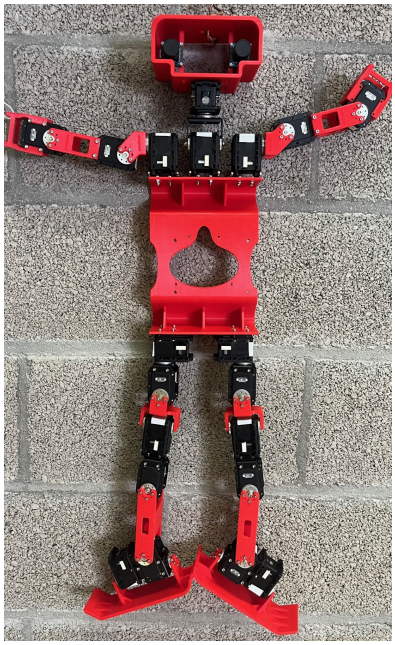
Development of the software architecture of a mobile robot

Auteur : Bounar, Nadir
Promoteur(s) : Boigelot, Bernard
Faculté : Faculté des Sciences appliquées
Diplôme : Master en sciences informatiques, à finalité spécialisée en "management"
Année académique : 2023-2024
URI/URL : http://hdl.handle.net/2268.2/20386

UNIVERSITY OF LIÈGE

ATFE0002-1: MASTER THESIS

Development of the software architecture for a mobile robot



Author:
Nadir Bounar, s193980

Supervisor:
Bernard BOIGELOT

Faculty of Applied Sciences

Master's thesis completed in order to obtain the degree of Master of Science in Computer Science

Academic year 2023-2024

"Compare yourself to who you were yesterday, not to who someone else is today."

— Jordan B. Peterson

UNIVERSITY OF LIÈGE

Abstract

Faculty of Applied Sciences
Department of Computer Science

Development of the software architecture for a mobile robot

Author: ***Nadir Bounar***
Supervisor: Mr. Bernard Boigelot
Academic Year: 2023-2024

This master's thesis focuses on building a software infrastructure for a unique form of humanoid robot, developed within the RoboCup competition framework. The primary aim was to integrate a real-time operating system, such as Preempt-RT, Xenomai, or RTAI, to minimize latency and optimize system performance. An appropriate Linux distribution was also chosen as the software development base. Following a thorough evaluation based on various criteria, including latency, Preempt-RT was selected as the real-time operating system, accompanied by the Fedora distribution, chosen for its minimal size.

The second part of this thesis concentrates on establishing efficient communication between electronic and software components. The chosen communication mode is USB (Universal Serial Bus), and results demonstrate satisfactory data transfer between these components.

The third part involves implementing an event logging system, based on four selectable log levels by the user, enabling quick issue tracking and resolution during robot program execution. Recorded logs are detailed, including time down to the nanosecond, message level, content, as well as the relevant line and file. Users can select the base level, filtering logged messages into a file located in a specific directory.

Furthermore, robust interfaces for task creation and inter-task communication have been designed to meet future needs. The primary goal was to optimize the efficiency and responsiveness of information exchange, thus contributing to the overall robot performance. The task-related interface facilitates the creation of tasks, assigning priorities and periods as needed, and, if required, assigning them to a designated processor core. The second interface enables tasks to communicate with each other: if a task requires another task's content, it can request and receive it once available. During this content wait, the task is paused to conserve processor resources.

To conclude, this thesis has established a robust software infrastructure for a humanoid robot intended for the RoboCup competition. Through the selection of Preempt-RT and Fedora, system performance has been optimized. The use of USB for communication and the implementation of a logging system have also been validated. The interfaces created for task management and communication meet future needs for efficiency and responsiveness.

Résumé

Développement de l'architecture logicielle d'un robot mobile

Autheur: ***Nadir Bounar***

Promoteur: M. Bernard Boigelot

Année Académique: 2023-2024

Cette thèse de master se concentre sur la construction d'une infrastructure logicielle pour une forme original de robot humanoïde, développée dans le cadre de la compétition RoboCup. L'objectif principal était d'intégrer un système d'exploitation temps réel, tel que Preempt-RT, Xenomai ou RTAI, afin de minimiser la latence et d'optimiser les performances du système. Une distribution Linux appropriée a également été choisie comme base de développement logiciel. Suite à une évaluation approfondie basée sur divers critères, dont la latence, Preempt-RT a été sélectionné comme système d'exploitation temps réel, accompagné de la distribution Fedora, choisie pour sa taille minimale.

La deuxième partie de cette thèse se concentre sur l'établissement d'une communication efficace entre les composants électroniques et logiciels. Le mode de communication choisi est l'USB (Universal Serial Bus), et les résultats montrent un transfert de données satisfaisant entre ces composants.

La troisième partie implique la mise en place d'un système de journalisation d'événements, basé sur quatre niveaux de journal sélectionnables par l'utilisateur, permettant un suivi et une résolution rapides des problèmes lors de l'exécution du programme du robot. Les journaux enregistrés sont détaillés, incluant l'heure jusqu'à la nanoseconde, le niveau du message, son contenu, ainsi que la ligne et le fichier pertinents. Les utilisateurs peuvent sélectionner le niveau de base, filtrant ainsi les messages enregistrés dans un fichier situé dans un répertoire spécifique.

De plus, des interfaces robustes ont été conçues pour la création et la communication des tâches, afin de répondre aux besoins futurs. Elles visent principalement à optimiser l'efficacité et la réactivité des échanges d'informations, contribuant ainsi à la performance globale du robot. L'interface de gestion des tâches facilite leur création, permettant d'attribuer des priorités et des périodes si nécessaire, ainsi que de les affecter à des cœurs de processeur spécifiques. La seconde interface permet aux tâches d'interagir entre elles : lorsqu'une tâche requiert le contenu d'une autre, elle peut en faire la demande et le recevoir dès qu'il est disponible. Durant cette attente, la tâche est mise en pause pour économiser les ressources du processeur.

En conclusion, cette thèse a développé une infrastructure logicielle solide pour un robot humanoïde compétitif. La sélection de Preempt-RT et de Fedora a amélioré les performances du système, tandis que l'utilisation de l'USB et la mise en place d'un système de journalisation ont été confirmées. Les interfaces conçues pour la gestion des tâches et la communication anticipent efficacement les futurs besoins en termes d'efficacité et de réactivité.

Acknowledgements

First and foremost, I wish to express my deepest gratitude to my promotor, Mr. Boigelot. His unwavering support and guidance have been instrumental in enabling me to complete my master's thesis on a subject that I am genuinely passionate about. This project has provided me with the opportunity to explore areas of great personal interest, enriching my understanding in numerous ways. I am particularly thankful for his accessibility, consistent supervision throughout the year, and the invaluable insights and feedback he has provided. His fervor for the fields of computer science and electronics has served as a wellspring of inspiration.

Furthermore, I extend my heartfelt appreciation to my close friends and the Robocup team, whose unwavering support has been a constant throughout the year, steadfast even during the most challenging moments.

Lastly, I am profoundly grateful to my parents, with a special mention to my mother. Their unyielding faith in me and their enduring support over the past five years have been indispensable pillars of strength.

Contents

List of Figures

List of Abbreviations

RTOS	**R**eal **T**ime **O**perating **S**ystem
USB	**U**niversal **S**erial **B**us
RT	**R**eal **T**ime
CAN BUS	**C**ontroller **A**rea **N**etwork **BUS**
RTAI	**R**eal **T**ime **A**pplication **I**nterface
IRQ	**I**nterrupt **Req**uest
GB	**G**iga**B**yte
MBPS	**M**ega**B**its **P**er **S**econd
CPU	**C**entral **P**rocessing **U**nit
EVL	**E**nhanced **V**irtual Machine **L**ayer
MSC	**M**ass **S**torage **C**lass
HID	**H**uman **I**nterface **D**evice
USB	**U**niversal **S**erial **B**us
CDC	**C**ommunications **D**evice **C**lass
ACM	**A**bstract **C**ontrol **M**odel

Chapter 1

Introduction

1.1 Objectives

In a world where technology is evolving at a breakneck pace, humanity stands on the brink of a new era, that of humanoid robots. This rapid advancement in technology has paved the way for the emergence of humanoid robots, machines that not only resemble humans but are also capable of mimicking human actions and interactions.

These anthropomorphic robots are designed to behave similarly to a human, enabling them to adapt and interact effectively with the human environment. The concept of humanoid robots, once confined to the pages of science fiction novels, has been popularized by visionaries such as Isaac Asimov [1]. These authors painted vivid pictures of a future where robots and humans would coexist and interact. What was once considered a fantasy is now a tangible reality, thanks to the rapid advancements in technology and engineering.

The first prototypes of humanoid robots appeared in the mid-20th century. However, it is in recent decades that we have witnessed a true revolution in this field. Humanoid robots, once confined to research laboratories, have now found their place in various sectors. Whether in research and development, education, healthcare, or elderly care, these anthropomorphic machines are redefining our interaction with technology.

To concretely illustrate this evolution, let's take the example of Honda's ASIMO [2], one of the first humanoid robots to walk autonomously, or Hanson Robotics' Sophia [3], who was the first humanoid robot to receive citizenship of a country. These examples not only show how far we have come, but also the potential that the future holds in the field of humanoid robots.

In this context, the RoboCup [4], an international competition promoting research in robotics and artificial intelligence, emerges. Its flagship event, the Robot Soccer World Cup, features teams of autonomous humanoid robots competing in a football tournament. The ultimate goal is to develop robots that can outperform a human team in an international match by 2050. The RoboCup [4] consists of various leagues, each showcasing specific types of robots.

Transitioning from the broader context to specific projects, Professor Bernard Boigelot and I have relived the project within the Montefiore Institute. This project, temporarily paused due to the Covid-19 pandemic, aims to involve various disciplines of the Faculty of Applied Sciences in the robot's development.

The project team comprises 20 students divided into three distinct teams—electronics, computer science, and mechanical—with the goal of pushing the boundaries of humanoid robotics through interdisciplinary collaboration.

Our team competes in the KidSize league [5], where robots between 40 to 90 cm compete on a 6 × 9-meter field for two 10-minute periods. In this league, robots must move using their legs and maintain balance without external assistance, relying solely on human-like actuators and sensors. They must determine their position and orientation, the ball's location, and identify their teammates and opponents using only a binocular vision system. Our objective is to construct the robot entirely from scratch, from inception to completion, within this rigorous framework.

1.2 State of the Project

Since its inception in 2017, the project has made significant progress in various fields, including master's theses and personal projects.

Master's theses include:

- Huber Woszczyk, *Simulation of complex actuators*, 2015-2016 [6]
- Guillaume Lempereur, *Development of an embedded servomotor controller*, 2015-2016 [7]
- Grégory Di Carlo, *Vision-based robot position estimation*, 2015-2016 [8]
- Tom Ewbank, *Efficient and precise stereoscopic vision of humanoid robots*, 2016-2017 [9]
- Odile Wauquaire, *Simulation of complex actuators*, 2016-2017 [10]

Personal projects include:

- Quentin Boileau, *Building a simulation environment for biped walking robots using Blender*, 2016-2017 [11]
- Pierre Nicolay, *Integration of Libopencm3 in FreeRTOS for STM32F4 and ST32F3 MCU*, 2018-2019 [12]

This year, the project has gained new momentum with the addition of team members who bring diverse skills and renewed enthusiasm. The primary objective of the computer science team is to develop a bootloader that enables simultaneous communication with multiple servo motors.

Simultaneously, the electronic team has developed a multifunctional board that must be capable of managing the battery resources optimally, handling communication between different parts of the robot, and efficiently powering all electronic components, notably the servo motors or the motherboard. Meanwhile, the mechanical team has focused on enhancing the flexibility of the robot limbs, integrating foot pressure sensors, and optimizing mobility.

A personal research project is also in progress during this year, aiming to establish effective communication between servo motors using the CAN BUS protocol [13]:

- Maxime Leonard, *Design of communication protocol over CAN bus for humanoid robots*, 2023-2024 [14]

The short-term goal is to enable the robot to walk autonomously in the Montefiore hallways and then to replicate the robot five times to form a complete team, with specific adjustments for the goalkeeper role. The ultimate goal is to win the RoboCup [4] Competition.

Despite the remarkable progress, several challenges remain. The electronic team needs to conduct a series of tests to validate the servo motor electronic boards and the multifunctional board, and address efficient battery connection and consumption. This includes developing methods to optimize battery usage and determining efficient ways to connect multiple battery cells. On the mechanical side, significant adjustments are required, including enhancing robot limb flexibility, integrating pressure sensors, and conducting comprehensive simulations to ensure optimal functionality. For the computer science team, several tasks remain, including implementing the robot's strategy, movement implementation, and the necessary algorithms for autonomous operation.

1.3 Objectives of this Master's Thesis

This thesis primarily focuses on developing a real-time platform to minimize latency in interactions between the robot's motherboard that is not yet configured and its electronic components through the electronic board that is also not yet configured.

The initial crucial step entails selecting and optimizing a Real-Time Operating System for the motherboard, which currently has no installed components. The objective is to attain response times comparable to human reflexes. We will explore various kernel options and implement strategies to minimize latency, thus facilitating seamless communication among electronic components such as actuators and sensors. It is important to note that we won't be re-implementing the kernel but rather examining all configurations that could affect latencies and selecting those that decrease them.

The second part is to implement a robust communication architecture between the motherboard and the electronic board The motherboard sends instructions to the electronic board to retrieve data, which is then transmitted back. Efficient protocols and interfaces, as neural connections in the human body, will be designed to facilitate smooth data exchange.

The third phase of the project entails the development of an exhaustive logging system to precisely document every action performed by the robot. This system will seize and timestamp each action, allowing for a detailed analysis and optimization of the system. This addresses the problem of inaccurate tracking of robot operations observed in past competitions. The objective is to simplify the debugging process.

Effective task management, along with the creation and communication between tasks, is equally vital. Tasks, which represent diverse functions of the robot, such as `vision_task`, necessitate efficient communication and resource handling. We will establish a robust interface for task creation and management, focusing on minimal memory allocation and CPU usage. This approach guarantees seamless coordination among various operations, thereby enabling the smooth execution of complex tasks.

Technical details will be provided in the *Appendix: A*, to aid others in understanding the project's foundations and continuing its development

Chapter 2

Hardware

2.1 Overview

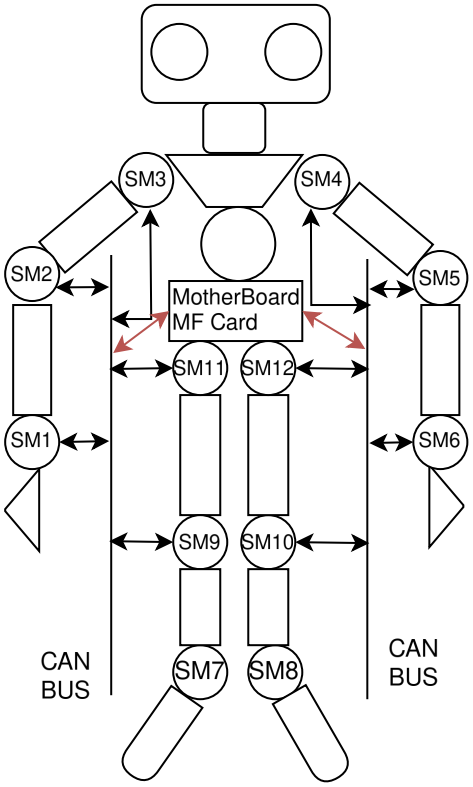


FIGURE 2.1: Simplified Representation Of The Robot

In this simplified representation of the robot, several servomotors, referred to as SM, are present. Typically, they consist of a motor coupled with a feedback mechanism, such as a potentiometer or an encoder, which allows for the accurate determination of the current position of the motor axis. These servomotors have been modified from the original versions, and details can be found in [7]. Their role is to manage the positioning of different parts of the body as precisely as possible.

Additionally, we have included two CAN buses [13] to facilitate communication between the servomotors and the multifunction board, denoted as MF in the schema.

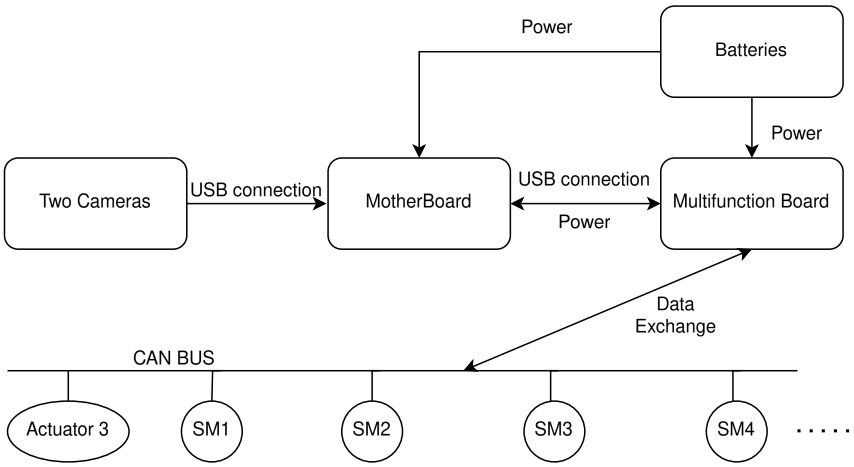


FIGURE 2.2: Sectional View Of The Robot

This plan represents a cross-sectional view of what is happening. The motherboard is connected via a USB 2.0 connection to the multifunction card and a USB 3.0 connection to the binocular vision system (Two Cameras), as it transmits data up to 1 GB/s, which is well supported by USB 3.0. In contrast, USB 2.0 has a speed of 480 Mbps.

All the servomotors and electronic components, including actuator 3, are connected to the CAN BUS, which interfaces with the multifunction board. The battery is connected to both the motherboard and the multifunction board to provide power to the system.

By the way, the two previous figures provided are simplified schematics. In reality, our robot is equipped with 25 servomotors, additional actuators, and other electronic components.

In the following subsections, we will define the roles of the motherboard and the multifunction card in more detail.

2.1.1 Motherboard

The first consideration in creating the robot was determining what would handle the computational aspect of the system. We needed a central element to manage the robot's data and make decisions, essentially serving as the robot's brain. This element should act as the intermediary between the user and the robot, allowing us to issue commands or instructions through the code we write. Moreover, it must be capable of managing and storing the information it receives from various parts of the robot, processing this data to extract useful insights. This central component must also be able to send commands to different parts of the robot as needed.

In order to select the motherboard that suits our needs, several requirements must be addressed. We particularly require an X86_64 motherboard, as software development is primarily conducted by students on this platform. We seek something simple and efficient, not overly complex to use.

Additionally, we operate within a constrained budget, and the more computational power we have, the faster the execution of various tasks, particularly image processing from cameras. However, this performance comes at a cost, as we also desire efficiency in power consumption due to the limited battery capacity of the robot.

Furthermore, we would like a USB connection to interface with the multifunction card and cameras. Given that the cameras transmit data at a rate of 1GB per second, USB3 seems to be the optimal choice. Storage capacity is another important aspect; the more storage we have, the better. Having a network interface would be beneficial, as we may wish to connect to the robot via SSH.

Based on these requirements, a suitable candidate emerged: the LattePanda Delta 3 [15]. It features an X86_64 architecture, an Intel quad-core processor capable of multithreading, WiFi and Ethernet interfaces, a USB 3 port, and two USB 2 ports. The USB 3 port will be utilized for the camera, enabling data transfer of up to 1GB per second, while one USB 2 port will be used for the multifunction card. The limitation of 480Mbps for the multifunction card's speed is not a significant concern.

In terms of power requirements, it operates on 12V, which is easily manageable. Comprehensive testing, including playing multiple videos simultaneously and running simplistic games, has confirmed the reliability and performance of the LattePanda, meeting the requirements of our project.

2.1.2 MultiFunction Card

Another essential element of the robot is the coordination of its numerous electronic components. It would be beneficial to have what could be called a conductor, managing these different electronic elements based on the data received from the motherboard. This component would also facilitate better coordination, ensuring that all electronic parts communicate through it, thereby centralizing the information. We could liken this component to the spinal cord of the robot, responsible for relaying information to each electronic element and receiving data from all of them to send to the motherboard. Without this component, the robot would not function properly. Thus, the idea of creating a multifunction board was conceived.

The multifunction card was developed by the electronics team to manage energy consumption and distribution among the various components of the robot. It regulates battery consumption and adequately powers different elements such as the motherboard, sensors, actuators, and vision modules.

It also incorporates an STM32F429 [16] card, a microcontroller featuring an ARM Cortex M4 processor. This component plays a crucial role in enabling data transmission. It facilitates communication with CAN busses [13], allowing for the sending and receiving of data from multiple servomotors, as well as communication with the motherboard to transmit, for instance, the robot's position based on its actuators.

It is worth noting that the servomotors are equipped with an STM32F3 processor, highly compatible with that of the multifunction card.

Chapter 3

Real-Time Operating Systems

3.1 Purpose

Achieving minimal latency within our technical infrastructure is imperative to fulfill the real-time demands inherent in our robot. Our focal point lies in optimizing kernel responsiveness, a pivotal aspect ensuring the reactivity and efficacy of our robotic operations. In other words, we need to minimize what we call latency. Latency, in this context, refers to the delay between issuing a command and the robot executing it. In a real-time operating system (RTOS) [17], minimizing latency involves ensuring that high-priority tasks are executed promptly, without unnecessary delays caused by lower-priority processes.

Within our system architecture, where the kernel on a motherboard is connected with the electronic card, which then engage with electronic components such as sensors, actuators, servomotors, and vision systems processing gigabytes per second of images, reducing latency is paramount. This reduction in latency guarantees swift processing and transmission of commands and data between these components, empowering our robotic system to function with minimal delay and maximum efficiency.

Furthermore, we will explore three real-time operating systems available to us, aiming to evaluate their effectiveness in minimizing kernel latency. Our primary focus is on Preempt-RT [18], RTAI [19], and Xenomai [20]. The idea is to change settings to optimize latency reduction. As articulated in the introduction, our objective is not kernel redesign but rather identifying options that influence latency and selecting values to minimize it. We also recognize that factors beyond kernel options, such as CPU allocation, can impact latencies.

Subsequently, we proceed to compile, install, and evaluate the performance of each kernel. Our evaluation strategy involves the utilization of testing tools such as cyclictest [21] across all three kernels, facilitating the measurement of kernel latency under diverse conditions. Additionally, we tailor tests to accommodate the unique characteristics of each kernel. For Xenomai , we utilize Latmus [22], the designated tool provided by its developers, while for RTAI, we rely on specific tests provided by the RTAI development team. Drawing from these comprehensive evaluations, we discern the most suitable solution for our project, prioritizing aspects such as response time predictability and efficient management of hardware resources.

Next, we select a distribution capable of accommodating this kernel. Considering our motherboard's storage capacity of 64GB, a crucial criterion is selecting a distribution with minimal space requirements.

Additionally, this distribution must be simple, efficient, and robust. Finally, based on these results, we determine the optimal kernel and distribution.

In summary, our methodology hinges on precise kernel optimization, centered on minimizing latency to guarantee the responsiveness and dependability of our robotic system in real-time scenarios, a crucial factor for enhancing system efficiency and ensuring temporal predictability. Typically, real-time operating systems maintain latencies on the order of a few microseconds, indicating their advanced efficiency. By prioritizing latency reduction and meticulously evaluating various real-time operating systems and kernels, we are poised to select the most optimal solution for our project's requirements.

3.2 Candidates

3.2.1 Preempt-RT

PREEMPT_RT [18], hosted at the Linux Foundation, is a patchset aimed at implementing a priority scheduler and other real-time mechanisms within the Linux kernel. While these patches are gradually being integrated into the mainline kernel, they are not yet fully upstream. The bulk of the PREEMPT_RT locking code was merged in Linux 5.15, signaling progress, but further work remains for complete integration.

Unlike other approaches such as Xenomai and RTAI, which utilize a 'co-kernel' alongside a soft Linux kernel, PREEMPT_RT distinguishes itself by directly impacting the kernel. Its objective is to achieve full preemption within the Linux kernel, encompassing critical sections while exempting only the most fundamental and critical code paths, such as entry code and low-level interrupt handling.

The primary aim of PREEMPT_RT is to subject all kernel-mode code to involuntary preemption, thus bringing most execution contexts under scheduler control. This is achieved through the replacement of locking primitives with variants that incorporate priority inheritance awareness, enforcement of interrupt threading, and the introduction of mechanisms to break up extended non-preemptible sections.

Furthermore, PREEMPT_RT replaces mutexes in the mainline Linux kernel with `rt_mutexes` to facilitate priority inheritance and mitigate priority inversion, enhancing kernel determinism. While PREEMPT_RT does not prioritize achieving the lowest latencies possible, unbounded latency would be considered a bug, as the patchset aims to render the Linux kernel preemptible, thereby ensuring deterministic response times.

Note about `rt_mutexes`: Real-time mutexes, or RT mutexes, are unique lock systems utilized in real-time operating systems. Their primary function is to counteract priority inversion, a situation where a task with a higher priority is obstructed by another task of lower priority that is currently holding a lock. To manage this, RT mutexes use a strategy known as priority inheritance. This strategy temporarily elevates the priority of the task holding the lock to align with the highest-priority task that is waiting, thereby guaranteeing execution that is both prompt and consistent.

3.2.2 RTAI

Real-Time Application Interface (RTAI) [19] is not a standalone kernel, but rather an extension of the Linux kernel that provides it with strict real-time capabilities. It does this by adding a layer between the hardware and the Linux kernel, effectively turning Linux into a fully preemptive real-time operating system. This means that RTAI can preempt the Linux kernel to ensure that real-time tasks are executed without delay. However, it is important to note that without RTAI, Linux operates as a standard operating system, not a real-time one. Therefore, RTAI is not a kernel, but an extension that transforms the Linux kernel into a real-time operating system.

This project was initiated in the late 1990s by Paolo Mantegazza and his team from the aerospace engineering department of the Polytechnic School of Milan.

RTAI is compatible with various architectures, including x86, PowerPC, ARM, and MIPS.

In technical terms, RTAI comprises a set of Linux kernel modules that offer real-time functionalities not available in a standard Linux environment. Once installed, RTAI remains dormant, initializing its control variables and capturing Linux's IRQ table addresses. When activated, RTAI assumes control of the machine, establishing its hardware clock handlers, intercepting hardware signals, and redirecting interrupts to its own handlers within a dedicated structure, the RTHAL.

At this point, Linux operates as a task managed by RTAI. RTAI ensures kernel preemption to meet the requirements of real-time applications. Additionally, RTAI provides extra modules to cater to the specific needs of real-time applications, such as shared memory management, FIFOs, semaphores, as well as a POSIX API and floating-point management.

This kernel extension was chosen for our project based on its successful use by a previous team participating in Eurobot.

However, despite being an open-source project, the latest version of RTAI was released on May 19, 2021, and is only compatible with Linux kernels up to version 4.19.

3.2.3 Xenomai

Utilizing a dual kernel architecture, Xenomai 4 [20] introduces real-time capabilities to Linux through its EVL core, following the footsteps of its predecessors in the Xenomai core series. This architecture incorporates a companion core into the kernel, dedicated to handling tasks with ultra-low and bounded response times to events. Termed as a dual kernel architecture, this methodology ensures stringent real-time assurances for select tasks while concurrently providing comprehensive operating system services to others. Within this framework, the general-purpose kernel and the real-time core operate nearly asynchronously, each catering to its designated set of tasks, with priority consistently accorded to the latter. As RTAI, Xenomai act as real-time extensions that add a layer to the Linux kernel. They are not standalone kernels themselves. Instead, they augment the Linux kernel with real-time capabilities.

The approach employed by Xenomai 4 with the EVL core offers an ingenious method of integrating real-time scheduling atop the Linux kernel without necessitating extensive modifications to support hard real-time scheduling. It achieves this by integrating a compact real-time kernel responsible for managing real-time tasks that function independently of regular tasks. This real-time kernel furnishes an alternative API tailored explicitly for hard real-time applications. Although standard "blocking" system calls are feasible, executing them triggers a fallback to non-real-time scheduling, thus preventing any potential stalling of the real-time kernel.

Xenomai 4 is a recently introduced kernel that has garnered positive reviews from multiple users. Being an open-source project, it benefits from active involvement by numerous contributors. However, while this collaborative effort offers several advantages, it may also introduce certain disadvantages that we will talk latter on.

3.3 Services Required

First of all, these three kernels must meet several specific needs we have. Firstly, it is essential for the operating system to allow us to define priorities for each task running on the robot. We also need operating system services that facilitate the use of certain libraries, such as the POSIX library. The ability to create periodic tasks and one-shot tasks, enable communication between tasks, and manage file writing for log journaling are indispensable operating system features.

We also need the operating system to provide a network interface for connecting via the SSH protocol. It is crucial that the operating system can utilize the four cores of the processor if necessary. Additionally, it is important that we can install a distribution of our choice and that these operating systems are simple, robust, and efficient.

Finally, the ability to use the USB port is essential, particularly for connecting to the multifunction board and cameras.

3.4 Installation

The idea is to provide a concise overview for each kernel through a diagram illustrating the installation steps. Subsequently, we will delve into a detailed discussion of each step involved in the kernel installation process. The installation of those kernel was performed on an x86_64 computer, using Fedora Workstation 39 as the distribution. During the distribution installation process, no libraries were pre-installed, enabling us to better identify the libraries required by the kernels. This allows us to provide the most comprehensive installation tutorial possible. Note that we used Fedora Workstation 39, as default for those kernels but a detailed explanation about the choice of distribution will be explained in the section 3.6

3.4.1 Preempt-RT

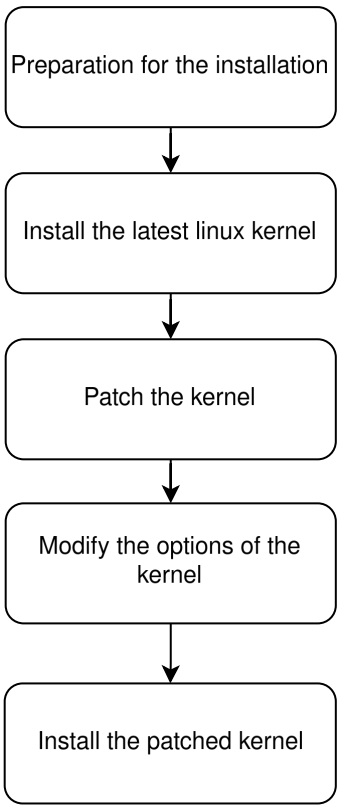


FIGURE 3.1: Overview of Preempt-Rt Installation

Preparation

During this initial step, several libraries and tools need to be installed to make the compilation of the kernel possible.

- `gcc, make`: These are necessary since the kernel is mostly written in C and compilation is managed using the make command.

- `libssl-dev`: Implements the SSL/TLS protocol and ensures the security of network communications. This library will be useful for the Download Kernel part.

- `libelf`: Provides tools for manipulating files in ELF format (Executable and Linkable Format), used particularly for executables and libraries in Unix systems.

- `wget`: Used to retrieve files from web servers via HTTP, HTTPS protocols. We will use it to download the kernel.

- `flex, bison`: Used to generate lexical and syntactic analyzers for programming languages or language description.

- `dwarves`: Manipulation and analysis of debugging files in Dwarf format, which is managed by the kernel during compilation.

- `gunzip`: Used to decompress .gz archives. This will be used during the extraction of the kernel and its associated patch.

Download Kernel

During this step, we utilize the `wget` command to access the original site containing the kernel archives. At the time of installation, the latest available kernel version was 6.5.2. Our objective is to install this latest kernel version and apply a patch that enhances its real-time functionality.

The `PREEMPT_RT` patch is a collection of modifications to the Linux kernel designed to enable certain processes to operate with a high level of real-time predictability. It achieves this by reducing the portion of kernel code that cannot be preempted by other processes. This involves converting spinlocks into sleeping spinlocks, implementing priority inheritance in mutexes, and executing interrupt handlers in their own threads, among other alterations. The outcome is a kernel capable of responding more swiftly and predictably to real-time tasks, making it well-suited for applications requiring precise timing and minimal latency.

Hence, we proceed to install kernel version 6.5.2 and apply the corresponding patch. It is crucial to ensure that the patch version matches the kernel version, in this instance, version 6.5.2. Once installed, we extract the kernel and apply the patch to it.

Configuration of the Kernel

The first step is to copy the `.config` file from the current kernel running on our computer (before installing the new one). This `.config` file allows precise customization and control over the kernel's features and configuration options based on the specific needs of the system it will run on. Next, we will execute a make oldconfig command, which will compile the kernel based on the `.config` file and present us with new options. It will be up to us to choose yes, no, or default for these options. It is crucial at this point to select yes for the Fully Preemptive Kernel option, which will enable our kernel to be real-time. Other new options can be left at default, accepting what the kernel proposes as default values. Alternatively, we can use the `make menuconfig` command to search for the Fully Preemptive Kernel option if preferred. The second step is to compile the kernel and modules and then install them.

Note: In Fedora, it is not necessary to update the initramfs, a temporary filesystem used by the Linux kernel during the boot process, nor the GRUB, a bootloader widely used in Linux-based operating systems. Appendix A, provides all the details of the installation.

The full installation of Preempt-rt by keeping only the modules and the initramfs file is: 600 MB.

3.4.2 RTAI

We first tested the RTAI kernel, by installing it in a Virtual Machine running Ubuntu. The reason we did not do this on Fedora is that it was difficult to install a 4.15 kernel on Fedora, whereas it was possible on Ubuntu. This is because every distribution comes with a kernel of a certain version installed. It is highly discouraged to downgrade from the base kernel version, as it leads to compatibility issues and kernel crashes, resulting in kernel panics. Therefore, we had to install a very old distribution, which is Ubuntu 18.04 LTS (Bionic Beaver), with a base kernel version of 4.15.

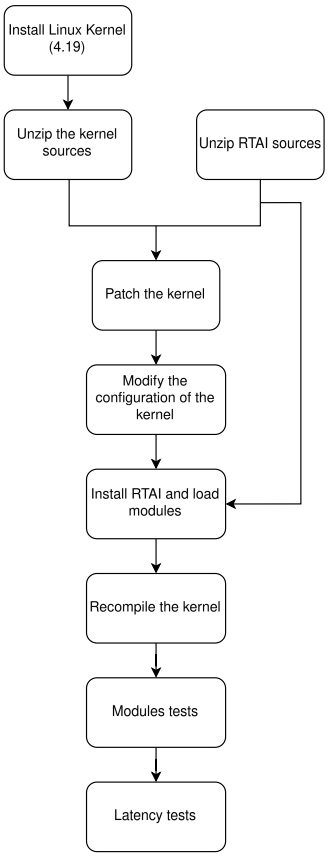


FIGURE 3.2: Overview of RTAI Installation

Preparation

The libraries to install here are similar to those of Preempt-RT, meaning that libssl and libelf will be required.

The tools required are the same as Preempt-RT: flex and bison.
However, two other libraries also need to be downloaded, which are:

- `Libncurses-dev`: programming library that provides an API for creating text-based user interfaces (TUIs). That is required by the `menuconfig`.

- `bc`: is a command-line calculator that provides arbitrary-precision arithmetic which is used during the compilation of the kernel.

Downloading the Kernel

In this context, we have opted for Linux kernel version 4.14.133 instead of the latest 4.19 version due to stability concerns. The 4.19 kernel exhibited instability during installation, leading to kernel panics. Furthermore, the patch provided by RTAI introduced compatibility errors with the 4.19 kernel.

The first step consists in download the RTAI 5.3 folder from the RTAI website. The next step is to download kernel version 4.14.133 from the official kernel website using the wget command, similar to the Preempt-RT installation process. Next, the downloaded folder must be placed in the `/usr/src/` directory. Similar to the Preempt-RT process, the kernel will then be patched. To achieve this, the patch file located in the RTAI folder need to be transferred to the linux folder and then patch the kernel with it. Finally, copy the `.config` file.

Kernel Configuration

Similar to Preempt-RT, the `.config` file must be copied. Once completed, the command `make menuconfig` must be run with some modified parameters, which have significant impacts on latencies. The selection of these parameters was based on documentation provided by RTAI and tests conducted by other RTAI users in similar configurations to ours. The following tutorial has been rewritten based on the source: [23].

- In the section `General setup`
  - Disable "Enable system-call auditing support" (`AUDITSYSCALL`) By disabling system-call auditing support, the kernel avoids the overhead of auditing system calls, such as maintaining audit logs and processing audit-related tasks.
  - The "Stack protector buffer overflow detection must be put to None (`CC_STACKPROTECTOR_NONE`) in order to execute all the tests of latencies correctly.
- In the section `Processor Type and Features`
  - Disable "Support x2apic" if present (`X86_X2APIC`). Disabling x2APIC support prevents the kernel from utilizing the advanced features of x2APIC, reverting to traditional interrupt handling mechanisms. While x2APIC can enhance interrupt handling performance in certain scenarios, it introduces additional complexity and overhead, which can impact latency.
  - Set the Maximum number of CPUs (`NR_CPUS`) as desired; it is optional.
  - Disable "Multi-core scheduler support" (`SCHED_MC`), this allow to activate the option for the CPU frequency scaling.
  - Enable "Interrup pipeline"(`IPIPE`). It essentially optimizes interrupt handling within the Linux kernel.

- In the section `Power Management and ACPI Options`
  - Disable ACPI Processor Support (`ACPI_PROCESSOR`. It prevents the kernel form dynamiccaly adjusting CPU frequecies and entering in power saving states.
  - Disable CPU Frequency Scaling (`CPU_FREQ`. Disabling CPU frequency scaling prevents the kernel from dynamically changing CPU frequencies, ensuring that the CPU operates at a fixed frequency.
  - Disable CPU Idle PM Support (`CPU_IDLE`. Disabling CPU idle power management support prevents the kernel from entering low-power idle states, keeping the CPU in an active state even during periods of inactivity.
- In the section `Device drivers`
  - In `Staging drivers`:
    * Disable Data acquisition support `COMEDI`. Disabling COMEDI support removes the overhead associated with the data acquisition framework from the kernel.
- In the section `Kernel Hacking`
  - In `Compile time checks and compiler options`:
    * Disable "Compile the kernel with debug info" (`DEBUG_INFO`). Disabling debug information compilation reduces the size of the kernel image by omitting symbols and debugging data.
    * Disable "Tracers" (`FTRACE`). Disabling kernel tracers eliminates the overhead associated with tracing and profiling kernel events.

The Appendix:A also explained how to modify manually the `.config` file (which is not recommended)

The next steps involve compiling the kernel and its modules for installation. Following this, the bootloader (GRUB) and initial ramdisk (initramfs), specific to the distribution, must be updated. Finally, the system needs to be rebooted.

Install RTAI

The first step consists to download the RTAI folder, in the `/usr/src/` repertory and then the `menuconfig` need to be lunched. Once on it several modifications and a verification should be done:

- In the `General` section the path to the linux tree should be the correct one.
- Under "Machine (x86) - Number of CPUs," the number of CPU's of our system should be written, the correct number of those can be found using the command `lscpu`.

Then RTAI and the modules need to be compiled. The next step is to update the library path and add RTAI binaries to the systems. In case of error after setting the number of CPU as explained before, one solution is to modify the grub file. Those information can be found in the AppendixA.

The full installation of RTAI keeping only the initramfs file and the modules is 2.4 GB.

3.4.3 Xenomai

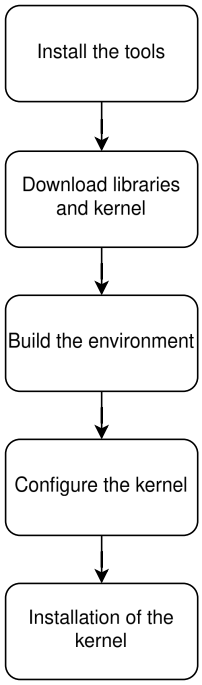


FIGURE 3.3: Overview of Xenomai Installation

Preparation

As for RTAI and Preempt-RT, some libraries and tools need to be installed:

- `meson`: Essential for configuring and compiling software projects, crucial for setting up Xenomai kernel.
- `ninja-build`: Used for efficient and fast building of software, necessary for Xenomai kernel compilation.
- `openssl`: Crucial for secure communication and cryptography, important for ensuring integrity in Xenomai kernel operations.
- `g++`: Essential for compiling C++ code, indispensable for Xenomai kernel development.
- `git`: Used to download the kernel.

Download the Kernel

Actually to download the kernel is by cloning the git of the open project Xenomai. Once downloaded, the folder that contains the kernel need to be renamed `linux-evl` and the one of library EVL as `libevl`

Pre-Configuration of the Kernel

This step consists in building the environment using `meson`. It streamlines the process of setting up the build environment and generating the necessary build scripts, ensuring a more efficient and organized development workflow.
It will install all compiled files and libraries in the folder Kernel, two folders will be created duing this compilation: `bin` and `tests` that will be useful later on.

Configuration of the Kernel

We run the command to have the configuration menu of the kernel. Once on it there are some options to change, in order that the kernel work well. The following tutorial has been rewritten from this source: [20].

- In the section `General Setup`
  - Disable `UAPI Headers`. The compilation process can be faster because it does not need to generate and process these additional header files and we indicates that the kernel is intended for internal use only and won't be providing user-space interfaces.
- In the section `Procesor Type And Features`
  - Enable `EVL`
    - The main component to be able to have the real time kernel.
  - Enable `EVL_LATMUS`
    - It is an option that enable us later to test the latencies of the kernel
  - Enable `EVL_HECTIC`
    - HECTIC aims to provide deterministic scheduling behavior for real-time tasks while supporting hierarchical scheduling structures and controlling the inheritance of scheduling parameters among tasks.
  - Enable `CONFIG_IKCONFIG`.
    - Enabling support for the ikconfig utility, which is used to extract kernel configuration information from a running kernel. When this option is enabled, the kernel configuration used to build the running kernel is embedded in the kernel image itself.

The `CONFIG_IKCONFIG` option is very important, when lunching EVL without it, it will not work correctly since he need to access to `/proc/config.gz` file without this option, an error appears: *evl-check:BROKEN.*

Installation

The next step is to compile the kernel and its modules, and then install them. After installation, a system reboot is required. However, launching EVL will not be possible initially. Since it is necessary to specify the path to the modules in the bash environment. A specific and detailed explanation can be found in the section *Command In Bash* in the Appendix A.

After installation, Xenomai occupies approximately 3 GB of disk space, which includes only the modules and the initramfs file, excluding compiled files.

3.5 Latency Analysis

One crucial aspect in the selection of the three kernels mentioned earlier is latency. Latency represents the duration between the moment when a task requests access to the processor and when it actually receives this access. This measurement is of paramount importance, especially the communication with electronic components, as previously mentioned.

Consider, for example, a scenario where a task requires updating the position of one servomotor. In such a situation, it is imperative that this action be performed within the shortest possible time frame. Any delay could lead to disastrous consequences, such as the robot falling or missing an action.

3.5.1 Tools Used

Regarding the tools we will utilize, firstly, we will employ `cyclictest`, a kernel independent library. This tool will be applied to all kernels, providing a consistent basis for our evaluations. Then, for certain kernels, we will utilize their own integrated latency measurement tools. Lastly, we will implement custom code that we have developed to test independently of any tools the latencies of each kernel.

Cyclictest

Cyclictest [21] performs precise and repetitive assessments of the time discrepancy between a thread's scheduled wake-up and its actual wake-up, offering insights into the system's latency characteristics. It has the capability to detect latencies originating from hardware, firmware, and the operating system in real-time environments.

Limitations: It is important to note that the latencies recorded by Cyclictest may exhibit a slight bias depending on the nature of the eventual application running on the system. This stems from the fact that Cyclictest employs nanosleep (sleeping threads voluntarily for a period time in nanoseconds) for its measuring threads, thereby minimizing their wake-up time to the shortest duration feasible for a real-time userspace task. In contrast, other application tasks may experience longer wake-up latencies, especially if their wake-up process is not executed directly within hard interrupt context (e.g., threaded interrupt context). This disparity introduces an optimistic bias in the latency measurements of Cyclictest, as the measuring threads operate with a level of efficiency not always replicated in real-world application scenarios.

How are latencies measured by Cyclictest ? This following section has been rewritten from this source [21].

Cyclictest analyzes latencies utilizing a specific method. It works utilizing the main elements and methods listed below:

1. *Master Thread:* A non-real-time master thread that is a member of the *SCHED_OTHER* scheduling class is used by Cyclictest. The latency measurements are coordinated by this master thread.
2. *Threading Measurements:* The master thread starts a predetermined number of threads mesuration that follow the priority scheduling class of the real-time kernel.
   `SCHED_FIFO`. Latencies are detected and recorded by these measuring threads.
3. *Wake-up Using Timer:* A cyclic alarm, that is a timer expiring at periodic intervals, wakes up measuring threads on a regular basis.
4. *Calculating Latency:* Each measuring thread computes the gap between the actual effective wake-up time and the programmed wake-up time once it has woken up. The measurement thread's delay is represented by this difference.
5. *Shared Memory*: The master thread is then informed of the computed latency values via shared memory. Efficient data exchange between threads is made possible through it leading to data exchanges faster. Also shared memory are used in this case to avoid inter-process communication that might increase the latency too.
6. *Reporting and Tracking Latencies:*The latency readings that the measuring threads provide are monitored by the master thread. It logs the average, maximum, and minimum latencies. The default behavior prints the results after each iteration; with the "*-quiet*" option, the results are printed after a predetermined number of iterations.

To accurately measure latencies, five specific options have been selected. Additional options and their descriptions are available using the command `cyclictest -h` [21].

Here is the command and the description of the selected options:

```
sudo cyclictest --mlockall --smp --priority=80 --interval=1000
--distance=0
```

- `mlockall`: Pages are not page out of memory.
- `smp`: Allow symmetric multiprocessing system.
- `priority`: Priority of the thread that is running.
- `interval`: Period desired.
- `distance`: Distance of thread interval.

The `sudo` command is employed because these operations entail configuring system-level parameters that are restricted to privileged users. By granting elevated privileges, cyclictest can accurately measure and report kernel latencies without being impeded by standard user-space restrictions, resulting in more precise and dependable outcomes.

Latmus

EVL has a tool called latmus [22] that allows users to gauge how quickly an interrupt occurs in various scenarios.

The latmus tool assists in locating possible lags that could occur between when a device raises an interrupt request and when a responder thread operating in the application space has a chance to respond to it.

Note that Latmus is specifically designed for use with the EVL core, namely Xenomai 4. It is not compatible with the other two kernels.

Personal Code

For each of the previously mentioned kernels, we will develop code tailored to their specifications to measure latency. The implementation specifics will be outlined in the following section 3.5.6, along with the resulting measurements.

3.5.2 Isolation of CPU

The latency of a kernel can be significantly improved by isolating a CPU core solely for kernel operations. This isolation minimizes interference from other processes and system tasks, allowing the kernel to execute more efficiently. We will observe the latency results obtained both with and without isolating the CPU to illustrate the impact of this optimization. The mechanism of the isolation of the CPU can be found in the A. In the following array, the parameters meaning are:

- `T`: Thread ID or Task ID, indicating the specific thread or task being monitored.
- `P`: Priority of the thread or task.
- `I`: Interval in microseconds at which the thread or task is being executed.
- `C`: Count of iterations or cycles executed.
- `Min`: Minimum latency observed during the execution of the thread or task, measured in microseconds.
- `Act`: Actual latency observed during the execution of the thread or task, measured in microseconds.
- `Avg`: Average latency observed during the execution of the thread or task, measured in microseconds.
- `Max`: Maximum latency observed during the execution of the thread or task, measured in microseconds.

.

3.5.3 Preempt-RT

- For Preempt-RT with isolated CPU core using Cyclic test over 9 hours, the results were:
  - T: 0 (2992) P:80 I:1000 C:33923445 Min: 2 Act: 2 Avg: 2 Max: 43
  - T: 1 (2993) P:80 I:1000 C:33923443 Min: 2 Act: 2 Avg: 3 Max: 47
  - T: 2 (2994) P:80 I:1000 C:33923440 Min: 2 Act: 2 Avg: 2 Max: 42

- For Preempt-RT without isolated CPU core using Cyclic test over 9 hours, the results were:
  - T: 0 (2992) P:80 I:1000 C:33324685 Min: 2 Act: 2 Avg: 2 Max: 94
  - T: 1 (2993) P:80 I:1000 C:33324683 Min: 2 Act: 2 Avg: 3 Max: 49
  - T: 2 (2994) P:80 I:1000 C:33324680 Min: 2 Act: 2 Avg: 2 Max: 45
  - T: 3 (2994) P:80 I:1000 C:33324676 Min: 2 Act: 2 Avg: 2 Max: 48

3.5.4 RTAI

- For RTAI with isolated CPU core using CyclicTest test over 8 hours, the results were:
  - T: 1 (3678) P:80 I:1000 C:30395361 Min: 1 Act: 4 Avg: 3 Max: 95
  - T: 2 (3679) P:80 I:1000 C:30395359 Min: 1 Act: 6 Avg: 7 Max: 86
  - T: 3 (3680) P:80 I:1000 C:30395356 Min: 1 Act: 2 Avg: 3 Max: 89

- For RTAI without isolated CPU core using CyclicTest test over 8 hours, the results were:
  - T: 0 (3677) P:80 I:1000 C:30235415 Min: 1 Act: 2 Avg: 3 Max: 159
  - T: 1 (3678) P:80 I:1000 C:30235412 Min: 1 Act: 4 Avg: 3 Max: 85
  - T: 2 (3679) P:80 I:1000 C:30235409 Min: 1 Act: 6 Avg: 7 Max: 86
  - T: 3 (3680) P:80 I:1000 C:30235405 Min: 1 Act: 2 Avg: 3 Max: 79

Based on the test provided by RTAI in the testsuite folder, we achieved a minimum latency of 10 ns and a maximum latency of 180 ns. With CPU core isolation, the minimum latency improved to 4 ns, while the maximum latency decreased to 135 ns.

3.5.5 Xenomai

- For Xenomai with isolated CPU core using CyclicTest test over 8 hours, the results were:
  - T: 1 (2653) P:80 I:1000 C:29263417 Min: 2 Act: 4 Avg: 27 Max: 101
  - T: 2 (2654) P:80 I:1000 C:29263415 Min: 2 Act: 14 Avg: 14 Max: 90
  - T: 3 (2655) P:80 I:1000 C:29263412 Min: 2 Act: 3 Avg: 6 Max: 78

- For Xenomai without isolated CPU core using CyclicTest test over 8 hours, the results were:
  - T: 0 (2770) P:80 I:1000 C:29063314 Min: 3 Act: 6 Avg: 4 Max: 162
  - T: 1 (2771) P:80 I:1000 C:29063312 Min: 3 Act: 10 Avg: 14 Max: 117
  - T: 2 (2772) P:80 I:1000 C:29063309 Min: 3 Act: 5 Avg: 14 Max: 153
  - T: 3 (2773) P:80 I:1000 C:29063305 Min: 3 Act: 8 Avg: 5 Max: 160

For Latmus the paramaters meanings are:

- `RTH`: Real-Time Histogram, indicating the type of measurement.
- `lat min`: Minimum latency observed during the execution.
- `lat avg`: Average latency observed during the execution.
- `lat max`: Maximum latency observed during the execution.
- `overrun`: Number of times tasks overrun their allotted time.
- `msw`: Maximum service time window, indicating the maximum time allowed for execution.
- `lat best`: Best-case latency observed during the execution.
- `lat worst`: Worst-case latency observed during the execution.

With Latmus and one CPU core isolated we obtained:

RTT| 07:29:46 (user, 1000 us period, priority 98, CPU1)

RTH	lat min	lat avg	lat max	overrun	msw	lat best	lat worst
RTD	−0.139	3.031	10.828	0	0	−0.335	87.30

With Latmus and no CPU core isolated we obtained: RTT| 07:29:46 (user, 1000 us period, priority 98, CPU0)

RTH	lat min	lat avg	lat max	overrun	msw	lat best	lat worst
RTD	−0.249	6.132	12.948	0	0	−0.555	114.20

3.5.6 Latency Using C Implementation

For each kernel, specific mechanisms need to be employed according to the kernel utilized. In the following sections, we will investigate the parameters to be configured for each kernel and present the corresponding results obtained. The code for measuring the latency is available in the GitLab repository [24]. The code is situated in the folder `Part 1/Latencies`. We have three files in it, one called `preempt-lat.c`, one called `rtai-lat.c`, and one called `evl-lat.c`, where the role of these three files is to measure the minimum, average, and maximum latencies, with two threads running in parallel and performing a simple task of incrementing a counter. Of course, mechanism that ensure mutual exclusion have been implemented. The folder also contains their header files: `evl-lat.h`, `preempt-lat.h`, and `rtai-lat.h`, along with a makefile and a README detailing the implementation and the commands that can be executed with the makefile.

Preempt-RT

The code `preempt-lat.c` implements a program that creates two threads, each representing a task that accesses a shared resource protected by a semaphore. The program measures the latency of resource access for each thread, recording the maximum, minimum, and average latencies.

The program initializes a semaphore to control access to the shared resource. Each thread, upon acquiring the semaphore, increments a counter to simulate work being done and then releases the semaphore. Latency is measured by capturing timestamps before and after acquiring the semaphore.
The program prints the latency for each thread's resource access and updates the maximum, minimum, and total latency values. After both threads complete their iterations, the program calculates and prints the maximum, minimum, and average latency values.

In the code `preempt-lat.c`, several options have been set up:

- We set the CPU1 only for the kernel by using the command `CPU_ZERO` and `CPU_SET`
- the `pthread_setaffinity_np` command is used to set the CPU affinity of a thread.
- the `pthread_setschedparam` command is used to set the scheduling of the thread, here its SCHED_FIFO, because threads have higher priority than other non-real-time tasks, such as those scheduled under SCHED_OTHER

The rest of the code calculates the maximum latency, the average latency and the best latency.

Here we use only two threads and the results that we obtain are:

- With CPU core isolation in preempt-rt and 1 million iterations:
  - Maximum latency: 11912 ns
  - Minimum latency: 40 ns
  - Average latency: 42 ns
- Without CPU core isolation and 1 millions iterations:
  - Maximum latency: 30635 ns
  - Minimum latency: 11 ns
  - Average latency: 42 ns

Note that to obtain the results without CPU core isolation, is to comment out the lines that assigns the CPU1 to the kernel.

Xenomai

The code `evl-lat.c` implements a program that creates two threads, each executing a task concurrently. The tasks involve acquiring and releasing a shared resource protected by semaphores, ensuring mutual exclusion. The program measures the execution time of each task and prints it to the console. It utilizes the EVL real-time extension for Linux, which provides capabilities for real-time scheduling and synchronization. Each task involves a loop where the shared resource is accessed, and the execution time is measured using the EVL clock. After both threads complete their tasks, the program calculates and prints the maximum, minimum, and average execution times. This program demonstrates real-time multitasking and synchronization using EVL in a Linux environment. Actually we measure the latences for 1 million iterations and we get the minimum, maximum and average latencies. As for Preempt-RT we have only two threads running in the code.

The functions used in the code using EVL mechanism are:

- `evl_get_sem(&sem)`: This function is used to acquire a semaphore.
- `evl_put_sem(&sem)`: This function is used to release a semaphore.
- `evl_create_sem(&sem, EVL_CLOCK_REALTIME, 1, EVL_CLONE_PRIVATE ,"my_sem")`: This function creates a semaphore. It initializes a semaphore structure (`sem`) and associates it with a semaphore object in the EVL subsystem. Parameters passed include the semaphore structure, the clock source (`EVL_CLOCK_REALTIME` for real-time clock), initial semaphore value (1 in this case, indicating it is available), a flag for private semaphore (`EVL_CLONE_PRIVATE` means the semaphore is private to the caller's process), and a name for the semaphore.
- `evl_close_sem(&sem)`: This function closes a semaphore. It deallocates the resources associated with the semaphore.

- `evl_attach_self("task1")` and `evl_attach_self("task2")`: These functions attach the calling thread to an EVL real-time thread with a specified name. This association allows the thread to be managed by the EVL scheduler.

- `evl_detach_self()`: This function detaches the calling thread from the EVL real-time thread it was previously attached to. This is typically done after the thread has completed its real-time tasks.

- `evl_create_sem(&sem, EVL_CLOCK_REALTIME, 1, EVL_CLONE_PRIVATE, "Sem")`: Similar to the `evl_create_sem(&sem, EVL_CLOCK_REALTIME, 1, EVL_CLONE_PRIVATE, "my_sem")` function, this creates a semaphore with a different name.

The results that we obtained are:

- With CPU core isolation and 1 million iterations:
  - Maximum latency: 130596 ns
  - Minimum latency: 48 ns
  - Average latency: 67 ns

- Without CPU core isolation and 1 million iterations:
  - Maximum latency: 113915 ns
  - Minimum latency: 48 ns
  - Average latency: 125 ns

RTAI

The code located in `rtai-lat.c` implements a program that creates two real-time tasks using RTAI (Real-Time Application Interface) to access a shared resource protected by a semaphore. The program measures the latency of resource access for each task, recording the maximum, minimum, and average latencies. The program initializes a semaphore to control access to the shared resource and creates two real-time tasks using RTAI. Each task, upon acquiring the semaphore, increments a counter to simulate work being done and then releases the semaphore. Latency is measured by capturing timestamps before and after acquiring the semaphore. Thread scheduling and priority settings are configured using RTAI functions to control task execution behavior.

- `rt_sem_wait(&resourceSemaphore)`: This function is used to acquire the semaphore.

- `rt_sem_signal(&resourceSemaphore)`: This function is used to release the semaphore.

- `rt_task_shadow(NULL, "Task")`: This function sets the task to run in shadow mode, ensuring that the task runs as a real-time task.

- `rt_task_set_priority(nam2num("T1"), 99)` and `rt_task_set_priority (nam2num("T2"), 100)`: These functions set the priority of the tasks. In this code, the task with thread ID 1 is set to a higher priority (99) than the task with thread ID 2 (100).

- `rt_task_init(&thread1, task, 1, 1024, 99, 0, 0)` and `rt_task_init( &thread2, task, 2, 1024, 100, 0, 0)`: These functions initialize the real-time tasks. They specify the task function, task argument, stack size, priority, and other parameters.

- `rt_task_resume(&thread1)` and `rt_task_resume(&thread2)`: These functions resume the execution of the real-time tasks.

- `rt_task_join(&thread1)` and `rt_task_join(&thread2)`: These functions wait for the real-time tasks to complete their execution.

- `rt_sem_delete(&resourceSemaphore)`: This function deletes the semaphore after it is no longer needed.

The results that we obtained are:

- With CPU core isolation and 1 million iterations:
  - Maximum latency: 196852 ns
  - Minimum latency: 126 ns
  - Average latency: 102 ns
- Without CPU core isolation and 1 million iterations:
  - Maximum latency: 222456 ns
  - Minimum latency: 224 ns
  - Average latency: 192 ns

To conclude, based on the results of all latency measurements, we found that the Preempt-RT kernel performed the best in terms of latency, followed by Xenomai, and then RTAI. Given the critical importance of low latency in our project, Preempt-RT has earned our trust as the most reliable option among the tested real-time solutions. This conclusion is supported by the tools we used and the custom code we developed for these tests.

3.5.7 Our Choices

Based on these results, we face two decisions: one regarding the distribution and another regarding the real time kernel to be chosen. To aid in this decision-making process, the following talk outline the advantages and disadvantages of each presented solutions.

	Preempt-RT	RTAI	Xenomai
Advantages	• Better latencies than other options • Easier to install and update as it is implemented by Linux • Comes with a set of tools for latency management	• Performs well in previous competitions • Low disk space requirement • Provides a set of tools for latency management	• Second place in latencies • Recently open-sourced, increasing community support • Provides tools for latency management
Disadvantages	• Latest version has poor latencies, necessitating use of previous versions	• Complex installation process • High number of parameters to configure • Heaviest kernel to install • Core issues may lead to kernel freezes • Poor worst-case latencies • Incomplete documentation	• Installation based on outdated kernel • USB functionality not supported • Frequent kernel freezes • Poor latencies even with provided tools • Complex installation process • Frequent kernel panics during installation

TABLE 3.1: Comparison of Real-Time Operating Systems

Based on the results obtained from our latency measurements, selecting Preempt-RT appears to be the best choice. Despite considering the various advantages and disadvantages of each kernel, Preempt-RT outperforms the others. Our primary concern is latency, and Preempt-RT consistently delivered the best results. Additionally, Preempt-RT is up-to-date, easier to install, and provides services that align well with our requirements.

Xenomai is the second-best option according to our measurements. While it currently lags behind Preempt-RT, we believe that Xenomai has the potential to surpass it in the coming years. Xenomai 4 is relatively new and still being actively developed by the community, which may explain its current performance.

On the other hand, RTAI came in last. It is no longer actively maintained by developers and uses an outdated kernel version (e.g., version 4) compared to the version we are currently using (e.g., version 6). Additionally, the lack of USB support is a critical issue for us since we rely on USB for some electronic components. Moreover, the latency values obtained with RTAI were not satisfactory, and the installation process was tricky.

3.6 Distribution Considerations

3.6.1 Criteria of Selection

In order to install the Preempt-RT kernel on the motherboard, we need to find a distribution to host this kernel.

The goal is to install a lightweight distribution on the motherboard, taking into account several criteria:

- **Distribution size**: Although the motherboard has 64 GB, it is desirable to preserve this space for other software development needs of the robot.
- **Simplicity**: The distribution should be easy to use, while avoiding excessive complexity.
- **Maintenance**: It is essential to choose a distribution that is regularly maintained and updated, to benefit from the latest improvements and bug fixes.
- **Stability**: The stability and reliability of the distribution are crucial to ensure the proper operation of the robot under various conditions.
- **Ease of Installation**: Evaluate the installation process to determine whether the distribution was straightforward to install without encountering any significant issues.

And of course, it must be able to install the choose real-time solution on this distribution.

3.6.2 Candidates

Lubuntu

Lubuntu [25] is described as a lightweight Linux distribution built on the Ubuntu base. It is user-friendly with a graphical interface while conserving system resources. Lubuntu is known for its simplicity and efficiency, receiving positive feedback from users with diverse needs. A simple installation of Lubuntu requires 4.3 GB of disk space.

Linux Voyager

Linux Voyager [26] is described as an Ubuntu/Debian-based distribution. As a lightweight functional operating system, it features small desktop environments like LXQt, targeting low-end hardware. It provides a user-friendly environment and is reputed for its simplicity and stability.

Puppy Linux

Puppy Linux [27] emerged as a notable option while searching for a small distribution. It requires around 300MB, is user-friendly, and highly customizable. It also comes with hundreds of derivatives called `puplets`.

Arch Linux

Arch Linux [28] is renowned for its minimalistic simplicity and being an up-to-date distribution. It utilizes a rolling release system that continuously updates the distribution with the latest GNU/Linux features, including the newest kernels. The Arch Linux community is highly active and allows users to customize their systems extensively. It offers high-quality packages managed by Pacman, the package manager.

Yocto

Yocto [29] was launched with the aim of creating custom Linux-based systems for embedded systems, independent of hardware constraints. The concept is to provide a toolkit developed by embedded device developers and build distributions using these tools. However, they also acknowledge that learning how to customize the distribution can be overwhelming for newcomers, as Yocto requires modifications and configurations for adding new packages. Consequently, developers may need to adapt to cross-build environments, which could be unfamiliar to some.

Fedora

Fedora [30] is a well-known Linux distribution developed by Red Hat, recognized for its robustness and up-to-date operating system. As an open-source project, it benefits from a large and active community of contributors. Fedora releases updates typically every six months, ensuring that users have access to the latest packages and security upgrade.

The Fedora community actively encourages user feedback to continually improve the distribution, such as through bug reporting. Fedora offers multiple desktop environments, each catering to different user preferences and needs. It is also renowned for its commitment to privacy and the comprehensive control it provides to its users.

By the way, Red Hat [31] is a software company specializing in operating systems, offering open-source software for businesses. It also provides cloud technologies, storage solutions, and operates in the domain of cybersecurity. This company is trusted by many organizations worldwide, making it a leader in the enterprise Linux market.

3.6.3 Our Choice

To make our choices we have based on the criteria said before:

- **Lubuntu** takes up 20 GB, faced initramfs issues, and encountered compatibility problems.
- **Linux Voyager** combining XFCE with Gnome, consumed 15GB of space.
- **Yocto** was difficult to install, and its source code did not compile without errors.
- **Puppy Linux**: operate solely in RAM. This suits not our needs, taking into account the accessibility of hard disk resources. Note that unless we manually save changes to a persistent storage device at the end of each session, modifications will not be retained. Additionally, allocating a significant portion of RAM by the distribution can result in performance degradation, particularly when executing our code. Therefore, it may be beneficial to leverage some gigabytes of our hard disk.
- **Arch Linux** was rudimentary to use and install, requiring significant adaptation time. The lack of a graphical installer made it less user-friendly, as every configuration had to be done through the terminal, including disk partitioning, bootloader setup, and file system configuration. This distribution did not come with a default installation, meaning each configuration had to be done manually, even for network settings. However, it included great documentation and a helpful community, with up-to-date releases. Despite offering the choice to install it, the configuration process was very difficult to set up, and troubleshooting issues within the distribution could be challenging.
- **Fedora**'s installation process was notably straightforward. We selected the minimal installation option from the graphical installation menu, allowing us to choose which libraries to pre-install. At present, no libraries have been selected during the pre-installation. By libraries, we mean, for example, Fedora may propose installing the network-manager packages or the build-essential package composed of several libraries and tools such as gcc, make, etc.

  Furthermore, Fedora demonstrated user-friendliness and strong support from the open-source community. Its usage proved to be intuitive, with Fedora showing resilience in managing both custom kernel installations and kernel updates. Notably, the minimal installation, devoid of graphical interfaces like GNOME, only occupies 1GB of disk space. This kernel has been used to install Xenomai and preempt-rt, and it works perfectly fine. The installation took around 1GB of disk space.

Therefore, we opted for Fedora. This distribution has proven its reliability in installing Xenomai and Preempt-RT, functioning seamlessly. While Debian distributions were attempted, they often resulted in kernel panics during updates to the initramfs file. This was due to their restricted nature, limiting experimentation with non-official kernels. In contrast, Fedora, stemming from Red Hat, fosters an environment conducive to such exploration. Kernels operations are easier on Fedora. Then the distribution is user friendly taking less than 1GB of space.

3.7 Optimization

Several optimizations can be implemented to improve system performance:

- **Reducing kernel module size:** In the case of Preempt-RT, kernel modules consume 5.5GB of disk space, but they can be compressed to 560 MB. Detailed instructions are provided in AppendixA for instructions.
- **Grub default selection:** Modify the Grub configuration file to specify the default kernel to boot, enabling direct booting into Preempt-RT, for example.
- **Limit kernel installation by default:** Adjust the configuration file of the distribution to restrict the number of installed kernels by default after each update or upgrade executed in the terminal.
- **Remove old kernels:** Delete unused old kernels, keeping in mind that Fedora retains a backup kernel for emergency situations such as kernel panics. This backup kernel cannot be deleted.
- **Libraries management:** Remove libraries that were installed for kernel compilation, and avoid choosing pre-installed libraries in the installation menu of the distribution.

Commands related to the first three optimizations are given in the Appendix A.

3.8 Conclusion

In this chapter, we have analyzed the potential kernel options and distributions available. Based on various tests conducted for latencies and other factors such as installation size and simplicity, Preempt-RT emerged as the preferred kernel, while Fedora stood out as the favored distribution. Consequently, we have determined that our motherboard will run the Preempt-RT kernel paired with Fedora as the distribution. However, it is important to note that this decision is not final; there the possibility of switching to a more recent kernel or distribution in the future. For instance, Xenomai 4, despite being in development, could some day emerge as a competitor to Preempt-RT, offering potentially better latencies or other advantages. Similarly, alternative distributions like Linux Voyage might surpass Fedora in performance. Notably, in previous competitions, Linux Voyage and RTAI were contenders, whereas now, Fedora and Preempt-RT are the leading choices. In summary, while we have identified our current candidates, it is conceivable that this may change in the coming years. With our motherboard configuration established, the next chapter will delve into establishing communication between this motherboard and the multifunction card.

Chapter 4

Communication with the Multifunction Board

4.1 Introduction

This chapter is oriented towards the STM32F4 [16] multifunction board, the central component of the multifunction board. As a reminder, the multifunction board's primary role is to provide a communication channel between the motherboard and the other electronic components of the robot such as the servomotors.

For instance, if a motherboard task requires data from an actuator to update its position, send the corresponding order to the multifunction board which then relays it to the actuator.

However, for this exchange to occur, the STM32F4 board and the motherboard must be physically connected and use a common communication protocol. We will use a USB connection, as the motherboard has three USB ports available, making it a logical choice. Several challenges need to be addressed to enable communication with the multifunction board. First, we must store data on the multifunction card to send messages or commands and execute code on it. Second, we need to receive data from the multifunction board through the USB connection. For this to happen, the multifunction board must be detected by the motherboard and allow the motherboard to write to a specific file on it.

Subsequently, the focus shifts to establishing communication between these components via USB. This chapter explores enabling communication through this interface. Specific libraries for the STM32F4 processor will be useful for implementing the code required for this communication with the motherboard. We will then examine various flags to achieve optimal communication.

Lastly, a brief test will assess data transfer speed and overall system functionality.

4.2 Communication With the Hardware

To enable communication between the multifunction card and the motherboard, we need to establish a USB connection between these two components. The challenge lies in developing code that operates on the motherboard to open the serial port, read from it, and write to it. This is necessary because the code from libopencm3 is designed for the STM32F429I board, not the motherboard. But first, let's define what USB is, the available classes, and the one we are going to choose.

4.2.1 Principles of USB

The Universal Serial Bus (USB) [32], invented in the 1990s, establishes communication protocols used over a bus between computers and electronic devices. Its primary purpose is to standardize connection methods. USB serves not only for communication but also for power supply. USB is a serial communication standard, meaning it transmits bits one by one over a single communication line, making it efficient and straightforward.

Four transfer types are defined:

- `Control`: Allows data transfer upon request.
- `Bulk`: Used for connecting hard drives, enabling data transfers without guaranteed delivery rates.
- `Interrupt`: Used with devices like mice to periodically transfer data.
- `Isochronous`: Primarily for streaming applications, such as displays.

In our case, an `Interrupt transfer` would be suitable for periodically transferring data with the multifunction card, especially from sensors.

USB devices are categorized into several classes to standardize their behavior across different electronic devices:

- `MSC (Mass Storage Class)`: Used for hard drives, facilitating fast file access.
- `HID (Human Interface Device)`: Enables user interaction with accessories like keyboards and game controllers.
- `Audio and Video`: Supports audio and video applications.
- `Battery Charging Class`: Manages power supply.
- `Hub Class`: Allows connecting multiple USB devices via a single USB port.

Our focus will be on the `CDC (Communication Device Class)`, which the STM32F4 board utilizes.

The CDC class handles data communication with devices and controls data transfer. It includes subclasses, with the relevant one being the `ACM (Abstract Control Model)`. ACM represents serial line control, allowing our board to connect via a serial port to the motherboard. This emulates the `RS-232` port (Recommended Standard 232) over USB, supporting speeds from 300 bit/s to 115200 bit/s for bidirectional bit-by-bit communication.

4.2.2 Library

Two library can help us to make our need satisfied:

- `LibOpenCM3`
  - LibOpenCM3 [33], formerly known as libopenstm32, is a freely available firmware library designed for ARM Cortex-M3 microcontrollers, including ST STM32 and more. It will facilitate setting up the connection between the STM32F429 board and the motherboard.

Another tool worth considering is:

- `STM32Cube`
  - STM32Cube [34], offered by STMicroelectronics, is a comprehensive development software for STM32 microcontrollers and microprocessors. It includes a suite of tools and embedded software libraries to streamline development, from configuration to code generation. However, it is not utilized due to its substantial installation size (up to 4GB), instability, and uncertainty regarding generated code functionality, which conflicts with the necessity for precise understanding and control in competition and development scenarios.

Therefore we decided to choose the LibopenCM3 library.

4.2.3 Implementation

To establish a CDC class communication between our motherboard and the multifunction card and to use the ACM subclass, we can utilize the functions provided by **libopencm3**, which have already been implemented for us.

In this section, we will explain in detail the code provided by libopencm3. The library offers an implementation for setting up the communication and ensuring the motherboard recognizes the board. All the code is available in the git repository [24]. If we look to the the `main.c` file located in `Part_2/stm32f429-USB_CDC/echo`, we can see the implemented code. The `main.c` file is the same as the one from the `libopencm3 library` [35]. The only modifications are in lines 202-212 and line 258, where the USB string name is defined and the function in `echo.h`, that will be explained in the subsection 4.4

The `main.c` code is for a USB communication device class (CDC) application, specifically an abstract control model (ACM). It is designed to run on an STM32F4 microcontroller, using the libopencm3 library for hardware abstraction.

The code typically starts with a series of #include directives, which instruct the preprocessor to include the contents of the standard libraries as well as libraries from the libopencm3 project, which provide functions and definitions for working with the STM32F4 microcontroller and USB communication.

Following this, the code defines a series of USB descriptors. These descriptors provide information about the USB device, its configurations, and its interfaces to the host computer.

For example, the `usb_device_descriptor` structure provides basic information about the device, such as its vendor ID, product ID, and maximum packet size. The `usb_config_descriptor` and `usb_interface_descriptor` structures provide information about the device's configurations and interfaces, respectively.

The `cdcacm_control_request` function is used to handle control requests sent from the host to the device. This function is called when the host sends a control request to the device, and it handles specific requests defined in the USB CDC specification.

The `cdcacm_data_rx_cb` function is a callback function that is called when data is received from the host. This function reads the received data into a buffer and then sends the transformed data back to the host.

The `cdcacm_set_config` function is called when the host sets the device's configuration. This function sets up the device's endpoints and registers the control request callback function.

The `setup_main_clock` and `setup_peripheral_clocks` functions are used to set up the microcontroller's main clock and peripheral clocks, respectively. These functions use functions from the libopencm3 library to configure the microcontroller's clock system.

Finally, the main function initializes the USB device, registers the configuration callback function, and enters a main loop where it continually polls the USB device. This polling allows the device to process USB events and handle data transmission and reception.

4.3 Serial Communication Library

In the previous section, we successfully set up the communication device class. However, another challenge arises: how to send and receive data between the motherboard and the multifunction card, and vice versa. To accomplish this, we need to be able to read from and write to the `/dev/ttyACM0` device path, which facilitates USB communication.

For instance, to send data from the motherboard to the multifunction card, the code opens the serial port on `/dev/ttyACM0`, writes the data to it, and the multifunction card reads from it. Conversely, if the multifunction card sends data to the motherboard, it will be written to this device path, and the motherboard will read from it.

To met this requirement a code have been implemented and he is located in the `Part 2/stm32f429-USB_CDC/PC part`. We can found the `serial-com.c` that contains the functions to open the serial port, write data and read data from it and close the port. There is also the `serial-com.h` that is the header file of the previous file. And also the `main.c` that is a test file that will be explained in the section 4.4.

So in this section the code that interest us is the `serial-com.c`.

4.3.1 Library

We have included several preprocessor directives in our program to enable various features and libraries, the macros define important parameters, such as buffer size and baud rate.

```
#include <stddef.h> // Provides definitions for common types and
    macros, such as size_t.
#include <stdio.h>  // Provides input/output functions, such as
   printf and scanf.
#include <string.h> // Provides functions for manipulating
   strings
#include <fcntl.h>  // Provides functions for manipulating file
   descriptors, such as open and close.
#include <errno.h>  // Provides error number definitions and
   functions for handling errors.
#include <termios.h> // Provides terminal I/O functions and
   structures, such as configuring the serial port settings.
#include <unistd.h> // Provides access to various POSIX
   operating system API functions, such as read and write.
#include <sys/time.h> // Provides functions and structures for
   working with time, such as measuring time intervals.
```

4.3.2 Implementation

```
int open_serial_port(const char *port_name);
```

This function takes a `port_name` as input and attempts to open the corresponding serial port for reading and writing (`O_RDWR`). It returns the file descriptor of the opened serial port if successful, or -1 if an error occurs.

```
int configure_serial_port(int serial_port);
```

This function takes the file descriptor of an opened serial port as input and configures its attributes. It uses the `termios` structure to modify various settings such as baud rate, character size, parity, stop bits, flow control, and timeout. It returns 0 if the configuration is successful, or -1 if an error occurs.

```
int write_serial_port(int serial_port, const unsigned char *data
   , size_t size);
```

This function takes the file descriptor of an opened serial port, a pointer to the data to be written, and the size of the data. It writes the specified data to the serial port and returns 0 if the write operation is successful, or -1 if an error occurs.

```
int read_serial_port(int serial_port, char *buffer, size_t size)
   ;
```

This function takes the file descriptor of an opened serial port, a pointer to a buffer where the read data will be stored, and the size of the buffer. It reads data from the serial port into the buffer until the specified size is reached. It returns the total number of bytes read if the read operation is successful, or -1 if an error occurs.

```
1 void close_serial_port(int serial_port);
```

This function takes the file descriptor of an opened serial port and closes it.

The code also includes some error handling. If any of the functions encounter an error, they print an error message using `printf` and return -1.

In summary, this code provides a set of functions that can be used to interact with a serial port, such as sending and receiving data. It abstracts away the low-level details of serial communication and provides a convenient interface for working with serial devices.

4.3.3 Parameters

In the header file two macros have been set:

```
1     #define NUM_BYTES 1000
```

This macro defines the buffer size for reading and writing data from/to the serial port. It is set to 1000, indicating that the buffer can hold up to 1000 bytes of data.

```
1     #define BAUD_RATE B4000000
```

This macro defines the baud rate for communication with the serial port. It is set to B4000000, which represents a baud rate of 4 Mbits/s. This is the maximum possible baud rate, but the actual rate will be limited by the capabilities of the USART (Universal Synchronous/Asynchronous Receiver/Transmitter) hardware. However, the baud rate in practice is therefore 115200.

4.3.4 Flags

In the `configure_serial_port` function is responsible for setting up the serial port configuration. It first retrieves the current attributes of the serial port using `tcgetattr`. Then, it manipulates various flags in the `termios` structure to achieve the desired configuration.

- `tty.c_cflag`: This flag is used to configure the control modes of the serial port. In this function, it is used to specify parameters such as parity, stop bits, and character size. The `PARENB` flag is cleared to disable the parity bit, the `CSTOPB` flag is cleared to set one stop bit, and the `CSIZE` bits are cleared to clear the character size bits. Additionally, the `CS8` flag is set to specify a character size of 8 bits, and the `CREAD` and `CLOCAL` flags are set to enable the receiver and ignore modem control lines.

- `tty.c_lflag`: This flag is used to configure the local modes of the serial port. In this function, it is used to specify parameters related to local echoing and canonical mode. The `ICANON`, `ECHO`, `ECHOE`, `ECHONL`, and `ISIG` bits are cleared to disable canonical mode, echoing, and signals.

- `tty.c_iflag`: This flag is used to configure the input modes of the serial port. In this function, it is used to specify parameters related to input processing and software flow control. Several bits including `IXON`, `IXOFF`, `IXANY`, `IGNBRK`, `BRKINT`, `PARMRK`, `ISTRIP`, `INLCR`, `IGNCR`, and `ICRNL` are cleared to disable software flow control and input processing.

- `tty.c_oflag`: This flag is used to configure the output modes of the serial port. In this function, it is used to specify parameters related to output processing. The `OPOST` and `ONLCR` bits are cleared to disable output processing and newline translation.

- `tty.c_cc[VTIME]` and `tty.c_cc[VMIN]`: These flags are used to configure the timeout and minimum number of characters to read, respectively. In this function, `VTIME` is set to 10 to specify a timeout in deciseconds, and `VMIN` is set to 0 to specify the minimum number of characters to read.

After configuring the `termios` structure, the function sets the baud rates for input and output using `cfsetispeed` and `cfsetospeed`, respectively. Finally, it applies the new settings to the serial port using `tcsetattr`. If any error occurs during the configuration process, error messages are printed, and the function returns -1.

4.4 Testing

To summarize, the `main.c` code located in the folder `Part 2/stm32f429-USB_CDC/echo` allows us to establish communication using the CDC class. The `serial-com.c` code, found in the folder `Part 2/stm32f429-USB_CDC/PC Part`, provides functions to open the serial port, write data, read data, and close the port. We need to test these two solutions with a concrete application.

Our approach involves creating a test function in the `echo.c` files located in `Part 2/stm32f429-USB_CDC/echo/`. This function will run on the STM32F4 board.

Additionally, the `main.c` file in the folder `Part 2/stm32f429-USB_CDC/PC Part` serves as another test file. It opens the serial port, sends data, receives it, calculates the speed, and then closes the port. First, let's delve into the details of the Echo program.

Echo Program

The primary concept is that the echo program runs within the STM32F429I board. When a user types a character into the terminal of the motherboard, this character is transmitted to the STM32F429I board. A specific function, termed `store_ascii_numbers`, modifies the input buffer by storing the ASCII values of the characters. If the character is alphabetical, it is converted to lowercase.

Conversely, if the character is not alphabetical, the function returns a hyphen (-) and then transmits it to the motherboard via the serial port. The code executing in the motherboard subsequently prints out the results.

This `store_ascii_numbers` function is invoked within the `cdcacm_data_rx_cb` function. As previously explained, this function is a callback function that is triggered when data is received from the host. It reads the received data into a buffer and then sends the transformed data back to the host.

The speed of this process is calculated within the `main.c` file of the `stm32f429-USB_CDC/PC` part. The program commences by specifying the name of the serial port to be used (`/dev/ttyACM0`) and attempts to open it using the `open_serial_port` function. If successful, it configures the port using the `configure_serial_port` function. Subsequently, it prepares a message to be sent over the serial port and attempts to write it using the `write_serial_port` function.

Upon successful transmission, the program prepares a buffer to store data read from the serial port. It then reads data into the buffer using the `read_serial_port` function. Afterward, it calculates the time taken for the data transfer by obtaining the current time before and after the transfer with `gettimeofday`.

The time taken is computed by subtracting the start time from the end time and converting the result into seconds. The speed of the data transfer is then calculated in megabytes per second (MBPS) by dividing the number of bytes transferred by the time taken. Additionally, the speed is converted to kilobytes per second (KB/s) for readability.

Finally, the program prints out the number of bytes read, the time taken, and the calculated speeds in both MBPS and KB/s. Upon completion, it closes the serial port and exits the program.

The transfer speed which currently stands at average 0.60 MB/s. This speed is limited not by the PC, but by the STM32F4 since it is an USB 1.0, therefore the USART limits the speed of transfer.

4.4.1 Makefile

The Makefile used is the same as the one from the work of Eric Herman [36] It merges makefile rules from different sources, specifically the `rules.mk` files and the `Makefile.include` file from the folder `libopencm3 example` [35]. Let's explained in detailled the makefile that allow us to write the executable file in the STM32F429I disco board:

The `LINDENT` variable is set to a command that formats C code according to a specific style.

The `BINARY` variable is set to the main output file of the project, and `LIBOPENCM3_DIR` is set to the directory of the libopencm3 library. The `LDSCRIPT` variable is set to the linker script used by the project.

The PREFIX variable is set to the prefix of the cross-compiler used for the STM32F4 microcontroller. The CC, LD, OBJCOPY, OBJDUMP, and GDB variables are set to the specific tools in the cross-compiler toolchain.

The CFLAGS variable is set to the compiler flags that are used when compiling the project's source code. The LDFLAGS variable is set to the linker flags that are used when linking the project's object files.

The OBJS variable is set to the object files that are part of the project.

The OOCD variables are set to the configuration for the OpenOCD on-chip debugger. The BMP_PORT variable is set to the port for the Black Magic Probe debugger.

The Makefile contains a series of rules for building the project. The all rule builds all the targets of the project. The images rule generates the binary, hex, srec, and list files from the project's ELF file. The flash rule writes the binary file to the microcontroller's flash memory.

The tidy rule formats the project's C code according to the style specified by the LINDENT variable. The clean rule removes all the generated files.

The -include directive at the end of the Makefile includes the dependency files generated by the compiler. These files specify the dependencies of the object files on the source and header files. If any of these files change, the corresponding object file is recompiled.

4.5 Notes

It is possible that the board may not be located at ACM0 and could be at a different location. To read from dev/ttyACM0 that is a device file path used to communicate with USB devices that used the ACM protocol, without requiring root privileges, one can add themselves to the dialout group using the following commands. Additionally, data transfer can be observed using the program minicom.

Note: Minicom [37] is a program designed for Unix operating systems, allowing users to communicate with connected devices, such as the STM32F4, through terminal emulation and serial communication, capable of handling RS232 interfaces.

Further details about these procedures can be found in the Appendix A

4.6 Conclusion

In this chapter, we established the communication infrastructure between the motherboad and the multufunction card.

To make it possible, we utilized the LibOpenCM3 firmware library, which provides a comprehensive set of functions and definitions for working with ARM Cortex-M3 microcontrollers. This library proved to be compatible, stable, and transparent in terms of code functionality, making it an ideal choice for our implementation.

We have also take a look to the STM32Cube development software, offered by STMicroelectronics, which provides a suite of tools and embedded software libraries for STM32 microcontrollers. However, due to its substantial installation size, instability, and uncertainty regarding generated code functionality, we decided not to utilize it in our project.

The implementation of the USB communication device class (CDC) application, specifically an abstract control model (ACM), was crucial for establishing communication between the STM32F4 board and the motherboard. This involved defining USB descriptors, handling control requests, and implementing callback functions for data transmission and reception.

Furthermore, we developed a serial communication library that abstracts away the low-level details of serial port communication. This library provides functions for opening, configuring, reading from, writing to, and closing a serial port. It simplifies the process of interacting with the serial port and provides a convenient interface for working with serial devices.

The Makefile efficiently orchestrates the compilation and deployment process for the STM32F429I disco board, utilizing variables to customize toolchain usage and providing rules for building, formatting, and cleaning the project.

To test the functionality of our implementation, we utilized the echo program, which operates on the STM32F4 board. This program receives data from the motherboard, transforms it by storing the ASCII values of the characters, and sends it back to the motherboard via the serial port. The motherboard program calculates the speed of data transfer by measuring the time taken for the transfer and the number of bytes transferred.

Chapter 5

Log Journal

5.1 Objectives

During its operations, it is possible that the robot may stop responding to commands or decide to halt for an unknown reason. It is also possible that errors may occur during the development phase. However, the question to ask is where this event originates in order to be able to delay the problem. To address this need, a log journal has been implemented. The idea behind this log journal is to write all events that occur between the moment we execute the code and the end of the code into a text file. The content written in this file will depend on the level of precision we desire. To achieve this, there are 5 levels:

- `DEBUG`: This level is intended for detailed information primarily used for debugging and operation tracking.
- `NORMAL`: This level is used for standard tracking information, not requiring immediate attention.
- `WARN`: This level is intended for warnings regarding potentially problematic but non-critical situations.
- `CRITICAL`: This level is reserved for critical errors that could compromise the overall operation of the system.

Under the `DEBUG` mode, we'll have access to all information, including those classified as critical, normal, and warnings. Critical messages will exclusively be displayed at the `CRITICAL` level, while warnings will be included at the `WARN` level. As for the `NORMAL` level, it will group regular messages along with critical ones and warnings.

In the log file, one will also find the precise location in the code where the event occurred, meaning which line and in which file, as well as at what moment, i.e., a precise timestamp down to the nanosecond.

5.2 Requirements

The needs are the following:

- Enable writing events to a file, such as recording when the servomotor moves 15 degrees during code execution.
- Allow messages to be associated with specific levels (e.g., `DEBUG`, `CRITICAL`, `NORMAL`, `WARN`).
- Provide the user with the option to choose the log level they want, such as only critical messages and above, or including debug messages.
- Write all events to a file in the `/tmp/` folder.
- Ensure that in the event of a sudden stop in code execution, such as a segmentation fault, the file remains accessible and complete.
- Record events with precise timestamps (date and time in nanoseconds), including the location of the event (file and line number).
- Implement mechanisms to avoid data corruption, ensuring mutual exclusion.

5.3 Variable Arguments

In the context of programming, a variable argument (also known as variadic argument or ellipsis) refers to a feature that allows functions to accept a variable number of arguments. This capability is particularly useful when the number of arguments that a function needs to handle is not fixed or known at compile time.

In the implementation of the log journal, variable arguments play a crucial role in enabling flexibility in message content. Instead of having a fixed number of arguments for logging functions, variable arguments allow developers to pass different numbers of arguments based on the specific message being logged. This flexibility enables diverse use cases, such as logging simple messages, formatting complex data structures, or including additional contextual information along with the message.

Variable arguments are typically used in conjunction with format specifiers, such as those used in the `printf` family of functions in C. These format specifiers define how the variable arguments should be formatted and inserted into the log message. For example, in the logging function `log_log`, the `format` parameter serves as the format string that specifies how the variable arguments should be formatted within the log message.

By utilizing variable arguments, the log journal implementation can accommodate a wide range of logging scenarios without being restricted by a fixed number of arguments. This flexibility enhances the utility and versatility of the logging module, allowing developers to tailor log messages to their specific requirements.

5.4 Implementation Details

The implementation is located in the git repository [24]. In the folder `Part 3/log`. We have three files: `log.c`, which contains the implementation of functions that meet our requirements explained in the previous section; the associated header file named `log.h`, which also contains macros and data structures and `main.c`, a test file that we will explain in detail in Section 4.4. Additionally, there is a `Makefile` that includes rules to compile the code, remove object files, and check for memory leaks using the `valgrind` tool, which is used for memory leak detection and debugging.

To meet the requirements, I found an interesting implementation that inspired me. The code can be found in [38]. The `log.c` file is different; I found that several mechanisms in his code were not necessary, so I wanted to implement it my way. However, it was a great source to know how to handle the VA argument and the macros to track line and files. Nevertheless, the `log.h` is similar since we have only one way to create macro of functions. The output is the same except that I wanted the clock in nanoseconds to have more precision, as it is clear, clean, and concise.

5.4.1 Data Structure

As we explained, we have four types of levels defined. To enumerate them, we have created an enumeration of these levels that will be useful. In the `log.h` file located in the `Part 3/log` folder, we have this data structure:

```
    typedef enum {
    LOG_DEBUG,     /**< Debug log level */
    LOG_NORMAL,    /**< Normal log level */
    LOG_WARN,      /**< Warning log level */
    LOG_CRITICAL   /**< Critical log level */
} LogLevel;
```

5.4.2 Functions

All the functions have been implemented in the `log.c` file located in `Part 3/log` folder.

5.4.3 log_init

```
int log_init(int lev);
```

Description: This function initializes the logging system by specifying the minimum severity level for messages to be recorded. It creates files with the `.log` extension and stores them in the directory `/tmp/`. The file is registered as `robot-log-xxxxxx .log`, where `xxxxxx` is a random ID. The `atexit` call registers the `log_close` function to be executed automatically when the program exits. Two macros are used in this function: the macro `FILE_SIZE`, which defines the size of the created file (256 bytes), and the macro `LOG_FILE_PREFIX`, which specifies the directory and filename prefix as `/tmp/robot-log`.

Parameters:

- **lev**: Minimum severity level for messages to be recorded. Can be `LOG_DEBUG`, `LOG_NORMAL`, `LOG_CRITICAL`, or `LOG_WARN`.

Return: Returns 0 if initialization is successful, -1 on failure.

5.4.4 log_write

```
void log_write(log *event);
```

Description: First, the function checks if the log file is open. If it is not, the function simply returns without doing anything. Next, a mutex lock is acquired to ensure thread safety using `pthread_mutex_lock`. This is important because multiple threads might be trying to write to the log file simultaneously.

The function then retrieves the current time using `timespec_get` and converts it to a human-readable format using localtime.The log message is then formatted using fprintf to include the timestamp, file name, line number, log level, and the actual log message. The vfprintf function is used to handle the variable arguments.

After writing the log message to the file, fflush is called to ensure that the message is immediately written to disk. Finally, the mutex lock is released using `pthread_mutex_unlock`. This allows other threads to access the log file.

Parameters:

- `level_str`: the log level based on the enumeration defined in the `log.h` file.
- `file`: the name of the file where the log message is being written.
- `line`: the line number in the file
- `format`: the format string for the log message
- `args`: the variable arguments for the format string

5.4.5 log_define

```
void log_define(int level, const char *file, int line, const
    char *format, ...)
```

Description

The function first checks if the provided level is less than a global `level_init` variable. If it is, the function returns immediately, not logging the message. This is a way to control the verbosity of the logging system - only messages with a level equal to or higher than `level_init` will be logged.

Next, the function uses a switch statement to convert the numerical level into a string representation. This is done for readability in the logs. The levels are `LOG_DEBUG`, `LOG_NORMAL`, `LOG_WARN`, and `LOG_CRITICAL`, in other words the ones defined in the enumeration in the file `log.h`. If the level doesn't match any of these, it defaults to `UNKNOWN`.

The function then declares a `va_list` named args. This is used to handle the variable number of arguments that were passed to the function. The `va_start` macro is used to initialize args to store the list of arguments that follow format.

The function `log_write` is then called with the level string, file name, line number, format string, and argument list. This function is presumably responsible for the actual logging.

Finally, `va_end` is used to clean up the memory reserved for args. This is important to prevent memory leaks.

Parameters

- `level`: integer level to represent the severity of the log message based on the enumeration of levels.
- `*file`: indicate the file where the log is being made,
- `line`: indicate the line number in the file
- `*format`: string format for the log message
- `...`: The ellipsis ... indicates that the function can take any number of additional arguments

5.4.6 log_close

```
void log_close();
```

Description: The code begins by acquiring a lock on a mutex called `log_mutex`.It ensures that the log file is closed safely without any conflicts. Next, it checks if the `log_file` variable is not `NULL`. The `log_file` variable is a pointer to a file object that represents the log file being used. If `log_file` is not `NULL`, it means that a log file is currently open. In that case, the function proceeds to close the file using the `fclose()` function. This ensures that any pending data is written to the file and the file is properly closed.

After closing the file, the `log_file` variable is set to NULL to indicate that no log file is currently open.

Finally, the function releases the lock on the `log_mutex` by calling `pthread_mutex_unlock()`. This allows other threads to access the log file if needed.

5.4.7 Macro

log_debug

```
#define log_debug(...) log_log(LOG_DEBUG, __FILE__, __LINE__,
    __VA_ARGS__)
```

log_normal

```
#define log_normal(...) log_log(LOG_NORMAL, __FILE__, __LINE__,
    __VA_ARGS__)
```

log_critical

```
#define log_critical(...) log_log(LOG_CRITICAL, __FILE__,
    __LINE__, __VA_ARGS__)
```

log_warn

```
#define log_warn(...) log_log(LOG_WARN, __FILE__, __LINE__,
    __VA_ARGS__)
```

The `log_debug` macro is used to log debug-level messages. When we use this macro in our code, it expands to a call to the `log_log` function with the debug level, the current file name, the current line number, and any other arguments we passed to `log_debug`. This is useful for logging messages that help in debugging the program, such as the values of variables or the flow of execution.

Similarly, the `log_normal` macro logs normal-level messages. These are typically informational messages that indicate the normal operation of the program. Like `log_debug`, this macro expands to a call to `log_log` with the normal level, the current file name, the current line number, and any other arguments.

The `log_critical` macro is used for logging critical errors. These are severe issues that may prevent the program from continuing to run. The macro expands to a call to `log_log` with the critical level, the current file name, the current line number, and any other arguments.

Finally, the `log_warn` macro is used for logging warning messages. These are issues that are not critical but may indicate a potential problem. Like the other macros, it expands to a call to `log_log` with the warning level, the current file name, the current line number, and any other arguments.

In all of these macros, the `__FILE__` and `__LINE__` macros are used to automatically include the file name and line number where the log message was generated. This can be very helpful when trying to trace the source of a message or error. The `__VA_ARGS__` macro is used to pass any additional arguments from the macro to the `log_log` function. This allows is to include additional information in the log message, such as the values of variables or the results of expressions.

FILE_SIZE

```
1    #define FILE_SIZE 256
```

This macro defines the maximum size of the log file.

LOG_FILE_PREFIX

```
1    #define LOG_FILE_PREFIX "/tmp/robot-log"
```

This macro specifies the directory and filename prefix.

5.5 Testing

The file `main.c` located in `Part 3/log` exemplifies the utilization of the previously explained log functions. The `simulate_activity` function acts as a temporary representation of a program activity. In its current implementation, it intentionally triggers a segmentation fault by attempting to dereference a null pointer, employed for testing purposes.

The `segfault_handler` function serves as a signal handler for segmentation faults. When a segmentation fault occurs (signaled by `SIGSEGV`), this function logs a critical error message and subsequently exits the program with a failure status.

The `thread_function` functions as the target executed by a new thread. It commences with a logged warning message, followed by the invocation of `simulate_activity`. The line `(void)arg;` signifies the deliberate disregard of the `arg` parameter.

Within the main function, the setup of the signal handler for segmentation faults is initiated using the `signal` function, followed by the initialization of the logging system via `log_init`. The objective is to configure the log to the debug level, meaning that all messages above the debug level – specifically, normal, debug, critical, and warning messages – will be printed. Failure leads to the printing of an error message to the standard error output and an exit with a failure status.

Subsequently, a normal message is logged, signifying the commencement of the program. This is followed by a debug message logging the initiation of the threads. A normal message then logs the creation of a new thread using `pthread_create` to execute `thread_function`. Failure in thread creation prompts a critical error message to be logged, followed by an exit with a failure status. Progressing, a warning log message signifies the entry into `simulate_activity`, where the function is invoked. The level of this log message is set to warn, as it intends to test all log levels in this code.

The main function awaits thread completion via `pthread_join`, logging a critical error message and exiting with a failure status upon failure. Finally, a normal message indicates program termination, returning a success status.

5.6 Output

The output below is from the main function located in `/log` as `main.c`. Since our logging level is set to debug, we can observe normal, warning, and critical messages. Note that the output is almost the same as in this code by the author `RXI` [38], which I found very clear and precise. The only difference is that we have the times in nanoseconds, which adds more precision to our file.

```
1 [22:45:48.148186099 24/05/2024] [main.c:64] [DEBUG] Start of the
     program
2 [22:45:48.148306155 24/05/2024] [main.c:67] [NORMAL] Start of
    the creation of threads
3 [22:45:48.148356951 24/05/2024] [main.c:75] [WARN] Enter in the
    function simulate_activity
4 [22:45:48.148372329 24/05/2024] [main.c:32] [CRITICAL] Segfault
    triggered
```

In the scenario where we set the journal initialization at the normal level:

```
1 [22:47:29.804980858 24/05/2024] [main.c:67] [NORMAL] Start of
    the creation of threads
2 [22:47:29.805187768 24/05/2024] [main.c:75] [WARN] Enter in the
    function simulate_activity
3 [22:47:29.805217273 24/05/2024] [main.c:32] [CRITICAL] Segfault
    triggered
```

Since we have set the logging level to normal, which is higher than the debug level, we receive all messages except those categorized as debug.

In the scenario where we set the journal initialization at the warn level:

```
[22:49:06.827026937 24/05/2024] [main.c:74] [WARN] Enter in the
    function simulate_activity
[22:49:06.827147904 24/05/2024] [main.c:32] [CRITICAL] Segfault
    triggered
```

Since we have configured the logging level to warn, similar to normal, only critical and warning messages will be logged.

The last scenario is when we set the level as critical:

```
[22:49:42.856312018 24/05/2024] [main.c:32] [CRITICAL] Segfault
    triggered
```

As we can see only critical messages will be logged.

5.7 Conclusion

In conclusion, the implementation of the log journal presents a significant advancement in tracking and debugging capabilities for our project. Unlike previous iterations, this journal offers enhanced precision, providing detailed timestamps down to nanoseconds, precise location information within source files, and robustness against common memory violations like segmentation faults.

One notable feature is the use of variable arguments, enabling flexibility in message content. This allows us to log diverse information, from simple messages to complex data structures, and facilitates the inclusion of additional contextual information alongside log messages. Moreover, the implementation effectively handles memory allocation, ensuring no memory leaks occur during operation.

The inclusion of multiple severity levels—DEBUG, NORMAL, WARN, and CRITICAL—provides flexibility in filtering and prioritizing logged events based on the development or operational context. This allow that the development team can focus on relevant information by choosing the level of the log.

This log journal implementation was useful for the implementation of the last part that we will address in the next chapter.

Chapter 6

Communication Architecture Between Tasks

6.1 Problem

In this chapter, we address a new issue: during the operation of the robot, multiple jobs need to be fulfilled. We define a "task" as a specific job required for the optimal functioning of the robot. These tasks can be related to electronic components such as sensors, actuators, or servomotors, or they can be unrelated to electronics.

For example, consider a servomotor: it sends its data to the motherboard via the multifunction board. The motherboard calculates the new position and sends the updated information to another electronic component that needs this data. On the other hand, a task not related to electronics might involve implementing the robot's strategy or communicating with what we call the "world." This "world" is a database linked to the states of the robot and its environment, allowing it to send data to other robots, for instance.

Currently, this is just a concrete example of the robot's operation; however, numerous tasks need to be created, especially those related to vision, as well as those that will be requested in the future as the robot continues to develop.

This chapter will therefore focus on developing an interface that facilitates the creation of tasks and communication between them. This interface aims to enable future programmers to develop new tasks as the robot continues to evolve and require additional functionalities. We will ensure the interface is modular and robust by implementing defensive programming to prevent misuse by future users. Additionally, to achieve fast computations and save battery life, we will explore options such as using threads and putting tasks to sleep when they have no work to perform.

We will also discuss various parameters that can be added for each task, such as its priority relative to other tasks, its periodicity and his exclusivity for a specific core if needed. Furthermore, we will cover how to customize error messages in case a function implementing a specific task fails, providing clearer insights into the reasons for the failure.

6.2 Thread or Process ?

To start with a solid general understanding, let's first define what a thread and a process are:

- **Process**: Processes are self-contained execution entities with their own dedicated memory. They operate independently of each other, contributing to improved stability and security as a failure in one process doesn't impact the others. While processes can execute concurrently on different cores, they need inter-process communication (IPC) to exchange data. This can be more intricate and less speedy than communication between threads.

- **Threads**: Threads represent the most minimal execution unit within a process. They utilize the same memory and resources as the parent process, facilitating efficient data exchange and communication among threads. Threads have the capability to operate concurrently on separate cores, which makes them suitable for executing multiple tasks at the same time and increasing computational speed. Given that they utilize the same resources, the management of threads requires fewer resources compared to processes.

As mentioned in the introduction, we aim for the calculations and operations performed by the tasks on the motherboard to be as fast as possible. Threads enable parallel execution, allowing multiple tasks to run simultaneously on different cores. This parallelization significantly enhances the efficiency and speed of task execution. Managing large tasks with threads is advantageous because it allows for efficient data sharing and communication within the same memory space, leading to faster computation and reduced overhead compared to using separate processes.

While this will slightly increase battery consumption, the increase is minimal. The power consumption of our motherboard ranges between 3W and 10W, with 10W being the peak usage during boot. The trade-off is worthwhile, as the benefit of achieving faster computations and smoother, more rapid execution of the robot's tasks far outweighs the minimal increase in power usage. This efficiency will provide a significant advantage during competitions.

6.3 Tasks

In this section, I will refer to some functions that have been implemented. This code is available in the GitLab Repository [24], in the folder `Part 3/communication`. The two files that are referenced are `task.c` and `task.h`. The other two files are linked to the section on Communication (Section 6.4).

6.3.1 Definition

As briefly explained in the section 6.1, a task is a requirement that must be met, whether it pertains to electronic components or other needs. For electronic components, tasks must interact with two crucial parts: the motherboard, which provides requested information or issues commands to electronic components such as actuators, and the multifunction card, which relays this information.

Notably, it serves as a communication point between the motherboard and electronic constituents, akin to a conductor in an orchestra. Also multiples tasks can represents the same constituent but with different requirements.

For instance, consider the example of vision: we may have a task responsible for computing the pixels received from it, and another task responsible for storing important information received from vision. In other words, for vision, we could have `"compute_vision"` and `"store_data_vision"` tasks.

For non-electronic components, tasks might involve implementing the robot's strategy, such as finding the best path to a goal or determining the shortest path avoiding opponents. Additionally, there might be a task responsible for storing data related to the states of the robot and its environment and providing this data to his teammates for instance.

6.3.2 Priority

In a real-time operating system (RTOS), tasks or processes may have different levels of urgency. Thus, tasks with higher importance are assigned a higher priority than those with lower importance. The kernel, in this case, Preempt-RT, follows a scheduling policy to determine the order in which tasks are executed. There are four types of scheduling policies: Round Robin, Earliest Deadline First, Rate Monotonic Scheduling, and Priority-Based Scheduling.

- `Round Robin scheduling`: allocates a fixed time slice to each task before moving on to the next task in a circular order.
- `Earliest Deadline First scheduling`: prioritizes tasks based on their deadlines, executing tasks with earlier deadlines first.
- `Rate Monotonic Scheduling`: schedules tasks based on their period or frequency, with shorter periods or higher frequencies being assigned higher priority.
- `Priority-Based Scheduling`: executes tasks based on their priority levels, with higher priority tasks being executed first.

For a task to be executed in real-time and benefit from the features offered by the Preempt-RT kernel, it must follow one of the four scheduling policies. In our case, we choose the `Priority-Based Scheduling` that meets our needs, since it is simpler, and we don't actually need the services provided by the other three scheduling policies.

In our context, we need this priority. For example, we have two tasks, one handling vision and the other related to the gyroscope. It might happen that we want the task linked to vision to be executed more urgently than the gyroscope task; therefore, we will assign, thanks to the scheduling policy, a higher priority.

In a Preempt-RT system, when a task with higher priority, such as vision management, enters the execution queue, the Linux kernel detects this need, often triggered by a hardware or software interruption.

In response, the kernel performs a context switch by preserving the state of the current active task (e.g., `gyroscope`) and loading the context of the new task (e.g., `vision`). Once the context is loaded, the processor begins executing the new task.This process of preemption can repeat if other tasks with even higher priorities appear during the execution phase.

Here's an example illustrating how this mechanism is implemented in the file `task.c` located in `Part 3/communication`. This code implements a function called `task_set_priority` that can be used by the program running in the motherboard of the robot. Note that the data structure of Task is available in the section 6.4.3.

```
int task_set_priority(Task *t, unsigned int priority) {
    if (!t) {
        log_critical("Task info is NULL");
        return TASK_MEMORY_ERROR;
    }
    if (priority > MAX_PRIORITY) {
        priority = DEFAULT_PRIORITY;
        log_warn("Invalid priority value: Default value set
        to 50.");
    }
    struct sched_param param;
    param.sched_priority = priority;

    // Attempt to set thread priority
    if (pthread_setschedparam(pthread_self(), SCHED_FIFO,
        &param) != 0) {
        // Error occurred while setting priority
        log_critical("Failed to set priority");
        return TASK_PRIORITY_ERROR;
    }
    return TASK_SUCCESS;
}
```

- The function checks if the task exists. If it does not, a special error return code `TASK_MEMORY_ERROR` will be returned by the function.

- The function ensures that task priorities comply with the constraints set by Preempt-RT. Preempt-RT establishes a maximum priority of 100 (`MAX_PRIORITY`). To address potential oversights, a macro is used to assign a default value of 50 to tasks that lack specified priorities. We choose 50 because it represents the middle of the priority range. If users try to assign priorities beyond the maximum allowed (e.g., 101), surpassing the `MAX_PRIORITY` threshold, the function resets the priority to `DEFAULT_PRIORITY` and generates a warning log message stating, "Invalid priority value: Default value set to 50."

- Then, a `sched_param` structure is defined, and its `sched_priority` field is set to the provided priority. This structure is used by the `pthread_setschedparam` function to set the scheduling parameters for a thread.

- The function then attempts to set the priority of the current thread (`pthread_self()`) to the provided priority using a First-In-First-Out (FIFO) scheduling policy. If this operation fails (i.e., `pthread_setschedparam` returns a non-zero value), a critical log message "Failed to set priority" is generated, and the function returns a `TASK_PRIORITY_ERROR` code, which is a special error return code.

- If everything went well, the function will return the special error return code `TASK_SUCCESS`.

- The special error return code will be explained in the subsection 6.3.6.

6.3.3 Periodicity

When implementing tasks for the robot, it is often necessary to have tasks executed periodically, meaning at regular intervals. For instance, a task checking the battery level must run at defined times to perform this check. Therefore, we require this task to be periodic, executing at each predefined time interval to verify the battery level.

Unlike RTAI, Preempt-RT does not provide native tools for this purpose. Thus, we utilize the standard library: `time.h`.

Here's how we accomplished this in the code located in the file `Task.c`. This code implements a function called `task_set_period` that can be used by the program running in the motherboard of the robot. Note that the data structure of Task is available in the section 6.4.3.

```
1  int task_set_period(Task *t, unsigned long period_ns) {
2      if (!t) {
3          log_critical("Task info is NULL");
4          return TASK_MEMORY_ERROR;
5      }
6      struct timespec period;
7      period.tv_sec = period_ns / 1000000000;
8      period.tv_nsec = period_ns % 1000000000;
9      // Attempt to set task period using clock_nanosleep
10     int result = clock_nanosleep(CLOCK_MONOTONIC, 0, &period,
       NULL);
11     if (result != 0) {
12         // Error occurred while setting period
13         log_critical("Failed to set period for task");
14         return TASK_TIMER_ERROR;
15     }
16     return TASK_SUCCESS;
17 }
```

- First, we check if the task exists. If not, a specific return value will be returned by the function.

- A `timespec` structure named `period` is defined. The `timespec` structure is a way to specify time periods with high precision; it consists of two fields: `tv_sec` for seconds and `tv_nsec` for nanoseconds. The function then calculates the seconds and nanoseconds parts of the period from the input `period_ns`

and assigns them to the `period` structure.

- The function then attempts to set the task period using the `clock_nanosleep` function. This function suspends the execution of the calling thread until either at least the time specified in `&period` has elapsed, or a signal is delivered that triggers the invocation of a signal-catching function. The `CLOCK_MONOTONIC` argument represents a clock that cannot be set and represents the amount of time since some unspecified point in the past.

- In the case the clock has not been set up, we handle it by writing to the log journal and returning a specific error code.

- If everything went well, the specific return value `TASK_SUCCESS` will be returned.
- The special error return code will be explained in the subsection 6.3.6.

6.3.4 Creation

In this section, we will explore the correct method of creating a thread to represent tasks. This is a crucial function implemented in the file `Task.c`.

The code in `Task.c` features a function called `task_create`, which is designed to be utilized by the program running on the robot's motherboard. Note that the data structure of Task is available in the section 6.4.3.

```
1  int task_create(void *(*start_routine)(void *), void *arg,
   Task *t, int cpu_core) {
2      pthread_t thread;
3      pthread_attr_t attr; // Declare pthread attribute object
4      if (!t) {
5          log_critical("Task info is NULL");
6          return TASK_MEMORY_ERROR;
7      }
8      // Initialize the pthread attribute object
9      if (pthread_attr_init(&attr) != 0) {
10         log_critical("Failed to initialize thread attributes");
11         return TASK_THREAD_ERROR;
12     }
13
14     // Set detach state attribute to PTHREAD_CREATE_JOINABLE
15     if (pthread_attr_setdetachstate(&attr,
   PTHREAD_CREATE_JOINABLE) != 0) {
16         log_critical("Failed to set thread detach state");
17         return TASK_THREAD_ERROR;
18     }
19
20
21     // Create thread with specified attributes
22     if (pthread_create(&thread, &attr, start_routine, arg) != 0)
       {
23         log_critical("Failed to create task");
24         pthread_attr_destroy(&attr); // Clean up attribute
   object
```

```
25          return TASK_THREAD_ERROR;
26      }
27
28      // Clean up attribute object
29      pthread_attr_destroy(&attr);
30
31      // Save the thread ID in the Task structure
32      t->task_id = thread;
33  }
34  // SKIP
35      return TASK_SUCCESS;
```

- The function begins by checking if the Task pointer `t` is NULL. If it is, a critical log message "Task info is NULL" is generated, and the function returns a `TASK_MEMORY_ERROR` code. This is a basic error handling mechanism to ensure that the function doesn't attempt to operate on a null pointer, which would lead to undefined behavior.
- Next, a `timespec` structure named `period` is defined. The `timespec` structure is a way to specify time periods with high precision, with two fields: `tv_sec` for seconds and `tv_nsec` for nanoseconds. The function then calculates the seconds and nanoseconds parts of the period from the input `period_ns` and assigns them to the `period` structure.
- The function then attempts to set the task period using the `clock_nanosleep` function. This function suspends the execution of the calling thread until either at least the time specified in the `period` variable has elapsed, or a signal is delivered that triggers the invocation of a signal-catching function. The `CLOCK_MONOTONIC` argument represents a clock that cannot be set and represents the amount of time since some unspecified point in the past.
- If the `clock_nanosleep` function call fails (i.e., it returns a non-zero value), a critical log message "Failed to set period for task" is generated, and the function returns a `TASK_TIMER_ERROR` code. This is an error handling mechanism to ensure that the task period is set correctly. Further elaboration on error handling will be provided in the following sections.

- The skipped part includes the code linked to the next sub-subsection: Attribution Of Core.
- If all the operations are successful, including the one for the attribution of core, the function returns a `TASK_SUCCESS` code, indicating that the task period was successfully set.

- The special error return code will be explained in the subsection (6.3.6).

Attribution of Core

A heavy task currently running on the robot is the vision system. The binocular vision system sends up to 1.5 GB/s of data. To process this data and perform necessary calculations, the task requires a significant amount of computational power.

As discussed in the Chapter 3, we can use Preempt-RT to assign a specific core for certain jobs. In C, using Linux libraries, it is possible to designate that a particular task runs on a defined CPU core. For the vision task, we will allocate a dedicated core to handle its processing. This allocation can be implemented as follows, note that this is the continuation of the create function seen above (the skip part) :

```
1       if (cpu_core > DEFAULT_CPU) {
2        cpu_set_t cpuset;
3        CPU_ZERO(&cpuset);
4        CPU_SET(cpu_core, &cpuset);
5        if (pthread_setaffinity_np(thread, sizeof(cpu_set_t), &
   cpuset) != 0) {
6            log_critical("Failed to set CPU affinity");
7            return TASK_AFFINITY_ERROR;
8        }
```

The code first checks if the `cpu_core` value is greater than `DEFAULT_CPU`. If it is, the code proceeds to set the CPU affinity. Because if the user did not define the CPU on which he want the task to run, we will give to this task a default CPU. In this case it is -1, meaning that we let the operating system to choose on which CPU the task can run.

A `cpu_set_t` variable named cpuset is declared. This data type is a bitset where each bit represents a CPU.

The `CPU_ZERO(&cpuset)` function call clears the set, meaning no CPUs are selected.

The `CPU_SET(cpu_core, &cpuset)` function call then adds `cpu_core` to the set, meaning the process will be able to run on this CPU.
The `pthread_setaffinity_np(thread, sizeof(cpu_set_t), &cpuset)` function call attempts to set the CPU affinity of the thread to the cpuset. If this function call fails (returns a non-zero value), a critical log message "Failed to set CPU affinity" is outputted and the function returns special code error: `TASK_AFFINITY_ERROR`.

6.3.5 Termination

Another feature we would like is the ability to check if a task has completed its work. Imagine wanting to stop a task, thinking it has finished. We need to be certain that it has indeed completed; otherwise, we risk corrupting data and encountering technical issues.
To address this, a function called `task_set_period` located in `Task.c` file have been implemented and can be used by the program running in the motherboard of the robot. Note that the data structure of Task is available in the section 6.4.3.

```
1  int task_wait(Task *t) {
2      if (!t) {
3          log_critical("Task info is NULL");
4          return TASK_MEMORY_ERROR;
5      }
6      else if (pthread_join(t->task_id, NULL) != 0) {
7          log_critical("Failed to wait for task");
8          return TASK_THREAD_ERROR;
9      }
10     return TASK_SUCCESS;
11 }
```

- The first thing to check is that the Task Object is allocated properly in memory, if not a critical message is logged and a special return value is returned.

- The second method is to wait for the completion of the thread associated with the Task Object using the `pthread_join` function, which takes two parameters: the task ID and a pointer to a location where the exit status of the thread can be stored. In this case, `NULL` is passed as the second parameter since the exit status is not needed.

- If the `pthread_join` function returns a non-zero value, it indicates that an error occurred while waiting for the thread. In this case, the function logs a critical error message using `log_critical` and returns a value `TASK_THREAD_ERROR` to indicate a thread-related error.

- If the `pthread_join` function returns 0, it means that the thread was successfully waited for and joined. The function then returns a value `TASK_SUCCESS` to indicate that the task was completed successfully.

6.3.6 Custom Return Value

As mentioned earlier, one of the objectives of this interface is to make it easier for the robot developers to use. However, debugging can be particularly tedious, especially when it is difficult to determine the origin of an error. One proposed solution is to create our own error codes for cases where a function does not execute correctly.

For example, suppose we decide to create a task and allocate it a specific core. If an error occurs, with a return code of -1, for instance, it is important to know where the problem occurred. In the `task.c` file, there are five places where this error could occur. Instead of checking each place individually to identify the source of the error, we could use specific error codes. These codes would allow us to know precisely what happened, thereby facilitating the debugging process. This code can be used on the program that is running on the motherboard of the robot.

The first step, therefore, was to define these error codes, let's take some of them that we can find in the file `task.h`:

```
1    #define TASK_SUCCESS              0x00
2    #define TASK_FAILURE              0x01
3    #define TASK_OS_ERROR             0x02
4    #define TASK_MEMORY_ERROR         0x03
```

- `TASK_SUCCESS` is defined as 0x00, which represents a successful completion of a task.
- `TASK_FAILURE` is defined as 0x01, indicating a general task failure.
- `TASK_OS_ERROR` is defined as 0x02, indicating an error related to the operating system.
- `TASK_MEMORY_ERROR` is defined as 0x03, indicating a memory-related error.
- `TASK_ERROR` is defined as 0x04, representing a general error.

Let's take for example, this code from the create function in `task.c`:

```
1    if (pthread_attr_init(&attr) != 0) {
2        log_critical("Failed to initialize thread attributes");
3        return TASK_THREAD_ERROR;
4    }
```

If the function fails to initialize thread attributes, `TASK_THREAD_ERROR` will be returned. Consequently, we will see 0x01 (which represents the value 1) as the return value in our terminal.

6.4 Communication

The code discussed in this section can be found in the GitLab repository, as referenced in [24]. Specifically, the relevant files are located in the `Part 3/communication` directory and are named `com.c` and `com.h`

6.4.1 Introduction

. In the previous section, we discussed the definition and role of a task. However, it may be necessary for these tasks to communicate with each other. Let's take a concrete example: imagine a task responsible for retrieving accelerometer data. The task managing the movements of the leg servomotors might need this data. Therefore, a means of transmitting this information from one task to another is required.

Additionally, if a task is interested in specific data, as in the previous example, it must be able to signal its interest and notify the relevant task that it wishes to receive this data.

Another point is that we must be able to communicate data from one task to another, as well as send messages such as a start command.

Another consideration is that a task might need to wait for data before it can proceed with its own work. As mentioned earlier, we aim to avoid wasting resources unnecessarily. Therefore, we need a mechanism that allows a task in waiting to avoid consuming resources during this time.

6.4.2 Principle of the Solution

To address the requirements of the situation, strategic decisions have been made. Firstly, a robust data structure is needed to serve as an orchestrator for storing tasks, their IDs, and types. This structure ensures efficient management of various tasks, such as those related to the camera, which are tagged with identifiers like `MSG_VISION`. A hashtable was selected for this purpose due to its constant time complexity for lookups, ensuring efficient access even with a large number of entries. Unlike arrays, which are susceptible to linear search complexities, hashtables provide faster access to data by directly mapping keys to indices, thereby minimizing lookup time.

To minimize collisions within the hashtable, a hash function is employed. This function takes the task ID as input, computes its multiplication with a prime number, and then takes the modulo of this product with the hashtable size.

This hashing technique distributes keys evenly across the hashtable, reducing the likelihood of collisions and ensuring efficient storage and retrieval of data based on their IDs.

However, despite the efforts to minimize collisions, they may still occur when attempting to place task information in an occupied entry. To address this issue, a technique called linear probing is employed. Linear probing involves searching for the next available slot by incrementing the index and wrapping around to the beginning of the hashtable if necessary when encountering occupied entries. By systematically searching for empty slots, linear probing helps maintain the efficiency of hashtable operations even in the presence of collisions.

Additionally, tasks often require access to data from other tasks they are interested in. To facilitate this, the data structure records the ID of the task and the relevant message. For example, if `servomotor 2` is interested in the content of the "position" task, the IDs of both `servomotor 2` and `MSG_POSITION` are recorded in the data structure.

For task-specific data, an array with a fixed size of 256 has been chosen, which is deemed sufficient for the expected message load. However, it is crucial to prevent task monopolization of the processor. Upon data availability, such as position data, the interested task (`servomotor 2`) is notified, awakened, gyroscope data transmitted to it, and then stored in a linked array data structure specific to the interested task. This approach ensures that processing resources are efficiently utilized while maintaining responsiveness to task-specific data requirements.

To adress this need several function have been implemented in the file `com.c`

- `create_hashtable`: Initializes the hash table.
- `hash`: That is the hash function.
- `insert_task`: Inserts a task into the hash table.
- `task_interested`: Sends a message to interested tasks.
- `task_get_message`: Retrieves a message from a task's input queue.

Let analyze in details in the following section.

6.4.3 Data Structure

To meet the aforementioned needs, we will need to assign attributes to these tasks.

First, we need a way to identify a task. For example, consider the `Task_wait` function mentioned earlier, which is responsible for determining if a task has finished its execution.

Second, we need a place to store the information that a task might require. For instance, if we have a task that manages the positioning of servo motor axes based on data received from the camera, we need to send these repositioning data to the task responsible for the servomotors.

Finally, we want to ensure that this task receives a message, such as a message instructing it to start.

To achieve this, we have defined a data structure for the task as follows:

```
typedef struct Task {
    unsigned int task_id; /**< Task ID */
    char input_queue[QUEUE_SIZE]; /**< Input queue for the task
   */
    struct Task *next; /**< Pointer to the next task */
    pthread_cond_t cond; /*< Condition variable */
    pthread_mutex_t mutex; /*< Mutex */
    bool message_received; /*< Flag to indicate if a message has
    been received */
} Task;
```

- The `task_id` is an unsigned integer variable that is used to uniquely identify each task.
- The `input_queue` is an array of characters with a length defined by `QUEUE_SIZE` that is 256. As explained is to store the data that the task needs.
- The `next` is a pointer to the next structure of the Task, usefull in the case of collision.
- The `cond` is a condition variable that will be useful when the task is waiting for a message and we want him to not use the CPU ressources.
- The `mutex` is usefull to assure mutual exclusion.
- The `message_received` is a condition variable that will be used in the case that a message is received in the `input_queue` of the task therefore if it is true we wake up the task.

Here is our implementation of the hashtable based on our needs:

```
/**
 * @brief Structure to represent a hash table entry.
 */
typedef struct hashtableEntry {
    int message; /**< Message associated with the entry */
    Task tasks[MAX_TASKS_PER_ENTRY]; /**< List of tasks
   associated with this message */
    size_t num_tasks; /**< Number of tasks */
    pthread_mutex_t mutex; /**< Mutex for the entry */
    struct hashtableEntry* next; /**< Pointer to the next entry
   */
} hashtableEntry;
```

```
11
12 /**
13  * @brief Structure to represent the hash table.
14  */
15 typedef struct {
16     size_t size; /**< Size of the hash table */
17     hashtableEntry entries[TABLE_SIZE]; /**< Array of entries */
18 } hashtable;
```

- `hashtableEntry` structure represents an entry in the hash table. It contains four fields:
  - `message`: This is an integer that represents the message associated with the entry. This could be a key used to identify the entry in the hash table.
  - `tasks`: This is an array of `Task` structures, with a maximum size of five that is defined as `MAX_TASKS_PER_ENTRY`. This array represents the list of tasks associated with the message. In our case we defined it actually as 5. In the context of a hash table, this could be used to handle collisions, where multiple keys (messages) hash to the same index in the table.
  - `num_tasks`: This is a `size_t` type (which is an unsigned integer type) that represents the number of tasks in the `tasks` array.
  - `mutex`: This is a `pthread_mutex_t` type, which is a mutex (short for "mutual exclusion") used for synchronizing access to the entry. This is particularly useful in a multithreaded environment where multiple threads might try to access or modify the same hash table entry simultaneously.
  - `next`: is used to create a linked list of hash table entries in case of collisions during hash table operations. Each entry in the hash table can have a pointer to the next entry with the same hash value. This allows for efficient handling of collisions and ensures that all entries with the same hash value are accessible.

- `hashtable` structure represents the hash table itself. It contains two fields:
  - `size`: This is a `size_t` type that represents the size of the hash table. This could be the number of entries currently in the hash table.
  - `entries`: This is an array of `hashtableEntry` structures, with a size of `TABLE_SIZE`. Actually we defined `TABLE_SIZE` as 100. This array represents the entries in the hash table. Each index in the array corresponds to a hash value, and the `hashtableEntry` at that index contains the message (key) and associated tasks (values) for that hash value.

6.4.4 Creation of the Hashtable

In the context of developing a multithreaded program, where concurrent access to shared resources is a concern, the implementation of a hash table with locking mechanisms becomes important. Hash tables serve as efficient data structures for storing and retrieving information, but in a multithreaded environment, their integrity can be compromised if not managed properly. Thus, incorporating locking mechanisms ensures that critical sections of code, responsible for initializing and accessing the hash table, are executed atomically. By employing locks, such as mutexes, we can enforce exclusive access to the hash table, preventing race conditions and maintaining data consistency. In the forthcoming implementation, we will delve into the creation of a hash table using locking mechanisms to safeguard against concurrent access issues, thereby ensuring the reliability and robustness of our program in a multithreaded setting.

Here is the way that we created our hashtable:

```
int create_hashtable(hashtable* table, size_t capacity) {
    // Lock the mutex before accessing the shared resource
    pthread_mutex_lock(&table_mutex);
    if (table == NULL) {
        log_critical("The table is empty!\n");
        // Unlock the mutex
        pthread_mutex_unlock(&table_mutex);
        return TASK_TABLE_NULL_ERROR;
    }

    table->size = capacity; // Set the size of the table
    // Initialize each entry in the table
    for (size_t i = 0; i < capacity; i++) {
        // Set the message to -1
        table->entries[i].message = -1;
        // Set the number of tasks to 0
        table->entries[i].num_tasks = 0;
    }
     // Unlock the mutex
    pthread_mutex_unlock(&table_mutex);

    return TASK_SUCCESS;
}
```

Let analyze this function located in `com.c`:

At the beginning of the function, a mutex lock `table_mutex` is acquired using `pthread_mutex_lock`. This is done to ensure that no other thread can access the shared resource (in this case, the hash table) while it is being initialized. This is a common practice in multi-threaded programming to prevent race conditions.

Next, the function checks if the table pointer is NULL. If it is, a critical log message is printed using `log_critical`, indicating that the table is empty, then the lock is release to avoid deadlock. Then the function returns a specific error `TASK_TABLE_NULL_ERROR`.

If the table pointer is not NULL, the function proceeds to initialize the hash table. It sets the size of the table to the provided capacity.

Then, it loops over each entry in the table (from 0 to capacity - 1) and initializes the message field to -1 and the `num_tasks` field to 0.

Finally, the function releases the mutex lock using `pthread_mutex_unlock`, allowing other threads to access the hash table. It then returns `TASK_SUCCESS` to indicate that the hash table was successfully created.

Note that specific error return will be explained in the subsection 6.4.10

6.4.5 Hash Function

```
size_t hash(unsigned int id) {
    return (id * 2654435761) % TABLE_SIZE; // Using a prime
   multiplication for better distribution}
}
```

The `hash` function is employed to minimize collisions within the hash table. Utilizing prime multiplication, it aims to enhance distribution across the table. By multiplying the task's `ID` with the prime number `2654435761` and subsequently taking the modulo of the `TABLE_SIZE`, which is set at 100, the function endeavors to evenly distribute hash values, thereby diminishing collision probabilities. This approach ensures a more balanced distribution of keys throughout the table, contributing to efficient performance in hash table operations.

6.4.6 Interest for a Message

To ensure the robust and safe insertion of tasks into the hashtable, it is imperative to address potential issues arising from parallelism. Consider a scenario where two tasks attempt to write to the hashtable simultaneously, which could lead to a deadlock. To mitigate this risk, implementing communication mechanisms such as locks becomes crucial. Locks ensure that data interleaving is controlled, preventing conflicts and ensuring thread safety during concurrent access. By employing synchronization techniques like locks, we can guarantee that tasks are inserted into the hashtable reliably and without risking data integrity. In our function we will implement those lock to be sure that only one thread can execute the critical section of code at a time, meaning writing in the Table, thus safeguarding against race conditions, data race and data corruption.

Quick reminder:

- `Data Race`: This occurs when two threads access the same mutable object (that can be changed after its creation) without proper synchronization.

- `Race Condition`: This occurs when the order of events impacts the program's correctness.

So the function has the role to insert a Task in the hashtable but not only, the essential part also of this function is that when putting the task in the hashtable, we specify, which type of task he is interested.

```
int insert_task(hashtable* table, unsigned int task_id, int
    message) {
    if (table == NULL) {
        log_critical("The table is empty!\n");
        return TASK_TABLE_NULL_ERROR;
    }
    // Lock the mutex before accessing the shared resource
    pthread_mutex_lock(&table_mutex);

    size_t index = hash(task_id);
    hashtableEntry* entry = &table->entries[index];

    // Check if the entry is free
    if (entry->num_tasks < MAX_TASKS_PER_ENTRY) {
        // Entry is free, proceed with insertion
        Task* newTask = &entry->tasks[entry->num_tasks];
        newTask->task_id = task_id;
        pthread_mutex_init(&newTask->mutex, NULL);
        pthread_cond_init(&newTask->cond, NULL);
        newTask->message_received = false;
        entry->message = message;

        // Initialize input_queue to an empty string
        newTask->input_queue[0] = '\0';

        entry->num_tasks++;
        // Unlock the mutex
        pthread_mutex_unlock(&table_mutex);
        return TASK_SUCCESS;
    } else {
        // Entry is already occupied, search for the next
        available slot
        // Start searching from the next index
        size_t next_index = index + 1;
        // Continue searching until we reach the starting index
        while (next_index != index) {
            if (next_index >= table->size) {
            /* Wrap around to the beginning of the table if we
            reach the end*/
                next_index = 0;
            }
            hashtableEntry* next_entry = &table
            ->entries[next_index];
            if (next_entry->num_tasks < MAX_TASKS_PER_ENTRY) {
                // Found an available slot with insertion
                Task* newTask = &next_entry
                ->tasks[next_entry->num_tasks];
                newTask->task_id = task_id;
                pthread_mutex_init(&newTask->mutex, NULL);
                pthread_cond_init(&newTask->cond, NULL);
                newTask->message_received = false;
                next_entry->message = message;

                // Initialize input_queue to an empty string
                newTask->input_queue[0] = '\0';
                next_entry->num_tasks++;

```

```
57                  // Unlock the mutex
58                  pthread_mutex_unlock(&table_mutex);
59                  return TASK_SUCCESS;
60              }
61              // Move to the next index
62              next_index++;
63          }
64
65          // Unable to find an available slot
66          log_critical("Unable to add task. No available slots!
67          \n");
68          pthread_mutex_unlock(&table_mutex);
69          return TASK_FAILURE;
70      }
71  }
```

In this code situated in the `com.c` file, if we want to say that a task is interested for a `MSG_START`, we insert the Task in the hashtable, with this ID, and the type of message that he is interested as:

```
1           insert_task(&table, 1, MSG_START);
```

Let's break down the code step by step:

- The `insert_task` function takes a `hashtable` pointer, a `task_id`, and a `message` as parameters. It is responsible for inserting a new task into the hash table.

- The function first checks if the table pointer is NULL. If it is, it logs a critical error message and returns `TASK_TABLE_NULL_ERROR`. This check ensures that the hash table is not empty.

- Next, the function locks a mutex called `table_mutex` using `pthread_mutex_lock`. This is done to ensure that only one thread can access the shared resource (the hash table) at a time. Locking the mutex prevents race conditions where multiple threads try to modify the hash table simultaneously.

- The function calculates the index in the hash table where the task should be inserted using the hash function. The hash function is not shown in the provided code, but it is assumed to generate a unique index based on the `task_id`.

- The function retrieves the entry at the calculated index from the hash table.

- The function checks if the entry has available slots for tasks. Each entry has an array of tasks, and `MAX_TASKS_PER_ENTRY` defines the maximum number of tasks that can be stored in each entry. If there are available slots, the function proceeds with the insertion.

- If there are available slots, the function initializes a new task (`newTask`) at the next available slot in the entry's task array. It sets the task ID, initializes the task's mutex and condition variable using `pthread_mutex_init` and `pthread_cond_init`, and sets the `message_received` flag to false. It also sets the `message` field of the entry to the provided `message`. The `pthread_cond_init` is important because once the task has been inserted, it will wait for a specific

message to manifest its interest in it. During this waiting time, it will not utilize CPU resources.

- The function initializes the `input_queue` of the new task to an empty string by setting the first character to '\0'.
- The function increments the `num_tasks` field of the entry to reflect the addition of the new task.
- Finally, the function unlocks the `table_mutex` using `pthread_mutex_unlock` and returns `TASK_SUCCESS` to indicate that the task insertion was successful.
- If there are no available slots in the entry, the function enters a loop to search for the next available slot in the hash table. It starts searching from the next index and continues until it reaches the starting index again. This ensures that it wraps around to the beginning of the table if it reaches the end.
- If an available slot is found, the function proceeds with the insertion in the same way as described earlier.
- If no available slot is found after searching the entire hash table, the function logs a critical error message and returns `TASK_FAILURE`.
- Note that specific error return will be explained in the subsection 6.4.10

Collisions are handled and when an entry is considered occupied: In this previous function, collisions occur when two or more tasks have the same hash value and need to be inserted into the same entry in the hash table. To handle collisions, the code uses a technique called linear probing. When an entry is already occupied, the code searches for the next available slot by incrementing the index and wrapping around to the beginning of the table if necessary. It continues this search until it finds an entry with available slots or until it reaches the starting index again.

An entry is considered occupied when the `num_tasks` field of the entry reaches the maximum number of tasks per entry (`MAX_TASKS_PER_ENTRY`). This means that all slots in the entry's task array are filled, and no more tasks can be inserted into that entry. In such cases, the code searches for the next available slot using linear probing.

6.4.7 Call Tasks

To effectively manage and handle incoming messages within a multithreaded environment. We need a way to handle whenever there is a need to notify tasks registered for a specific message (done through the insert function).

In essence, we need to traverses through the entries of a given hash table to locate tasks interested in a particular message. Upon finding such tasks, we need to update

their content meaning their input queue with the message content but being regardless about safety to avoid corrupt data. We need also to be sure that the data given in the task respect a predefined size to be sure to not load so much the task .

```
// Function to send a message to all tasks interested in a
    specific message
int task_interested(hashtable* table, int message) {
    // Lock the mutex before accessing the shared resource
    pthread_mutex_lock(&table_mutex);
    // Iterate over each entry in the table
    for (size_t i = 0; i < table->size; i++) {
        // Get the entry
        hashtableEntry* entry = &table->entries[i];
        /* Check if the entry's message matches the specified
        message*/
        if (entry->message == message) {
            // Get the first task in the entry
            Task* current_task = entry->tasks;
            // Traverse the linked list of tasks
            while (current_task != NULL) {
                // Lock the task mutex
                pthread_mutex_lock(&current_task->mutex);
                // Get the length of the message
                size_t message_length = strlen(message_strings
                [message]);
                // Get the length of the input queue
                size_t current_queue_length = strlen(
                current_task->input_queue);
                /*Check if the input queue has enough space
                for the message*/
                if (current_queue_length + message_length
                < QUEUE_SIZE - 1) {
                    // Copy the message to the input queue
                    memcpy(current_task->input_queue +
                    current_queue_length, message_strings
                    [message], message_length);
                    // Ensure null-termination
                    current_task->input_queue[
                    current_queue_length + message_length] = '\0
    ';
                    // Set the message received flag to true
                    current_task->message_received = true;
                    // Signal that a message is available
                    pthread_cond_signal(&current_task->cond);
                }
                else {
                     // Log a warning message
                    log_warn("Input queue is full for task %u.
                    Discarding the message.\n", current_task
                    w->task_id);
                }
                // Unlock the task mutex
                pthread_mutex_unlock(&current_task->mutex);
                // Move to the next task in the linked list
                current_task = current_task->next;
            }
        }
```

```
52      }
53      // Unlock the mutex
54      pthread_mutex_unlock(&table_mutex);
55      // Return success
56      return TASK_SUCCESS;
57  }
```

Here is a description of what this function situated in `com.c` does:

The purpose of this function checks if a task is interested in a certain message. It takes two parameters: a pointer to a hashtable structure and an integer message.

The function starts by looping over each entry in the hash table. For each entry, it checks if the message field of the entry matches the provided message. If it does, the function proceeds to process each task in the entry's task list.

For each task, the function first acquires the task's mutex lock using `pthread_mutex_lock`. This is done to ensure that no other thread can access the task while it is being processed. This is a common practice in multi-threaded programming to prevent race conditions.

Next, the function calculates the length of the message and the current length of the task's input queue. If the total length of the message and the current queue is less than `QUEUE_SIZE` - 1, meaning 255, the function copies the message to the end of the task's input queue using `memcpy`. It then ensures that the input queue is null-terminated by setting the character following the message to \0. The task's message_received field is set to true, and a condition variable cond is signaled to indicate that a new message has been received.

If the total length of the message and the current queue is not less than `QUEUE_SIZE` - 1 meaning 255, a warning log message is printed using log_warn, indicating that the input queue is full and the message is being discarded.

Finally, the function releases the task's mutex lock using `pthread_mutex_unlock`, allowing other threads to access the task. It then moves on to the next task in the list. If all tasks have been processed, the function returns `TASK_SUCCESS` to indicate that the operation was successful.

Note that specific error return will be explained in the subsection 6.4.10

To sum up, This function is designed to be called whenever there is a need to notify tasks registered for a specific message. In essence, the `task_interested` function traverses through the entries of a given hash table to locate tasks interested in a particular message. Upon finding such tasks, it updates their input queue with the message content, ensuring proper synchronization and thread safety through the utilization of mutex locks. Additionally, the function takes precautions to prevent input queue overflow by discarding messages if the queue is full, thus maintaining the integrity of task execution. Overall, the `task_interested` function serves as a crucial component in a multithreaded system, facilitating efficient communication and coordination between tasks in response to incoming messages.

6.4.8 Efficient Waiting

An essential consideration arises when we have a scenario where out of 100 tasks, 85 are waiting to retrieve the data they require. In such instances, it might not be optimal for all 85 tasks to actively utilize the processor. This is because diverting resources from these tasks could potentially enhance the computational power available for other tasks. Additionally, conserving processor usage in this context can contribute to saving battery resources, thereby optimizing overall system efficiency.

To adress this issue, we need to find a way to not use the processor. An effecient way has been implemented. Since we are using multithreaded environment the posix funxtion allow us to make a task to not use the processor.

But actually if the data is available, well the task is waken up and written the data into the buffer of the task caller.

Here is the way that has been implemented in the file `com.c`:

```
// Function to retrieve message for a task
int task_get_message(hashtable* table, unsigned int task_id,
    char* message_buffer) {
      // Lock the mutex before accessing the shared resource
     pthread_mutex_lock(&table_mutex);
     // Check if the table or message buffer is empty
     if (table == NULL && message_buffer == NULL) {
         // Log an error message
         log_critical("The table is empty!\n");
         // Log an error message
         log_critical("The message buffer is empty!\n");
         // Unlock the mutex
         pthread_mutex_unlock(&table_mutex);
         // Return an error code
         return TASK_TABLE_NULL_ERROR;
     }
     // Get the entry at the hash index for the task ID
     hashtableEntry* entry = &table->entries[hash(task_id)];
     // Get the first task in the entry
     Task* current_task = entry->tasks;
     // Traverse the linked list of tasks
     while (current_task != NULL) {
         // Check if the task ID matches
         if (current_task->task_id == task_id) {
             // Lock the task mutex
             pthread_mutex_lock(&current_task->mutex);
             /*Wait until a message is received or the
             input queue is non-empty*/
             while (!current_task->message_received
             && current_task->input_queue[0] == '\0') {
                 // Log a normal message
                 log_normal("Waiting for signal\n");
                 // Wait for the condition variable
                 pthread_cond_wait(&current_task->cond,
                 &current_task->mutex);
             }
             // If a message is received, copy it to the buffer
             if (current_task->message_received) {
```

```
38                  // Get the length of the message
39                  size_t message_length = strlen(current_task
40                  ->input_queue);
41                  // Check if the message is too large for the
    buffer
42                  if (message_length >= MAX_MESSAGE_SIZE) {
43                      // Unlock the task mutex
44                      pthread_mutex_unlock(&current_task->mutex);
45                      // Unlock the mutex
46                      pthread_mutex_unlock(&table_mutex);
47                      // Return an error code
48                      return TASK_FAILURE;
49                  }
50                  // Copy the message to the buffer
51                  memcpy(message_buffer, current_task
52                  ->input_queue, message_length);
53                  // Ensure null-termination
54                  message_buffer[message_length] = '\0';
55
56                  // Reset state for next message
57                  current_task->message_received = false;
58                  // Set the input queue to an empty string
59                  current_task->input_queue[0] = '\0';
60                   // Unlock the table mutex
61                  pthread_mutex_unlock(&table_mutex);
62                  // Unlock the task mutex
63                  pthread_mutex_unlock(&current_task->mutex);
64                  return TASK_SUCCESS;
65              }
66              // Unlock the task mutex
67              pthread_mutex_unlock(&current_task->mutex);
68              // Unlock the table mutex
69              pthread_mutex_unlock(&table_mutex);
70              // Return an error code
71              return TASK_FAILURE;
72          }
73          // Move to the next task in the linked list
74          current_task = current_task->next;
75      }
76      // Unlock the table mutex
77      pthread_mutex_unlock(&table_mutex);
78      // Return an error code
79      return TASK_NOT_FOUND_ERROR;
80  }
```

Here is a description of what this function situated in `com.c` does:

The function takes three parameters: `hashtable table`, `unsigned int task_id`, and `char message_buffer`. These parameters are used to access a hash table, identify a specific task, and store the retrieved message.
The function begins by locking a mutex called `table_mutex`. This is done to ensure that only one thread can access the shared resource (the hash table) at a time. Locking the mutex prevents race conditions and ensures thread safety.
The function then checks if the table pointer or the `message_buffer` pointer is `NULL`. If either of them is `NULL`, it logs an error message using a function called `log_critical`, unlocks the `table_mutex`, and returns an error code `TASK_TABLE_NULL_ERROR`.

If the table and message_buffer are not NULL, the function proceeds to retrieve the task entry from the hash table based on the provided task_id. It uses the hash function to calculate the index in the hash table where the entry should be located.Once the entry is obtained, the function retrieves the first task in the entry's linked list of tasks.
The function then enters a while loop that checks two conditions: whether a message has been received for the current task (current_task→message_received) and whether the input queue for the task is empty (current_task→input_queue[0] == '\0').

If both conditions are true, it means that the task is waiting for a message. In this case, the function logs a normal message using log_normal, waits for a signal indicating that a message is available using pthread_cond_wait, and releases the task mutex (current_task→mutex). This allows other threads to access the shared resource while the current thread is waiting.

Once a message is received or the input queue is non-empty, the function checks if a message has been received (current_task→message_received). If a message is received, it copies the message from the input queue to the message_buffer using memcpy. It also ensures that the message_buffer is null-terminated by adding a null character at the end.

After copying the message, the function resets the state for the next message by setting current_task→message_received to false and clearing the input queue (current_task→input_queue). It then unlocks the table_mutex and the task mutex (current_task→mutex).

Finally, the function returns TASK_SUCCESS to indicate that the message retrieval was successful.

If a message is not received or the input queue is empty, the function unlocks the task mutex and the table_mutex, and returns TASK_FAILURE to indicate an error. If the provided task_id is not found in the hash table, the function unlocks the table_mutex and returns TASK_NOT_FOUND_ERROR to indicate that the task was not found.

Note that specific error return will be explained in the subsection 6.4.10

In summary, the task_get_message function is responsible for retrieving a message for a specific task from a hash table. It ensures thread safety by using mutexes to control access to shared resources and provides error handling for cases where the table or message buffer is NULL, the task is not found, or the message is too large for the buffer.

Note that the way that we handled, the fact that are task that is in waiting state does not consume CPU resources is handled thanks to the pthread_cond_wait(). He get the signaled in the function task_interested. Once he gets the signal the task write his content in the buffer of the task that is interested by his content.

The way that we checked that of course this task when he is in a waiting state does

not consume the resource is by using the command `top`. Here is the command used to check this:

```
./task
ps
top -p 40660
```

Using `ps` command we know the id of the `./task` process that is running. Here it is 406600. Then using `top -p id` where p is the process, we see the number as 0 meaning that he is not using the CPU.

Quick reminder, a thread have actually fives states:

- `New`: The thread has been created but he is not currently ruuning
- `Ready`: The thread is ready to run and waiting for the CPU to execute it.
- `Running`: The thread is currently being executed by the CPU.
- `Blocked`: The thread is waiting for a resource or event (such as I/O operation or a lock) to become available.
- `Terminated`: The thread has finished its execution or has been explicitly terminated and cannot be executed further.

In our case, since our task is waiting to receive the signal from the condition variable `pthread_cond_init`, and this task is being executed by the threads, the thread is currently in a blocked state as it waits for the resource to become available. Once the resource is available and the thread receives the signal, it transitions to the running state.

6.4.9 Print Content

To ease debugging it may be useful to know what are the content in the buffer of a specific task.

To do so a function has been implemented that we can find in `com.c` that write the content of the task in the log file.

```
// Function to print input queue for a task
int print_queue(hashtable* table, unsigned int task_id) {
    // Lock the mutex before accessing the shared resource
    pthread_mutex_lock(&table_mutex);
    // Check if the table is empty
    if (table == NULL) {
        log_critical("The table is empty!\n");
        // Unlock the mutex
        pthread_mutex_unlock(&table_mutex);
        // Return an error code
        return TASK_TABLE_NULL_ERROR;
    }
    // Get the hash index for the task ID
    size_t index = hash(task_id);
    // Get the entry at the hash index
    hashtableEntry* entry = &table->entries[index];
    // Get the first task in the entry
    Task* current_task = entry->tasks;
    // Traverse the linked list of tasks
```

```
20      while (current_task != NULL) {
21          // Check if the task ID matches
22          if (current_task->task_id == task_id) {
23              // Log the input queue
24              log_normal("Input queue for task %u:\n%s\n",
25              task_id, current_task->input_queue);
26              // Unlock the mutex
27              pthread_mutex_unlock(&table_mutex);
28              // Return success
29              return TASK_SUCCESS;
30          }
31          // Move to the next task in the linked list
32          current_task = current_task->next;
33      }
34      // Log a warning message
35      log_warn("Task %u not found\n", task_id);
36      // Unlock the mutex
37      pthread_mutex_unlock(&table_mutex);
38      // Return an error code
39      return TASK_NOT_FOUND_ERROR;
40  }
```

At the start of the function, a mutex named `table_mutex` is locked using `pthread_mutex_lock`. This is done to prevent simultaneous access to the shared resource (the hash table) by multiple threads, which could lead to data inconsistencies or race conditions.

Next, the function checks if the provided hash table is NULL. If it is, a critical log message is written in the log file, the mutex is unlocked and the function returns with an error code `TASK_TABLE_NULL_ERROR`.

If the table is not NULL, the function calculates the index in the hash table for the given task ID using a hash function. It then retrieves the corresponding hashtableEntry and sets `current_task` to the first task in the linked list of tasks at that hash table entry.

The function then enters a while loop, which continues as long as `current_task` is not NULL. Inside the loop, the function checks if the ID of the `current_task` matches the provided task ID. If it does, the function writes the input queue for that task in the log file, unlocks the mutex, and returns with a success code `TASK_SUCCESS`.

If the `current_task` ID does not match the provided task ID, the function moves to the next task in the linked list. If the function goes through all tasks in the list without finding a match, it writes a log message in the log file indicating that the task was not found, logs a warning, unlocks the mutex, and returns with an error code `TASK_NOT_FOUND_ERROR`.

6.4.10 Custom Return Value

As for the task part, several custom return value have been implemented. The goal remains the same as for the task part, it allows to ease debugging when a particular function or portion of code is not executed correctly.

In the case of the communication here is those that have been created in the next of the ones for task that we can found in the file `task.h`

```
1    #define TASK_TABLE_NULL_ERROR 0x09
2    #define TASK_OVERFLOW_ERROR 0x10
3    #define TASK_NOT_FOUND_ERROR 0x11
```

- `TASK_TABLE_NULL_ERROR`: This error code might be used when there is an attempt to access or manipulate a task hashtable that is NULL or uninitialized. It indicates that the task hashtable, which is likely a data structure used to store tasks, is not available or has not been properly initialized.

- `TASK_TABLE_NULL_ERROR`: This error code might be used when there is an attempt to access or manipulate a task hashtable that is NULL or uninitialized. It indicates that the task hashtable, which is likely a data structure used to store tasks, is not available or has not been properly initialized.

- `TASK_NOT_FOUND_ERROR`: This error code might be used when there is an attempt to perform an operation on a task that does not exist in the task hashtable. It indicates that the specified task could not be found in the task table, possibly because it has been removed or was never added in the first place.

Therefore if for example, we tried to access to a task that is not on the hashtable, as a result we get a value of 17 (0x11) returned in the terminal.

By the way, we used also in the `com.c` file, the custom return value that we have been used in `task.c` !

6.4.11 Testing

To ensure that all aspects of the communication and task components function correctly, several tests have been conducted and are located in the file `com.c`. Here are the tests in a sequential manner.

```
int main() {
    if (log_init(LOG_DEBUG) != 0) {
        log_critical("Error during journal initialization\n");
    }
    /* Used for debugging purpose using top (check if the task
    waiting do not use CPU)*/
    log_normal("PID: %d\n", getpid());

    create_hashtable(&table, TABLE_SIZE);

    log_normal("Insertion of Tasks \n");
    for (int i = 1; i <= NUM_TASKS; ++i) {
        insert_task(&table, i, i % 6);
    }
    log_normal("End of Inseration\n");

    log_normal("Call tasks interested of specific message\n");
    for (int i = 0; i < 6; ++i) {
        task_interested(&table, i % 6);
    }
    log_normal("End of calling tasks\n");
    log_normal("Print the queue of each tasks \n");
    for (int i = 1; i <= NUM_TASKS; ++i) {
        print_queue(&table, i);
    }
    log_normal("End of printing queue of each task\n");

    char message_buffer[MAX_MESSAGE_SIZE];
    log_normal("Start of the recpetion of msg of each task\n");
    for (int i = 1; i <= NUM_TASKS; ++i) {
        if (task_get_message(&table, i, message_buffer)
        == TASK_SUCCESS || i == 5) {
            log_normal("Message retrieved for task %d :
            %s\n", i, message_buffer);
        } else {
            log_warn("Impossible to retrieve the message
            of task %d\n", i);
        }
    }
    log_normal("End of the reception of msg of each task\n");
    log_normal("Freeing memory\n");
    pthread_mutex_destroy(&table_mutex);
    log_normal("End of the program\n");
    return TASK_SUCCESS;
}
```

In this example, we tested the hashtable to ensure it functions correctly, and it performed as expected. Even with a number of tasks close to the size of the hashtable, all tests passed successfully, and the output in the log file was satisfactory. The Valgrind test, used to detect memory leaks, found none, confirming that all allocations were correctly freed.

This code located in `com.c` has been implemented to handle tasks in a parallel manner, utilizing the functions implemented in `task.c` as well.

```
1
2  void *send_messages(void *arg) {
3      for (int i = 0; i < NUM_TASKS; i++) {
4          insert_task(&table, i, i % 6);
5          task_interested(&table, i % 6);
6          print_queue(&table, i);
7          task_get_message(&table, i, message_buffer);
8      }
9      pthread_exit(NULL);
10 }
11
12 int main() {
13     if (log_init(LOG_DEBUG) != 0) {
14         log_critical("Error during journal initialization\n");
15         return TASK_FAILURE;
16     }
17     log_normal("Starting program\n");
18     printf("PID: %d\n", getpid());
19     if (create_hashtable(&table, TABLE_SIZE) != TASK_SUCCESS) {
20         log_critical("Failed to create hashtable\n");
21         return TASK_FAILURE;
22     }
23     log_normal("Hashtable created\n");
24     Task task_list[NUM_TASKS];
25
26     for (int i = 0; i < NUM_TASKS; i++) {
27
28         if (task_create(send_messages, (void *)&task_list[i],
29         &task_list[i], -1) != TASK_SUCCESS) {
30             log_critical("Failed to create task %d\n", i + 1);
31             return TASK_THREAD_ERROR;
32         }
33         log_normal("Task %d created\n", i + 1);
34         if (task_set_priority(&task_list[i], NUM_TASKS - i)
35         != TASK_SUCCESS) {
36             log_critical("Failed to set priority for task
37             %d\n", i + 1);
38             return TASK_PRIORITY_ERROR;
39         }
40         log_normal("Task %d priority set\n", i + 1);
41         if (task_set_period(&task_list[i], (i + 1) * 10000000)
42         != TASK_SUCCESS) {
43             log_critical("Failed to set period for task
44             %d\n", i + 1);
45             return TASK_AFFINITY_ERROR;
46         }
47
48
```

```
49
50          log_normal("Task %d period set\n", i + 1);
51          pthread_join(pthread_self(),NULL);
52          log_normal("Task %d finished\n", i + 1);
53
54      }
55      log_normal("Freeing memory\n");
56      pthread_mutex_destroy(&table_mutex);
57      log_normal("End of the program\n");
58      return TASK_SUCCESS;
59  }
```

Note that this code will not work in a non-real-time operating system, as it relies on mechanisms specific to real-time operating systems, such as setting task priorities. However, in a real-time environment, the code functions correctly, and the log file output appears consistent with no errors reported.

6.5 Conclusion

In this chapter, we have meticulously explored the rationale behind our choice of threads over processes, detailing each function within the `task.c` file that facilitates task creation and priority setting. But also the creation of periodic tasks and the assignment of specific CPUs for critical tasks, such as the vision task. Defensive programming measures have been implemented to handle cases such as where users set impossible priorities, ensuring robust performance.

The `com.c` file plays a crucial role in managing inter-task communication, storing task IDs in a hash table along with their respective messages. Tasks are paused while waiting for necessary messages to conserve CPU resources, and linear probing is employed to prevent collisions. Additionally, we provided a function to view the contents of each task, along with two test functions that integrate the log system but also the functions of `task.c` and `com.c`. To ensure maximum reliability and robustness, especially during competitions, we have minimized dynamic allocations to prevent unexpected interruptions. Our thorough approach ensures that the code is resilient, keeping the robot operational and competitive.

Chapter 7

Conclusions

As announced, we have established a computing infrastructure, both in hardware and software, providing the necessary tools for developers to continue implementing the remaining functionalities of the robot in a pre-configured environment.

First, we selected a real-time operating system (RTOS) capable of performing the robot's operations with minimal latency. We chose a compact RTOS that is regularly updated, offers services tailored to our needs, and is compatible with the various components of the motherboard. After thorough evaluation, we opted for the Preempt-RT kernel and the Fedora 39 distribution, ensuring a balance between performance, space efficiency, and ease of use.

The second phase of the project involved establishing a USB connection between the motherboard and the multifunction board. We developed code on the multifunction board to send data to the motherboard, and code on the motherboard to display the results and send commands back to the multifunction board. Data transfer tests showed satisfactory results in terms of transfer speed.

Next, we implemented a logging system that records events from the robot's central program in a text file. This system logs messages according to five severity levels, allowing users to filter the information they want to see in the log. Events are recorded with nanosecond precision, including the time, date, location in the code, and the relevant source file.

Finally, we developed an interface providing a set of functions for task management. This interface allows for the creation and management of tasks, including setting periods and priorities in a multithreaded environment. It also includes an inter-task communication system, offering functions to define message types of interest, notify tasks of incoming messages, display message contents, and efficiently and robustly store hundreds of tasks.

In conclusion, these steps have created a robust and flexible computing foundation, facilitating future development of the robot's functionalities and ensuring an optimized environment for performance and resource management.

7.1 Perspectives

Looking ahead, the Robocup project has been relaunched this year. As mentioned in the introduction, several works have already been completed or are currently in progress. A notable achievement has been the developmentof the multifunction card by the electronics team. However, significant work remains on the software side, particularly in terms of strategy, implementation of movements, and shape detection. This thesis provides a solid foundation for them to begin their implementations and test them directly.

Currently, we have 20 active members who plan to work through the holidays with the sole aim of seeing the first robot function and take its first steps in the Montefiore hall. This milestone is ideally scheduled for the 2024-2025 academic year. Let's not forget that the ultimate goal is to win the Robocup. In the coming months, a few new members will join the Robocup team. We also plan to call for sponsors; an ASBL has already agreed to sponsor us for the purchase of new consumables.

This project progresses thanks to the efforts of the involved students and the support of several faculty members, particularly the Robocup project coordinator, the supervisors of the works mentioned in the introduction, and the promoter of this thesis, Mr. Bernard Boigelot. Mr. Olivier Bruls also assists the mechanical team, and Mr. Pascal Harmeling provides crucial help to the electronics team regarding the soldering of the multifunction card components. It is important to note that other professors, assistants, and members of the University of Liège have offered their support if needed. We are immensely grateful for their help and contributions, which not only advance the Robocup project but also allow us to learn beyond what is taught in our cursus.

Appendix A

Technical Documentation

The Git link for the implementation of each part described in the report is available at this link: https://gitlab.uliege.be/Macgyver/test

A.1 Chapter 3: Installation of Kernels

Note that for this tutorial, it has been done on a Debian workstation.

A.1.1 Preempt-RT

Installation Of Libraries

```
sudo dnf install gcc make git libssl-dev libelf-dev flex bison
   openssl dwarves  wget
```

Download the Kernel

Here we downloaded the kernel.

```
    wget https://mirrors.edge.kernel.org/pub/linux/kernel/v6.x/
   linux-6.5.2.tar.xz
```

Know, we are going to download the patch associated with this kernel to let us to have a preemptive kernel.

```
    wget https://mirrors.edge.kernel.org/pub/linux/kernel/
   projects/rt/6.5
    /patch-6.5.2-rt8.patch.gz
```

Extract the archive and run the patch:

```
    xz -cd linux-6.5.2.tar.xz | tar xvf -
    gunzip patch-6.5.2-rt8.patch.gz
    cd linux-6.5.2
    patch -p1 </home/macgyver/Preempt-RT/patch-6.5.2-rt8.patch
```

Configuration of the Kernel

```
    cp -v /boot/config-$(uname -r) .config
    make oldconfig
```

Note: He will ask us for *Preemption Model*, select the *Fully Preemptive kernel* and let the default value for the rest. We don't touch other options since it is apatch that makes everything for us, and ask our for the options that needs to be modified.

We can also do it with *make menuconfig*

Installation of the Kernel

Before doing ths step, we can check if everything have been configure correctly, here we can use *nano .config* and check if the preemptive kernel has been enabled. If yes, we are free to do the next step.

```
make -j6
sudo make  modules_install
sudo make  install
sudo update-grub
```

Check the Installation

After doing the step 4, we can reboot and check with the command *uname -r* if it corresponds to the rt kernel.

Links & Sources

All the installation process explained above, have been rewritten from this source: [39]

A.1.2 RTAI

Download the RTAI Folder

- Visit the official RTAI website at https://www.rtai.org/.
- Choose the required version (e.g., version 5.3).
- Download the folder and identify the patch compatible with our kernel version (details below).
- For our installation, we used kernel version 4.14.133.

Download the Specific Kernel

- Use the `wget` command to download the kernel from the official kernel archive:

```
wget https://cdn.kernel.org/pub/linux/kernel/v4.x/linux
-4.14.133.tar.xz
```

- Decompress the downloaded archive using the following command:

```
sudo tar -xvf linux-4.14.133.tar.xz -C /usr/src
```

  Use the `-f` flag for verbose output, allowing us to see the files being decompressed. This command will decompress the folder and put it into the repository /usr/src.

- Access the kernel folder using:

```
sudo ln -snf linux-4.14.13/ linux
cd /usr/src/linux-4.14.133
```

- Install the necessary libraries using the following command:

```
sudo apt-get install libncurses-dev libssl-dev bc flex
bison libelf-dev
```

- Apply the patch using the following command:

```
patch -p1 </.../rtai-5.3/base/arch/x86/patches/hal-
linux-4.14.133-x86-7.patch
```

  Be aware to replace `username` with our username that we are currently using on our laptop.

- Copy the configuration of our current kernel into the kernel folder:

```
cp -v /boot/config-$(uname -r) .config
```

  Using the `-v` flag for verbosity, we should see an output like: `-> /boot/config-4.15.0-20 -> '.config'`

- Run `make menuconfig` to configure the kernel as needed.

Kernel Configuration Settings

Configuration can be found also in: `/usr/local/src/rtai/README.CONF_RMRKS`

The description of options to be modified is explained in the subsection 3.4.2.

Instead of doing it through the `menuconfig`, an alternative is to do it directly from the `.config` file but it is not recommended.

- Open the `.config` file in our kernel source directory using the `nano` text editor or any text editor of our choice:

```
nano /usr/src/linux-4.14.133/.config
```

- Inside the `.config` file, we can modify kernel options by changing the values from:
  - `=y` to enable the feature.
  - `=n` to disable the feature.
  - `=m` to make the feature modular.

  For example, to enable a feature, change:

```
# CONFIG_FEATURE_NAME is not set
```

  to:

```
CONFIG_FEATURE_NAME=y
```

To disable a feature, change:

```
CONFIG_FEATURE_NAME=y

```

to:

```
# CONFIG_FEATURE_NAME is not set

```

To make a feature modular, change:

```
CONFIG_FEATURE_NAME=y

```

to:

```
CONFIG_FEATURE_NAME=m

```

- Save our changes and exit 'nano'. If `nano` has been used, press `Ctrl+O` to save, then `Ctrl+X` to exit.
- After modifying the `.config` file, then we check that we have not warning about the "unmet dependencies" then we continue the kernel compilation process
- Compile the kernel with the following command, using the number of available CPUs on the machine for parallel compilation:

```
make -j $(nproc)

```

- Install the kernel itself:

```
sudo make install
```

- Update the kernel images. This is crucial to avoid kernel panic:

```
sudo update-initramfs -c -k 4.14.133
```

- Update the GRUB bootloader configuration:

```
sudo update-grub
```

- Reboot our system:

```
sudo reboot
```

RTAI Installation

- Copy the RTAI source to the `/usr/src` directory:

```
sudo cp rtai-5.3-tar.bz2 /usr/src/
```

- Decompress the RTAI source:

```
sudo tar -xvf rtai-5.3.tar.bz2
```

- Create a symbolic link named `rtai` pointing to the RTAI directory:

```
sudo ln -snf rtai-5.3/ rtai
```

- Navigate to the RTAI directory:

```
cd rtai
```

- Configure RTAI using the following command:

```
sudo make menuconfig
```

  - Check in the `General` section if the path to the Linux tree is correct.
  - Under `Machine (x86) - Number of CPUs`, set it to our actual number of CPUs (we can check using 'lscpu').
  - Exit the configuration menu.

- Compile and install RTAI:

```
make -j4
make modules -j4
sudo make modules_install -j4
sudo make install
```

- Add an RTAI configuration file to set the library path:

```
sudo nano /etc/ld.so.conf.d/rtai.conf
```

  - Write the following line in the file:

```
/usr/realtime/lib
```

- Run the following command to update library paths:

```
sudo ldconfig
```

- Modify the `.profile` file to add RTAI binaries to the system's PATH:

```
    nano ~/.profile

```

  - Add the following lines to the end of the file:

```
    PATH="$PATH:/usr/realtime/bin"
    export PATH
    LD_LIBRARY_PATH="$LD_LIBRARY_PATH:/usr/realtime/lib
    "
    export LD_LIBRARY_PATH

```

- If we encounter the "RTAI running less than CPU" error, modify the GRUB configuration:

```
    sudo nano /etc/default/grub

```

  - In the line starting with `GRUB_CMDLINE_LINUX_DEFAULT`, add the following parameter (replace "our number" with our desired number of CPUs):

```
    quiet splash maxcpus=our number

```

- Save the GRUB configuration and update GRUB:

```
  sudo update-grub

```

Links & Sources

All the installation process explained above, have been rewritten from those source.

- Kernel installation: [40] [41]
- Bugs: [42]
- Configuration of the kernel: [43]
- RTAI installation: [44]

A.1.3 Xenomai

Install of Librairies

If we are using **Ubuntu**, we can download the previous tools (i.e based on RTAI) and download also *meson*:

```
        sudo apt-get instal meson ninja-build rt-tests
        sudo apt-get update && sudo apt-get upgrade
```

If we are using **Fedora**, we can do the following commands:

```
    sudo dnf install dwarves meson openssl g++ "Development
   Tools"
    "Development Librairies" rt-tests dwarves
    sudo dnf update && sudo dnf upgrade
```

Download the kernel

```
    cd
    mkdir Kernel
    cd Kernel
    git clone https://source.denx.de/Xenomai/xenomai4/linux-evl.
   git // linux-evl
    git clone https://source.denx.de/Xenomai/xenomai4/libevl.git
    // libevl
```

Notes: Rename the linux-evl folder as *linux-evl* and the libevl folder as *libevl*. Also we are using the last version of Xenomai based on the last release kernel that was *6.6-evl-rebase*.

Configuration with Meson

```
    mkdir /tmp/build-ative && cd /tmp/build-native
    meson setup -Dbuildtype=release -Dprefix=/home/macgyver/
   Kernel -Duapi=/home/macgyver/Xenomai/linux-evl
    . /home/macgyver/Kernel/libevl
    meson compile && meson install
```

This command will install all compiled files and libraries in the folder Kernel, where the folder *bin* is located and the folder *tests* that will be useful later on.

Configuration of the Kernel

```
    make O=../linux-build mrproper
    make O=../linux-build oldconfig
    make O=../linux-build menuconfig (*) // Need to read
   carefully the note before going on.
```

Notes: Follow those following steps:

- **In General Setup** → *Disable* UAPI Headers

- **In General Setup** → Local version - append to kernel release → Add *-nb* for example.

- This step is very important for the LattePanda, do *nano .config* and search for *CONFIG_MODULE_ALLOW_BTF_MISMATCH* that is not set and set it:

```
#CONFIG_MODULE_ALLOW_BTF_MISMATCH=y

```

- Enable the following options in **Processor Types and Features**, theoretically it should be asked when using the *oldconfig* option:

```
CONFIG_EVL is set
CONFIG_EVL_LATMUS is set
CONFIG_EVL_HECTIC is set
CONFIG_IKCONFIG is set

```

The **CONFIG_IKCONFIG** is very important, when evl is lunched without it it will not work correctly since he need to access to the `/proc/config.gz` file but without this option he cannot do it and raised the error: *evl-check:BROKEN*.

Installation

```
make O=../linux-build bzImage -j8 (in Fedora)
or make O=../linux-build -j8 (in Ubuntu)
sudo make O=../linux-build modules
sudo make O=../linux-build modules_install
sudo make O=../linux-build install
sudo update-initramfs -c -k initrd.img-6.1.54-xenomai+  (-v)
(only in Ubuntu)
sudo update-grub (only in Ubuntu)
```

Additional Set Up

When running evl, we need to go in *Kernel/bin* and then run *sudo ./evl* . But with this following modification we will be able to lunch evl anywhere in the terminal and without going to the bin folder or using the sudo.

```
export PATH="$PATH:/home/macgyver/Kernels/bin"
source ~/.bashrc
```

It might happens that this step does not work correctly, meaning that when opening another terminal it does not recognise evl to fix it:

```
sudo nano ~/.bashrc
// Add this at the end of the bash file.
export PATH="$PATH:/home/macgyver/Kernels/bin"
source ~/.bashrc
```

Now on terminal, if we write evl or latmus or hectic it should work correctly.

```
sudo visudo
// read carefully the note after before doing it.
Defaults    secure_path = /usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/var/lib/snapd/snap/bin:/home/macgyver/Kernels/bin:
```

Notes:

In the default line that is in the visudo, we will need to add the path linked to our bin folder in my case i added: `:/home/macgyver/Kernels/bin:`

Links & Sources

All the installation process explained above, have been rewritten from this source: [45]

A.1.4 Remark

In this tutorial, I explain my usual procedure for building using only make, but we can also follow using the command *make -j($nproc)*. While I typically execute this process, there are instances when it runs overnight, making it challenging to monitor every detail of the installation. To address this, i've discovered a helpful command that enables us to capture the output in a text file. By using the *&* symbol, we can redirect the stderr (standard error) associated with our command, and by using the > symbol, we can redirect the stdout (standard output) related to our command. This approach ensures that we have a record of the installation process for later review. For example:

```
make -j6 &> installation.txt
```

A.1.5 Optimization

As mentioned in Chapter 3, in the section on optimization, here are the commands (done on Fedora):

Reducing Kernel Module Size

In the kernel folder, once we have already compile the kernel and the modules we can lunch this command. The strip will lower the size of the modules.

```
sudo make INSTALL_MOD_STRIP=1 modules_install
sudo make install
```

Grub Default Selection

List all installed kernels

```
sudo grubby --info=ALL | grep -E "^kernel|^index"
```

The output should be something like this:

- index=0: kernel="/boot/vmlinuz-6.8.9-200.fc39.x86_64"
- index=1: kernel="/boot/vmlinuz-6.6.22-rt27-nb"
- index=2: kernel="/boot/vmlinuz-6.5.2-rt8"
- index=3 kernel="/boot/vmlinuz-0-rescue-747e474ca6fc44a58657bc975f9183c0"

Now we can change the default kernel by using the index number for example:

```
sudo grubby --set-default-index=1
```

Verify the default kernel

```
sudo grubby --default-title
```

Limit Kernel Installation By Default

If we want to limit the number of kernel installed when we update in our terminal first thing is to see what is the limit number:

```
cat /etc/dnf/dnf.conf
```

Then we can limit for example i want to limit it to two:

```
$ sudo sed -i 's/installonly_limit=3/installonly_limit=2/' /etc/dnf/dnf.conf
```

Remove Old Kernels

First of all list all our kernels:

```
rpm -q kernel-core
```

Then select the kernel that we want to remove for example:

```
sudo dnf remove kernel-core-6.8.9-200.fc39.x86_64
```

A.2 Chapter 4: Communication Hardware

Here is more explanation about what is explained in the Chapter 4.

To view data on the PC terminal, two options are available: using minicom with the command `minicom -D /dev/tty/ACM0` (this operation requires root privileges) or utilizing the code provided in `robocup-info`. The latter allows both reading from and writing to the ACM0 file to communicate directly with the board. Various options have been integrated to enhance transfer efficiency.

By default, it is typically detected as ACM0, but if disconnected and reconnected rapidly, it may be detected as ACM1. To verify if the board is correctly recognized, the `lsusb` command can be used, and the virtual COM port should be displayed. It is important to note that the board is connected via two USB connectors, the ST Link and the virtual COM port, with the latter being used for data transfer. To determine which ACM port is active, the command `ls /dev/` can be utilized. From that, we can see which value is associated with ACM.

Dialout

To read from `dev/tty/ACM0` without requiring root privileges, it is possible to add oneself to the dialout group using the following commands:

```
sudo adduser --gid dialout macgyver
newgrp dialout
```

Note that `macgyver` is actually my username; in the case that we don't know our username, we can use the `whoami` command. By doing so, users can access the ACM0 port without needing elevated permissions.

Command

```
make
./serial-com
```

Bibliography

[1] Isaac Asimov. *The Robots*. 1950. URL: https://fr.wikipedia.org/wiki/Les_Robots.

[2] Asimo Robot. *Honda*. 2014 - 2022. URL: https://fr.wikipedia.org/wiki/ASIMO.

[3] Hanson Robotics. *Sofia Robot*. URL: https://www.hansonrobotics.com/sophia/.

[4] Robocup. *Description of the competition*. URL: https://www.robocup.org/.

[5] Robocup. *Kidsize competition*. URL: https://www.robocup.org/leagues/29.

[6] Hubert Woszczyk. "Simulation of complex actuators". Master Thesis. University of Liege, 2015-16. URL: http://hdl.handle.net/2268.2/1446.

[7] Guillaume Lempereur. "Development of an embedded servomotor controller". Master Thesis. University of Liege, 2015-16. URL: http://hdl.handle.net/2268.2/1641.

[8] Grégory Di Carlo. "Vision-based robot position estimation". Master Thesis. University of Liege, 2015-16. URL: http://hdl.handle.net/2268.2/1393.

[9] Tom Ewbank. "Efficient and precise stereoscopic vision for humanoid robots". Master Thesis. University of Liege, 2016-17. URL: http://hdl.handle.net/2268.2/3144.

[10] Odile Wauquaire. "Locomotion control system of a humanoid robot - A biologically inspired model". Master Thesis. University of Liege, 2016-17. URL: http://hdl.handle.net/2268.2/3169.

[11] Quentin Boileau. "Building a simulation environment for biped walking robots using Blender". Personal Project. University of Liege, 2016-17.

[12] Pierre Nicolay. "Integration of Libopencm3 in FreeRTOS for STM32F4 and STM32F3 MCU". Personal Project. University of Liege, 2018-19.

[13] Texas Instruement. *Paper Can Bus Explanation*. Pg 1 - 16. URL: https://www.ti.com/lit/an/sloa101b/sloa101b.pdf?ts=1716696603393&ref_url=https%253A%252F%252Fwww.google.com%252F.

[14] Maxime Leonard. "Design of communication protocol over CAN bus for humanoid robots". Personal Project. University of Liege, 2023-2024.

[15] Latte Panda 3 Delta. *User Manual*. 2020. URL: https://www.farnell.com/datasheets/3699523.pdf.

[16] STM32F429I DISCO. *User Manuel*. URL: https://www.st.com/en/evaluation-tools/32f429idiscovery.html.

[17] Alexander T Gillis. *What is a real time operating system*. 2022. URL: https://www.techtarget.com/searchdatacenter/definition/real-time-operating-system.

[18] Bootlin. *Understanding Linux real-time with PREEMPT_RT training*. 2004-2024. URL: https://bootlin.com/doc/training/preempt-rt/preempt-rt-slides.pdf.

[19] Giovanni Racciu. *RTAI 3.4 user manual*. 2006. URL: https://www.rtai.org/userfiles/documentation/documents/RTAI_User_Manual_34_03.pdf.

[20] Xenomai 4. *Xenomai Overview*. 2023. URL: https://v4.xenomai.org/overview/.

[21] Cyclictest. *Overview of Cyclictest*. 2023. URL: https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest/start#technical_explanation.

[22] Xenomai Team. *Testing using Latmus*. 2023. URL: https://evlproject.org/core/testing/.

[23] Makertai. *Installation RTAI*. Makertai, 2021. URL: https://github.com/relacs/makertai.

[24] Bounar Nadir. *Development of the software architecture for a mobile robot*. 2023-2024. URL: https://gitlab.uliege.be/Macgyver/test.git.

[25] Lyn Perrine. *Lubuntu Manual*. 2016-2024. URL: https://manual.lubuntu.me/stable/.

[26] Linux Voyager. *Voyager official website*. 2024. URL: https://voyagerlive.org/voyager-22-04-lts/.

[27] Puppy Linux. *Puppy Linux official website*. 2014-2024. URL: https://puppylinux-woof-ce.github.io/.

[28] Arch Linux Fr. *Arch Linux Fr official website*. 2024. URL: https://archlinux.fr/.

[29] Yoctor Project. *Yocto documentation*. 2010-2024. URL: https://docs.yoctoproject.org/.

[30] Fedora. *Fedora Workstation Documentation*. 2024. URL: https://docs.fedoraproject.org/en-US/workstation-docs/.

[31] Red Hat. *Red Hat official webpage*. 2024. URL: https://www.redhat.com/en.

[32] Enrico Marinoni. *Introduction to USB*. URL: http://www.emcu.it/Acronimi/USB/IntroToUSB.pdf.

[33] Libopencm3. *Home Page*. Libopencm3, 2023. URL: https://libopencm3.org/docs/latest/stm32f4/html/modules.html.

[34] STM32CUBE. *STM32CUBE official website*. 2024. URL: https://www.st.com/en/ecosystems/stm32cube.html.

[35] Libopencm3. *cdcacm.c file*. 2023. URL: https://github.com/libopencm3/libopencm3-examples/blob/master/examples/stm32/f4/stm32f429i-discovery/usb%5C_cdcacm/cdcacm.c.

[36] Eric Herman. *Makefile*. 2016. URL: https://github.com/ericherman/stm32f4-discovery-example/blob/master/rotate-chars/Makefile.

[37] Miquel van Smoorenburg. *Man Page Minicom*. 2005. URL: http://www.delafond.org/traducmanfr/man/man1/minicom.1.html.

[38] RXI. *Log Implementation*. 2020. URL: https://github.com/rxi/log.c/blob/master/src/log.c.

[39] Siddharth Chenna. *How to Setup PREEMPT-RT on Ubuntu 18.04*. 2020. URL: https://chenna.me/blog/2020/02/23/how-to-setup-preempt-rt-on-ubuntu-18-04/.

[40] Arturo Deza. *RTAI Manual*. URL: http://arturodeza.wdfiles.com/local--files/data-log/RTAI%20_FINAL.pdf#page39.

[41] *RTAI User Manual*. URL: https://www.rtai.org/userfiles/documentation/documents/RTAI_User_Manual.pdf.

[42] *RTAI Mailing List Archive: November 2020*. URL: https://mail.rtai.org/pipermail/rtai/2020-November/028219.html.

[43] RELACS. *makertai*. URL: https://github.com/relacs/makertai.

[44] *Real-Time Applications Using RTAI*. URL: https://www.youtube.com/watch?v=Gnyf_xf6kig.

[45] *Build Steps for the EVL Project*. URL: https://evlproject.org/core/build-steps/.