheduling interrupts or function call interrupts which appeared on all cores. A confirmation of this was done by reading */proc/interrupts* on the RPi which showed which core each interrupt was running on.

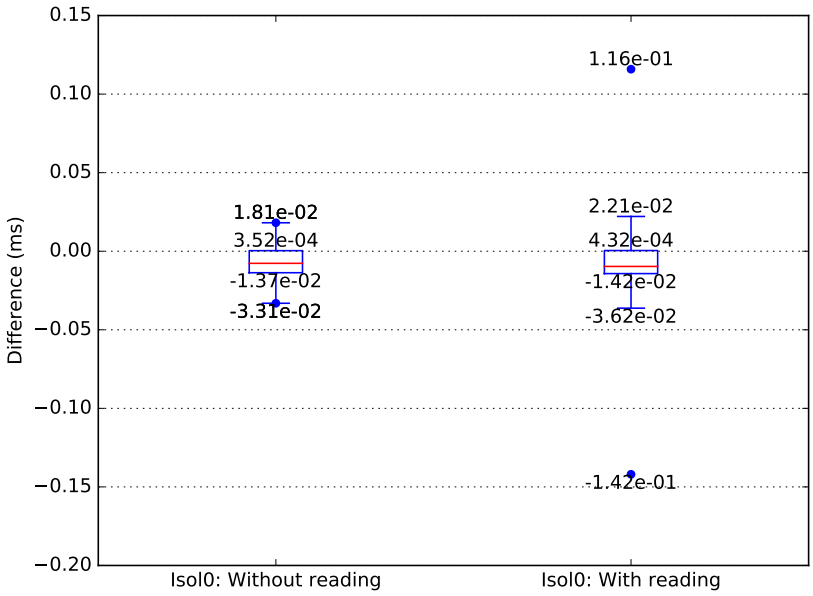


Figure 6.4: Box plot of writing differences comparing with and without reading

6.4 Summary

This chapter has shown details about the results sought out through the thesis. The results have been analyzed and compared based on information given from previous chapters.

7 Conclusions & Future Work

This chapter describes the conclusions of the project. The final results are shown afterward and discussed. The requirements set of the beginning of the project are then compared with the result. Lastly, examples of possible future work are described and motivated.

7.1 Conclusions

The final conclusions will be based on the measurement results from the previous chapter, on the process of the Xenomai installation and the application implementation, and also on the automation process. The results will then be evaluated based on the requirements from the project description in section 1.2.

Installing Xenomai on RPi 3 was proven to be more difficult than anticipated. Much time was spending on researching on other developers success with having Xenomai on RPi and trying to follow their guidelines. The Xenomai installation was deemed unsuccessful as the RPi 3 would either not boot or be unstable. Thanks to the backup, it was possible to continue on the project and gather the data needed. Because of the Xenomai setup was unsuccessful, an initial idea of configuring the kernel in ways to try to improve the measurements were scrapped, as it was not possible with the backup plan.

Implementing the application was a success. Thanks to the available exercises on how to use Xenomai's API and the documentation of Xenomai's API.

7.1.1 Final results

The final results which are used to determine if the requirements are met are from the measurement type Isolate Core as they had the best performance. A summary can be seen below in Table 7.1.

Table 7.1: Final results

	Together			Separate		
	Min	Max	Range	Min	Max	Range
Reading	−28.10 µs	38.10 µs	66.20 µs	−28.10 µs	38.10 µs	66.20 µs
Writing	−142 µs	116 µs	258 µs	−38.10 µs	18.10 µs	56.20 µs

What can be seen in the table is when the task Read and task Write was scheduled together, task Read caused worse performance for task Write, as earlier explained. What is interesting to see is that task Write has better performance than task Read when they were executed separately.

7.1.2 Results Versus Requirements

The project had requirements for the implementation in order to deem whether it was successful or not. The requirements can be seen in the list below.

- ☑ The approach chosen need to support the hardware of RPi 3.
- ☑ The GPOS preferred is Raspbian.
- ☑ The approach need hard real-time support.
- ☑ The system needs to handle real-time tasks which won't be interrupted or degraded while using generic tasks from GPOS.
- ☑ The system needs to be able to set up communication between real-time tasks and general-purpose tasks.
- ☒ The implementation of the system needs to be able to read and write GPIO pins with a time precision of $\pm 10\,\mu s$, and include timestamps within the same precision.
- ☑ The implementation of the system needs to be able to monitor and log data, either during or after the real-time measurements have been performed.

What can be seen from the list is that the time precision requirement was not met. This is because of the minimum and maximum result in Table 7.1 exceeded the required time precision of $\pm 10\,\mu s$.

7.1.3 Positive Remarks

The measurement data shown in the box plots in chapter 6 showed short boxes, meaning that the majority of measurement data had very small variance. This means that with enough test sample, the minority of the measurement data could be discarded. In other words, the system can be used for firm real-time characteristics with better precision than hard real-time.

7.1.4 Negative Remarks

Because of applying a stress test to the automation setup was unsuccessful (section 5.3), an attempt on measuring worst-case execution time could not be

done. The measurements shown displayed only the measurements for a system executed in an idle state, which can be interpreted as ideal execution time instead. The latency test in section 6.2.1) was however done with stress testing and could be used as a reference point on how the GPIO measurements could be extended to.

7.2 Future Work

Future work on the project can be made. A few of them are described in the list below.

- Modifying the RTDM GPIO device driver to read and write pins using bit-masks instead of selecting separated pins. This would allow users to read multiple of pins concurrently and also write multiple pins concurrently.
- Creating an experimental setup with efficient stress testing on the system. This would create more reliable data as the Cobalt core is sharing resources with the Linux kernel.
- Adding PREEMPT_RT patch to the Linux kernel together with Xenomai Cobalt. The PREEMPT_RT makes the Linux kernel more preemptible, thus improving the unpredictability whenever a real-time task in Cobalt need to make a syscall to the Linux kernel.

7.3 Summary

This chapter has described the overall results and conclusions of the project. The final results have been shown in Table 7.1. The project has been compared with its requirements which were set at the beginning of the project. All requirements except for one were met. Examples of future work were also described and motivated.

References

[1] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems*. Vol. 24. Real-Time Systems Series. DOI: 10.1007/978-1-4614-0676-1. Boston, MA: Springer US, 2011. ISBN: 978-1-4614-0675-4. URL: http://link.springer.com/10.1007/978-1-4614-0676-1 (visited on 26/01/2018).

[2] Phillip A. Laplante and Seppo J. Ovaska. *Real-Time Systems Design and Analysis: Tools for the Practitioner*. 4th ed. the Institute of Electrical and Electronics Engineers, Inc.: John Wiley & Sons, Inc., 2012. ISBN: 978-1-118-13660-7.

[3] *Power Supply - Raspberry Pi Documentation*. 2018. URL: https://www.raspberrypi.org/documentation/hardware/raspberrypi/power/README.md (visited on 07/05/2018).

[4] *Power Management Statistics: Information Technology - Northwestern University*. Oct. 2017. URL: http://www.it.northwestern.edu/hardware/eco/stats.html (visited on 07/05/2018).

[5] Victor Yodaiken. "The RTLinux Manifesto". In: *Linux Expo* 5 (1999), p. 12.

[6] *Real-Time Linux Wiki*. Aug. 2016. URL: https://rt.wiki.kernel.org/index.php/Main_Page (visited on 07/05/2018).

[7] *RTAI - the RealTime Application Interface for Linux*. Jan. 2018. URL: https://www.rtai.org/ (visited on 09/02/2018).

[8] *Xenomai*. 2018. URL: https://xenomai.org/ (visited on 09/02/2018).

[9] Jonathan Corbet. *Deadline scheduling for Linux [LWN.net]*. Oct. 2009. URL: https://lwn.net/Articles/356576/ (visited on 07/05/2018).

[10] A. Stahlhofen and D. Zöbel. "Linux SCHED DEADLINE vs. MARTOP-EDF". In: *2015 IEEE 13th International Conference on Embedded and Ubiquitous Computing*. 2015 IEEE 13th International Conference on Embedded and Ubiquitous Computing. Oct. 2015, pp. 168–172. DOI: 10.1109/EUC.2015.28.

[11] Philip Axer et al. "Building timing predictable embedded systems". In: *ACM Transactions on Embedded Computing Systems (TECS)* 13.4 (2014), p. 82.

[12] Steven J. Vaughan-Nichols. *Twenty Years of Linux according to Linus Torvalds*. ZDNet. Apr. 2011. URL: http://www.zdnet.com/article/twenty-years-of-linux-according-to-linus-torvalds/ (visited on 07/02/2018).

[13] *Mobile operating systems' market share worldwide from January 2012 to December 2017*. 2017. URL: https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/ (visited on 07/02/2018).

[14] *GNU Operating System*. What is Copyleft. 2018. URL: https://www.gnu.org/copyleft/ (visited on 07/02/2018).

[15] *Kernel Definition*. May 2004. URL: http://www.linfo.org/kernel.html (visited on 26/01/2018).

[16] Linus Torvalds. *Linux kernel stable tree*. May 2018. URL: https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git (visited on 08/05/2018).

[17] Philippe Gerum. "Xenomai-Implementing a RTOS emulation framework on GNU/Linux". In: *White Paper, Xenomai* (2004).

[18] Claudio Scordino and Giuseppe Lipari. "Linux and real-time: Current approaches and future opportunities". In: *ANIPLA International Congress, Rome*. 2006.

[19] *Real-Time Linux*. May 2017. URL: https://wiki.linuxfoundation.org/realtime/start (visited on 26/01/2018).

[20] Paul McKenney. *A real-time preemption overview*. Aug. 2005. URL: https://lwn.net/Articles/146861/ (visited on 19/01/2018).

[21] Paul McKenney. *Sleeping Spinlocks*. July 2017. URL: https://wiki.linuxfoundation.org/realtime/documentation/technical_details/sleeping_spinlocks (visited on 09/05/2018).

[22] Paul McKenney. *Technical details of PREEMPT_RT patch*. Feb. 2017. URL: https://wiki.linuxfoundation.org/realtime/documentation/technical_details/start (visited on 09/05/2018).

[23] J. Calandrino and H. Leontyev and A. Block and U. Devi and J. Anderson. "LITMUSRT: A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers". In: *IEEE Real-Time Systems Symposium* 27 (Dec. 2006), pp. 111–123.

[24] B. Brandenburg. "Scheduling and Locking in Multiprocessor Real-Time Operating Systems". PhD thesis. Chapel Hill: UNC, 2011.

[25] *LITMUS-RT: Linux Testbed for Multiprocessor Scheduling in Real-Time Systems*. 2017. URL: https://www.litmus-rt.org/index.html (visited on 20/02/2018).

[26] Karim Yaghmour et al. *Building embedded Linux systems: concepts, techniques, tricks & traps*. Ed. by Karim Yaghmour. 2. ed. [incl. real-time variants]. Beijing: O'Reilly, 2008. 439 pp. ISBN: 978-0-596-52968-0.

[27] *Start Here: Xenomai*. 2018. URL: https://xenomai.org/start-here/ (visited on 15/02/2018).

[28] J. Kiszka. "The real-time driver model and first applications". In: *7th Real-Time Linux Workshop, Lille, France*. 2005.

[29] Wind River Systems, Inc. *VxWorks: Product Overview*. Sept. 2016.

[30] *Raspberry Pi*. Jan. 2018. URL: https://en.wikipedia.org/w/index.php?title=Raspberry_Pi&oldid=820745037 (visited on 18/01/2018).

[31] ARM. *ARM Cortex-A53 MPCore Processor Technical Reference Manual*. 2013. URL: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0500d/DDI0500D_cortex_a53_r0p2_trm.pdf (visited on 17/01/2018).

[32] *Raspberry Pi 3 is out now! Specs, benchmarks & more*. The MagPi Magazine. Feb. 2016. URL: https://www.raspberrypi.org/magpi/raspberry-pi-3-specs-benchmarks/ (visited on 20/02/2018).

[33] *Raspberry Pi Downloads - Software for the Raspberry Pi*. Raspberry Pi. 2018. URL: https://www.raspberrypi.org/downloads/ (visited on 12/02/2018).

[34] TalOrg. *Build 64-bit kernel for Raspberry Pi 3, using native tools*. TalOrg. Mar. 2017. URL: http://www.tal.org/tutorials/raspberry-pi3-build-64-bit-kernel (visited on 12/02/2018).

[35] *Kernel source tree for Raspberry Pi Foundation*. Jan. 2018. URL: https://github.com/raspberrypi/linux (visited on 26/01/2018).

[36] Youssef Zaki. "An embedded multi-core platform for mixed-criticality systems: Study and analysis of virtualization techniques". master thesis. Stockholm, Sweden: KTH, 2016. 65 pp.

[37] Nitin Kulkarni. "Real-time audio processing for an embedded Linux system using a dual-kernel approach". master thesis. KTH, 2017.

[38] Salman Rafiq. "Measuring Performance of Soft Real-Time Tasks on Multicore Systems". master thesis. KTH, 2011.

[39] Adam Lundström. "Finding strategies for executing Ada-code in real-time on Linux using an embedded computer". master thesis. Stockholm: KTH, 2016. URL: http://kth.diva-portal.org/smash/get/diva2:931386/FULLTEXT01.pdf.

[40] Radosław Rybaniec and Piotr Z. Wieczorek. "Measuring and minimizing interrupt latency in Linux-based embedded systems". In: *Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments 2012*. Vol. 8454. 2012. URL: https://doi.org/10.1117/12.2000230.

[41] *Installing Xenomai 3.x: Xenomai*. 2018. URL: https://xenomai.org/installing-xenomai-3-x/ (visited on 23/02/2018).

[42] Koide Masahiro. *Raspberry Pi 3 and real-time kernel, introduction of Xenomai*. Japanese. Aug. 2016. URL: http://artteknika.hatenablog.com/entry/2016/08/23/143400 (visited on 18/06/2018).

[43] Christophe Blaess. *Xenomai sur Raspberry Pi 3 : bon espoir mais bilan mitigé*. French. Mar. 2017. URL: https://www.blaess.fr/christophe/2017/03/20/xenomai-sur-raspberry-pi-3-bilan-mitige/ (visited on 18/06/2018).

[44] Harco Kuppens. *Raspberry Pi image for the Pi zero,1,2,3 with Xenomai 3.0.5 on Raspbian linux 4.1.y Debian 8 jessie*. Lab. Aug. 2017. URL: http://www.cs.kun.nl/lab/xenomai/ (visited on 13/02/2018).

[45] *Xenomai 3.0.5 API*. 2018. URL: https://xenomai.org/documentation/xenomai-3/html/xeno3prm/index.html (visited on 07/03/2018).

[46] IVI Foundation. *The VISA Library*. VXIplug & play Systems Alliance, Oct. 2017. (Visited on 09/03/2018).

[47] PyVISA. *PyVISA: Control your instruments with Python*. 2016. URL: https://pyvisa.readthedocs.io/en/stable/ (visited on 09/03/2018).

[48] *Ubuntu Manpage: stress-ng - a tool to load and stress a computer system*. Ubuntu manuals. Mar. 2016. URL: http://manpages.ubuntu.com/manpages/xenial/man1/stress-ng.1.html (visited on 09/03/2018).

[49] Xenomai. *I-Pipe download page*. Oct. 2017. URL: https://xenomai.org/downloads/ipipe/ (visited on 15/02/2018).

[50] *Kernel building - Raspberry Pi Documentation*. 2018. URL: https://www.raspberrypi.org/documentation/linux/kernel/building.md (visited on 16/02/2018).

[51] *Xenomai Git Repositories*. 2018. URL: http://git.xenomai.org/ (visited on 16/02/2018).

[52] Harco Kuppens. *Xenomai 3 on RPI with GPIO*. Aug. 2017. URL: https://github.com/harcokuppens/xenomai3_rpi_gpio (visited on 24/04/2018).

A Xenomai Setup (Unsuccessful Attempt Using RPi Kernel)

This approach is using a guide created by Harco Kuppens[52] where he stated to successfully setup Xenomai Cobalt on a RPi 3. In order to have a Xenomai cobalt core on the RPi 3, a few steps was needed to be done. A kernel version needed to be decided. This was done by comparing the available versions for I-Pipe. It is important that the version is as close as possible because either I-Pipe is version dependent and was be later patched to the kernel.

A patch is a file created from using a git diff, which essentially shows all the differences between latest commit and unstaged files. The patches can then be applied on different kernel repositories.

Both the kernel and the Xenomai API was needed to be built, which was done by using a cross-compiler. A cross-compiler is essentially an application which translates the source code into machine code for a different architecture. Cross-compilers are usually used for building to embedded systems because the embedded systems have usually much slower hardware than a host computer.

A.0.1 Preparation

The parts needed was: the kernel source, a cross-compiler, the Xenomai source, and the I-Pipe patch.

Acquiring I-Pipe

As mentioned earlier, the kernel and the I-Pipe needed to have the same version or as close as possible. The available releases of I-Pipe can be acquired from the Xenomais download page[49]. It did not however include RPi 3, but a similar version was acquired instead. In addition to the patch was another one downloaded to fix certain configurations in the kernel source tree. A third file was acquired which is an updated device driver for the GPIO used by RPi 3. It was updated to be used with I-Pipe.

```
~/$ wget https://raw.githubusercontent.com/harcokuppens/
    ↪ xenomai3_rpi_gpio/master/install/patches_for_pi2+/ipipe-
    ↪ core-4.1.18-arm-9.fixed.patch
~/$ wget http://www.blaess.fr/christophe/files/article
    ↪ -2016-05-22/patch-xenomai-3-on-bcm-2709.patch
```

```
~/$ wget -O pinctrl-bcm2835.c http://git.xenomai.org/ipipe.git/
    ↪ plain/drivers/pinctrl/bcm/pinctrl-bcm2835.c?h=vendors/
    ↪ raspberry/ipipe-4.1
```

Acquiring Kernel

The RPi has its own kernel source tree accessible on The Raspberry Pi Foundations official git repository[35]. The kernel source tree was downloaded using command:

```
~/$ git clone https://github.com/raspberrypi/linux.git ~/rpi-
    ↪ kernel
```

The latest version on the kernel source tree was different from the sought out version, therefore it was needed to change it into 4.1.21, that was done by changing a branch which has the needed kernel version as latest commit.

```
~/$ git checkout rpi-4.1.y
```

Acquiring Cross-Compiler

A cross-compiler was needed as mentioned before. The chosen cross-compiler was from the kernel building guide[50]. It was downloaded by command:

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

Then to make it easier to specify the cross-compiler, the $PATH environment was updated using the commands below:

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

Acquiring Xenomai Source

The Xenomai source can be found on Xenomais git repository page[51]. The latest Xenomai version was 3.x and downloaded with command:

```
~/$ git clone http://git.xenomai.org/xenomai-3.git/ ~/xenomai
```

A.0.2 Installing Cobalt Core

The preparations was then done and next step was to continue with the Xenomai setup. The steps for Cobalt core and Mercury core are similar but the Cobalt core have slightly more configurations that was needed to be made.

Configure Kernel

Preparing the kernel source with a Xenomai script was needed. What this script does is adding the Cobalt core into the kernel source together with the I-Pipe patch. This is done by running this command:

```
~/xenomai$ ./scripts/prepere-kernel.sh --linux=~/rpi-kernel --
    ↪ arch=arm --ipipe=../ipipe-core-4.9.18-arm-9.fixed.patch
~/rpi-kernel$ patch -p1 < ../patch-xenomai-3-on-bcm2709.patch
~/rpi-kernel$ cp ../pinctrl-bcm2835.c ./drivers/pinctrl/bcm/
```

It was then necessary to setup a configuration on the kernel to specify what hardware the kernel is for. This was be done by extracting the kernel configuration from a running RPi 3. Getting the configuration was done by using commands:

```
pi@rpi:~/$ sudo modprobe configs
pi@rpi:~/$ cp /proc/config.gz .
pi@rpi:~/$ gunzip ./config.gz
```

The configuration file was then copied over to the host computer and replaced $/rpi-kernel/.config$. After that could the configuration of the kernel be done. Configuration was done by using the built-in menuconfig, which was ran by command:

```
~/rpi-kernel$ make ARCH=arm menuconfig
```

Some features was needed to be edited in order for Xenomai to work properly.

CONFIG_CPU_FREQ This option allows the CPU frequency to be modulated depending on the work load. This feature add unpredictability and was needed to be disabled.

CONFIG_CPU_IDLE This feature allows the CPU to enter sleep states. As it takes time for the CPU to be awoken, it adds latencies and unpredictability. The feature could also cause timers for Xenomai to stop working. It was therefore needed to disable this feature.

CONFIG_KGDB Is a kernel debugger which could only be enabled for X86 architectures. As RPi 3 is an ARM architecture, it was needed to disable it.

CONFIG_CONTEXT_TRACKING_FORCE This option was automatically disabled with the I-Pipe patch.

XENO_DRIVERS_GPIO_BCM2835 Is the option to use the GPIO on RPi, it was enabled with "m", meaning that it would be a loadable kernel module.

Installing the Kernel

When the configurations was done, the kernel could finally be built. This is done by command:

```
~/rpi-kernel$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf-
    ↪ zImage modules dtbs -j12
```

The next step was to install the kernel on the proper destination. The Raspbian OS was already installed on the SD card which was mounted in path */mnt/rpi*. The mount contains two partitions. Boot is the partition which only handles the boot sequence which selects the kernel and its arguments. Rootfs is the other partition which contains the rest of the entire OS.

The parts needed to be installed was the kernel image, modules, and device tree blobs (dtbs). These was installed with the command:

```
~/rpi-kernel$ sudo make ARCH=arm CROSS_COMPILE=arm-linux-
    ↪ gnueabihf- INSTALL_MOD_PATH=/mnt/rpi/rootfs
    ↪ modules_install
~/rpi-kernel$ sudo cp arch/arm/boot/zImage /mnt/rpi/boot/kernel7.
    ↪ img
~/rpi-kernel$ sudo cp arch/arm/boot/dts/*.dtb /mnt/rpi/boot
~/rpi-kernel$ sudo cp arch/arm/boot/dts/overlays/*.dtb* /mnt/rpi/
    ↪ boot/overlays
```

The kernel was then finally installed with Xenomai Cobalt. The next step was to install the Xenomai API so that real-time applications could be implemented.

A.0.3 Installing Xenomai API

The Xenomai source was acquired by Git, and because of this was a configuration script needed to be executed in order to generate the necessary Makefiles. The automatic configuration script was be executed by command:

```
~/xenomai$ ./scripts/bootstrap
```

The next step required a build_root directory and a staging path. The build_root was needed to store the files the coming command was going to generate. The staging path is usually a path to store the installed files temporary before moving them to the final location. In this case however could the files be stored directly on the final path on the SD card.

It was needed to generate a installation configuration for the specific platform which was going to be used, this was done by commands:

```
~/xenomai$ cd buildroot
~/xenomai/buildroot$ ../configure \
```

```
CFLAGS="-mcpu=cortex-a53␣-march=armv7-a" \
LDFLAGS="-mcpu=cortex-a53␣-march=armv7-a" \
--enable-smp \
--disable-debug \
--build=i686-pc-linux-gnu \
--host=arm-linux-gnueabihf \
--with-core=cobalt
```

The arguments in the scripts describes the architecture on the platform. The CPU used on RPi 3 is an ARM cortex-A53. The OS used was Raspbian which is a 32bit OS, meaning that the instruction set armv7-a is necessary.

When the configuration step was done, the installation of the API could finally start. The installation was done by using command:

```
~/xenomai/buildroot$ make DESTDIR=/mnt/rpi/rootfs install
```

B GPIO Template

B.1 Makefile

```
# Xenomai setup
XENO_CONFIG  := /usr/xenomai/bin/xeno-config
XENO_CFLAGS  := $(shell $(XENO_CONFIG) --rtdm --alchemy --cflags)
XENO_LDFLAGS := $(shell $(XENO_CONFIG) --rtdm --alchemy --ldflags)
CC           := arm-linux-gnueabihf-gcc

# The Target
TARGET := main

# The Directories, Source, Includes, Objects
SRCDIR   := src
INCDIR   := inc
BUILDDIR := obj
SRCEXT   := c
INCEXT   := h
OBJEXT   := o

# Flags, Libraries and Includes
CFLAGS  := $(XENO_CFLAGS) -Wall -O3 -std=c11
LDFLAGS := $(XENO_LDFLAGS) -Xlinker -Map=$(TARGET).map
INC     := -I$(INCDIR)
INCDEP  := -I$(INCDIR)

#
# DO NOT EDIT BELOW THIS LINE
#
SOURCES     := $(wildcard $(SRCDIR)/*.$(SRCEXT))
INCLUDES    := $(wildcard $(INCDIR)/*.$(INCEXT))
OBJECTS     := $(patsubst $(SRCDIR)/%,$(BUILDDIR)/%,$(SOURCES:.$(SRCEXT)=.$(OBJEXT)))

# Default Make
all: $(TARGET)

# Make the Directories
directories:
  @mkdir -p $(BUILDDIR)

# Clean
clean:
  @$(RM) -r $(BUILDDIR) $(TARGET) *.map
  @echo "cleaned"

# Link
$(TARGET): $(OBJECTS)
  @echo Linking $@
  @$(CC) -o $(TARGET) $^ $(LDFLAGS)
```

```
  @size $(TARGET)

# Compile
$(BUILDDIR)/%.$(OBJEXT): $(SRCDIR)/%.$(SRCEXT)
  @echo "building "$@
  @mkdir -p $(dir $@)
  @$(CC) $(CFLAGS) $(INC) -c -o $@ $<

# Non-File Targets
.PHONY:  clean
```

B.2 Code

```
//! @file    gpio_template.cpp
//! @author  Gustav Johansson <gusjohan@kth.se>
//! @brief   A template for using RTDM GPIO on RPi3 with Xenomai
//!          Cobalt kernel. Don't forget to load the GPIO device
//!          driver module with  "modprobe xeno-gpio-bcm2835"
//!          before running.

//! The device driver in mainline linux tree is very different from the rpi
//! linux tree. A problem with this is that the gpio pins that appears with the
//! gpio module does not have the same numbers as on a regular RPi. The pins can
//! be seen in /dev/rtdm/pinctrl-bcm2835/. For me started the pins at 970 and
//! ended at 1023. The pins work however without any problems, you only need to
//! know which pin corresponds to a regular pin which is easy. 970+16=986
//! corresponds to gpio pin 16 (bcm).
#include <stdio.h>
#include <alchemy/task.h>
#include <rtdm/gpio.h>

//! @brief  Initialize a GPIO pin either as input/inputIRQ or output.
//! @param  path      Path to the pin.
//!                   Example /dev/rtdm/pinctrl-bcm2835/gpio986
//! @param  openArg   Arguments for open. O_RDONLY,
//!                                       O_RDONLY|O_NONBLOCK
//! @param  ioctlArg  Arguments for ioctl
//!                   -input:     GPIO_RTIOC_DIR_IN
//!                   -output:    GPIO_RTIOC_DIR_OUT
//!                   -input IRQ: GPIO_RTIOC_IRQEN
//! @param  edge      Specify edges or start value
//!                   -input: NULL
//!                   -output: Start value
//!                   -input IRQ: GPIO_TRIGGER_EDGE_RISING or
//!                               GPIO_TRIGGER_EDGE_FALLING
int pinInit(const char* path, int openArg, int ioctlArg, int* edge)
{
  int retval;

  int pin = open(path, openArg);
  if(pin < 0){
    rt_printf("Error:%d couldn't open file. Check path or load module\n", pin);
  }
  retval = ioctl(pin, ioctlArg, edge);
  if(retval < 0){
    rt_printf("Error:%d couldn't request ioctl. Run as sudo\n", retval);
```

```
  }

  return pin;
}

// read from GPIO pin
int pinGet(int pin)
{
  int retval;
  int val=-1;
  retval = read(pin, &val, sizeof(val));
  if(retval < 0){
    rt_printf("Error:%d couldn't read pin.\n", retval);
  }
  return val;
}

// write to GPIO pin
void pinSet(int pin, int val)
{
  int retval;
  retval = write(pin, &val, sizeof(val));
  if(retval < 0){
    rt_printf("Error:%d couldn't write pin.\n", retval);
  }
}

// real-time task
RT_TASK task;
void rtTask(void* arg)
{
  int edges  = GPIO_TRIGGER_EDGE_RISING | GPIO_TRIGGER_EDGE_FALLING;
  int writeValue = 0;
  int retval;

  //! @brief  open and read value from GPIO21 (BCM).
  rt_printf("Input\n");
  int pinInput = pinInit("/dev/rtdm/pinctrl-bcm2835/gpio991",
    O_RDONLY|O_NONBLOCK,
    GPIO_RTIOC_DIR_IN,
    NULL);
  rt_printf("read value:%d\n", pinGet(pinInput));

  //! @brief  open and read value from GPIO20 (BCM) using interrupt.
  //!         input change detect raising and falling edge.
  rt_printf("INPUT IRQ\n");
  int pinInputIRQ = pinInit("/dev/rtdm/pinctrl-bcm2835/gpio990",
    O_RDONLY,
    GPIO_RTIOC_IRQEN,
    &edges);
  rt_printf("read value:%d\n", pinGet(pinInputIRQ));

  //! @brief  open and write value to GPIO16 (BCM).
  rt_printf("OUTPUT\n");
  int pinOutput = pinInit("/dev/rtdm/pinctrl-bcm2835/gpio986",
    O_WRONLY,
    GPIO_RTIOC_DIR_OUT,
```

```
    &writeValue);
  for(int i=0; i<5; ++i){
    pinSet(pinOutput, writeValue = !writeValue);
    retval = rt_task_sleep(500000000);
    if(retval < 0){
      rt_printf("Error:%d couldn't put task to sleep", retval);
    }
  }
}

int main(int argv, char* argc[])
{
  int retval;

  // create task
  retval = rt_task_create(&task, "rtTask", 0, 99, T_JOINABLE);
  if(retval < 0){
    rt_printf("Error:%d couldn't create task.\n", -retval);
  }
  // start task
  retval = rt_task_start(&task, rtTask, NULL);
  if(retval < 0){
    rt_printf("Error:%d couldn't start task.\n", -retval);
  }
  // join task
  retval = rt_task_join(&task);
  if(retval < 0){
    rt_printf("Error:%d couldn't join task.\n", -retval);
  }

  return retval;
}
```

TRITA TRITA-EECS-EX-2018:179