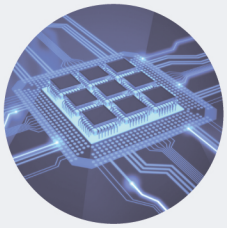
INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA DE SISTEMAS COMPUTACIONAIS CRÍTICOS



Space is So Monotonic: Introducing Dynamic Schedulers to Satellite Software

ALEXANDER PINHEIRO PASCHOALETTO
outubro de 2025

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Space is So Monotonic: Introducing Dynamic Schedulers to Satellite Software

Alexander Pinheiro Paschoaletto

Dissertation submitted in partial fulfilment of the requirements for the Master's Degree in Critical Computing Systems Engineering

Supervisor: Paulo Baltarejo Sousa

Evaluation Committee:

President:

Luis Lino Ferreira, Instituto Superior de Engenharia do Porto

Members:

Cláudio Maia, Sword Health

Paulo Baltarejo Sousa, Instituto Superior de Engenharia do Porto

Porto, October 1, 2025

Statement of Integrity

I hereby declare having conducted this academic work with integrity. I have not plagiarised or applied any form of undue use of information or falsification of results along the process leading to its elaboration.

I declare that the work presented in this document is original and my own, and has not previously been used for any other purpose.

I further declare that I have fully followed the Code of Good Practices and Conduct of the Polytechnic Institute of Porto.

ISEP, Porto, October 1, 2025

Abstract

The space industry has seen a trend shift in the recent decades by a handful of perspectives. The increase in competition via the introduction of more participants (both state-related and private), the expansion in mission scopes from simple research and defense to others such as internet service and even tourism, and the growing computational demand to handle these new missions are a few of them. On this scenario, satellites come as a fundamental element in most use cases. Just as any spacecraft, they need to be developed to withstand the harsh physical conditions of space, which imply techniques such as radiation hardening in some components to operate reliably, and are deployed in environments where maintenance is complicated. They are also expected to work autonomously for years, even decades.

Given these and other challenges, satellites traditionally have a long and expensive development phase, and are slow-paced when it comes to incorporating recent technologies. For the on-board computers that go within them, single-core processors of outdated architectures, fixed-priority schedulers and low CPU workloads are dominant. This paradigm works for now, but may not in the years to come as trends such as Artificial Intelligence (AI) and real-time video streaming may also reach the space domain.

The lastest iterations of space-oriented software do tackle the issue of development costs by introducing a greater code re-usability across missions, but little seems to be done regarding the software performance itself. In this context, this Thesis aims at bringing modern software paradigms into play by introducing the support of three widely known schedulers - Rate Monotonic (RM), Earliest Deadline First (EDF), and Constant Bandwidth Server (CBS) - into KARVEL, a space-oriented software originally developed by Critical Software. We evaluate their performance, advantages and shortcomings in both synthetic (by emulation work as busy-wait routines) and real-world workloads (by deploying it into a robot), and demonstrate that indeed dynamic algorithms such as EDF are capable of outperforming RM even on overloaded scenarios.

Keywords: Space, RTOS, Scheduling, EDF, RM, CBS

Resumo

A indústria espacial tem testemunhado, nas últimas décadas, uma tendência de mudança em vários de seus pontos. O aumento da concorrência através da introdução de mais participantes (tanto estatais como privados), a expansão do tipo das missões, passando de áreas como investigação e defesa para outras como serviços de Internet e até turismo, e a crescente demanda computacional para lidar com estas novas missões são algumas delas. Neste cenário, os satélites surgem como um elemento fundamental na maioria das finalidades. Tal como qualquer nave espacial, eles precisam de ser desenvolvidos para suportar as condições físicas adversas do espaço, o que implica técnicas como proteção adicional contra radiação em alguns componentes para operar de forma confiável, e são implantados em ambientes onde a manutenção é sempre complicada. Também se espera que funcionem de forma autónoma durante anos, ou mesmo décadas.

Diante desses e de outros desafios, os satélites tradicionalmente têm uma fase de desenvolvimento longa e dispendiosa, e são lentos quando se trata de incorporar tecnologias mais recentes. Para os computadores de bordo que os equipam, predominam processadores single-core de arquiteturas desatualizadas, escalonadores de prioridade fixa e baixas cargas de trabalho da CPU. Este paradigma funciona por enquanto, mas pode não funcionar nos próximos anos, uma vez que tendências como a Inteligência Artificial (IA) e o streaming de vídeo em tempo real também podem chegar ao domínio espacial.

As últimas iterações de software orientado para o espaço abordam a questão dos custos de desenvolvimento, introduzindo uma maior reutilização de código entre missões, mas pouco parece ter dado atenção ao desempenho do software em si. Neste contexto, esta tese visa colocar em prática paradigmas de software modernos, introduzindo o suporte de três escalonadores amplamente conhecidos - Rate Monotonic (RM), Earliest Deadline First (EDF) e Constant Bandwidth Server (CBS) - no KARVEL, um software desenvolvido para o espaço e mantido pela Critical Software. Para estes escalonadores avaliamos o desempenho, vantagens e desvantagens tanto em cargas de trabalho sintéticas (por meio de emulação como rotinas "busy wait") quanto em cargas de trabalho reais (por meio da implementação do software em um robô) e demonstramos que, de fato, algoritmos dinâmicos como o EDF são capazes de superar o RM mesmo em cenários de sobrecarga.

Contents

List of Figures

List of Tables

List of Abbreviations

WCET	**W**orst **C**ase **E**xecution **T**ime
GPOS	**G**eneral-**P**urpose **O**perating **S**ystem
RTOS	**R**eal-**T**ime **O**perating **S**ystem
UDFP	**U**ser-**D**efined **F**ixed **P**riority
OS	**O**perating **S**ystem
RR	**R**ound **R**obin
RM	**R**ate **M**onotonic
DM	**D**eadline **M**onotonic
SS	**S**poradic **S**erver
CBS	**C**onstant **B**andwidth **S**erver
FLP	**F**ixed and **L**ower **P**riority
SJF	**S**hortest **J**ob **F**irst
EDF	**E**arliest **D**eadline **F**irst
LLF	**L**east **L**axity **F**irst
EDZL	**E**arliest **D**eadline until **Z**ero **L**axity
EDF-WM	**EDF** with **W**indow-constraint **M**igration
NASA	**N**ational **A**eronautics and **S**pace **A**dministration
cFS	**C**ore **F**light **S**ystem
cFE	**C**ore **F**light **E**xecutive
MC	**M**ission **C**ontrol
SMED	**S**ingle-**M**inute **E**xchange of **D**ie
NVA	**N**on-**V**alue **A**dded

List of Symbols

C	Worst-Case Execution Time	s
Q	Budget	s
P	Period	s
U	Utilization Factor	%
τ_m	Real-time Periodic Task m	
J_n	Aperiodic Job n	

Chapter 1

Introduction

1.1 Context

The 20th century was arguably the most significative for human history in terms of technology advancements. What started as a race for the heavier-than-air flight that featured its first successful public demonstration in 1906 by the hands of inventor Alberto Santos-Dumont and his 14-bis eventually culminated, in 1969, into the first man leaving earth in a rocket towards the moon. Agricultural production saw a continuous increase in productivity thanks to the widespread adoption of machines such as the tractor, and the introduction of affordable contraceptives led to both great advancements in women's labor and reproduction rights and to a collapse in the foundations of Malthusianism. The development of polymers and the introduction of the assembly line in the factory floor allowed for the democratization of access to goods, and the rise of computers after World War II expanded horizons into nearly every conceivable aspect of civilization worldwide. The internet, originally developed as a military project, became mainstream and enabled quick and reliable communication across the globe, being nowadays considered an essential element of our everyday lives.

Another noteworthy advancement of the 20th century was the development of satellites, which are essentially radiation-hardened electronic devices deployed in orbit (of Earth or other planets such as Mars) and which provide unique data collection and communication services. Satellites were originally developed for military purposes within the space race context, but have naturally expanded their applications and are nowadays used by governments and private companies in weather forecasting, global positioning systems (GPS), environmental monitoring and, more recently, in providing reliable internet connectivity to areas such as the Amazon rainforest, which traditionally posed extensive challenges to ground infrastructure.

1.2 Motivation

However, despite their crucial roles and the intuition that satellites are highly advanced pieces of technology, the development cycle of spacecraft in general has been traditionally slow (10 years or more) for a myriad of reasons, of which could be cited certification, Verification and Validation, and tailor-made software development. On the latter, spacecraft diverge from

personal computers and smartphones in the sense that while the latter comes loaded with operating systems meant for a variable spectrum of usages, the first is developed having a collection of tasks (i.e. a *mission*) and operational guarantees in mind. An application that stops working in a smartphone generates annoyance to the end user; An application that stops working in a spacecraft generates loss of service at best and loss of lives at worst. Moreover, performing maintenance activities in a satellite or a rover exploring Mars (which has been needed in the past (Jones 1997)) is much more complicated than servicing a malfunctioning laptop.

These unique characteristics that constitute spacecraft missions make developing them a costly and complex endeavor. For this reason, the space industry is traditionally very conservative when it comes to the technology chosen for their projects. In terms of computational hardware, many current satellites operate using single-core processors, often with decades-old architectures and lower clock rates to limit power consumption. In software domain, the usage of multiprogramming Real-Time Operating Systems (RTOS) is ubiquitous, however fixed-priority schedulers are still dominant even when more promising dynamic options have been around for nearly the same time.

The recent years of space exploration saw a trend shift in both players involved (private companies vs state agencies) and mission scopes (tourism and broad internet vs research and surveillance) that begun showing need for acceleration in project development (due to the larger competition) and a higher demand for on-board software capabilities. Still, the outer space environment remains harsh and unforgiving, which means costs are still prohibitively high for deploying unproved solutions. The most notable consequence of these changes has been the regained interest in re-usability for both spacecraft components (such as rocket launchers) and on-board software: instead of crafting every element in the project from scratch, build the new projects from previously validated solutions. With re-usability, the time and money costs associated with the initial development phase gain the potential to decrease significantly. In the software landscape, two examples that can be cited are the NASA Core Flight System (cFS) and KARVEL. Both provide a software abstraction layer for a selection of possible RTOS, meaning one same application can be integrated for different projects (or adapt to changes in scope and requirements for the same project) with minimal time and effort.

However, there is still room for exploring the usage of tools provided in real-time theory, namely the adoption of more efficient real-time schedulers and SMP. To the best of our knowledge, the cFS only supports used-defined fixed priorities (UFDP) and KARVEL offers Rate Monotonic (RM) as well as UFDP. There is no support for dynamic priorities or scheduling servers to provide bandwidth guarantees to tasks, likely because these algorithms are still uncommon on RTOS implementations as well. This scenario translates into a contribution opportunity that resulted in the work conducted in this Thesis.

1.3 Contributions

In this document we extend the selection of RTOS supported by the KARVEL software platform, originally developed and maintained by Portuguese firm Critical Software, to comprise Zephyr RTOS, a fairly recent, scalable and feature-rich operating system currently maintained by the Linux Foundation, and which contributions made by us beforehand in the schedulers topic could be integrated into KARVEL. In the process, we also leverage our experience with this RTOS to implement new schedulers and enhance the architecture of existing ones. As a result, we provide a working implementation following the theoretical models for the aforementioned (but not yet existant in Zephyr) Rate Monotonic (RM), Earliest Deadline First (EDF) and Constant Bandwidth Server (CBS), as well as practical evaluation of their performance with both emulated (with busy-waits) and real-world (performing tasks in a robot) payloads. We show the real payloads are way more unpredictable than emulation, which emphasizes the need for testing in more realistic scenarios for a true assessment of the system performance, and prove that on overloaded systems EDF shows a much more balanced response than RM, being able to deliver shorter delays and keep the service of all tasks without starving the lower priotity ones.

1.4 Outline

The remainder of this document is organized as follows. Chapter 2 provides a brief overview of the base concepts used on real-time systems and the main scheduling algorithms for uniprocessors. Chapter 3 goes through some of the most widely known open-source RTOS used in both academia and commercial domains, highlighting their core features and support for scheduling algorithms. Chapters 4 and 5 analyze the two aforementioned software platforms, NASA cFS and KARVEL, providing some historical context, architecture, and casting brief analysis of their similarities and differences, including the choice for the open- vs closed-source approaches presented by each.

In Chapter 6, we describe the implementation work conducted in bringing support of KARVEL to Zephyr RTOS, highlighting the main design decisions, challenges, and the unit tests developed for each feature described. Furthermore, we provide the rationale behind every adaptation needed to enable compatibility between KARVEL and Zephyr features (for example creating an unified semaphore entity with dynamically configurable priority inheritance whereas Zephyr has mutexes with mandatory inheritance and counting semaphores without it). In this chapter we also provide the synthetic (busy-wait based) tests for the scheduling algorithms we implemented.

Then, Chapter 7 describes the requirements and architecture of the test application built on top of PegasoV3, a desktop robot arm of industrial topology and six degrees of freedom. We also provide a description of additional contributions made to the KARVEL abstraction layer to enable a proper functioning of the robot. Chapter 8 proceeds with the description and

execution of the tests conducted in the application, with increasing degrees of complexity, to verify the dynamics of each developed scheduling algorithm and their impact on the robot behavior as a whole. Each result is accompanied with the respective analysis of the results obtained. Finally, Chapter 9 concludes this document with the final remarks.

Chapter 2

Real-Time Concepts

This chapter will provide a brief overview of real-time theory concepts which will be relevant to understand the work conducted for this Thesis.

2.1 Real-Time Operating Systems

Operating Systems (OS) refer to a category of software usually written in a low-level language (such as C) and dedicated to manage the resources and execution context of various concurrent tasks in a way that they appear to be running in parallel even on single-core processors, where true parallelism is not physically possible. General-Purpose Operating Systems (GPOS) refer to OS usually made for a wide public and a handful of workloads such as document writing, web browsing and video rendering, thus being flexible and optimized for a good overall response of the many tasks it is subjected to process. Real-Time Operating Systems (RTOS), on the other hand, are usually designed for specific applications (such as robot firmware, airplanes, cars, satellites and stock exchange) and feature well-defined tasks and requirements that tend to exist for the lifetime of the application, trading the overall responsiveness of a GPOS for the determinism that the most important tasks will be executed under known timing constraints. The narrower scope of an RTOS makes it favorable for embedded systems, and as a consequence most RTOS are compatible with a myriad of architectures and hardware capabilities (while GPOS tend to be ported only for more powerful hardware due to their generalist design and end user approach).

2.2 Scheduler

The OS entity that governs the decision of which task to run based on priorities is known as the *scheduler*, and the decision itself is based off a *scheduling algorithm*. The algorithm performs the prioritization based on a given task feature - for instance, a simple numeric index informed by the application code, or the task activation instant, or a given deadline associated with it - and may interrupt an ongoing task in favor of another if that becomes active with a higher priority. The act of interrupting a task to start or resume another task is known as *preemption*. Most scheduling algorithms in the literature assume preemptions

are enabled within a task set, but that is not always the case in real-life systems. Some tasks may not be preemptible due to the nature of their work - for example, the uninterrupted transmission of a data packet through an Ethernet port - and most RTOSes offer means of disabling preemptions for a subset of the tasks.

2.3 Priority

All widely used RTOSes are designed in a way that allows the tasks to be assigned a *priority*, so that the task with the highest priority within a pool of active tasks is always the one executing in the CPU. When these priorities are known and decided before the task activation (often before the system initialization, even), they are called *static* or *fixed*. Conversely, when priorities are subject to change mid-execution and dependent on the current state of the tasks, they are called *dynamic*. Tasks are usually represented as threads or processes at implementation level.

2.4 Period

In modern real-time theory, the period is considered to be the *minimum* interval between activations of any given task. The first scheduling algorithms such as (Liu and Layland 1973) assumed that most tasks within a real-time system would be related to digital control loops, with goals of gathering sensor data and driving actuators. Thus, said tasks would be periodic, i.e. have a constant amount of time between their activations. This assumption is valid to this day, but since most modern real-time systems have both time-triggered and event-triggered types of work to perform, the minimum interval is used instead. Tasks that have truly constant activation rates are called *periodic*, and tasks with maximum known activation rates are called *sporadic*. If a task has undetermined activation rates, it is considered *aperiodic*. The period of a task is denoted by T.

2.5 Deadline

For most real-time systems, it is imperative that any requested work must be complete before the next execution request comes. Failing to respect this rule might lead to a multitude of consequences that can scale from a degrade in service quality (in soft systems) to actual loss of lives (in hard systems). On this note, the length of time that a task has available to execute the request is known as the *relative deadline* and is denoted by D. Conversely, the absolute time instant that is D time units ahead of the respective task activation (i.e. the longest allowed instant of execution) is known as the *absolute deadline* and is denoted by d.

2.6 Execution Time

The time that a task spends to fulfill its work once activated is known as *Execution Time*. In GPOS and RTOS with less critical tasks being executed, is is common to estimate an average execution time for a task in order to assess the system responsiveness. However, for critical task sets that cannot afford to miss a deadline it is typically estimated the longest possible time that a task can eventually spend. This value is known as *Worst-Case Execution Time* and is denoted by $WCET$ or simply C. It should be noted that, for a number or reasons related to the way modern processors are designed, it is actually hard to obtain an exact estimation for the worst-case scenario. This problem and relevant techniques to mitigate the effects of variable execution times are considered in Section 2.8.5.

2.7 Utilization Factor

Knowing the amount of CPU time a task might demand per request and the frequency of requests are fundamental measures to assess if a given task set is *feasible*, i.e no task will ever miss a deadline. In this regard, the ratio between a task's execution time C and period T (i.e. C/T is known as the *utilization factor* or *bandwidth* and it denoted by U. For a task set Γ, the total CPU utilization therefore corresponds to the sum of all individual tasks' utilization, i.e.:

$$U = \sum_{i \in \Gamma} \frac{C_i}{T_i} \tag{2.1}$$

Since a higher feasible U means the processor is able to run more tasks without missing any deadlines, the search for the maximum U that guarantees schedulability has been one of the main points of analysis for scheduling algorithms in real-time literature. Fixed priority policies typically guarantee lower amounts of U than dynamic policies (more details in Chapter 2). Furthermore, U can never be higher than 100% for a single CPU.

2.8 Scheduling Algorithms

As the decision of which task should be occupying the processor at each given instant directly affects the response time of the concerned tasks within a system, the selection for the scheduling algorithm might cast a significant influence in the system performance. As a consequence, works focused on developing algorithms that can efficiently handle the tasks date as far back as the first commercial-grade computers (IBM 1965), with works in academia rising in the same period (Kruskal 1969). The remainder of this chapter will explore some of the schedulers proposed in literature for real-time systems.

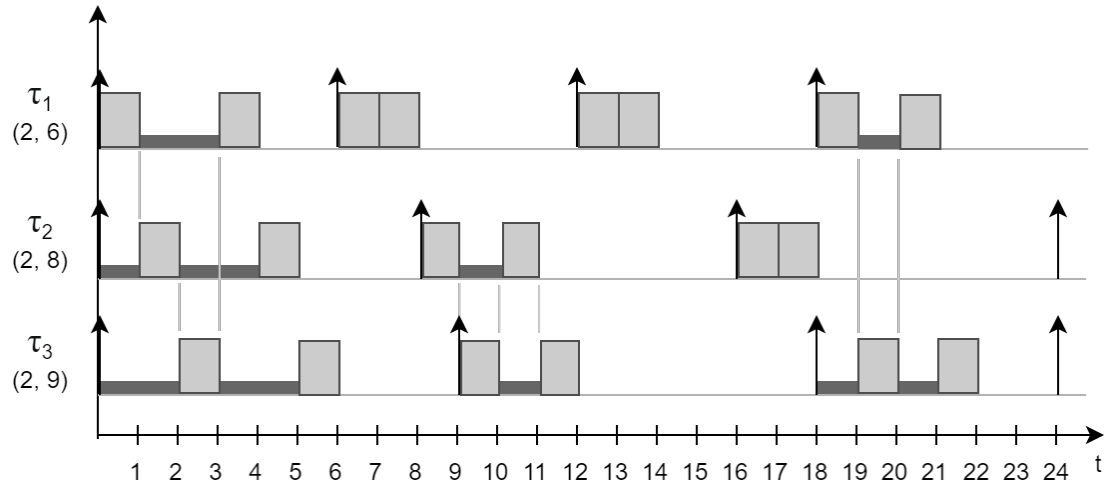


Figure 2.1: RR Scheduling.

2.8.1 User-Defined Fixed Priority

This is the simplest scheduling algorithm, if an algorithm at all. Here, the priority of each task is decided based on an arbitrary value given by the application designer, and the task with the highest (or lowest, depending on the RTOS of choice) numeric value denotes the highest priority. There is no dependency on any of the task's attributes such as period, laxity or deadline. Although the application can change any given task's priority at any given moment, it often remains the same through the lifetime of the application - hence why it is considered a fixed scheduling algorithm.

Due to its simplicity both in terms of kernel implementation and application usage, the User-Defined Fixed Priority (UDFP) approach is ubiquitous in the commercial RTOS domain. To the best of our knowledge, though, and due to the inexistent relation between task's priorities and runtime attributes, there are no specific schedulability analysis targeted for this algorithm. If the concerned tasks can have their CPU utilization estimated (for example, by estimating execution times and periods of the tasks), however, it is possible to infer the schedulability with tools such as the ones explained in Section 2.8.3.

2.8.2 Round-Robin

The Round-Robin (RR) is also a very simple scheduling algorithm as it requires no information from the tasks it allocates for execution (such as deadlines or periods). Instead, the jobs of these tasks are served in a First-In-First-Out (FIFO) manner and are allowed to occupy the CPU for at most a given amount of time, called a *time slice* or *quantum* (S). If the task finishes before the end of its quantum, it leaves the CPU normally and the next task in the ready queue proceeds to execute. If S is achieved, however, the task is preempted and moved to the end of the ready queue.

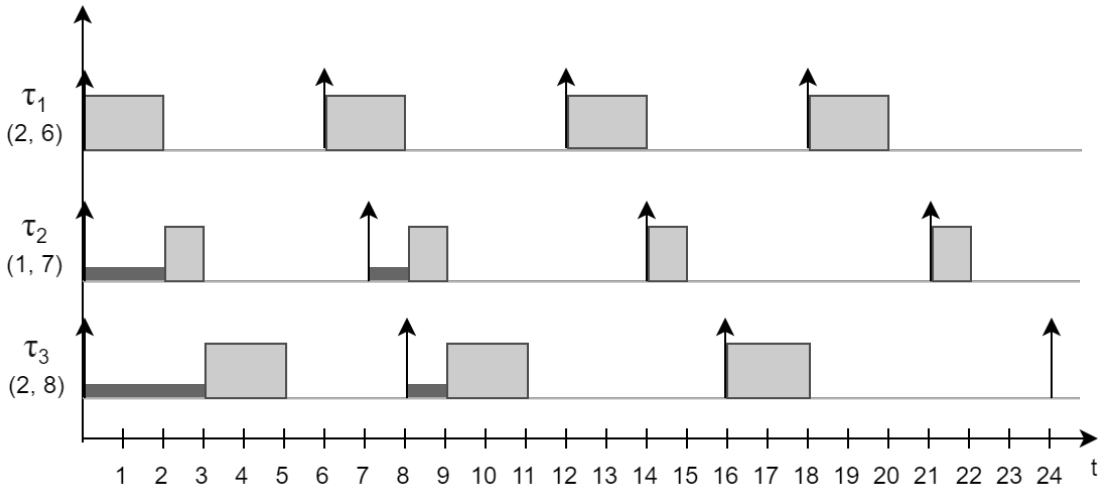


Figure 2.2: RM Scheduling.

The RR scales easily and predictably for any number of new jobs that might be inserted in the ready queue, but any new addition implies that all of the task set will be subject to an increased delay in response time. Figure 2.1 shows an example for a periodic task set running under the RR scheduler with $S = 1$. The tasks [τ_1, τ_2, τ_3] have periods of [6, 8, 9] time units and $C = 2$. One may notice that at every elapsed quantum, the current task is preempted in favor of the next unless there is no awaiting task for execution. The RR therefore guarantees that no task will *starve* (i.e. wait indefinitely for execution), but it tends to cause more preemptions than other scheduling algorithms.

2.8.3 Rate-Monotonic

The Rate-Monotonic (RM) was proposed in (Liu and Layland 1973) for the multiprogramming environment (i.e. an operating system running on single-core hardware) under the assumption that the concerned tasks are periodic, and that every time a task is activated, it must complete its work before the next activation comes. The deadlines are then equal to the periods, and the task with the smaller period has the highest priority. For tasks that don't have an explicit activation period, the minimum inter-activation time must be considered instead. Moreover, the same authors of the RM algorithm also proved it is optimal among fixed priority schedulers in the multiprogramming context, which ultimately means that if a task set cannot be scheduled by RM, it won't be schedulable by any other fixed priority policy.

Figure 2.2 shows the execution result of a set of tasks [τ_1, τ_2, τ_3] with periodic releases started at t=0 under the RM algorithm. Task τ_1 has a period (T) of 6 time units, while τ_2 has T=7 and τ_3 has T=8 time units. Under these circumstances, τ_1 has the highest priority of the set and τ_3 has the lowest. Thus, whenever τ_1 activates, it will preempt any running task and start executing immediately.

Schedulability Analysis

Note that knowing a task's period is enough for assigning priorities, but the execution times of the tasks are not taken into consideration. they are, however, used for the schedulability analysis of the task set. (Liu and Layland 1973) proposes a tool in this regard: for a set of m tasks, the system will have guaranteed feasibility if $U \leq m(2^{\frac{1}{m}} - 1)$. That condition is known as the *Least Upper Bound* (LUB_m) and is valid for any fixed-priority scheduling algorithm. Thus, for values of U of up to LUB_m, the task set is feasible. For values between LUB_m and 1, it requires further analysis and may or may not be feasible. Thus, the LUB_m is a sufficient, but not necessary, condition. Alternatively, (Bini, G.C. Buttazzo, and G.M. Buttazzo 2003) proposed the Hyperbolic Bound (HB) test to assess the schedulability, in which a task set would be feasible for RM if:

$$\prod_{i=1}^{m}(\frac{C_i}{T_i} + 1) \leq 2 \tag{2.2}$$

This condition offers similar complexity as LUB_m but an increase of the guaranteed schedulability by a factor of $\sqrt{2}$ as the task set size grows, being therefore less "pessimistic" than LUB_m.

2.8.4 Earliest Deadline First

The authors of RM also presented in (Liu and Layland 1973) the "Deadline-Driven" scheduling policy for single-core environments which would later be known as Earliest Deadline First (EDF). Whereas the DM algorithm selects which task to run based on the relative deadlines of the active tasks (the lowest the value, the highest the priority), the EDF scheduler makes the decision based on the absolute values of the deadlines instead. For that to work, at every instant *t* where a given task τ_i is activated for a new cycle, the scheduler defines its absolute deadline as $d_i = t + D_i$. Thus, at any given moment, the active task with the earliest absolute value has the highest priority.

The fact that priorities are constantly recalculated at runtime makes EDF a dynamic scheduling algorithm. A set of *m* tasks running under the EDF policy will be schedulable if:

$$\sum_{i=1}^{m}\frac{C_i}{D_i} \leq 1 \tag{2.3}$$

Which also means that the utilization factor is of up to 100% when the relative deadlines are equal to periods. This equation also shows that a) the periodicity is not required for the tasks, so EDF can easily be applied for both periodic and aperiodic tasks within the same set, and b) unlike the LUB_m condition for fixed priority algorithms there is no relation between the number of tasks and the schedulability guarantee. In fact, the utilization might be of 100% for any number of tasks.

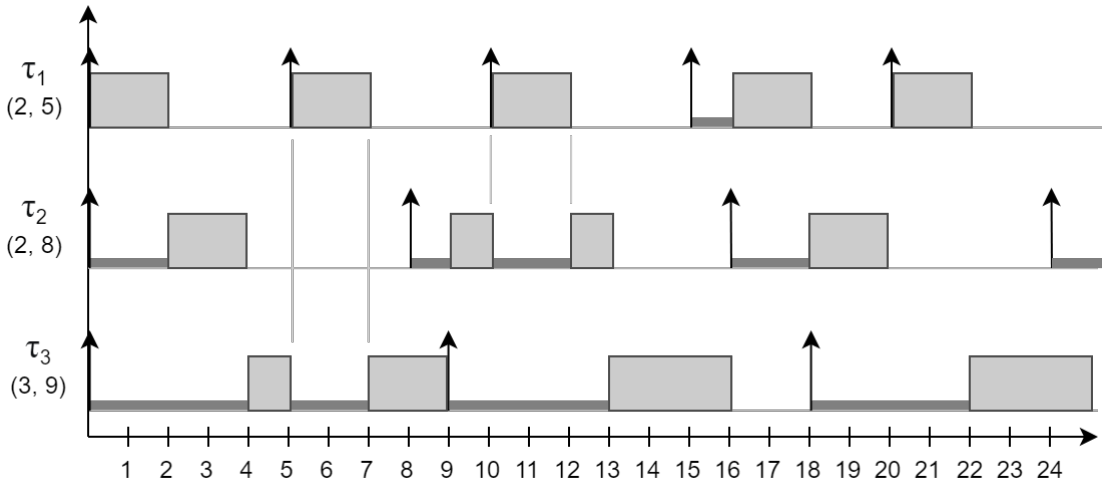


Figure 2.3: EDF Scheduling.

Figure 2.3 illustrates an example of three periodic tasks [τ_1, τ_2, τ_3] with (C_i, T_i) of [(2, 5), (2, 8), (3, 9)] time units, respectively, and deadlines equal to periods for simplicity. The combined utilization factor is $40\% + 25\% + 33\% = 98\%$, which means the system is feasible according to Equation 2.3. Such guarantee can't be made for either RM or DM, as $LUB_3 \approx 78\%$, and in fact this task set is not schedulable with static priorities: if RM was applied, the priorities would be fixed as $\tau_1 > \tau_2 > \tau_3$ and therefore at instant t=8 τ_2 would preempt τ_3, inevitably leading to a deadline miss at t=9. The EDF algorithm therefore allows for greater system usage and typically incurs in much less preemptions than RM, even if requiring a somewhat more complex implementation than the fixed-priority counterparts (G. C. Buttazzo 2003).

2.8.5 Scheduling Servers

Nearly all aforementioned scheduling algorithms rely on knowing the worst-case execution time (C) of the tasks involved to enable estimations of the system usage (U) and, therefore, infer the feasibility of the task set. The theory developed behind said algorithms, such as in (Liu and Layland 1973), also imply, for example, that the tasks have their jobs released periodically or with known minimum inter-arrival periods and that aperiodic tasks don't have critical deadlines. However, these conditions are not always the case for the systems that make use of these algorithms. Industrial robots, for example, generally have tasks that are both time-triggered (e.g. reading encoders, driving motors) and event-triggered (e.g. reacting to an emergency button press) with similar criticalities.

Besides periodicity, it is equally common that one same system might have to deal with both hard and soft real-time tasks. An example of soft real-time task would be the refreshing of an User Interface (UI), which although desirable to be made as quickly as possible, is not usually taken as the most critical part of any particular system (specially in RTOS domain). One possible way of dealing with both periodic and aperiodic tasks of different criticalities can be seen in Figure 2.4, which shows a hard real-time pair [τ_1, τ_2] running under RM with (C, T) of

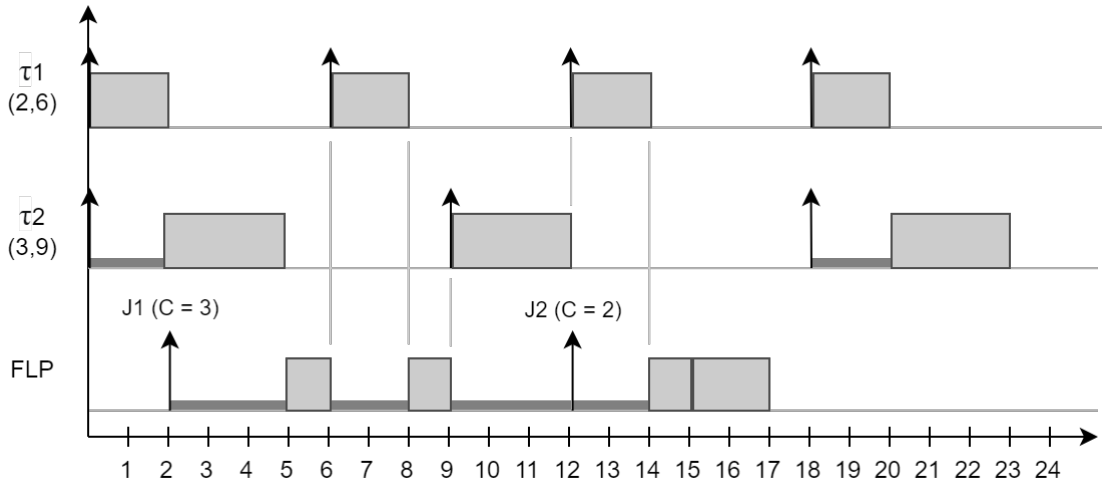


Figure 2.4: RM + FLP Scheduling.

[(2,6), (3,9)], respectively ($U = 66,6\%$). Alongside these tasks, a separate Fixed and Lower Priority (FLP) ready queue serves incoming soft real-time jobs in a FIFO manner only when the RM tasks are not executing. The FLP is easy to implement, portable to most algorithms and secures the schedulability of τ_1 and τ_2, but the usage of the laxity to serve the soft tasks ultimately means they are subject to large response times. In fact, Figure 2.4 shows that J_1 arrives at t=2 but only begins execution 3 time units later, being preempted twice in the meantime and only finishing at t=15. The delay in J_1's response is reflected in J_2's execution as well, which is forced to start 1 time unit later than it presumably would.

Thus, although the FLP strategy is presumed valid for serving soft real-time tasks, the observed delays might translate into a significative downgrade in the Quality of Service (QoS) of the concerned payload. It also should be noted that more often than not tasks require inputs from the environment (such as packets from a communication bus such as I^2C or CAN) that can vary significantly in response time, and non-linearities associated with modern computational systems such as speculative execution, cache misses and Direct Access to Memory (DMA), coupled with application design choices such as the use of unbounded recursion, make asserting the actual value of C a difficult problem to solve (Wilhelm et al. 2008). A common strategy to obtain C is to run the desired system for a while, collecting samples of execution times, but the truly worst-case response might never be achieved during simulation. Thus, the usual approach in such case is to note the longest sample and multiply it by a safety margin to yield C. If the estimated C is oversized, the system will behave as expected but significative computational resources might be wasted.

An alternative approach proposed in academia to deal with the aforementioned unpredictabilities on systems with real-time requirements and improve the tardiness implied by tools such as the FLP is to employ *scheduling servers*, which are essentially controlled ways of allowing aperiodic and soft real-time tasks to coexist with periodic and hard real-time tasks in a way that all scheduling guarantees remain secured. This Section presents two options among the

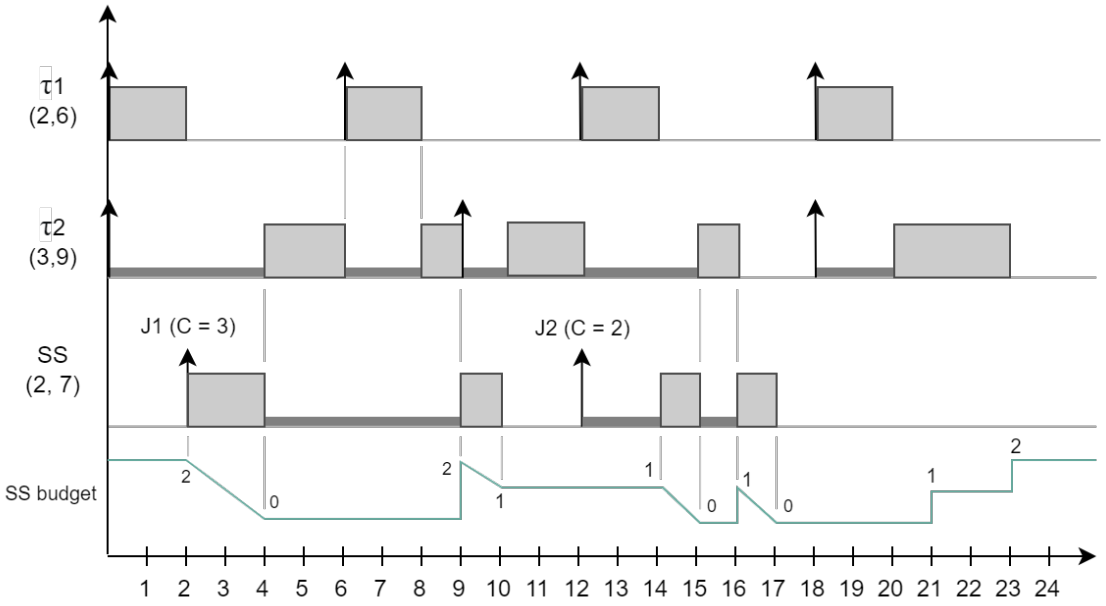


Figure 2.5: RM + SS Scheduling.

myriad of solutions proposed in literature and which reportedly have found use in commercial RTOS solutions (explored in Chapter 3).

Sporadic Server

The Sporadic Server (SS) is a solution originally proposed in (Sprunt, Sha, and Lehoczky 1989) to work as an extension of the RM algorithm, but which can be easily adapted for other fixed-priority policies as well. The SS is designed to receive aperiodic jobs and, depending on how it's dimensioned, it can either improve the response times for soft payloads or guarantee that the hard ones will meet their deadlines. It is defined by two static (configurable) attributes, a maximum *budget* Q_S and a *period* T_S, and three runtime (calculated) attributes, the current budget q_S, the replenishment instant RT and the replenishment amount RA.

While Q_S is equivalent to the maximum amount of time a job can be served before being obligated to relinquish the CPU, somewhat similar to the time slice in the RR scheduler, RT is the amount of time a job that has exhausted the budget will need to wait before being eligible to resume execution. On that note, An SS is said to be *active* if it is executing jobs or if preempted, yet with budget available (i.e. $q_S > 0$). It is said *idle* otherwise (no jobs to execute or preempted with $q_S = 0$). Thus, it has two moments of runtime calculation which are performed whenever it transitions from active to idle and vice-versa. When it becomes active at instant t_1, it sets $RT = t_1 + T_S$ and starts executing the given job. When it becomes idle, at instant t_2, it sets $RA = t_2 - t_1$.

Figure 2.5 modifies the example shown in Figure 2.4 by exchanging the service of jobs J_1 and J_2 for a Sporadic Server with (Q_S, T_S) = (2, 7). Considering task τ_1 has a period of 6 time units and τ_2 has a period of 9 time units, the SS sits right in between in terms of priority and will

always preempt τ_2 when it becomes active. As a result, it can be observed that in Figure 2.5 the server starts executing right away at t=2, when τ_1 has just ended its first job. At this very moment the SS has become active, so $RT_2 = 2 + 7 = 9$.

However, at t=4 the SS budget is exhausted by J_1. Thus, $RA_4 = 2$ (the amount of budget consumed). Since q_S has dropped to 0, the SS becomes inactive. At t=9 it is fully restored and, since J_1 needs to finish, the SS becomes active again. As a consequence, $RT_9 = 9 + 7 = 16$. J_1 completes at the next instant (t=10), leaving the SS with $q_S = 1$ and $RA_{10} = 10 - 9 = 1$. As there are no further jobs to execute at this point, the SS becomes idle.

Then, J_2, the next job, is inserted in the SS queue at t=12, but since τ_1 is also contending for the processor at this point, J_2 may only start execution at t=14. This is the moment the SS becomes active (when it starts the job, not when the job is inserted), so $RT_{14} = 14+7 = 21$. The budget runs out at t=15 and leaves SS idle until t=16, when RT_{16} is calculated as $16 + 7 = 23$ and the SS budget is restored in 1 time unit thanks to RA_{10}. The job J_2 finishes at t=17, leaving the SS idle and with $q_S = 0$. Finally, q_S is gradually restored to its full capacity at t=21 and t=23 thanks to RT_{14} and RT_{16}, respectively.

Note that the execution times of J_1 and J_2 are irrelevant for the analysis and calculations of RA and RT; the SS doesn't require any information on jobs given to it, as all required attributes for schedulability purposes can be derived from Q_S and T_S. As the SS was originally designed to work alongside RM, its priority is given by the defined T. However, if adapting for e.g. DM, the SS could be regarded as having $D = T$, or there could be an extra D attribute to represent its relative deadline. If adapting for UDFP, it would suffice to select a value to represent the SS priority.

When compared to FLP (Figure 2.4), it's possible to note that J_1 starts earlier and finishes faster, also not causing a delay in J_2. However, it should also be noted that if J_2 had $C = 3$ time units instead of 2, it would only be able to finish at t=22 due to the lack of available SS budget between t=17 and t=21. This condition is aggravated by the fact that the processor remains completely idle between t=17 and t=18, effectively meaning a waste in computational resources while still doubling the response time of J_2. Moreover, the hard task τ_2 is able to finish its first job on time, but right at its deadline, which in a real application could be a dangerous situation considering the difficulties of estimating C (Wilhelm et al. 2008).

Constant Bandwidth Server

The Constant Bandwidth Server (CBS) is another option of scheduling server whose main purpose is to guarantee temporal isolation for soft and non-real-time tasks within a dynamic priority environment. Just like the SS was originally designed to work with RM, the CBS was originally designed to work with EDF, with a target use case for multimedia applications such as Continuous Media (CM) (i.e. audio and video streams). These activities are very sensitive to delay and jitter and yet can feature highly variable execution times due to the frequent usage of compressed payloads (Abeni and G. Buttazzo 1998). As the encoding/decoding of frames

can have a way lengthier worst-case performance than the average response, and given that hard task sets are dimensioned with the worst-case scenario rather than average execution time, serving CM-related tasks directly in the task set might waste significative CPU resources. Thus, it's not an adequate solution.

Just like the SS, the CBS works with a budget Q_S that is consumed by the jobs assigned to it and a period T_S that is used to calculate the server absolute deadline. Unlike the SS, though, the CBS performs the budget replenishment immediately whenever it is exhausted, and when it does, however, it postpones its own absolute deadline as well. As a consequence, the CBS remains eligible for execution right after exhausting the budget, but with a lower priority; If even so it has the earliest absolute deadline among the active tasks, it will resume execution normally. This behavior makes the CBS a *work-conserving* server, which effectively contributes for an improved response time of the jobs served by it.

The CBS works with the notion of a guaranteed *bandwidth* (hence its name). The bandwidth is the ratio Q_S/T_S and represents the maximum percentage of the processor that the CBS is allowed to reclaim, being equivalent to the Utilization Factor (U) in the schedulability analysis (G. C. Buttazzo 2011). In order to assure the CBS will always use *at most* its assigned bandwidth, there are two key moments where it perform calculations to possibly intervene. The first is when the CBS transitions from *idle* (no jobs in the queue) to *active* (at least one job executing or pending)[1], and the second is the aforementioned moment where the running job completely exhausts the current budget (q_S). In the first case, the CBS was idle and had just received a new job to execute, so it verifies if the current bandwidth is compatible with the ratio Q_S/T_S. In other words, the CBS verifies if:

$$\frac{q_S}{d_S - t} \geq \frac{Q_S}{T_S} \tag{2.4}$$

If this condition holds, the CBS would serve the incoming job with a higher bandwidth than configured, which is not allowed (as it possibly jeopardizes the hard real-time guarantees of the remaining tasks in the system). Thus, it replenishes the budget to its full capacity and sets the new absolute deadline d_S as $t + T_S$. Then later, when the job completely exhausts q_S, it is once again replenished to Q_S but at the cost of redefining $d_S = d_S + T_S$.

Figure 2.6 demonstrates the CBS in action with the same task set as Figure 2.5, but with J_2 extended to $C = 3$ to better illustrate the capabilities and life cycle of the server. Interestingly, although in Figure 2.5 the algorithm was RM and the current example is following EDF, the scheduling output of tasks τ_1 and τ_2 yields the same execution order. The CBS receives the first job at t=2, therefore transitioning from idle to active, and since the absolute deadline wasn't set just yet, is is calculated as $d_S = 2 + 6 = 8$. The CBS therefore has d_S with an earlier value than d_{τ_2}, so it enters the CPU instead of τ_2. Then at t=4 the budget runs out, so q_S is

[1] Note that although q_S might drop to 0, it is immediately restored to its full capacity whenever this happens. Thus, unlike the SS, the budget level doesn't matter for the state of the CBS (if active or idle) - just if there are jobs to execute or not.

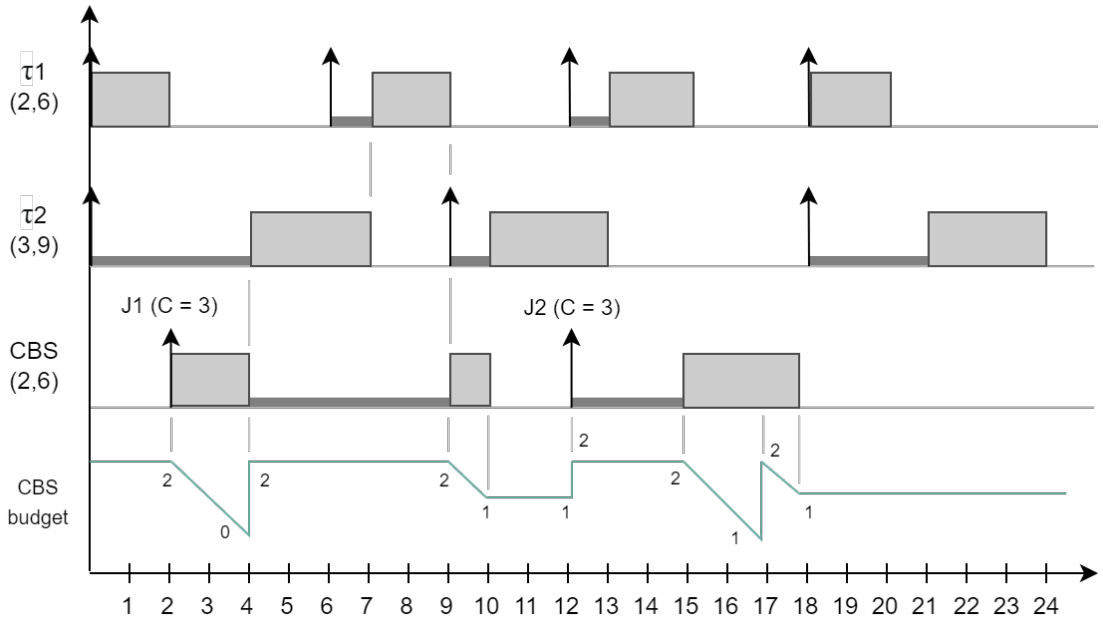


Figure 2.6: EDF + CBS scheduling.

replenished and $d_S = d_S + 6 = 14$. Thus, the CBS yields the CPU in favor of τ_2, which now has the earliest deadline of all. At t=6 the second job of τ_1 is activated with $d_{\tau_1} = 12$, so the CBS is only able to resume at t=9, when it finally returns to having the earliest deadline. J_1 finishes at t=10, leaving the CBS idle and with $q_S = 1$.

Then, at t=12, the CBS becomes active once again with the request for job J_2. It proceeds to bandwidth check: since $q_S/(d_S - t) = 1/(14 - 12) = 0.5$, it is greater than $Q_S/T_S = 0.33$ and therefore not allowed. Thus q_S is replenished and $d_S = 12 + 6 = 18$. The new d_S at this point coincides with τ_1 and τ_2 own deadlines, so in this particular case it is up to the kernel to decide how to break the tie - the most efficient way is to preserve the ongoing τ_2 and serve τ_1 next, as it keeps the cache values and favors the hard tasks instead of the soft job J_2, but any execution order would be valid scheduling-wise. We chose to illustrate J_2 being served last, as mentioned, and by doing so it exhausts the budget once again at t=17; Thus once again q_S is restored and $d_S = d_S + 6 = 24$. Note that there are no other active tasks in the system at this point, so even with the postponed deadline the CBS manages to keep running J_2.

Chapter 3

Open-Source Real-Time Operating Systems

We now proceed to an overview of the main features regarding some of the most widely adopted Open-Source RTOS that have active development, involvement and maintenance by the community.

3.1 FreeRTOS

FreeRTOS is a simple C/C++ library designed for embedded systems that easily introduces threads and real-time capabilities to otherwise simple bare-metal applications. It features user-defined fixed priorities, global round-robin time slicing for competing threads of the same fixed priority, and multiprocessing for truly parallel execution of threads is present on both Symmetric (SMP) and Asymmetric (AMP) variants. Further functionalities include queues for exchanging messages between threads, mutexes and semaphores, software timers and different strategies for dynamic memory allocation (AWS Open Source 2024).

The documentation declares official support for over 40 Microcontroller (MCU) architectures, and the convenience and portability of the library allows for easy integration on external toolchains such as Espressif's ESP-IDF. However, the kernel falls short on the selection of scheduling algorithms, synchronization services and optimized data structures for the run queue, with a simple double-linked list being adopted.

The simplicity of FreeRTOS also means it contains little support for non-kernel functionalities, limiting itself to a thread-safe implementation for the TCP/IP stack. This means that, official version considered, the bare metal approach is still required for most part of the application development and peripheral handling, such as GPIO and Bluetooth. As one of the most widely known RTOS, though, FreeRTOS has a great amount of external community contributions that introduce the missing functionalities. The project is currently maintained by Amazon Web Services and hosted on GitHub under the MIT license.

3.2 Zephyr

In contrast, Zephyr offers a more comprehensive framework with a larger set of supported boards and a python-based meta-tool to streamline the configuration, upgrading, building, flashing and debugging phases down to a single command-line interface. The scheduler features the same user-defined fixed priority strategy as FreeRTOS, but extended to allow the coexistence of preemptive and cooperative threads within the same taskset. The Round-Robin for same-priority threads is supported as well, but also with the possibility of adoption by specific threads instead of the entire kernel only (Linux Foundation 2015).

Earliest Deadline First (EDF) is also supported as a scheduling algorithm, and the ready queue can be structured with simple linked lists, multi-queues or red/black trees. True parallel execution is also enabled with the presence of SMP, although the scheduler remains global and the ready queue becomes limited to liked linked lists if one enables CPU affinity. Moreover, the kernel by default runs in a "tickless" mode, which contributes to less energy consumption and allows for the implementation of higher resolution timers.

Interestingly, One of the many corporative entities behind the Zephyr development and which provided most of the original codebase is Wind River, which already has an RTOS of its own. VxWorks is purposefully built for mission-critical projects involving military and aerospace applications, and a leading market player with certifications such as DO-178C/ED12C (Wind River Systems, Inc. 2024). It is also one of the supported RTOS of NASA's Core Flight System (see Chapter 4). However, VxWorks is not Open Source and features a different set of functionalities such as ROS2 and OpenCV compatibility. Zephyr, then, acts as a complement which allows the company to provide different options for its clients based on their needs.

Zephyr also distinguishes itself by having a Devicetree structure (DTS) for declaring each supported CPU and related peripherals for each specific supported board. There are also native drivers for wireless and IoT protocols such as Wi-Fi, Bluetooth, CAN, CoAP and MQTT. Although feature-rich, the build system is made with Kconfig Macros that can include or remove most functionalities. Thus, Zephyr remains scalable and compatible with resource-constrained devices. Zephyr is maintained by the Linux Foundation, hosted on GitHub and released under the Apache 2.0 license.

3.3 RTEMS

RTEMS (Real-Time Executive for Multiprocessor Systems) stands out as an RTOS certified for, and widely used in, space applications, having been selected as the RTOS of choice by European Space Agency (ESA) and National Aeronautics and Space Administration (NASA) in a handful of missions such as the Curiosity Rover on Mars. RTEMS is also one of the currently supported RTOS of the Core Flight System framework. It is implemented as a single address-space kernel and supports applications written in C/C++, Ada, Erlang, Fortran, Python/Micropython and Rust. Parallel abstraction directives such as OpenMP are also supported to some

extent. The RTOS is hosted on GitLab, and most of its source code and documentation are released under the Apache 2.0 license (RTEMS Project 2025a).

The RTEMS API is compliant with the POSIX standard, which facilitates the transition from other operating systems such as Linux. Like Zephyr, the project is also a framework with built-in meta-tool and compiler toolchains. The kernel offers a comprehensive set of features; For the scheduler, the base method is, as with the aforementioned RTOS, the user-defined fixed priority with a per-thread Round-Robin to alternate between same-priority threads. However, it is possible to also employ Rate Monotonic and EDF, with the latter limited to having deadlines always equal to periods. The ready queues are implemented with red-black trees when running under the EDF scheduler (RTEMS Project 2025b).

RTEMS also stands out on its feature set for having, on single-core systems using the EDF scheduler, a "hard" implementation of the Constant Bandwidth Server (CBS). As explained in Section 2.8.5, the CBS is a mechanism primarily designed to provide temporal isolation for soft and non-real-time tasks, allowing them to be executed alongside a hard task set without jeopardizing the given timing constraints. The RTEMS version of CBS, however, downgrades their priority to "background" level if the budget is exhausted until the next period comes[1] (RTEMS Project 2025b). Moreover, in RTEMS's implementation of the CBS the available bandwidth is checked more frequently, whenever the task attempts to unblock (instead of just when the task is activated), which implies a more granular control at the cost of some additional runtime overhead. The budget is also measured at the system tick instead of using a dedicated timer.

3.4 Apache NuttX

Apache NuttX is another option of RTOS which seeks a small footprint, wide architecture support and compliance with the POSIX and ANSI standards (in some cases where functionalities are not covered by those, it defaults to the models provided by Unix and VxWorks). Just like Zephyr, NuttX features a highly configurable Kernel with the use of Kconfig Macros, and can be set to run in "tickless" mode for compatible architectures (Apache software Foundation 2023). The available scheduling algorithms are the ubiquitous combination of user-defined fixed priority (UDFP) and round-robin (RR) for competing tasks of the same priority. NuttX also supports a protective mechanism for yielding a time-controlled execution environment for soft and non-real-time tasks called Sporadic Scheduling, which downgrades the thread priority if it executes for longer than a configured interval.

Moreover, the kernel also features event logging, multiple strategies for both dynamic memory allocation and file system management, message queues and microsecond-resolution timers. Access control to shared resources can be implemented with semaphores or mutexes,

[1] Note that although the term "background" is often used to denote a lower-priority execution when the core is free and no other task is ready (same as FLP scheduling), the RTEMS kernel actually suspends the task instead.

and priority inheritance is supported to help mitigate eventual problems concerning the usage of these mechanisms. An extensive set of device drivers and protocols such as I2C, PWM, CAN, WiFi, TCP/IP and Bluetooth LE. NuttX is implemented in C/C++ and hosted on GitHub under the Apache License.

3.5 Linux

Linux is the name of the open-source kernel that powers a plethora of operating systems such as Ubuntu and Android, and is currently used on most supercomputers (Elad 2023). Although it was not originally developed with real-time capabilities in mind, Linux started extending support for more latency-critical tasks with the introduction of `SCHED_FIFO` (First In, First Out) and `SCHED_RR` (First In, First Out with timeslicing) alongside the default `SCHED_OTHER` ('fair' scheduling for non-critical tasks) algorithm. Although the introduced options improved the response time for the real-time workload, they were not yet capable of guaranteeing deadlines (Reghenzani, Massari, and Fornaciari 2019). Linux is, along with RTEMS and VxWorks, supported by the NASA cFS (see Chapter 4).

In 2024, with Linux version 6.12, the last features of the `PREEMPT_RT` patch, which had been in development and discussion since 2005, had been merged into the mainline kernel. The most relevant contributions made over the years that enabled Linux to become real-time capable were high-resolution timers in version 2.6.16, which yielded nanosecond resolution for the ticks, an improved mutex and priority inheritance mechanisms in version 2.6.18, which allowed for safer resource sharing, the `NO_HZ` setting in version 3.10, which enabled tickless operation on idle cores, and the `SCHED_DEADLINE` algorithm in version 3.14, which introduced a new preemptive scheduling algorithm with higher precedence over the previous `SCHED_FIFO` and `SCHED_RR` (Reghenzani, Massari, and Fornaciari 2019, Corbet 2023 and Vaughan-Nichols 2024).

The introduced scheduler consists of a base Earliest Deadline First (EDF) algorithm augmented with yet another variation of the previously explained Constant Bandwidth Server (CBS) (refer to Section 2.8.5 for details). Yet, just like RTEMS the CBS is also modified for hard reservations. Under `SCHED_DEADLINE`, tasks declare a maximum "runtime" budget Q, a relative deadline D and a period P. The absolute deadline is dynamically calculated by the scheduler, and is initially set to 0. Whenever a task T becomes ready for execution, be that for the first time since activation or after recovering from a blocked state, the scheduler checks the following condition:

$$\frac{remaining\_runtime}{absolute\_deadline - current\_time} > \frac{Q_\tau}{P_\tau} \tag{3.1}$$

This condition is similar to the one introduced with the theoretical CBS model in (Abeni and G. Buttazzo 1998), but as previously mentioned is applied at every moment the task enters the ready queue (whereas in the theoretical model it is only applied when the task is activated). If

the absolute deadline is smaller than the current time or if the condition holds, the remaining runtime is replenished to the original value Q and the new absolute deadline is recalculated to $current\_time + D$.

On the other hand, if none of those are true, then the task remains with the previous values of deadline and runtime. It should be noted that the declared period P is primarily used only for admission control and in equation 3.1; The scheduler needs to know the expected CPU bandwidth required by the task and therefore assert the task set schedulability. If the required bandwidth is too elevated, the kernel might reject the task scheduling request.

Having the absolute deadline set, `SCHED_DEADLINE` proceeds with the selection of which task should run based on regular EDF parameters. The remaining runtime of a running task is decreased at every system tick, and once exhausted, the task is suspended until its absolute deadline. When this moment comes, the runtime is replenished, the deadline is postponed in P time units and the task becomes ready to resume execution. This approach is once again inspired by the CBS but not applied *ipsis litteris*, as in the theoretical model the runtime exhaustion would immediately trigger a deadline recalculation and keep the execution of the task if it kept having the earliest deadline of all (Abeni and G. Buttazzo 1998).

3.5.1 Suitability for embedded applications

These features lead to the observation that, with the integration of the `PREEMPT_RT` patch into the mainline kernel, a decisive chapter in the development of Linux was achieved. The kernel now enables greater suitability for critical systems and can be considered for a broader range of real-time applications such as industrial control plants, robotics and vehicle automation. However, Linux still remains targeted towards more powerful hardware, being advised for processors with a Memory Management Unit (MMU) that are typically not included within microcontrollersCarlson 2023. Thus, Linux plays a complementary role to options such as RIOT, which is built from the ground up for restricted hardware.

Chapter 4

The NASA Core Flight System

Space applications diverge from most in the sense they tend to value reliability over performance; after all, servicing a malfunctioning satellite already in orbit or a robot deployed in Mars is obviously complicated. For this reason, while Earth's bleeding edge technology (such as autonomous cars and humanoid robots) historically evolved at rapid pace, the trend of upgrading hardware and software capabilities in space has been traditionally very slow. The life cycle of the project phase in a satellite can be of up to 20 years, and single-core applications are still dominant. As a consequence, the pursuit for schedulers which could guarantee greater levels of system usage without endangering task set feasibility, as seen in Chapter 2, hasn't been historically a concern in the space domain.

In this context rises the Core Flight System (cFS), a software framework designed for space applications which can be a major source of inspiration for the works conducted in the Thesis. Thus, throughout this chapter we'll perform an overview of its architecture and main features. All the information presented had their sources taken from the documentation (NASA 2025) and training sessions (NASA 2021) provided by NASA. Other references will be eventually disclosed throughout the paragraphs.

4.1 Historical Perspective

the cFS was designed to tackle the re-usability of software in the space domain, where projects developed are traditionally tailor-made for their end applications. Each spacecraft used to have a bare-metal design that tightly coupled the selection of embedded processor, operating system (if any) and peripherals, and high degrees of rework were associated with every new project even though most of those share common functionalities such as communication with Mission Control (MC) on ground.

The cFS therefore seeks to provide the support to these common functionalities as well as an abstraction layer for the underlying system, which ultimately accelerates the project development phase. It isn't an OS, but a programming and runtime environment that runs on top of one and should be treated as a starting point for the software development of spacecraft.

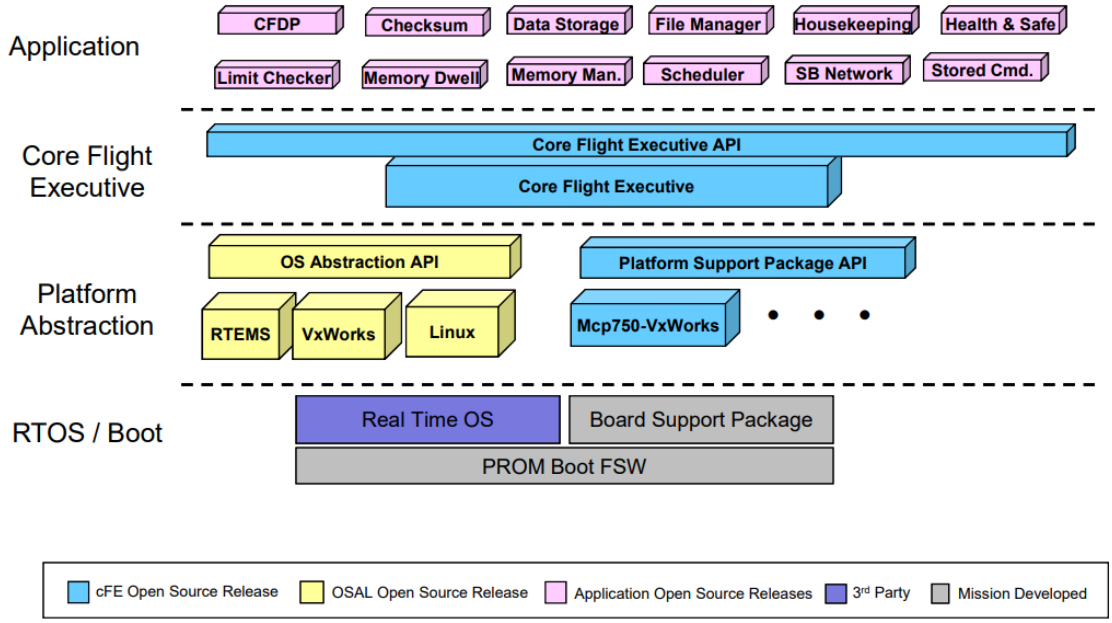


Figure 4.1: overall cFS architecture (NASA 2021).

In fact, the cFS currently supports Linux, RTEMS and VxWorks, and provides an API for the application that enables an independent development from the underlying OS of choice.

What makes the cFS adequate for space missions is that most of its core features have been contributed by NASA itself, amassing a knowledge base obtained from the many successful missions that can be traced back to the SAMPEX program launched in 1992 (Prokop 2014). Moreover, the re-usability of a codebase with an embedded test suite and a history of actual usage in space missions helps building trust in the software. The cFS is maintained by NASA Goddard Space Flight Center and has an open source version since 2015. This version is currently available on GitHub (NASA 2015).

4.2 System Architecture

cFS is a framework (i.e. a group of tools and applications to facilitate the development, test and deploy of software) for the space domain. It is specifically built with the goals of I) accelerating the time to deploy high quality flight software for a myriad of applications, contemplating mission durations of 10 years or more; II) reduce the need and complexity of maintenance; and III) enforce the adoption of common standards all around the project.

Figure 4.1 gives an overview of the most important elements within the cFS, divided by layers. The top layer is the collection of development-time test tools and runtime external systems (such as MC) that test and communicate with the space application. The subsequent layer comprises the application itself, with pre-developed modules that implement most of the common functionalities of spacecraft (this is the intended layer for regular users of the cFS).

The middle layer is known as the Core Flight Executive (cFE) and poses as the central piece of software that provides the platform-independent API infrastructure for the layer above, processing and routing requests to the underlying OS.

The cFE, however, also does not directly communicate with the OS. Instead, it relies on an internal API for that. The intent of having two API layers is to make the maintenance and development of upcoming cFE features easier, thus facilitating the open-source community involvement. At the same layer but aside from the OS there exists the suite of platform-specific drivers API, also used internally by the cFE, that enable support for peripherals such as MIL-STD 1553 bus. Finally, the RTOS/Boot layer is the external, machine-side edge which comprehends the actual OS. The OS must allow preemptive and real-time execution of multiple tasks, and as referred in Section 4.1, currently supports Linux, RTEMS and VxWorks.

For the remainder of this Chapter we'll examine the main aspects of the middle layer, cFE, which poses as the core implementation of the cFS bundle.

4.3 cFE Services

the cFE has 5 services that collectively deliver the core functionalities provided by cFS. A *service* is composed of one library and one application; the library declares the API, and the application is composed of one active task that processes the incoming requests through the API. Interestingly, some of these core services (e.g. the Software Bus) often uses the others (e.g. Executive Services) for specific functionalities (e.g. allocating memory for a pipe). The current Section casts an overview of these services.

4.3.1 Executive Services

The Executive Services (ES) is an ever-running task that whose purpose is to allow applications to request OS functionalities and oversee the whole application pool. In a nutshell, the ES is an active entity responsible for keeping execution straight; It can perform the start, termination and reset of specific applications that either requested it or threw exceptions at runtime, as well as reboot the whole cFE when specific circumstances are met. When errors happen they are logged by the ES, and when commands arrive from MC, it is up for the ES to process the request. The ES is also responsible for providing a basic file system for the applications, collecting performance metrics, storing data and managing dynamic memory allocation, for example. The sending of telemetry messages to MC is also within the chores assigned to the ES.

4.3.2 Software Bus

The Software Bus (SB) allows applications to publish and subscribe to messages in a similar approach as done with nodes in ROS2, with the fundamental difference that while ROS2 follows

a completely decentralized and network-based approach, the SB is meant to be only used internally by the flight software (recall that the exchange of information with MC is performed by ES). Messages exchanged with SB by default follow the Consultative Committee for Space Data Systems (CCSDS) Space Packet Protocol, but it's possible to set a custom data format by replacing the dedicated message module. Moreover, following the publish/subscribe paradigm, the publisher applications have no idea of who will receive the messages. The subscribers, by their turn, can either poll or pend for incoming messages.

Any messages exchanged with the SB are stored in *pipes*, which are nothing more than FIFO buffers that hold incoming messages until they are read out by a task. Pipes are created by, and assigned to, individual tasks, so each pipe can only be read by one task (although one task can have multiple pipes). For telemetry purposes, the SB stores runtime statistics of the pipes, such as the current queue length and high water mark, and is able to provide these information upon request of MC or application-generated command. The memory required for the pipes is statically allocated by the ES service.

4.3.3 Event Services

Similar to kernel logs, *event messages* are asynchronous ways of informing that a particular event has occurred in the system. The Event Services (EVS), by its turn, is a task whose purpose is to provide a centralized way of dealing with the event messages, allowing the requiring application to send, filter and log events. The messages are considered asynchronous because they are not generated periodically. There are 4 possible event types: Debug, Informational, Error and Critical.

4.3.4 Time Services

More often than not, applications need to perform time-triggered actions (such as the periodic gathering of sensor data) or simply keep track of the passage of time for various purposes. In space applications, it is also useful to have an external element as an absolute time reference instead of relying solely on the conventional time since system boot. To address these issues, the cFE provides the Time Services (TIME) utility.

The TIME application is a generic approach to time-keeping functionality, since different space missions might need specific ways of measuring time. Thus, it provides many time relations for application use. The *Ground Epoch*, for instance, marks an absolute timestamp for spacecraft calculations which is usually set as the same epoch defined in MC host computers or, more recently, the midnight of January 1st, 2001. The *Spacecraft Time* is the period since Ground Epoch, and the *Mission Elapsed Time* (MET) is the period since launch of the mission.

Other definitions provided by TIME include the tracking of Leap Seconds, which is an occasional adjustment added to the clock to synchronize atomic clocks with solar time (Laboratory 2025), and the *Tone*, which signals that the MET has incremented. The Tone is similar to a system tick in most RTOS, but it can be configured for a handful of sources other than the

conventional hardware timer, such as incoming payload from the GPS receiver or the MIL-STD 1553 bus.

4.3.5 Table Services

The Table Services (TBL) is a centralized way of storing and setting non-volatile environment variables, somewhat similar to a database. These variables may dictate to some extent the behavior of the flight software, like defining the attitude controller PID gains and telemetry filters, and are not expected to change frequently during runtime. The TBL is typically used by operations personnel in MC which oversee and remotely interact with the spacecraft, not by any other core application.

Mission-specific applications, however, can make use of tables as they wish. Any application can use TBL to register a new table and will be its "owner", being able to "share" it with other applications that will typically only perform read operations. Tables usually have two images, Active (currently in use by applications) and Inactive (a copy of the Active image that can be manipulated by the MC). After performing updates to the Inactive image, the MC can send a command to replace the Active counterpart with the new content, but the precise timing of the update is up for the application which owns the table to decide (thus, the update request is received synchronously but the effective update is performed asynchronously). Moreover, it is possible to assign a validation function to a table when it is registered that will perform an inspection on the table contents to check if they are logically correct. This validation is performed whenever a new version of the table is about to become Active.

4.4 cFE Scheduler

Although not one of the aforementioned core services, the Scheduler service (SCH) is also available as part of the official open-source distribution provided by NASA, and can be included in the project if the spacecraft could benefit of a synchronization mechanism for the many executing tasks (NASA 2017). Despite the name, the SCH does not substitute or directly interact with the kernel scheduler in the underlying OS, nor does it work based on priorities. Instead, it basically provides a method of periodically sending messages using the SB core application. These messages are effectively regarded as rescheduling signals that tasks can use to voluntarily yield the CPU, thus allowing others to execute. The SCH operates under a Time Division Multiplexed (TDM) algorithm, which provides a deterministic execution order for the task set.

Other than the SCH application, there is no other scheduling-related feature available in cFE; Mostly this is because the actual task scheduling is a concern of the underlying OS, and thus outside the scope of cFE. Ultimately, this means that the availability of kernel-level scheduling algorithms is completely determined by the OS and the support level of the destination

platform [1]. As a generic framework, the cFS only supports basic fixed-priorities defined by the user, which are set for each application's main task (and echoed to any created child tasks) at their start.

4.5 cFE Applications

Tasks in cFE are implemented as threads or processes in the OS down below. Each application is typically composed of one 'main' task, but it's possible for an application to create one or more child tasks from the main. These tasks will always be associated with the parent task. In the case of RTOS with unified memory address space like VxWorks, both parent and child tasks are completely independent from each other in practice and any relation between them exists only at the cFE layer. In process-oriented options such as Linux, where virtual memory addresses are adopted instead, the main task is translated as a process and the child tasks are regarded as pthreads.

The application users might start or stop the execution of tasks at any moment, provided that the concerned task is the main. Child tasks can only be created programmatically by the main through API calls for the ES core application, which also keeps track of the number of registered and active applications. There are two ways of starting an application: using the startup script or through a request issued by MC. The act of Interrupting an active application, however, can only be accomplished with another dedicated MC request.

Interrupting an ongoing application means to shut down all processes involved in the application, as well as releasing any memory resources previously allocated. The cFE, through its core ES application, performs the shutdown in a controlled fashion: it fires a timer and sends a cease request for the target application, giving it some time to clean its own resources and terminate execution. If the task doesn't acknowledge the termination until the timer expires, the ES forces the shutdown. This alternative is the last resource, since the cFE cannot guarantee that the memory has been properly liberated.

4.6 cFE Startup Script

The cFE initialization phase consists in reading a *Startup Script*, which is nothing more than a text file with a list of entries (one per column). Each entry specifies which applications should start executing automatically and which execution rules should be applied for each one. The system first checks for the *CFE_PLATFORM_ES_VOLATILE_STARTUP_FILE* at every processor reset, and if this file doesn't exist, the system proceeds to checking if there's a file passed as argument to the *CFE_ES_Main* function. Each entry contains a handful of attributes such as:

- *Object Type*: a flag to specify if the object to be initialized is a regular application or a library;

[1] The same case applies for multi-core execution: the cFE has no current specifications in this regard, although explicit support for SMP is within the open-source roadmap (NASA 2015)

- *Filename*: A name or path for the application in the internal cFE virtual file system. This attribute is relevant for the cFE alone and has nothing to do with the underlying OS' file system;
- *Entry Point*: The function that represents the task workload;
- *Priority*: The static priority of the application;
- *Stack Size*: Amount of stack to be allocated for the application; and
- *Exception Action*: How the cFE should proceed if the application threw and exception. 0 means to reset the processor only, and non-zero means to reset only the specific application.

It should be noted that if any error happens while processing either of the scripts, the cause of these errors will be registered in the System Log using the EVS core application.

Chapter 5

The KARVEL Software Platform

The most recent decades of the space exploration age saw a trend shift in both players and mission goals which ultimately led to the creation of the terms *Old Space* and *New Space* to broadly divide each period's dynamics: while in Old Space government agencies such as NASA and Roscosmos carried missions oriented to research and defense, in New Space the private sector is now progressively more involved and focusing on commercial services such as Low Earth Orbit (LEO) tourism and satellite internet (Baber and Ojala 2024).

Some examples of private firms currently involved in space exploration are SpaceX, Virgin Galactic, Turion Space, Boeing and Airbus, which subsequently partner with other companies such as Thales Alenia and Critical Software to develop, test and maintain the software and hardware involved in missions. Partnerships between private and public sectors are equally common in New Space, with, for example, NASA having elected SpaceX's spacecraft for several manned missions in the recent past (O'Callaghan 2020) and ESA awarding a contract to Critical Software for the Verification and Validation activities of the Gateway Space Station's International Habitat (I-Hab) (Software 2023).

Although the cFS explored in Chapter 4 provides an open-source, common and re-usable codebase for space applications which can largely decrease costs and time spent in the development phase, it can't completely eliminate the need for writing the code to support each mission's idiosyncrasies as they tend to be very unique. Some missions might not be well fitted for the cFS architecture, memory usage or power requirements, or require a smaller kernel footprint, which in turn favors a more direct abstraction layer between application and operating system, if any. Furthermore, the fact that the cFS is maintained by NASA, an American state agency, also raises concerns regarding the impact of eventual moments of political instability on projects developed outside the United States of America. As space exploration is still a very expensive endeavor, it's a risk that can't be neglected.

In this landscape enters KARVEL, a «software platform» which, at a glance, offers similar purposes as the cFS: allow a faster project development by enabling code re-usability. It has, however, some key differences in the design, features and development context which will be explored throughout the remainder of this Chapter. The KARVEL platform is also the implementation target of this Thesis, of which contributions were made in two fundamental areas:

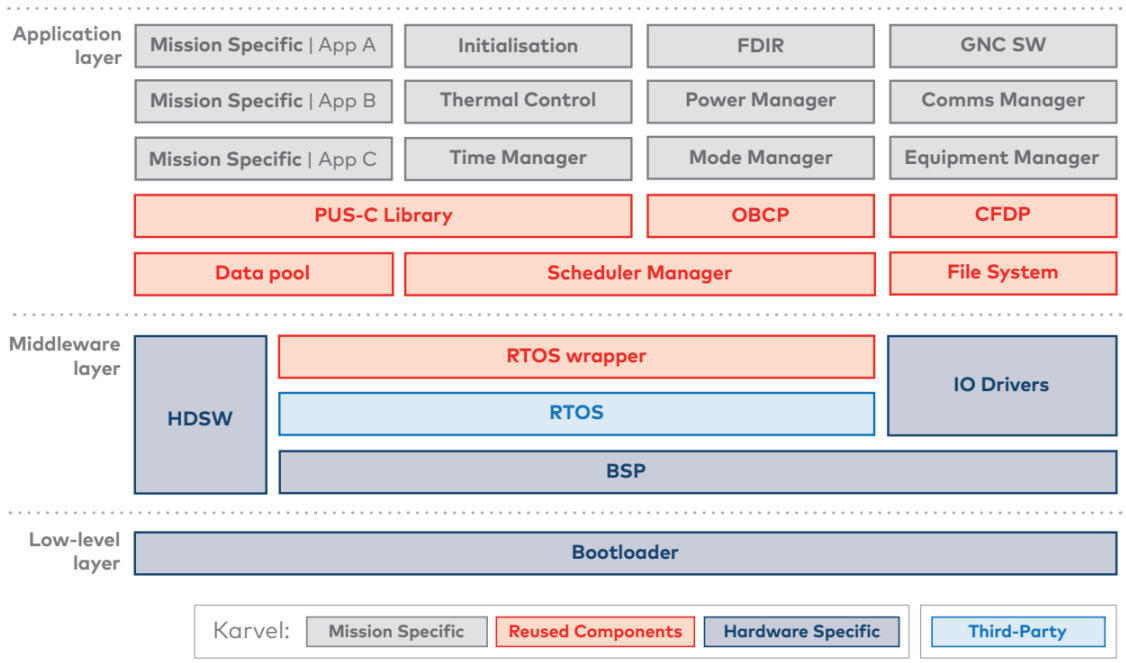


Figure 5.1: overall KARVEL architecture (Software 2025).

first, the support to a new RTOS, Zephyr, which will be described in Chapter 6, and second, the addition of new scheduling algorithms and support for multi-core execution.

5.1 Features Overview

KARVEL is a software solution developed by the Portuguese private company Critical Software (CSW) targeting space applications and built on the knowledge obtained from over 30 missions in which the firm has taken part since 1998 (Software 2025). In a similar goal as cFS, KARVEL aims at offering a single API, from the application standpoint, that is able to internally handle a myriad of hardware platforms (e.g. SPARC LEON3, NOEL-V, Xilinx Zynq and NVIDIA Jetson Xavier NX) and operating systems (e.g. RTEMS, FreeRTOS, VxWorks and Linux), thus making one same application easily portable and extensible for future missions to come. In contrast to most of the software analyzed in the previous chapters (including cFS), KARVEL is not open-source and access to the codebase for the purposes of this Thesis was granted by CSW.

Figure 5.1 illustrates the overall architecture of the platform, in which it's possible to see the pre-existing modules and interfaces supported by it. Notably, there is out-of-the-box support for a selection of standards and libraries, of which is worth mentioning:

- OBCP, which stands for On-Board Control Procedure and refers to a software engine that goes embedded in the spacecraft and processes scripts intended to control it in a highly adaptive manner, be that autonomously, in reaction to specific events or by intervention from Mission Control (MC) on Earth (Tipaldi et al. 2015);

- PUS-C, which means Packet Utilization Standard and is an application-level extension of the CCSDS Space Packet protocol for defining the format of telemetry and command payloads exchanged between MC and the spacecraft (PUSopen 2025); and
- CFDP, which expands to CCSDS File Delivery Protocol and allows the exchange of files between spacecraft and MC under harsh transmission conditions and with a configurable delivery assurance level (Yamcs 2025).

The architecture overview provided in Figure 5.1 also allows one to spot core differences and a few similarities with NASA's cFS (Figure 4.1). On both alternatives there are mission components at the application layer which need to be tailored for each project needs, however in KARVEL those involve functionalities which in cFS would be there by default and likely managed by the cFE; for example, the Time Manager application in KARVEL would be equivalent to the TIME core service in cFS, and the ES core service of cFS likely covers, among other things, the procedures ran by KARVEL's Initialization application.

The need to implement such components for every mission might seem to go against the reusability principle, however, this architecture choice is deliberate for a few reasons. One is that although the cFE services are designed to include the most common functionalities of space applications, it implies a generalist approach that may not be entirely needed for the suite of missions that the company behind KARVEL might have participated in. Hence, adopting just a subset of what would be a core cFE service might be a better idea if it already fits the intended purpose.

5.1.1 OS Abstraction

Another possible reason, and the main driver for the selection of KARVEL as a starting point for this thesis implementation, is that there is, a priori, no other executive component in the architecture stack aside from the OS kernel, which means the abstraction layer is primarily designed to interface directly with the underlying operating system. In other words, KARVEL does what the OSAL would do in the cFS. Hence, every API call internally maps to one or more kernel primitives, which contributes for a smaller code footprint at the expense of making the codebase less modular than the cFS counterpart.

This approach means that the development of new kernel features, such as multi-core support and the new scheduling algorithms that may be implemented in the context of this Thesis, are potentially more straightforward to do. It also means that for every new combination of RTOS and hardware platform, most parts of the API implementation need to be made from the ground up, but since KARVEL is a proprietary software with no open-source variant, the code maintenance and extension is normally performed by a dedicated team within the company. Thus, the project evolves in a more controlled fashion - and the modularity feature is not as crucial as compared to the cFS. Moreover, code quality can still be ensured by internal Continuous Integration and Development (CI/CD) policies, which is nowadays a common practice for any production-level software.

Interestingly, aside from the selection of directly supported operating systems (which comprise RTEMS, FreeRTOS, VxWorks, Linux and Windows with WSL) there is also a dedicated implementation of KARVEL that is compatible with the NASA cFS. In this particular case, the KARVEL API internally maps to the Flight Executive API rather than to the underlying RTOS. For this arrangement, there is no clear benefit compared to just using the cFS directly aside from enabling application portability to some of the other targets, should the mission requirements change.

5.1.2 RTOS Wrapper

The heart of KARVEL lies within the RTOS wrapper, which is where the API to all kernel services is defined and implemented; consequently, it's where all work related to this Thesis will be conducted. The main services available are listed below.

- **Logs:** set of methods to register arbitrary messages at runtime. The output of these messages is configurable, being normally UART for microcontrollers and the CLI interface for devices with a shell. According to the nature of each log, they are usually classified as INFO, DEBUG, ERROR or FATAL. KARVEL supports all these levels.
- **Barriers:** synchronization mechanism in which threads can stop and wait for others before continuing. A barrier can be set to automatically release when the number of awaiting threads reach a configurable amount, and individual threads can also set a time limit to give up on waiting if the barrier takes too long to fill.
- **Semaphores:** a counting mechanism used by threads to avoid race conditions when contending for shared resources. When a thread enters a critical section, it can iterate on the semaphore count to either signal the remaining threads where it is or actually block the access to the same resource by any other thread. In KARVEL, semaphores are used solely in binary mode (count is 0 or 1, i.e. section is *locked* or *unlocked*), which in real-time literature is also referred to as a *mutex*.
- **Interrupts:** asynchronous events usually associated with external devices and data buses (such as GPIO and UART) and which, when triggered, immediately preempt the current thread and execute a callback function defined by the application. The callback has a few restrictions associated with it - can't block waiting for resources, may take no arguments and must return void. The usage of this resource in KARVEL is application-specific.
- **Timers:** a special type of interrupt associated with the clock register, and which executes the application-defined callback after a given time has elapsed. A timer is typically started and stopped multiple times on the course of an application, and multiple timers can be defined. Most RTOS implement blocking functions on top of timers. Their usage in KARVEL is also application-specific.

- **Message Queues:** channels used by threads to asynchronously send messages to each other. Typically, any thread can send or read the payload of any queue, and there is usually no predefined format imposed by the system (that is specified when the queue is created). The messages are also usually sent and retrieved in a FIFO manner, however KARVEL also allows an urgent message to be pushed to the beginning of the queue. Other than this method, there is no priority distinction between the messages.

Tasks and Loader

KARVEL tasks are the kernel entities which represent the application logic, being implemented as processes in Linux and threads in all other supported RTOS. The main difference between a process and a thread is that processes have a virtually isolated address space, and threads don't. A process can spawn multiple threads to parallelize its inner work, and they will share the process address space, but a thread that spawns other threads will have no predefined relation with them.

Regardless of these definitions, KARVEL treats all tasks equally: they are always virtually independent from each other, in contrast to cFS which keeps track of relations between parent and child tasks even on RTOS that don't support it. This design choice is mostly because, although it is possible to launch new tasks at arbitrary moments of execution, in KARVEL the default behavior is to have the applications started altogether right after the initial boot by an auxiliary entity known as Loader (LDR), therefore being independent by default. Similarly to the cFE Startup Script, all applications that should start executing automatically are declared in a list.

The LDR is the closest equivalent to the ES core service in NASA cFS, as it manages the lifecycle of application tasks. However, unlike cFS, it is not designed to be an active kernel entity but a collection of APIs executed in the context of whoever calls each of its methods. During the initialization phase, the main task invokes the LDR startup function, which parses the application list and starts all of its entries (hence, the execution context is the main task). Later on, when any given application desires to finish, it calls the LDR termination function (thus, the execution context is the ending task), which attempts to release all resources allocated by the task before shutting it down.

The LDR also hosts most of the data used by the services listed in the beginning of this Section, such as semaphores and barriers, and keeps track of some runtime information regarding the task set which can be used for synchronization. For example, it is expected that a KARVEL task will announce to the LDR the phase it is currently in: initializing, ready, running, blocked or ended. The LDR keeps this information at hand, and offers a specific API function that allows calling tasks to block until every other task in the set has also achieved the "ready" phase. This mechanism ensures the whole system is ready before proceeding.

Scheduler and Multi-core

When it comes to the selection of kernel scheduling algorithms, KARVEL offers slightly more options than NASA's cFS: while cFS is limited to user-defined static priorities (explained in Section 2.8.1), KARVEL takes an extra step by offering Rate-Monotonic (RM, explained in Section 2.8.3) as well, which is interesting for real-time applications but still implies that the task set behaves periodically (which is not always the case, specially for interrupt-driven tasks). A similar situation happens in terms of multi-core: while the open-source distribution of the cFS has no official way of enabling SMP (although support for it is planned within the roadmap (NASA 2015)), KARVEL offers the basic CPU affinity at the task launch, which allows for the usage of partitioned scheduling. It should be noted that the support for SMP is also dependent on the target hardware and selected RTOS.

5.2 Considerations on Open- vs Closed-Source

The approach taken by cFS of being an open-source software is both a benefit and challenge. The major upside is that as the project codebase becomes available for community contributions, it might release the original authors from the responsibility of maintaining the project in the long run. For single developers such as Linus Torvalds, the creator of the Linux kernel, this means one can retire from the role of maintainer without risking the continuity of the project in which so many parties might depend on. For companies, it means the project can be less subject to the risk of losing complete support on bankruptcy events or, more commonly, intentional decisions to move on to new products and discontinue the old ones. However, it should be noted that an open-source project might become too dependent on community efforts for support, which is often made by people in their spare time, and come with no guarantees of liability.

Two examples of open-source software that have successfully taken advantage of this characteristic are the aforementioned Zephyr RTOS, which in its 2016 public launch had 4 companies as founding members but nowadays counts with official involvement from more than 40 software and hardware firms (Nashif 2022 and Foundation 2023), and Cura, a 3D printing slicer tool which was originally developed and is still managed by Ultimaker but leverages a strong community involvement, acknowledging their support at every new version released (Ultimaker 2023). Another example of an open source project that also clearly showcases this advantage is the Demoiselle, a single-seater light airplane developed by Alberto Santos-Dumont in the early 1900's and made publicly available in the Popular Mechanics magazine. The files are still accessible to the current date, more than a century after the project was released (Joerin 1910).

However, the clear exposure of a codebase to broad access poses the challenge of cybersecurity simply because the code also becomes directly available to malicious agents, which

can then examine and exploit any vulnerabilities found within it much easier than on closed-source counterparts. On the modern age where most services rely on internet connectivity, this is a major concern and requires quick and effective actions taken by maintainers once attacks are discovered. There are recent examples that back this claim: in 2024 it was found that some Linux-based distros were subject to a "backdoor" attack through a widely used library related to OpenSSH that ultimately allowed remote code execution on infected machines. This vulnerability received the highest criticality score possible in CVE metrics, and was actually inserted in the codebase by a trusted contributor (Red Hat 2024 and Akamai 2024).

NASA's cFS is not free from attacks, either: for instance, a 2025 security assessment conducted on version 6.7.0-Aquila of the cFS (the latest stable release at the time) revealed multiple vulnerabilities in many components of the framework, from the Memory Management (MM) module to the OS Abstraction Layer (OSAL), which opened a breach for attacks such as denial of service, file writing outside designated directories and remote code execution on the invaded spacecraft (Ferreira 2025).

Thus, although satellites and other spacecraft are not currently a typical target of regular hackers, it's worth to note that the trend shift in space exploration occasioned by the participation of private companies (which tends to grow in upcoming years), specially in the field of satellite internet, might eventually increase interest from attackers as well. Moreover, space equipment employed by military forces can be equally targeted by conflicting nations in war efforts, and cyberattacks can be just as effective as missiles when it comes to connected systems.

A priori, as a closed-source platform, KARVEL has limited information publicly available and thus offers a considerably smaller attack surface when compared to the cFS, which despite not guaranteeing security, might be more adequate for certain missions. However, if on one side the source code is proprietary, on the other most interfaces used in space and naturally supported by KARVEL, such as CAN and MIL-STD 1553, have vulnerabilities of their own that should be considered (De Santo et al. 2021). Furthermore, as KARVEL is still in essence an abstraction layer, it runs on top of open-source options of RTOS which, by their turn, could be exploited.

It should also be noted that the field of cybersecurity evolves at a rapid pace, which implies that security patches should be occasionally applied as part of the maintenance plan in every modern software. As a consequence, when electing the software platform for future space missions (which may have a planned durations in the order of a decade), the apparent cybersecurity edge of a closed-source solution such as KARVEL should be carefully weighted with concerns on long-term support. As previously mentioned, the major benefit of an open-source solution is that it can become decoupled from the original author's availability to correct bugs, handle reported vulnerabilities and add new features to the codebase. This might not be the reality for KARVEL, which is solely maintained by CSW, but the company's tight involvement in the project might also mean a quick and centralized response when it comes to maintenance efforts.

Chapter 6

Extending KARVEL to Zephyr RTOS

As previously stated in Chapter 5, The KARVEL platform was selected as the implementation target for the work of this Thesis due to its adoption in space applications and a more direct relation with the underlying RTOS when compared to the NASA cFS, which would presumably reduce the complexity of extending the original API. Having this in mind, the contribution of this Thesis was decided to be twofold. First, leveraging both the author's familiarity with Zephyr RTOS, a whole new implementation of KARVEL was developed for this operating system having the RTEMS variant as a starting point. Second, considering that KARVEL, like cFS, offers basic support to scheduling algorithms and multi-core execution, it was decided to extend the existing API in a way to also include more algorithms. The remainder of this chapter will thus describe the contributions made to port KARVEL to Zephyr RTOS.

6.1 Starting Point

RTEMS was the operating system of choice to base the port for Zephyr because there are many similarities between both RTOS when it comes to the feature set. for instance, both operate on a single, shared address space and have kernel primitives such as timers, message queues, semaphores and mutexes with priority inheritance. The intended usage for each primitive (as in the way they should be configured and used in runtime) is also very similar across the entire feature set, which again was considered ideal for both accelerating development time and ensuring a consistent behavior of the KARVEL API to the eyes of the application.

Naturally, even though similarities in design and features exist on both RTOS, it wasn't always possible to just replace the RTEMS API for the Zephyr equivalent because there are occasional differences in configuration options and metadata stored for some objects. For example, RTEMS supports the creation of semaphores (binary or not) with the option to leave priority inheritance enabled or disabled, whereas Zephyr applies the distinction by leaving mutexes with priority inheritance enabled and counting semaphores without it. Since KARVEL follows the RTEMS convention, extra implementation had to be carried on top of Zephyr semaphores to enable them to have optional inheritance as well.

As a consequence, for the whole porting phase, the RTEMS documentation and source code were frequently compared with the Zephyr counterparts in order to guarantee the correct behavior of the resulting implementation. Whenever the need for adaptation arose, a subsequent analysis followed to decide whether the implementation should happen at the abstraction layer or directly in the kernel code. When the latter path was chosen, the resulting additions to the codebase were later converted into contributions to the Zephyr RTOS repository, in a way to participate in the open source community.

6.2 Porting Results

We now proceed with describing the work conducted on most of the kernel services supported by KARVEL, highlighting the main differences and similarities with the original RTEMS port.

6.2.1 Barriers

A barrier is a synchronization mechanism in which all participating tasks proactively block their own execution and wait for a given condition to be achieved before carrying on. When a task is waiting on a barrier, it remains inactive and therefore consumes no CPU time. The condition for release can be arbitrary (such as a global variable achieving a given value), but it is often related to a specific amount of waiters being achieved.

A real-world example of a barrier usage would be the coordination of movement in a multi-axis robot. Robots are often manufactured with stepper motors to drive the joints, as they tend to have fine-grained precision and are easy to control. In a complex movement where multiple joints are employed, the application must calculate each joint step interval in a way that they all start and finish their move together. However, due to the inherent coarse resolution of digital math, that is almost impossible to guarantee and some motors will typically stop moving slightly after the others. For this reason, the application cannot rely on timers or simply watching a given motor step count to consider a movement done; it's better to have a barrier in which all motor-driving tasks should wait after completing their own movement.

Thus, while semaphores locally synchronize the access of shared data, barriers are noteworthy for synchronizing the application workflow in general. It is therefore an important topic in parallel programming, being adopted by standards such as OpenMP (ARB 2018) and fully supported by RTEMS. KARVEL abstracts most of functionalities implemented by the latter, of which can be cited:

- `TLCH_BarrierCreate`: initializes a static memory region with the control block of a new barrier. Here a configuration parameter should be passed to indicate whether the barrier is automatic (i.e. will release once a given amount of waiters is achieved) or manual (i.e. keeps blocking until a task decides to release it), as well as the maximum amount of waiters if the automatic option is chosen. This function returns the ID of the barrier,

which is used by tasks to reference which one to wait on, and will return an error code if the maximum amount of barriers have already been created.

- `TLCH_BarrierDelete`: releases the barrier referenced by the ID passed as a parameter (thus unblocking all waiters of this barrier) and clears the memory region allocated for the control block. Will return an error code if the given ID does not refer to any barriers. There is no ownership concept applied (meaning any task which has the barrier ID can delete it, not just the task that created it).
- `TLCH_BarrierWait`: invoked by tasks which intend to wait on a barrier. The ID must be specified, as well as the maximum time in milliseconds that the task is willing to wait before giving up and resuming execution. Returns an error code if the given ID is not associated with any barriers.
- `TLCH_BarrierRelease`: called by any task that wishes to arbitrarily release a barrier instead of allowing it to reach the maximum count of waiters. Accepts the barrier ID and returns the number of released tasks upon a successful call. Alternatively, an error is returned if the ID is not associated with any barriers.

This API subset is closely mapped with the functionality offered by RTEMS, meaning that unlike the aforementioned logging API, porting to this RTOS has been pretty straightforward. For Zephyr, however, a different approach had to be made as it has no explicit support for barriers, but a somewhat similar feature called *conditional variables*. Basically, a conditional variable is a FIFO queue where tasks can wait until a given system state is reached. This state is necessarily application-specific and could be represented through a collection of variables, for example. When some running task identifies the state has been reached, it may *signal* the event to the queue by invoking either `k_condvar_signal` or `k_condvar_broadcast`. While the first releases only the task in the queue head, the latter releases *all* tasks in the queue.

```
1  typedef struct {
2      ...
3      uint32 waiters;
4      uint32 max_waiters;
5      struct k_condvar condvar;
6      ...
7  } TLCH_Barrier_t;
```

Listing 6.1: Core barrier data structure.

Thus, it was clear that the broadcast function could be coupled with some state variables to achieve the purposes of a barrier. A data structure to store these variables was created, and the core fields can be seen in Listing 6.1. Namely, they are the current and maximum amount of waiters, respectively, and Zephyr's `k_condvar` struct that represents the conditional variable. The usage of the created `TLCH_Barrier_t` data structure is then performed internally by the KARVEL API.

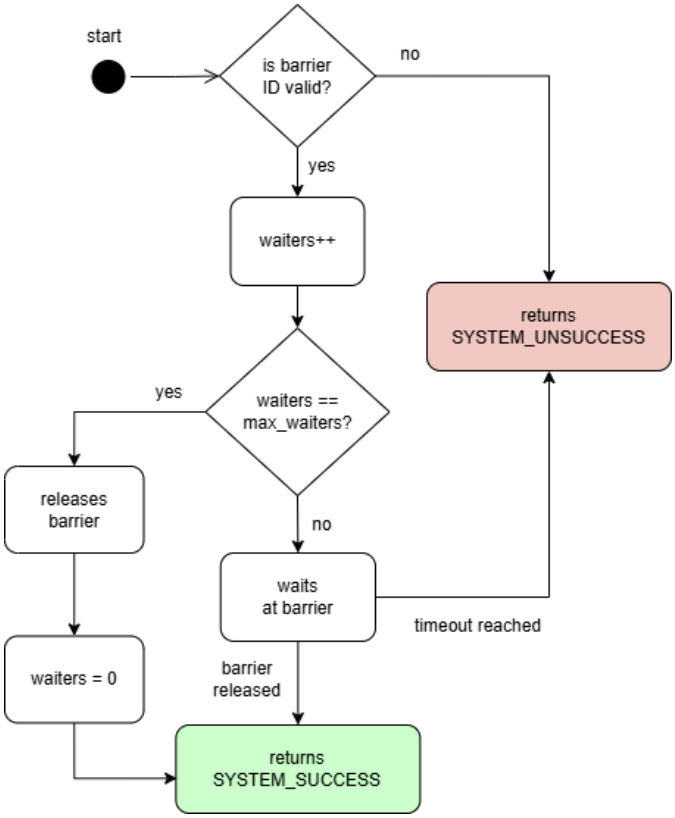


Figure 6.1: TLCH_BarrierWait execution flow.

Figure 6.1 illustrates the resulting execution flow of the `TLCH_BarrierWait` implementation on Zephyr. The first step in this function is to verify if the given ID is valid and, if so, increment the number of waiters in the barrier. If the new value is equal to the maximum count, then the barrier is automatically released (i.e. `k_condvar_broadcast` is invoked) and the waiter count resets. If not, then the task calls `k_condvar_wait` and blocks in the barrier until it gets released by another task or the timeout (if any) is reached.

Tests

Besides the porting of the KARVEL API to Zephyr, a few unit and functional tests were developed to evaluate the correctness of the resulting implementation. For the barriers API, these targeted the creation, deletion and regular usage of barriers:

- **creation:** As the memory space for storing all barriers' data is statically declared as an array of type `TLCH_Barriers_t`, there is a limited (yet configurable) amount of barriers that can exist at once in the system. Thus, this test was developed to attempt creating more barriers than the maximum allowed;
- **deletion:** The same principle as creation, but in reverse: the test created x barriers and attempted to delete more than x. The reason to first create barriers was to verify if they were correctly deleted before attempting to delete non-existing ones; and

```
[00:00:05.014,000] <inf> KARVEL: TEST 1: 5 threads will wait on a 10-thread barrier. Expected fails: 5
[00:00:08.018,000] <err> KARVEL: Waiting on barrier ID 0 (timeout 3000 ms or 30000 ticks): timed out.
[00:00:08.518,000] <err> KARVEL: Waiting on barrier ID 0 (timeout 3000 ms or 30000 ticks): timed out.
[00:00:09.031,000] <err> KARVEL: Waiting on barrier ID 0 (timeout 3000 ms or 30000 ticks): timed out.
[00:00:09.528,000] <err> KARVEL: Waiting on barrier ID 0 (timeout 3000 ms or 30000 ticks): timed out.
[00:00:10.028,000] <err> KARVEL: Waiting on barrier ID 0 (timeout 3000 ms or 30000 ticks): timed out.
[00:00:10.030,000] <inf> KARVEL: TEST 1: Passed.

[00:00:10.531,000] <inf> KARVEL: TEST 2: 5 threads will wait on a 5-thread barrier. Expected fails: 0
[00:00:12.541,000] <inf> KARVEL: Thread 4 released 1 ms after waiting in barrier 0.
[00:00:12.542,000] <inf> KARVEL: Thread 0 released 2010 ms after waiting in barrier 0.
[00:00:12.542,000] <inf> KARVEL: Thread 1 released 1498 ms after waiting in barrier 0.
[00:00:12.542,000] <inf> KARVEL: Thread 2 released 1001 ms after waiting in barrier 0.
[00:00:12.542,000] <inf> KARVEL: Thread 3 released 501 ms after waiting in barrier 0.
[00:00:12.543,000] <inf> KARVEL: TEST 2: Passed.

[00:00:13.058,000] <inf> KARVEL: TEST 3: 5 threads will wait on a 3-thread barrier. Expected fails: 2
[00:00:14.070,000] <inf> KARVEL: Thread 2 released 0 ms after waiting in barrier 0.
[00:00:14.070,000] <inf> KARVEL: Thread 0 released 1012 ms after waiting in barrier 0.
[00:00:14.070,000] <inf> KARVEL: Thread 1 released 499 ms after waiting in barrier 0.
[00:00:17.576,000] <err> KARVEL: Waiting on barrier ID 0 (timeout 3000 ms or 30000 ticks): timed out.
[00:00:18.074,000] <err> KARVEL: Waiting on barrier ID 0 (timeout 3000 ms or 30000 ticks): timed out.
[00:00:18.075,000] <inf> KARVEL: TEST 3: Passed.
```

Figure 6.2: Regular usage test of barriers.

- **regular usage:** This test simply created one barrier with x maximum waiters and y tasks to wait on it, all with a defined timeout. Then, multiple rounds were conducted with different values of x and y to verify changes in the release dynamics.

While the creation and deletion tests are important to check the correct allocation of memory resources for barriers, the most significative test from the execution standpoint is probably the regular usage one, as it allows to observe if the barriers are behaving accordingly. Figure 6.2 shows the outcome of this test in three cases, where the amount of tasks was fixed at 5 and the maximum waiters for the barrier varied between 10, 5 and 3 tasks, respectively. The idea was that in the first test the maximum count would never be reached (leading to timeout for all), whereas in the second the barrier would release on the 5th task entering the barrier and in the third it would release on the 3rd awaiting task, leading the following two tasks to an inevitable timeout.

To ensure a better visualization, the tasks entered the barrier at different moments. This can be verified in Figure 6.2 by inspecting the timestamps, which are $500ms$ apart from each other when the timeouts occur. Conversely, whenever the barrier is released the timestamps coincide (as it should). The first task to be released is always the one that reached the maximum count followed by the order of arrival in the barrier, which explains the sequence 4-0-1-2-3 for the second test and 2-0-1 for the third test. The observed outcomes allows one to conclude the implementation of barriers in KARVEL were successful.

6.2.2 Semaphores

Semaphores are simply integer variables which are atomically iterated on by the tasks that use it. A semaphore typically has a count value which is decremented when a task *takes* it,

and incremented when a task *gives* it. When this count reaches zero, any future task that attempts to take the semaphore will block waiting for other task to give it. When the maximum count of a semaphore is greater than 1, it can be used as a non-blocking alternative to barriers in which tasks signal others they have reached a certain point in execution. When the semaphore maximum count is only 1, however, every attempt of taking it leads to blocking if it has already been taken before. For this reason, semaphores with counts of up to 1 are frequently used to mediate access to critical sections by multiple tasks. They are also known as binary semaphores or simply *mutexes*.

The KARVEL API only makes use of binary semaphores. It does, however, enable the application to choose between a *simple* and a *regular* semaphore at the time of creation. The difference is that regular semaphores offer priority inheritance (if a semaphore is already taken by low-priority task x when high-priority task y attempts to take it, then for as long as x holds the semaphore its priority will be raised to be the same as y) and nested access (the task can recursively take the semaphore, but if so should also recursively give it), and simple semaphores don't. Both types, however, use the task's own priority to organize the wait queue for taken semaphores (as opposed to a more simple FIFO scheme). The main functions concerning the KARVEL semaphore API are:

- `TLCH_SemaphoreCreate`: initializes a static memory region with the control block of a new regular semaphore. Just like barriers, it returns the ID of the initialized object, and an error if the maximum amount of created semaphores has already been achieved. Unlike barriers, there is no need to point configuration options on creation, as the only selection is between simple and regular semaphores - and for that distinction, a `TLCH_SemaphoreCreateSimple` variant also exists.

- `TLCH_SemaphoreDelete`: simply wipes out the data associated with the semaphore. If the semaphore is regular, any tasks waiting for it are unblocked before deletion. If simple, however, the deletion will only succeed if the semaphore is not currently taken. Returns an error code otherwise, or if the ID passed as argument doesn't map to a valid semaphore.

- `TLCH_SemaphoreTake`: attempts to lock a semaphore, waiting indefinitely if it's already taken. Comes alongside two other variants, `TLCH_SemaphoreTakeNoWait` (returns immediately with an error if the semaphore is already taken) and `TLCH_SemaphoreTakeWithTimeout` (which waits up until a time limit, in milliseconds, for taking the semaphore, returning an error otherwise).

- `TLCH_SemaphoreGive`: releases a locked semaphore if the calling task is the current owner of that semaphore. If other tasks are blocked waiting for it, the highest priority one becomes the new owner immediately. Returns an error if the ID passed as argument is not related to any semaphore, if the semaphore is not locked in the first place, or if it's locked by some other task.

- `TLCH_SemaphoreFlush`: releases all tasks blocked in the semaphore wait queue while still keeping the semaphore locked by the current owner. As such, any task can invoke this function (owner or not). Returns an error if the ID passed as argument doesn't point to a valid semaphore.

This choice between regular or simple semaphores is derived from the configurations of the underlying RTEMS primitives, which allows the creation of both types on a flexible, single kernel object. Zephyr, on the other hand, has different data structures for mutexes and counting semaphores, and apply hard-coded distinctions between them: mutexes have priority inheritance, counting semaphores don't, and both mandatorily enable nested access. The porting to Zephyr would therefore require some adaptations, and after caferul consideration we decided to not use Zephyr mutexes at all. Instead we only used semaphores, and applied the logic for priority inheritance and nested access restriction at the abstraction layer. Similarly as done for barriers, a dedicated data structure comprising Zephyr primitives and necessary metadata was created. Listing 6.2 shows the main elements included: the count of recursive takes, the original owner priority, the current owner of the semaphore and the kernel semaphore data itself.

```
1    typedef struct {
2        ...
3        uint32 lock_count;
4        int32 owner_original_priority;
5        struct k_thread *owner;
6        struct k_sem semaphore;
7        ...
8    } TLCH_Mutex_t;
```

Listing 6.2: Core semaphore data structure.

The most significant challenge of implementating the semaphores naturally lied in the routines to take and give them, as there would be the need to check the object type, ownership and iterate over the introduced fields to keep track of the semaphore state. Figure 6.3 illustrates the resulting steps for taking a semaphore. If a task attempts to take one, the KARVEL API will first verify if it's free for the taking or not. If it is, then the task becomes the owner of that semaphore and its original priority is noted down. if it's not, then a second check is performed to see if the task is already the owner (which would figure as a recursive lock) or not.

The next steps differ between simple and regular semaphores: for the simple, a confirmed recursive lock would yield an error, while for the regular it would just iterate on the introduced `lock_count` variable and return successfully. In the alternative case of the semaphore being taken by another task, the regular type would check both tasks' priorities for inheritance applicability, whereas the simple one would just block for the specified timeout.

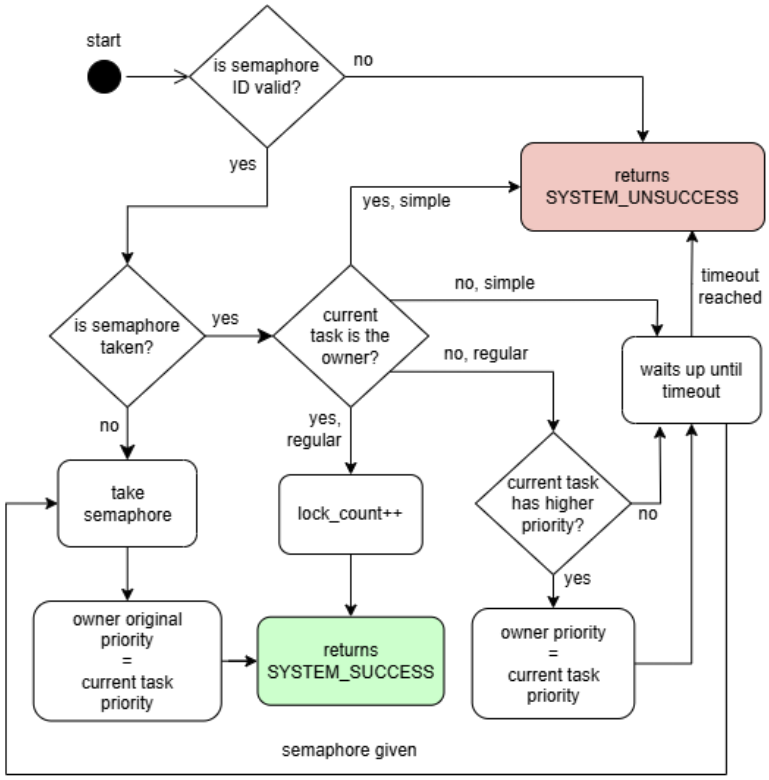


Figure 6.3: TLCH_SemaphoreTakeBase execution flow.

Tests

Just like with barriers, the entire semaphore API was also subject to unit and functional tests to verify if the implementation was working as intended. The most relevant tests were necessarily about taking and giving of semaphores, since they concentrate the runtime behavior and, as explained in the previous paragraphs, were responsible for most of the implementation required to port the KARVEL semaphores to Zephyr. The many nuances that tell apart a simple and a regular semaphore, however, required the development of a more comprehensive set of test cases when compared to barriers.

Figure 6.4 shows the output for three of these test cases, all applied to the `TLCH_SemaphoreTake` function illustrated in Figure 6.3, in which it's possible to verify how it adapts to each semaphore type. In the first case, two tasks attempt to recursively take one semaphore each, however one of those (of $ID = 0$) is regular and the other ($ID = 1$) is simple. As a result, the task taking semaphore 0 succeeds while the other fails. Test cases 2 and 3, by their turn, are conceptually the same: three tasks are launched with a slight offset between them, and a different priority each. The low-priority task 0 goes first, followed by the medium-priority task 1 $500ms$ later and the high priority task 2 $1000ms$ after the first. All three tasks announce their activation, start and finish, busy-waiting for $2000ms$ in the meantime. However, only tasks 0 and 2 make use of a common semaphore.

```
[00:00:05.011,000] <inf> KARVEL: TEST 1: Testing recursive semaphore takes. Expected fails: 5.
[00:00:05.013,000] <inf> KARVEL: Test 1: Task 0 recursively took semaphore 0.
[00:00:05.413,000] <inf> KARVEL: Test 1: Task 0 recursively took semaphore 0.
[00:00:05.526,000] <err> KARVEL: Obtaining semaphore ID 1: Simple semaphore forbids recursive locks.
[00:00:05.526,000] <err> KARVEL: Obtaining semaphore ID 1: Simple semaphore forbids recursive locks.
[00:00:05.526,000] <err> KARVEL: Obtaining semaphore ID 1: Simple semaphore forbids recursive locks.
[00:00:05.526,000] <err> KARVEL: Obtaining semaphore ID 1: Simple semaphore forbids recursive locks.
[00:00:05.526,000] <err> KARVEL: Obtaining semaphore ID 1: Simple semaphore forbids recursive locks.
[00:00:05.813,000] <inf> KARVEL: Test 1: Task 0 recursively took semaphore 0.
[00:00:06.213,000] <inf> KARVEL: Test 1: Task 0 recursively took semaphore 0.
[00:00:06.615,000] <inf> KARVEL: Test 1: Task 0 recursively took semaphore 0.
[00:00:07.015,000] <inf> KARVEL: TEST 1: Passed.

[00:00:07.516,000] <inf> KARVEL: TEST 2: Testing priority inheritance for regular semaphore. Expected fails: 0.
[00:00:07.517,000] <inf> KARVEL: TEST 2: Task 0 activated.
[00:00:07.518,000] <inf> KARVEL: TEST 2: Task 0 starting (took semaphore 0).
[00:00:08.017,000] <inf> KARVEL: TEST 2: Task 1 activated.
[00:00:08.017,000] <inf> KARVEL: TEST 2: Task 1 starting.
[00:00:08.517,000] <inf> KARVEL: TEST 2: Task 2 activated.
[00:00:09.518,000] <inf> KARVEL: TEST 2: Task 0 ending (will give semaphore 0).
[00:00:09.518,000] <inf> KARVEL: TEST 2: Task 2 starting (took semaphore 0).
[00:00:11.518,000] <inf> KARVEL: TEST 2: Task 2 ending (will give semaphore 0).
[00:00:11.518,000] <inf> KARVEL: TEST 2: Task 1 ending.
[00:00:11.518,000] <inf> KARVEL: TEST 2: Passed.

[00:00:12.020,000] <inf> KARVEL: TEST 3: Testing priority inheritance for simple semaphore. Expected fails: 0.
[00:00:12.020,000] <inf> KARVEL: TEST 3: Task 0 activated.
[00:00:12.020,000] <inf> KARVEL: TEST 3: Task 0 starting (took semaphore 1).
[00:00:12.521,000] <inf> KARVEL: TEST 3: Task 1 activated.
[00:00:12.521,000] <inf> KARVEL: TEST 3: Task 1 starting.
[00:00:13.020,000] <inf> KARVEL: TEST 3: Task 2 activated.
[00:00:14.521,000] <inf> KARVEL: TEST 3: Task 1 ending.
[00:00:14.521,000] <inf> KARVEL: TEST 3: Task 0 ending (will give semaphore 1).
[00:00:14.521,000] <inf> KARVEL: TEST 3: Task 2 starting (took semaphore 1).
[00:00:16.521,000] <inf> KARVEL: TEST 3: Passed.
```

Figure 6.4: test case results for semaphore takes.

The fact two tasks dispute the same semaphore and one doesn't is the key for observing different results between Test Case 2 and 3, which only practical distinction is that one uses a regular semaphore (which enables priority inheritance) and the other doesn't. On both cases, task 0 will acquire the semaphore and be preempted by 1, which runs without taking anything and is eventually preempted by task 2. This is the point where the cases differ, as task 2 will try to lock the semaphore still owned by 0. The regular semaphore has inheritance enabled, so task 0 will have its priority elevated to match task 2's, thus becoming greater than task 1's and resuming execution for as long as it holds the semaphore. As soon as it releases, its priority falls back to the original value and task 2 takes over. Thus, for Test Case 2 the observed execution should be $0-0-1-1-2-0-2-2-1$, as Figure 6.4 confirms.

For Test Case 3, the semaphore has no priority inheritance and therefore task 0 would remain with the lowest priority at all times. As a consequence, Task 1 gets to finish before Task 2, which means a longer response time for the highest-priority Task 2 and a guaranteed *priority inversion* phenomena as well, which is usually troublesome for real-time applications. Still, that's the intended behavior for the simple semaphore. The expected outcome for Test Case 3 is therefore $0-0-1-1-2-1-0-2-2$, which Figure 6.4 also attests. These results, alongside the observed outcome for the rest of the developed tests, allows one to confirm that the semaphore API is behaving as intended.

API extensions

The porting of KARVEL to Zephyr naturally led to improvement opportunities, and one of them was to extend the semaphore API with a method to allow any given task to release all semaphores it possesses at once. The motivator behind this contribution was that, when a task voluntarily invokes the `TLCH_EndTask` method, it expects the underlying RTOS to wipe it off the system entirely, leaving behind only a record in the Loader (LDR) data that it has finished execution. The Zephyr documentation, however, makes it clear that its `k_thread_abort` method does little more than removing the task from the scheduling context and any references to its control block. It doesn't free any resources such as dynamically allocated memory or mutexes (Z. Project 2025).

Since it's a valid use case to have one task taking multiple semaphores at once, it would be likely problematic to have an application ending its execution without releasing some or all of the taken objects. A common approach could be to program the application logic to manually release semaphores before terminating, but that requires a discipline from the system developers which could be easily dismissed with an automated alternative. Hence, the introduced solution was to create a `TLCH_SemaphoreGiveAllTakenBy` method, which simply runs through the list of existing semaphores and automatically releases all those held by the task passed as a parameter.

LDR refactoring

On a related note, most of the task deletion logic was migrated to the LDR due to problems encountered with self-deleting tasks on Zephyr: although the documentation mentions that a task can invoke `k_thread_abort` for itself, practical tests ran on a number of emulated and real devices found that this procedure inevitably led to a fatal system error. Moreover, when a task is deleted on Zephyr, its stack might become available for other tasks to use but its data remains untouched. Although not wiping the information is good for performance, the no longer used, yet still existing data could lead to security exploits and execution problems, which are particularly sensitive topics in critical systems.

In this context, the developed solution was to turn the LDR into an active task, with a higher priority than all applications and an operation based on demand. In this new format, the LDR launches all tasks as usual and then becomes inactive, listening for specific application requests to come up. At implementation level, this means the LDR is now the main function, and that infinitely polls on a dedicated message queue after starting the applications. Thus, when a task desires to end, it now sends a termination request for the LDR message queue and suspends. The LDR wakes up with the incoming message, processes the request (in this case, calls the aforementioned `TLCH_SemaphoreGiveAllTakenBy` method, deletes the task and resets the respective stack), and goes inactive once again waiting for the next request.

The initially implemented requests at this point were `LDR_ObjectAction_CLEANUP` for deleting individual tasks or `LDR_ObjectAction_SHUTDOWN` for deleting the whole task set, but this refactoring has been made with extensibility in mind for future types of request.

6.2.3 Timers

Timers are kernel primitives that enable arbitrary actions to be scheduled to happen in the future. In its generic form, a timer is given an application-defined *callback function* and a time interval (usually relative to the moment of activation) to wait before executing the given callback. Timers are usually implemented as time-driven interrupts with the power to preempt whatever task is currently executing in the CPU in order to ensure the correct fulfillment at the end of the configured interval. Timers are leveraged by the OS for implementing sleep functions, control watchdogs and timeouts for blocking operations.

Although similar in concept and usage, the actual implementation of timers can vary significantly between kernels according to their target balance between precision, accuracy and portability. In FreeRTOS, for example, timers are actually handled by a service task instead of interrupts to favor portability, and this task can be given a low priority by the application if desired (which may severely increase the latency between the expected activation and the de facto execution of the callback[1]) (Source 2024b). The task-based approach is also followed by RTEMS, but in this case the task has the absolute highest priority of all the taskset (R. Project 2017). The Linux kernel has two APIs for timers, of which one is suited for general purpose, non critical use cases and the other is meant for real-time applications. Both have a somewhat similar usage, but differ in terms of data structure and interval resolution (Corbet 2023).

In Zephyr, the timers API is slightly more flexible than the aforementioned counterparts. While there exists the ubiquitous callback for when the interval is achieved (known as the *expiry function*), there can also be configured a secondary callback (known as the *stop function*) to be invoked synchronously when a timer is cancelled before expiring. Moreover, while the timeout units are fixed as ticks in FreeRTOS and RTEMS and jiffies/nanoseconds in Linux, Zephyr accepts an opaque data type which abstracts different magnitudes by using macros such as `K_SECONDS(3)` for 3 seconds and `K_USEC(500)` for 500 microseconds. Although these macros internally convert the specified values to ticks, Zephyr has a *tickless* kernel by default and the ticks are only used to regulate the period resolution. The KARVEL API for timers is made of onlt three methods:

- `TLCH_TimerNew`: initializes a timer, starting it immediately upon successful initialization. A timer in KARVEL must be declared a type (if one-shot or periodic), as well as the time interval (in milliseconds) and the callback function to be invoked at the end of this interval. If needed, this API function can be invoked on a pre-initialized timer, in which case it just resets all its attributes.

[1] The time interval is also tied to the system tick in FreeRTOS, of which itself is responsible for pacing the scheduler activation (as this RTOS is *tickful*). Thus, high tick rates (above 1000Hz) are discouraged because they may reduce the system efficiency and increase energy consumption (Source 2024a).

- `TLCH_TimerDelete`: stops the timer, if running, and wipe its associated control block data clean. The control block is a C `struct` like any other, which means that, if not dynamically allocated by the application, will remain with the memory region reserved for the lifetime of the variable scope.
- `TLCH_TimerReset`: restarts the timer if not running, or resets the interval if running. Can be applied to both one-shot and periodic timers. The callback function, as well as the argument passed to it, is kept intact (thus reinforcing the need of the application to first call the aforementioned `TLCH_TimerNew` before this one).

Notably there is no stop function even though both Zephyr and RTEMS (and Linux, and FreeRTOS, of which KARVEL has built-in support) have this option available within their own APIs. There is likely no specific reason for the lack of this functionality except maybe that no application needed it yet, and therefore no implementation was required. As for the works in porting, every KARVEL API could be easily mapped to an existing Zephyr timer API and therefore no major challenges were encountered. One of the few adaptations needed, though, was in the callback function entity, which in Zephyr receives a pointer to the timer as the argument but in KARVEL receives an application-defined argument, as illustrated in Listing 6.3.

```
1  // Zephyr-like timer callback
2  void callback_function ( struct k_timer *timer );
3
4  // KARVEL-like timer callback
5  void callback_function ( void *arg );
6
7  // resulting implementation for KARVEL in zephyr
8  static void internal_callback ( struct k_timer *timer ) {
9      TLCH_Timer_t *handle = CONTAINER_OF(timer, TLCH_Timer_t, timer);
10     handle->callback(handle->arg);
11 }
```

Listing 6.3: Timer Callbacks in Zephyr and KARVEL.

The solution in this case was to actually define the callback internally, within the abstraction layer, and make it a simple wrapper for the application-defined callback and argument. The Zephyr timer structure is stored in the opaque `TLCH_Timer_t` data type, which also stores the provided callback and its argument. Thus, the most elegant solution was to find these elements with the `CONTAINER_OF` macro, which Zephyr inherited from Linux, and execute them afterwards. Listing 6.3 also showcases the resulting arrangement.

Tests

As KARVEL timers, unlike barriers and semaphores, have no static limit of how many can exist at once in the system, and considering that a call to `TLCH_TimerNew` already starts the timer after a successful initialization, there was no need to develop specific tests for the creation path.

Instead, the tests already implied the creation success or not by asserting their regular usage. The most relevant test thus consisted of starting two 300-millisecond timers, one configured as one-shot and the other as periodic, with the same function pointed as the callback. This function merely logged a message and incremented a counter, of which would be checked after a while to verify how many times the callback was executed.

```
int32 test_TLCH_TimerNew(int32 id){
    ...
    int32 callback_count = 0;
    char *phrase = "This is the timer callback";

    // callback function
    void do_something(void *what){
        callback_count++;
        char *message = (char *) what;
        TLCH_Log(LOG_INFO, "TEST %d: %s. Count is: %d",
            id, message, callback_count);
    }

    // initialize the timer
    TLCH_Timer_t handle;
    TLCH_TimerNew(&config, &handle);

    // let it run for a bit, then delete it
    TLCH_DelayTask(1000);
    TLCH_TimerDelete(&handle);

    if(callback_count == expected){
        TLCH_Log(LOG_INFO,
            "TEST %d: callback count is correct.", id
        );
    } else {
        fails++;
        TLCH_Log(LOG_ERROR,
            "TEST %d: Unexpected callback count: %d (expected: %d)",
            id, callback_count, expected
        );
    }
}
```

Listing 6.4: Timers main usability test.

Listing 6.4 illustrates the main steps of the test. First the timer is created with a period of 300 milliseconds, the callback `do_something` and the phrase *"this is the timer callback"* as the argument. The test function then calls `TLCH_TimerNew` to start the timer, waits 1000 milliseconds, deletes the timer and check the value of `callback_count`. This same flow is executed twice, one for each type of timer, so for the one-shot variant it is expected that a 300-millisecond period

```
[00:00:05.013,000] <inf> KARVEL: TEST 1: Testing one-shot Timer with 300ms period. Expected fails: 0
[00:00:05.314,000] <inf> KARVEL: TEST 1: This is the timer callback. Count is: 1
[00:00:06.024,000] <inf> KARVEL: TEST 1: Timer callback count is correct.
[00:00:06.025,000] <inf> KARVEL: TEST 1: Testing periodic Timer with 300ms period. Expected fails: 0
[00:00:06.325,000] <inf> KARVEL: TEST 1: This is the timer callback. Count is: 1
[00:00:06.625,000] <inf> KARVEL: TEST 1: This is the timer callback. Count is: 2
[00:00:06.925,000] <inf> KARVEL: TEST 1: This is the timer callback. Count is: 3
[00:00:07.028,000] <inf> KARVEL: TEST 1: Timer callback count is correct.
[00:00:07.028,000] <inf> KARVEL: TEST 1: Passed.

[00:00:07.831,000] <inf> KARVEL: TEST 2: Attempting to delete an existing timer. Expected fails: 0
[00:00:07.832,000] <inf> KARVEL: TEST 2: Timer deleted successfuly.
[00:00:07.832,000] <inf> KARVEL: TEST 2: Attempting to delete an already deleted timer. Expected fails: 1
[00:00:07.832,000] <err> KARVEL: Deleting Timer (handle 0x8000c630): timer not found.
[00:00:07.832,000] <inf> KARVEL: TEST 2: Attempting to delete a NULL timer. Expected fails: 1
[00:00:07.832,000] <err> KARVEL: Deleting Timer (handle 0): timer not found.
[00:00:07.832,000] <inf> KARVEL: TEST 2: Passed.

[00:00:08.333,000] <inf> KARVEL: TEST 3: Launching one-shot Timer with 300ms period.
[00:00:08.633,000] <inf> KARVEL: TEST 3: This is the timer callback. Count is: 1
[00:00:09.335,000] <inf> KARVEL: TEST 3: Done. Relaunching timer through reset. Expected fails: 0
[00:00:09.635,000] <inf> KARVEL: TEST 3: This is the timer callback. Count is: 1
[00:00:10.337,000] <inf> KARVEL: TEST 3: Done. Reconfiguring timer for a 2-second period.
[00:00:10.338,000] <inf> KARVEL: TEST 3: Done. Resetting timer multiple times. Expected fails: 0
[00:00:11.342,000] <inf> KARVEL: TEST 3: Reset.
[00:00:12.345,000] <inf> KARVEL: TEST 3: Reset.
[00:00:13.347,000] <inf> KARVEL: TEST 3: Reset.
[00:00:15.346,000] <inf> KARVEL: TEST 3: This is the timer callback. Count is: 1
[00:00:16.349,000] <inf> KARVEL: TEST 3: Done. Timer callback count is correct.
[00:00:16.349,000] <inf> KARVEL: TEST 3: Passed.
```

Figure 6.5: test case results for the timers API.

will yield precisely one callback execution during the 1000 milliseconds waited. Conversely, for the periodic timer, the same conditions should yield three consecutive calls, 300 milliseconds apart from each other. The results are shown in Figure 6.5, as TEST 1.

Additionally, two other test cases were developed to specifically check the other APIs. The second test simply attempted to invoke `TLCH_TimerDelete` with 1) an initialized timer, 2) a non-initialized timer and 3) a NULL pointer passed as argument, in order to see how it adapts. Fortunately, while the first attempt succeeds at deleting, the following erroneous attempts are correctly handled by the API, as TEST 2 of Figure 6.5 illustrates.

For the third and last test, the idea was to assess the behavior of the `TLCH_TimerReset` function under different circumstances. First, a one-shot timer is launched with a 300-millisecond period only to be reset a second later. This action should ignite a new run for this timer, so the consequence would be a second callback execution 1300 milliseconds after the first launch. Then, for the last part of the test, the same timer is reconfigured to have a 2000-millisecond period. During this interval, the test function resets this timer 3 timers, with one-second between each reset. The consequence then is that the callback function should be delayed in 3 seconds, thus only being invoked a whole 5-second interval after the initial launch. As the TEST 3 of Figure 6.5 showcases by inspecting the timestamps, the described outcome is correctly observed.

6.2.4 Message Queues

Message queues are a powerful and asynchronous way of exchanging information between executive units such as threads or ISRs. Messages can be usually configured to be of any format and size, and while some operating systems require that a format is kept the same for all messages sent to a specific queue, others allow different types of message to exist at once in a queue, only requiring a *maximum* message size to be specified instead (this is the case for RTEMS and Apache NuttX). All messages in a queue are stored in a memory buffer, and conventionally retrieved in a FIFO manner. When a message is *sent* to a queue, it is copied *to* the buffer, and when it is *received*, it is copied *from* the buffer. This feature makes it adequate for exchanging messages from ISRs, of which memory stack is usually short-lived, but may imply some performance overhead.

The message queue API in KARVEL is closely related with the features available in RTEMS, allowing applications to create, delete and send messages in both FIFO and LIFO ordering. Tasks intending to receive messages from a queue can poll for new payloads if the queue is empty, but interestingly enough tasks that wish to send messages are not allowed to wait for a spot to open up if the queue is already full (a restriction shared by RTEMS). However, perhaps due to the lack of necessity, or the requirement for compatibility with other RTOS, not all APIs available in RTEMS are abstracted in KARVEL (for example, there is no option for broadcasting a received message to every pending task). The available functions are:

- `TLCH_MsgQueueNew`: initializes the control block of a message queue. The application is required to provide the maximum message size and message count, which allows to infer the total buffer size, and optionally might as well provide the buffer location. If a buffer is not provided, this function dynamically allocates the required memory for it (which may not be recommended for critical applications). Another optional parameter is a `buffer_free` callback, which is invoked later when the queue is deleted.
- `TLCH_MsgQueueDelete`: wipes out the control block data allocated for the message queue. If the buffer was dynamically allocated, frees the related memory region. If a `buffer_free` callback was declared in initialization, executes it. A message queue can only be deleted if there are no pending tasks on it, so a call to this API will return an error otherwise.
- `TLCH_MsgQueueSend`: Simply sends a message to the end of the specified message queue, thus configuring a FIFO delivery. The application is required to point the desired queue as well as the message and its size. Returns an error if the queue is invalid or already full or messages.
- `TLCH_MsgQueueSendUrgent`: does the same as `TLCH_MsgQueueSend`, but inserting messages at the beginning (head) of the queue instead of at the end. Thus, this method configures a LIFO delivery scheme. Also returns an error if there is currently no space left in the queue to insert the message.

- `TLCH_MsgQueueReceive`: retrieves the message in the head of the queue, copying it to a location pointed by the calling task. If the queue is currently empty, the task may specify a timeout to wait for a new payload before giving up. Pending tasks will be unlocked upon the arrival of new messages by the order of their priority (either static or EDF).
- `TLCH_MsgQueuePending`: gets the number of messages awaiting retrieval in a queue.
- `TLCH_MsgQueueClear`: deletes all existing messages in a queue.

As for Zephyr, the built-in support for message queues is a bit different than the one in RTEMS, so some extra work was required in the abstraction layer in order to make it compatible with the KARVEL standards. For example, whereas messages might only be bounded to a maximum specified size in RTEMS, in Zephyr they are always required to have the same exact length. If it's true that this might theoretically narrow the application use case of message queues, it's also true that in the majority of times these queues are created with a specific data format in mind (for example, a struct containing information of a sensor or the motion planning for an actuator), which itself has a fixed length.

The fact that a variable size increases the message handling complexity and requires extra per-message runtime data doesn't seem to match the benefit of flexibility, specially considering that, as the buffer is invariably pre-allocated with the worst-case assumption of every message having the maximum allowed size, the variability doesn't bring any advantage in terms of memory usage either. Having these considerations in mind, it was decided that the effort of altering Zephyr to support variable-sized messages was a non-priority; hence, in this port of KARVEL the messages would need to keep a fixed length. In the occasion of a call to `TLCH_MsgQueueSend` specified a different message size than configured, the abstraction layer would simply return an error code.

Another major difference between the approaches of Zephyr and RTEMS regarding message queues is that RTEMS supports the LIFO delivery scheme for urgent messages, and Zephyr doesn't. In the latter there is only the `k_msgq_put` method for inserting messages at the end of the queue, even though the underneath implementation follows a ring buffer design (where the head and tail pointers of the buffer are always moving around on read and write operations) that could extend the LIFO option with relative ease. Being Zephyr an open-source project, this option has been previously submitted as a feature request and later as an implementation, but the analysis phase stalled and never made it to the kernel (Wu 2024).

Still, as the "urgent" sending mode is a core part of the KARVEL queue API, it was deemed necessary to extend Zephyr to support it. There were two possible strategies to achieve this: either within the abstraction layer, in a similar procedure to what's been done with the semaphores API, or directly through modifications in the kernel code, which could potentially yield a valid contribution to the open-source community. Although different in execution

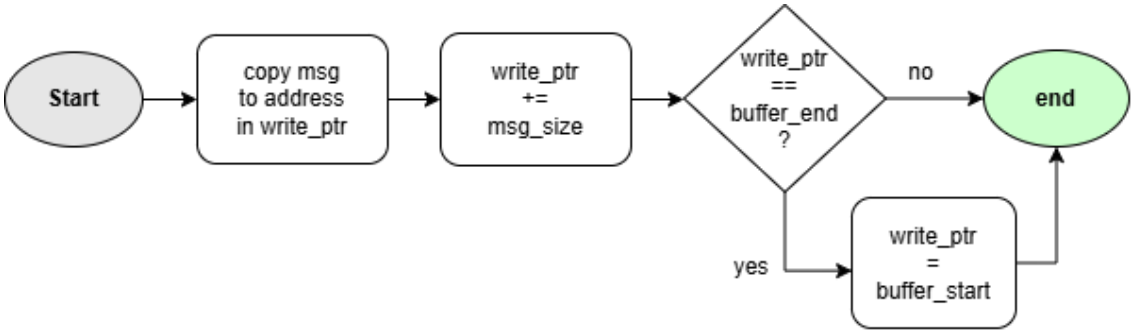


Figure 6.6: typical FIFO write operation of a message queue.

and requirements, the logic steps for performing the operation would remain remarkably unchanged: first manipulate the pointers of the queue, then perform the write operation, then restore these pointers to their original values, if needed.

A ring buffer distinguishes itself by having the aforementioned head and tail pointers to denote where the queue starts and finishes, and they "move" around the buffer memory region as read and write operations take place. It is conventional for a FIFO scheme that the start (head) of the queue points to the first message to be retrieved in a read operation, while the end (tail) of the queue marks where the incoming message will be stored. Since in a message queue every content is passed by copy, a write operation consists of writing to the memory address stored in the tail pointer and incrementing it by the size of the newcoming message. Conversely, a read operation copies the message indicated by the head pointer and also increments it accordingly.

In Zephyr, these pointers are named write and read pointers instead to better reflect their usage. A typical write operation (where there is room for newcoming messages in the queue) is represented in Figure 6.6, and in fact that is the core procedure of Zephyr's existing `k_msgq_put` API. So, in order to enable the LIFO scheme for the write operation with minimal changes to the existing algorithm, a sufficient modification before writing would be to decrement the read pointer by `msg_size` to open room in the head of the queue for the incoming message, then change the write pointer to match the read pointer; this maneuver would effectively mean writing the message in the beginning of the queue, as the LIFO scheme requires. After the writing has completed, an extra step would be to revert the write pointer to its original value, as `k_msgq_put` expects it to always point to the end of the queue; the read pointer already reflects the new head of the queue, though, and requires no further change.

```
1 int32 TLCH_MsgQueueSendUrgent(TLCH_MsgQueue_t* handle, void* msg, size_t
      msg_size) {
2     ...
3     k_msgq *queue = handle->msgq;
4
5     //pointer manipulation
6     char *original_write_ptr = queue->write_ptr;
```

```
7       queue->read_ptr -= queue->msg_size;
8       queue->write_ptr = queue->read_ptr;
9
10      //actual writing
11      k_msgq_put(&(handle->msgq), msg, K_NO_WAIT);
12
13      //pointer restoring
14      queue->write_ptr = original_write_ptr;
15      ...
16  }
```

Listing 6.5: Minimal code to send messages in the LIFO scheme.

If applied at the KARVEL abstraction layer, the aforementioned algorithm would yield a similar result as shown in Listing 6.5. In fact, the unit tests described in the next Section were conducted for this implementation and the LIFO scheme was observed successfully (and without jeopardizing the pre-existing FIFO alternative). As one of the two options studied for enabling the "urgent" sending of messages, restricting the implementation to the abstraction layer has the advantage of speed and simplicity, as there is no requirement for writing additional documentation, complying with contribution standards or awaiting for feedback from reviewers.

Only working with existing kernel primitives is also interesting from the dependability standpoint, as those have been well tested before by the community, however for this specific case this alternative has two major issues that discourage it. First, one of the actions performed by `k_msgq_put` aside from the flow described in Figure 6.6 is to make use of a queue-specific mutex to protect the operation from race conditions. This means that the internal pointer manipulation is done in a thread-safe manner, whereas the approach of Listing 6.5 has no such guarantee. It would be possible to add an extra mutex lock/unlock pair in the abstraction layer to resolve it, but as a consequence the original lock/unlock operations would become redundant.

Secondly, there are three steps in Figure 6.6 which specifically manipulate the write pointer value only for it to be reverted to its original state later, at the end of Listing 6.5. If the value is invariably restored in the introduced logic, these steps have no reason to execute, and therefore keeping them is inefficient. In the light of such observations, following the alternative of introducing a new Zephyr API for the LIFO scheme of messages seems considerably better, and as such it was chosen as the implementation route for this Thesis. Moreover, it's worth remarking that although there is no guarantee of acceptance by the maintainers of Zephyr, this decision also meannt a possible impact in the open-source RTOS as a contribution of this Thesis.

The resulting implementation as a dedicated API in Zephyr has its main steps represented in the diagram of Figure 6.7 and, as it can be observed, there is no longer any manipulation necessary for the write pointer. Instead, the implementation leverages the fact that the read

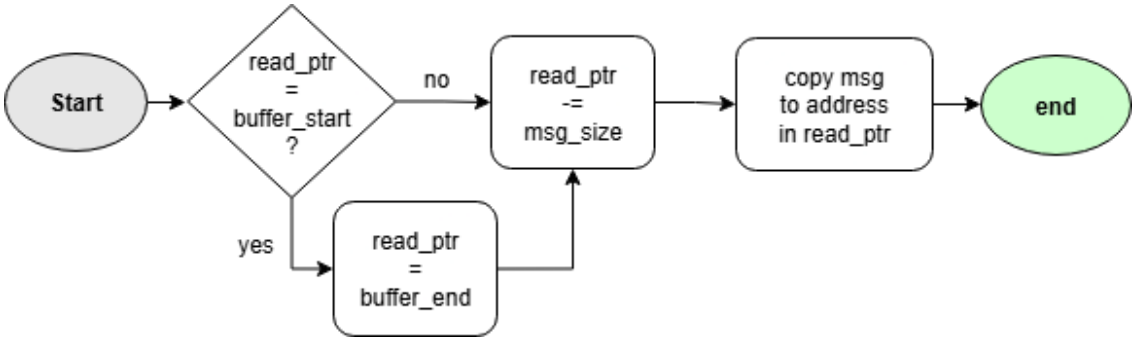


Figure 6.7: proposed Zephyr API algorithm for LIFO write operation.

pointer already stores the desired location for the incoming message right after being subtracted by `msg_size`, and thus places the message there. From a consistency standpoint, this solution is not optimal; the write pointer is supposed to be used for write operations, not the read pointer. However, with this strategy the LIFO scheme was able to have its complexity reduced to match the existing FIFO sending of messages, which is optimal in terms of performance. As the procedure is not too complex, it was deemed OK to go with it.

The contribution led to the creation of a new `k_msgq_put_front` method, and was submitted as a Pull Request on GitHub which has been approved and merged (Paschoaletto 2025c). As for the rest of the KARVEL message queue API, their functionality was closely related with the pre-existing Zephyr API and, apart from the necessary defensive code to ensure the function parameters were valid (for example, if the provided message queue pointer is not null), the bulk of the work consisted of directly mapping the abstraction layer to the underlying RTOS API. For example, the `TLCH_MsgQueueSend` method mapped to `k_msgq_put` function, but since there is no timeout option for the first, the latter received the constant `K_NO_WAIT` to indicate an error should be returned if the message couldn't be sent immediately. `TLCH_MsgQueueNew`, by its turn, can receive a statically declared buffer for the queue as a parameter, and if not supplied by the application, will dynamically allocate one using the maximum number of messages and the message length to compute the total size. Zephyr supports these options as well, allowing each thread to have a dynamic memory region known as a "resource pool" and offering the `k_msgq_alloc_init` API to allocate the buffer as needed.

Tests

Six test cases were developed in order to check if the KARVEL message queue API was behaving correctly, and the results can be verified in Figure 6.8. The first three refer to test cases of the message queue creation: in the first case, `TLCH_MsgQueueNew` is invoked with various invalid parameters such as empty configuration, or no handler appointed, or a message size equal to zero. The second creation test supplies valid parameters, including the address of a statically allocated buffer to receive the messages, and performs a quick usage test by sending and retrieving a few messages from the newly created queue. The third and final creation test

```
[00:00:05.012,000] <inf> KARVEL: TEST 1: Attempting to create message queues with invalid parameters. Expected fails: 5.
[00:00:05.012,000] <err> KARVEL: Creating message queue: Invalid configuration parameters.
[00:00:05.012,000] <err> KARVEL: Creating message queue: Invalid configuration parameters.
[00:00:05.012,000] <err> KARVEL: Creating message queue: Invalid configuration parameters.
[00:00:05.012,000] <err> KARVEL: Creating message queue: Invalid configuration parameters.
[00:00:05.012,000] <err> KARVEL: Creating message queue: Invalid configuration parameters.
[00:00:05.013,000] <inf> KARVEL: TEST 1: Passed.

[00:00:05.514,000] <inf> KARVEL: TEST 2: Attempting to create a message queue with provided buffer. Expected fails: 0.
[00:00:05.515,000] <inf> KARVEL: TEST 2: Done. Quickly testing message queue usage. Expected fails: 0.
[00:00:05.821,000] <inf> KARVEL: TEST 2: Passed.

[00:00:06.322,000] <inf> KARVEL: TEST 3: Attempting to create a message queue within thread resource pool. Expected fails: 0.
[00:00:06.325,000] <inf> KARVEL: TEST 3: Done. Quickly testing message queue usage. Expected fails: 0.
[00:00:07.727,000] <inf> KARVEL: TEST 3: Passed.

[00:00:08.229,000] <inf> KARVEL: TEST 4: Testing the sending of messages 1-7. Expected fails: 0.
[00:00:08.939,000] <inf> KARVEL: TEST 4: Quick test succeeded. received sequence: '1234567'.
[00:00:08.939,000] <inf> KARVEL: TEST 4: Passed.

[00:00:09.441,000] <inf> KARVEL: TEST 5: Testing the sending of messages 1-7. urgent: '4' and '7'. Expected fails: 0.
[00:00:09.442,000] <inf> KARVEL: TEST 5: Quick test succeeded. received sequence: '7412356'.
[00:00:09.442,000] <inf> KARVEL: TEST 5: Passed.

[00:00:09.943,000] <inf> KARVEL: TEST 6: Attempting to delete a message queue with awaiting tasks. Expected fails: 1.
[00:00:10.944,000] <err> KARVEL: Deleting message queue: at least one task pending in the queue.
[00:00:10.945,000] <inf> KARVEL: TEST 6: Attempting to delete a message queue with no waiters. Expected fails: 0.
[00:00:10.945,000] <inf> KARVEL: TEST 6: buffer freeing callback executed (as it should).
[00:00:10.946,000] <inf> KARVEL: TEST 6: Attempting to delete a message queue that doesn't exist. Expected fails: 1.
[00:00:10.946,000] <err> KARVEL: Deleting message queue: queue not found.
[00:00:10.946,000] <inf> KARVEL: TEST 6: Passed.
```

Figure 6.8: test case results for the message queue API.

is equal to the second, with the exception that the buffer is not provided and thus must be implicitly allocated by the API. As Figure 6.8 illustrates, all these tests execute successfully.

The following tests, 4 and 5, simply verify the correctness of methods to send messages to a queue, `TLCH_MsgQueueSend` and `TLCH_MsgQueueSendUrgent`. Both send the same sequence of characters from '1' to '7', but test 5 invokes the "urgent" variant specifically for characters '1', '4' and '7'. The character '1' should have no meaningful difference as it's the first one to be sent, but characters '4' and '7' should appear before the rest. So while the expected sequence is "1234567" for test 4, test 5 expects to receive "7412356" - and, as Figure 6.8 showcases, this is indeed the observed result for each case.

The final test simply attempts to delete existing message queues under three circumstances. In the first there is at least one task waiting for messages, in which case deletion is not allowed. Then, a queue is deleted with no waiters, which is accepted. Finally, the last attempt tries to delete an non-initialized message queue, which is also forbidden. The deletion callback is only supposed to execute upon a successful case, and indeed that's the outcome Figure 6.8 illustrates. With these results, the implemented KARVEL message queue API was considered adequate.

6.2.5 Rate Monotonic

KARVEL closely follows the RTEMS convention on its support for Rate Monotonic (RM) (explained in Section 2.8.3). In RTEMS, the task first initializes the data structure known as a *RM manager* during setup and then falls into a loop, where it *announces* the next period at every

new iteration. Listing 6.6 exemplifies, for RTEMS, the minimum required steps for an RM task with a period of 100 ticks.

```
void periodic_task(void *arg){
    rtems_name        handler_name;
    rtems_id          handler;
    ...
    handler_name = rtems_build_name('T', 'E', 'S', 'T');
    rtems_rate_monotonic_create(handler_name, &handler);

    while(true){
        rtems_rate_monotonic_period(handler, 100);
        do_periodic_work();
    }

    // no more periodic work to do
    rtems_rate_monotonic_delete(handler);
}
```

Listing 6.6: Minimal code for a rate monotonic task in RTEMS.

The initialization method does little more than allocating and initializing the necessary memory for the manager, and the deletion method basically wipes it clean. All memory for allocating managers is statically defined through a `CONFIGURE_MAXIMUM_PERIODS` macro in RTEMS configuration. The `rtems_rate_monotonic_period` method, by its turn, is what truly enables RM scheduling in RTEMS: it both defines the task period and regulates the execution pace to match what's been configured. Figure 6.9 illustrates the steps it executes: in the first loop cycle, this function will just set the current period value and calculate the absolute point in time which would correspond to the end of the cycle; then, the loop may proceed as normal. The function is invoked again at the beginning of the next cycle, but since this time there was a previously configured period on the run, the function will first check if it exceeded that period or not.

If not, it means the expected case where the previous loop finished on time, and therefore all that's left to do is wait until the end of the current period before proceeding to the next. Conversely, if the second iteration started *after* the first period was due, then the prior execution took longer than expected or was preempted for too long. Either way the period was exceeded, which configures a deadline miss to the eyes of RM. There is no formal specification for what to do when a deadline is missed for most scheduling algorithms; in RTEMS, the period simply resets and the function returns an error code without waiting, allowing the task to decide what to do with it.

The KARVEL port for RTEMS leveraged this RTOS' existing API with very little additions (mostly logs, actually) for its own rate monotonic API, however the same can't be said for Zephyr because it has no built-in support for RM or any standard way of running periodic tasks. Timely activations can be achieved with a task suspending itself upon completing a cycle and a timer

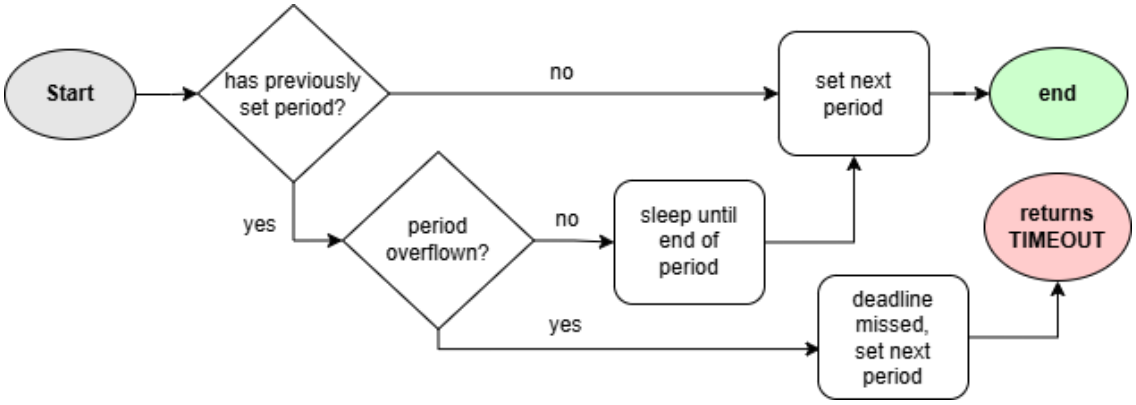


Figure 6.9: rtems_rate_monotonic_period algorithm.

to wake it up by the next cycle start, for example, but this is a generic (although perfectly acceptable) solution that requires multiple unrelated kernel entities and an application-level coordination to work together.

Surely enough, this coordination could be transposed to the abstraction layer with relative ease. However relying on timers to perform the periodic activations is not really compatible with the flexibility of the original RTEMS API that KARVEL extends. Under these circumstances, using a timer would likely cause a race condition if the task misses a deadline, as the task would suspend *after* the timer had tried to wake it up for the new period; as a consequence, for that next period the task would remain inactive, "skipping" a whole cycle. Thus, it was deemed better to simply invoke a regular sleep routine for the suspension interval instead.

```
int32 TLCH_RateMonotonicPeriod(PeriodID, periodMilliseconds){
    ...
    if(rm_handler->activation_instant == 0){
        // this is the first cycle
        rm_handler->activation_instant = now;
    } else {
        int64 next_activation =
            rm_handler->activation_instant + rm_handler->period;

        // task was already in a cycle, check deadline
        if(now > next_activation) {
            // deadline miss
            rm_handler->activation_instant = now;
            status = SYSTEM_UNSUCCESS;
        } else {
            // execution has gone within period (as expected),
            // so we can just sleep for for the remainder of
            // the current period before starting the next
            k_sleep(K_MSEC(next_activation - now));

            // we have just slept until nextActivation, so
            // this becomes our new activationInstant
```

```
23              rm_handler->activation_instant = next_activation;
24              status = SYSTEM_SUCCESS;
25          }
26      }
27
28      // there only needs to be a rescheduling if
29      // the new period is different than the last
30      reschedule_if_needed(task, rm_handler, periodMilliseconds);
31
32      ...
33      return status;
34  }
```

Listing 6.7: `TLCH_RateMonotonicPeriod` main implementation for Zephyr.

Listing 6.7 illustrates the main parts of the implementation of the `TLCH_RateMonotonicPeriod` method, and in practice it performs the same steps as the algorithm depicted in Figure 6.9. The major concern in using sleep functions instead of timers, which are interrupt-driven and recommended for more precise timekeeping, is that the activations may drift over time. The solution to mitigate this effect was to track the theoretical activation instants, and use them instead of the actual moment of which the function was invoked. As Listing 6.7 shows, the only moment where the current timestamp is used to mark an activation is in the first cycle, as there is no previous reference to compare to; the following cycles simply update the activation instant based on the previous value and the theoretical period itself.

Curiously enough, the best conceived solution for applying RM priorities at a kernel level in Zephyr was to adapt the existing EDF infrastructure. Zephyr has exactly one API which is specific to EDF: `k_thread_deadline_set`, available when the Kconfig macro `CONFIG_SCHED_DEADLINE` is enabled. What this API does is receive a relative deadline, in hardware cycles, and internally calculate the absolute deadline from it. The calculated value is then stored in a dedicated field in the thread control block called `prio_deadline`, and consulted by the kernel whenever the task is placed in the ready queue. There is no admission control to check whether the declared deadline is in the past or not, nor does the kernel take any action when a deadline is missed. Thus, to enable RM in this architecture, one could simply have a way of declaring *absolute* deadlines directly, bypassing the internal calculations, and providing the task period as the deadline value. Since the smallest value yields the highest priority, This would effectively enable RM (and in fact any other deadline-oriented scheduling algorithm) with minimal changes required in the kernel. Moreover, it would stand in a clear position on the priority hierarchy: first FP, then RM, then EDF, as RM periods would likely be far smaller than EDF absolute timestamps.

Given this solution was better implemented at kernel level instead of abstraction level, and considering this contribution would bring the benefit of facilitating the introduction of new

schedulers to Zephyr, a decision was made to implement a new kernel API for absolute deadlines. As a result, a new `k_thread_deadline_set_absolute` method was proposed, but it unfortunately received little attention from Zephyr maintainers. The only reviewer deemed the contribution dangerous, alleging the EDF in Zephyr is meant to accept the thread CPU times as their relative deadlines, and therefore declaring absolute values made no sense (Paschoaletto 2025b).

This conclusion made by the sole maintainer that reviewed the Pull Request gives one the impression that the deadline API was actually tailored for the Shortest Job First (SJF) algorithm, in which the priority is highest for the task with the lowest required CPU time, rather than EDF, however in practice the design choice for what the relative deadline *represents* seems irrelevant at the implementation level. In fact, the EDF algorithm, implemented as the theory defines it, has been successfully achieved with the existing API (See Section 6.3.1), and the introduction of an absolute deadline setter did in fact enable RM scheduling, as the tests described in the following Section illustrate.

```
1    static inline void reschedule_if_needed(task, rm_handler, newPeriod){
2        if(rm_handler->period == newPeriod) return;
3
4        rm_handler->period = newPeriod;
5        K_SPINLOCK(&_sched_spinlock) {
6            k_thread_deadline_set_absolute(task, K_MSEC(newPeriod));
7        }
8        k_reschedule();
9    }
10
```

Listing 6.8: `reschedule_if_needed` main implementation.

So although the contribution was not accepted in the mainline kernel, the rationale explained for the refusal seems to be based on limited (or plain wrong) assumptions of the system design. Thus, a decision was made to keep this contribution for the KARVEL fork of Zephyr and use it for the RM (or any other new scheduling algorithms) implementation. Listing 6.8 shows the rescheduling logic invoked at the end of `TLCH_RateMonotonicPeriod`. When the new period is different from the last, the newly created `k_thread_deadline_set_absolute` API is invoked under a protective spinlock and followed by `k_reschedule`, which applies the priority changes to the scheduler immediately. This last step was found necessary because the deadline setters update the priorities in the task control block, but don't wake up the scheduler right away for efficiency reasons (Paschoaletto 2024 and Mitsis 2024).

```
[00:00:05.010,000] <inf> KARVEL: TEST 1: Attempting to create 26 Rate Monotonic handlers (allowed: 16).
[00:00:05.010,000] <inf> KARVEL: TEST 1: Handler 0 created.
[00:00:05.010,000] <inf> KARVEL: TEST 1: Handler 1 created.
[00:00:05.010,000] <inf> KARVEL: TEST 1: Handler 2 created.
[00:00:05.010,000] <inf> KARVEL: TEST 1: Handler 3 created.
[00:00:05.010,000] <inf> KARVEL: TEST 1: Handler 4 created.
[00:00:05.010,000] <inf> KARVEL: TEST 1: Handler 5 created.
[00:00:05.010,000] <inf> KARVEL: TEST 1: Handler 6 created.
[00:00:05.010,000] <inf> KARVEL: TEST 1: Handler 7 created.
[00:00:05.011,000] <inf> KARVEL: TEST 1: Handler 8 created.
[00:00:05.011,000] <inf> KARVEL: TEST 1: Handler 9 created.
[00:00:05.011,000] <inf> KARVEL: TEST 1: Handler 10 created.
[00:00:05.011,000] <inf> KARVEL: TEST 1: Handler 11 created.
[00:00:05.011,000] <inf> KARVEL: TEST 1: Handler 12 created.
[00:00:05.011,000] <inf> KARVEL: TEST 1: Handler 13 created.
[00:00:05.011,000] <inf> KARVEL: TEST 1: Handler 14 created.
[00:00:05.011,000] <inf> KARVEL: TEST 1: Handler 15 created.
[00:00:05.011,000] <err> KARVEL: Creating Rate Monotonic handler '16': No space left.
[00:00:05.011,000] <err> KARVEL: Creating Rate Monotonic handler '17': No space left.
[00:00:05.011,000] <err> KARVEL: Creating Rate Monotonic handler '18': No space left.
[00:00:05.011,000] <err> KARVEL: Creating Rate Monotonic handler '19': No space left.
[00:00:05.011,000] <err> KARVEL: Creating Rate Monotonic handler '20': No space left.
[00:00:05.011,000] <err> KARVEL: Creating Rate Monotonic handler '21': No space left.
[00:00:05.011,000] <err> KARVEL: Creating Rate Monotonic handler '22': No space left.
[00:00:05.011,000] <err> KARVEL: Creating Rate Monotonic handler '23': No space left.
[00:00:05.011,000] <err> KARVEL: Creating Rate Monotonic handler '24': No space left.
[00:00:05.011,000] <err> KARVEL: Creating Rate Monotonic handler '25': No space left.
[00:00:05.011,000] <inf> KARVEL: TEST 1: Passed.

[00:00:05.512,000] <inf> KARVEL: TEST 2: Created 5 RM handlers. Attemping to delete 10. Expected fails: 5
[00:00:05.512,000] <inf> KARVEL: TEST 2: Handler 0 deleted.
[00:00:05.512,000] <inf> KARVEL: TEST 2: Handler 1 deleted.
[00:00:05.512,000] <inf> KARVEL: TEST 2: Handler 2 deleted.
[00:00:05.512,000] <inf> KARVEL: TEST 2: Handler 3 deleted.
[00:00:05.512,000] <inf> KARVEL: TEST 2: Handler 4 deleted.
[00:00:05.512,000] <err> KARVEL: Deleting Rate Monotonic handler with ID 5: Handler not found.
[00:00:05.512,000] <err> KARVEL: Deleting Rate Monotonic handler with ID 6: Handler not found.
[00:00:05.512,000] <err> KARVEL: Deleting Rate Monotonic handler with ID 7: Handler not found.
[00:00:05.512,000] <err> KARVEL: Deleting Rate Monotonic handler with ID 8: Handler not found.
[00:00:05.512,000] <err> KARVEL: Deleting Rate Monotonic handler with ID 9: Handler not found.
[00:00:05.512,000] <inf> KARVEL: TEST 2: Passed.
```

Figure 6.10: test case results for creating and deleting RM managers.

Tests

Following the examples of other features in the KARVEL API, the tests developed for the RM functionality covered the creation and deletion[2] of RM managers as well as their usage on a regular application. In terms of memory, the implementation on KARVEL followed the same path of barriers and semaphores of having a static array to allocate managers and a Kconfig macro to set the maximum allowed number that can exist at once in the system. Aside from that, the creation routine takes no configuration arguments that could be incorrectly provided, nor there is any evident problem in deleting a manager at any point of a task execution.

Thus, a sufficient test case for both creation and deletion of RM managers was to attempt creating more than allowed, and then attempt deleting more than existent. Figure 6.10 shows the execution results for these attempts. For the first test, the maximum amount of managers

[2] The deletion of RM managers was not originally part of KARVEL, probably because there was no need for it in any KARVEL application developed so far. To keep the consistency with the other APIs of having both a creation and deletion methods, this API was included as part of this Thesis.

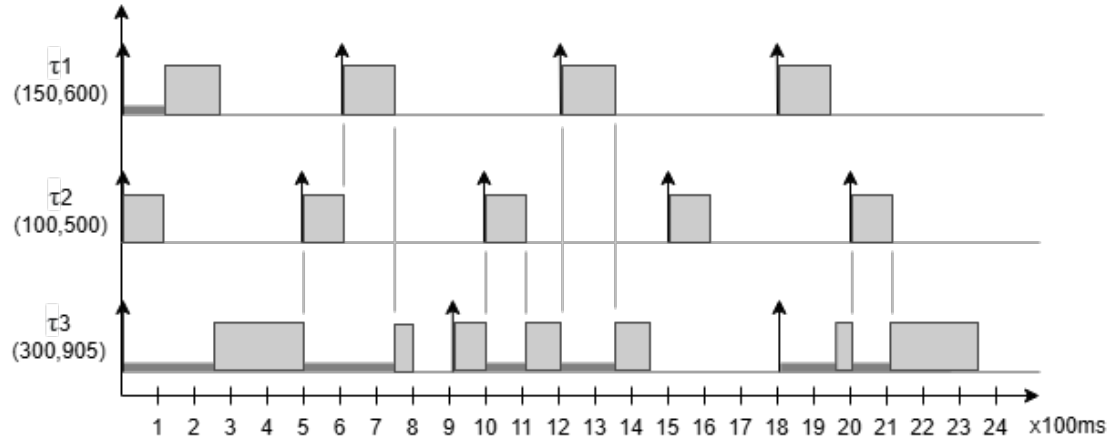


Figure 6.11: theoretical scheduling result for the chosen task set.

is configured as 16, and it's possible to see the API denying the creation of managers once this limit is reached. Then, the deletion test proceeds with the creation of 5 managers only to try deleting 10 afterwards, and once again the API stops the deletion once there is no longer a manager left. Thus, these tests were successful.

As for verifying the `TLCH_RateMonotonicPeriod` method, a classic schedulability analysis was conducted: a given task set with constant periods and CPU times was defined and launched on the critical instant, and the resulting execution was compared with the theoretical outcome expected for RM. Each task was designed to cast logs at their execution start and end, and alongside the log timestamps included by default, allowed one to verify the system behavior and compare it with the theory. Figure 6.11 illustrates expected execution for the chosen task set, which has a combined CPU usage of 78,15% and is schedulable according to the Hyperbolic Bound (HB) analysis (Equation 2.2).

The tasks had their computation times implemented as calls to Zephyr's `k_busy_wait` method, which allows the task to occupy the CPU for approximately the time passed to it, in microseconds. The practical results obtained with the tests are available in Figure 6.12, where the numbers displayed within parenthesis show the time since all tasks were launched (whereas the ones within brackets show the absolute time since boot). As expected, the times corresponding to the start and ending of each task cycle closely match the theoretical outcome of Figure 6.11. Notably, at the critical instant task τ_2 and its smallest period of all goes first, followed by τ_1 and τ_3. τ_3 starts approximately $250ms$ after the beginning, but is preempted by the second cycle of the remaining tasks before being able to finish. Overall it takes nearly $550ms$ between the start and end logs, exactly like the theoretical outcome.

The last developed test aimed at verifying the period overrun detection. A single task was created running the same procedure as those used for the schedulability test, but with a configured period of $1000ms$ and an execution time of $1500ms$, which is clearly unfeasible. Every

```
[00:00:06.031,000] <inf> KARVEL: TEST 3: Testing normal execution of tasks under Rate Monotonic (RM) scheduling. Expected fails: 0.
[00:00:06.037,000] <inf> KARVEL: [ 2 ] start    (at 6ms)
[00:00:06.138,000] <inf> KARVEL: [ 2 ] finish   (at 107ms)
[00:00:06.138,000] <inf> KARVEL: [ 1 ] start    (at 107ms)
[00:00:06.289,000] <inf> KARVEL: [ 1 ] finish   (at 258ms)
[00:00:06.289,000] <inf> KARVEL: [ 3 ] start    (at 258ms)
[00:00:06.537,000] <inf> KARVEL: [ 2 ] start    (at 506ms)
[00:00:06.637,000] <inf> KARVEL: [ 2 ] finish   (at 606ms)
[00:00:06.637,000] <inf> KARVEL: [ 1 ] start    (at 606ms)
[00:00:06.787,000] <inf> KARVEL: [ 1 ] finish   (at 756ms)
[00:00:06.832,000] <inf> KARVEL: [ 3 ] finish   (at 801ms)
[00:00:06.941,000] <inf> KARVEL: [ 3 ] start    (at 910ms)
[00:00:07.037,000] <inf> KARVEL: [ 2 ] start    (at 1006ms)
[00:00:07.138,000] <inf> KARVEL: [ 2 ] finish   (at 1107ms)
[00:00:07.236,000] <inf> KARVEL: [ 1 ] start    (at 1205ms)
[00:00:07.386,000] <inf> KARVEL: [ 1 ] finish   (at 1355ms)
[00:00:07.476,000] <inf> KARVEL: [ 3 ] finish   (at 1445ms)
[00:00:07.537,000] <inf> KARVEL: [ 2 ] start    (at 1506ms)
[00:00:07.638,000] <inf> KARVEL: [ 2 ] finish   (at 1607ms)
[00:00:07.837,000] <inf> KARVEL: [ 1 ] start    (at 1806ms)
[00:00:07.987,000] <inf> KARVEL: [ 1 ] finish   (at 1956ms)
[00:00:07.987,000] <inf> KARVEL: [ 3 ] start    (at 1956ms)
[00:00:08.037,000] <inf> KARVEL: [ 2 ] start    (at 2006ms)
[00:00:08.137,000] <inf> KARVEL: [ 2 ] finish   (at 2106ms)
[00:00:08.377,000] <inf> KARVEL: [ 3 ] finish   (at 2346ms)
[00:00:08.751,000] <inf> KARVEL: TEST 3: RM scheduling sequence is correct.
[00:00:08.752,000] <inf> KARVEL: TEST 3: Passed.
```

Figure 6.12: obtained scheduling result for the chosen task set under RM.

```
[00:00:09.254,000] <inf> KARVEL: TEST 4: Testing detection of RM period overruns. Expected fails: 5.
[00:00:09.255,000] <inf> KARVEL: [ 1 ] start    (at 1ms)
[00:00:10.756,000] <inf> KARVEL: [ 1 ] finish   (at 1502ms)
[00:00:10.756,000] <err> KARVEL: Rate Monotonic handler 0: Exceeded period by 501ms.
[00:00:10.756,000] <inf> KARVEL: [ 1 ] start    (at 1502ms)
[00:00:12.256,000] <inf> KARVEL: [ 1 ] finish   (at 3002ms)
[00:00:12.256,000] <err> KARVEL: Rate Monotonic handler 0: Exceeded period by 500ms.
[00:00:12.256,000] <inf> KARVEL: [ 1 ] start    (at 3002ms)
[00:00:13.757,000] <inf> KARVEL: [ 1 ] finish   (at 4503ms)
[00:00:13.757,000] <err> KARVEL: Rate Monotonic handler 0: Exceeded period by 501ms.
[00:00:13.757,000] <inf> KARVEL: [ 1 ] start    (at 4503ms)
[00:00:15.257,000] <inf> KARVEL: [ 1 ] finish   (at 6003ms)
[00:00:15.257,000] <err> KARVEL: Rate Monotonic handler 0: Exceeded period by 500ms.
[00:00:15.257,000] <inf> KARVEL: [ 1 ] start    (at 6003ms)
[00:00:16.757,000] <inf> KARVEL: [ 1 ] finish   (at 7503ms)
[00:00:16.757,000] <err> KARVEL: Rate Monotonic handler 0: Exceeded period by 500ms.
[00:00:16.757,000] <inf> KARVEL: TEST 4: Passed.
```

Figure 6.13: period overrun detection test for the RM API.

cycle should therefore exceed the period, and all attempts should return a deadline miss error. In the implemented API, the error log comes with the account of how big was the overrun. Figure 6.13 depicts the result for a 5-cycle run, and as it can be noticed all deadline misses were observed with an excess of $500ms$, which is the expected outcome. In the light of these results, the implementation of the RM API is considered successful.

6.3 New API

The previous sections highlighted the main aspects of porting the existing KARVEL API to Zephyr, with the occasional inclusion of new methods to complement the already developed functionalities (for example, the extension of the Semaphore API to include a the option to release all semaphores taken by any given task at once). With the porting of the original features

completed, and since the work of this Thesis relates to the practical performance comparison of scheduling algorithms, the next phase consisted of developing a completely new API to introduce new schedulers. Their design and implementation will be described in the following sections.

6.3.1 Earliest Deadline First

The realization of the EDF scheduler, of which theory has been explained in detail in Section 2.8.4, ended up being very unique and the single largest contribution of this Thesis to KARVEL: it greatly enhanced the capabilities of the pre-existing Zephyr EDF API, as this has an over-simplified implementation of EDF, while still offering more flexibility than the RTEMS counterpart, which requires adherent tasks to be periodic and have periods equal to deadlines (EDF in RTEMS is handled by the same kernel object that handles RM, thus sharing some of its characteristics). The goal was to develop an API with an usability close to RM's, so that tasks would be able to alternate between EDF and RM with minimal effort, while also providing means for synchronous deadline miss detection and the ability to run EDF on single-shot tasks as well.

```
1  void periodic_task(void){
2      uint32_t handler;
3      int32_t  id;
4      int32_t  period;
5      int32_t  relative_deadline;
6      ...
7      TLCH_EDFCreate(id, &handler);
8      TLCH_EDFSetPeriod(handler, period);
9
10     while(true){
11         TLCH_EDFDeadlineStart(handler, relative_deadline);
12         do_periodic_work();
13     }
14
15     // no more periodic work to do
16     TLCH_EDFDelete(handler);
17 }
```

Listing 6.9: Minimal code for a periodic EDF task in KARVEL

At the bare minimum, a periodic EDF task can be achieved with the code illustrated in Listing 6.9, which is remarkably similar to the paradigm employed by RTEMS on its RM manager (see Listing 6.6 for comparison); there is only one extra method required to set the period, which is opt-in by default. Since KARVEL provides a close abstraction to the RTEMS implementation, it's therefore possible for a periodic task to switch between schedulers with ease; Such approach facilitates direct performance comparisons between schedulers, simplifies the logic required

at the application level, and brings consistency to the KARVEL API set as a whole. The complete set of methods developed for enabling the EDF scheduler is given below.

- `TLCH_EDFCreate`: initializes a static memory region with the control block of the EDF manager, assigning it to the calling task and resetting the runtime data to their default values. The EDF timer, used to keep track of deadline misses, is initialized as well, but not yet started.

- `TLCH_EDFSetPeriod`: optionally set the period of which the EDF task should be running, in milliseconds. Note this function is merely a setter and will produce no immediate effect on the task scheduling. If configured, the period will be later enforced by calls to the `TLCH_EDFDeadlineStart` method, yielding the same effect as `TLCH_RateMonotonicPeriod`. If desired, the task can invoke this method more than once to update its own period. Doing so will cause the current cycle to immediately shift towards the new value[3].

- `TLCH_EDFSetDeadlineMissCallback`: optionally sets a callback function, as well as a custom argument for it, to be executed whenever a task misses its absolute deadline. This method is also just a setter and will not produce any immediate effect apart from updating the EDF manager control block. A task may invoke this API more than once to update or enable/disable the synchronous track of the deadline state.

- `TLCH_EDFDeadlineStart`: this is the most important method for enabling EDF: it effectively updates the task's absolute deadline by implicitly invoking Zephyr's `k_thread_deadline_set` API. If the task has configured a period, applies suspensions between cycles to provide regular activation intervals, in a similar fashion as the RM API (also warning about period overruns, if detected). Furthermore, automatically starts/stops the EDF timer in case a deadline miss callback is configured.

- `TLCH_EDFDeadlineCancel`: this is a complementary method for tasks which only execute once, or which only desire to have arbitrary sections of their execution scheduled by EDF (thus, without a period). In this scenario, a task may only invoke the `TLCH_EDFDeadlineStart` a few times, and thus eventually need a way of leaving the EDF scheduler and/or stopping the aforementioned deadline timer. The EDF manager data is preserved, which means a subsequent call to `TLCH_EDFDeadline` will simply return the task to EDF.

- `TLCH_EDFDelete`: Does the same as `TLCH_EDFDeadlineCancel` above, but freeing the memory region associated with the EDF manager control block afterwards. Thus, this API is meant for a permanent exit from the EDF scheduler, usually invoked when the task is guaranteed to have no further intention of running under EDF. Returning later to the scheduler is possible, but requires the EDF handler to be re-initialized.

[3] the next activation is always calculated based on the most updated period, so if a shorter one is configured mid-cycle, the task will always consider the new value when calculating the time to suspend until the next cycle. Defining a period smaller than the task's current execution time simply means no task suspension at all.

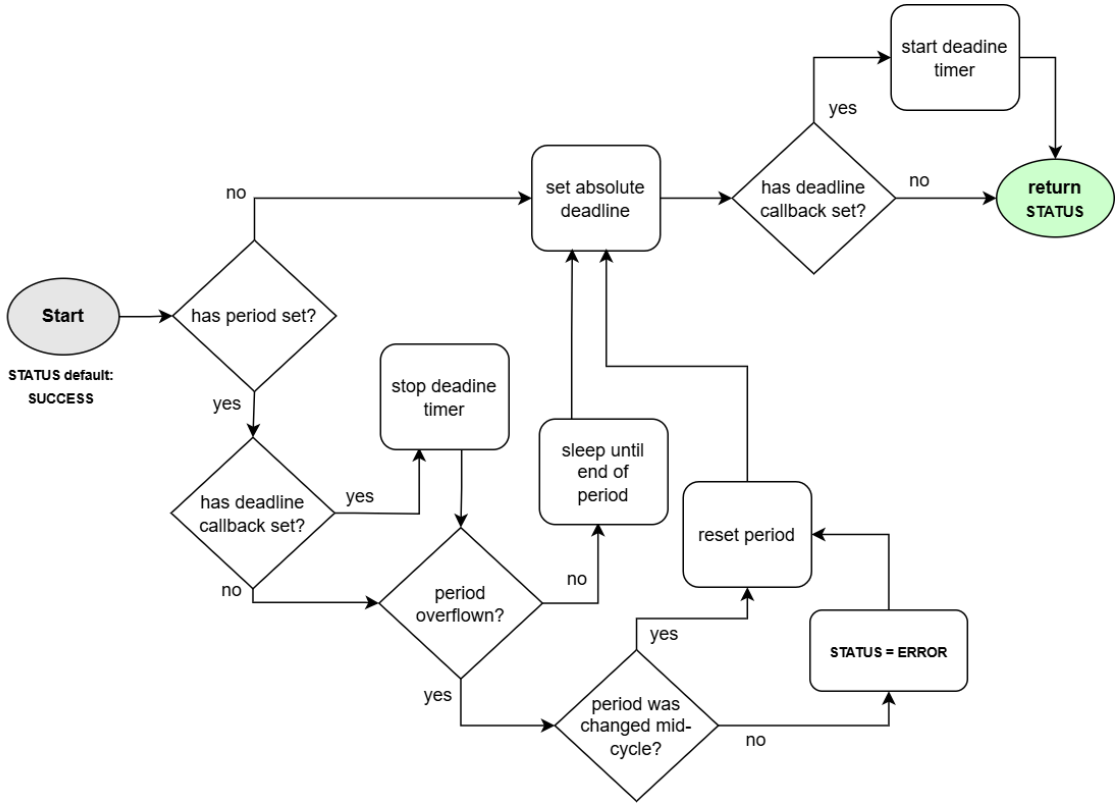


Figure 6.14: TLCH_EDFDeadlineStart algorithm.

Thus, the present API offers a handful of options for a task to be scheduled by EDF. One can configure it for a periodic section or just enable EDF for parts of its logic which execute only once. It's also possible for a task to enable a deadline miss detection, and through a custom callback decide what to do when an overrun happens. Period overflows are also detectable, and all settings can be changed or toggled mid-execution. Moreover, all these settings are put in motion by a single method, `TLCH_EDFDeadlineStart`, and in practice any given task already fit for RM can switch to EDF with minimal adaptations required.

Figure 6.14 illustrates the execution flow of the aforementioned `TLCH_EDFDeadlineStart`, which is the most important method in the EDF API and seeks to perform virtually the same role as RM's `TLCH_RateMonotonicPeriod`. Both APIs define a task's priority and can be invoked at the beginning of a periodic section, but the EDF method executes noticeably more steps due to the extra responsibilities assigned to it. If no period or deadline callback is configured, however, this method will simply set the absolute deadline of the calling task by invoking Zephyr's `k_thread_deadline_set`. If a deadline callback is configured, the EDF timer will start right after the deadline is set. This flow is the upper path in Figure 6.14.

If there's a defined period, the lower path is taken instead, and this method will behave very similarly to `TLCH_RateMonotonicPeriod` - being responsible for both the end of the last cycle and the beginning of the next. As such, the first thing to do is check if the previous cycle finished on time, and, if true, forcing the task to suspend for the remainder of the configured period.

By the time that is completed, the new cycle begins and the task may set its new absolute deadline, following the same steps of the upper path. Naturally, if at every time this method is invoked the current cycle is about to end, the EDF timer must be interrupted before anything else to prevent false deadline miss detections. That's the reason why the configuration or not of a callback is verified on both routes: the same check is performed, but the outcomes derived from it are different.

It's interesting to note that, although the behavior between the RM and EDF methods for applying the deadline are very similar when the period is enabled on the EDF counterpart, the latter performs an extra check for whether or not the task updated the period length during its last cycle. Such verification is not needed in `TLCH_RateMonotonicPeriod` simply because any modifications to the period can both only be set and applied synchronously by this method, whereas in `TLCH_EDFDeadlineStart` the period is an extra setting that can be altered with a call to the auxiliary `TLCH_EDFSetPeriod` API. For the EDF method, this verification only needs to take place if there was a period overflow in the latest cycle.

Two possibilities arise from this event: either it happened because the task executed (or was preempted) for longer than expected, or it deliberately altered this value to be shorter than the previous execution time. The latter implies that the application knows what it's doing and is not considered an error, but the former clearly denotes a troublesome situation which would be considered a deadline miss in RM. Under these terms, the `TLCH_EDFDeadlineStart` will return an error status upon completion, allowing the calling task to take action if desired. Otherwise the execution returns the default successful status. Thus, for the developed EDF API, deadline miss events are detected immediately (with the assistance of timers) and period overflow events are detected asynchronously, by the time the `TLCH_EDFDeadlineStart` method gets to run again. Although undesired, period overflows are not necessarily critical in EDF, hence why it was assumed an asynchronous detection would be sufficient.

Tests

Both unit and functional tests, four in total, were developed to evaluate the correctness of the newly introduced EDF API for KARVEL. While the unit tests were more generic and focused on checking the creation and deletion of EDF managers, the functional tests mirrored the RM suite by assessing the scheduling outcome of a given task set and the accurate detection of deadline misses. The creation and deletion tests went in line with previous tests of the same nature, as EDF is architecturally very similar to RM: attempt to create more managers than allowed by the value of `CONFIG_KARVEL_MAX_EDF_HANDLERS` (a Kconfig option included as part of this EDF contribution to KARVEL), then attempt to delete more managers than what's been previously created. Results for this test were therefore identical to the ones depicted in Figure 6.10: being the Kconfig option of max handlers set to e.g. 10, the API won't allow more than this amount to be created.

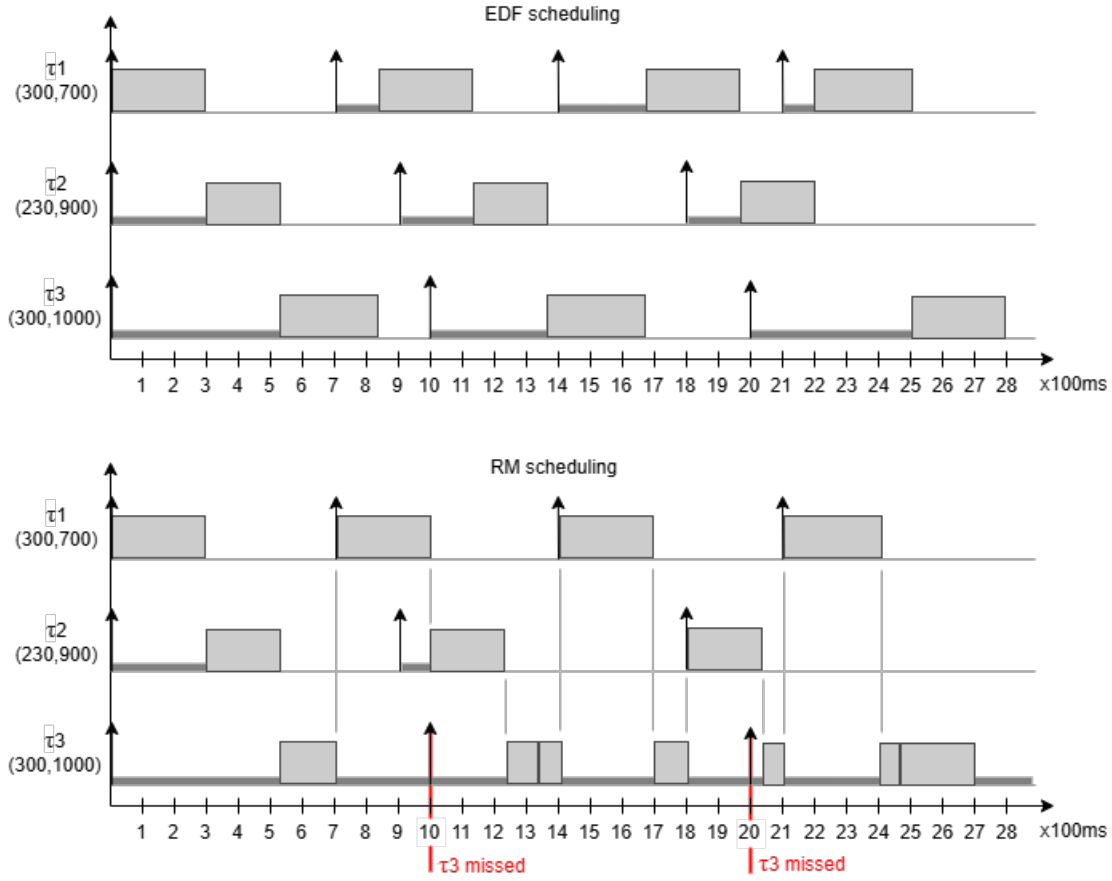


Figure 6.15: theoretical results for EDF and RM of one same task set ($U = 98\%$).

The schedulability tests were also in line with the ones designed for RM, with the major difference being a new task set, purposefully selected for having a combined CPU usage of around 98% (which is schedulable for EDF according to theory but unfeasible for RM by any analysis method explored in Section 2.8.3). This set was selected to provide comparisons with the output of RM. The expected theoretical output for the critical instant under each scheduler is illustrated in Figure 6.15. Once again the tasks involved were periodic, had deadlines equal to periods and leveraged Zephyr's `k_busy_wait` API to emulate their execution times. For EDF's deadline overrun detection, a boolean flag was set per thread as false at the beginning of each cycle, and should a miss event happen, a configured callback would update it to true. At the end of the cycle, then, this flag would be verified and a failure counter would be incremented accordingly.

The results to these tests are available in Figures 6.16 (for EDF) and 6.17 (for RM). In addition to the timestamps relative to the test beginning, EDF also appends the respective task's deadline to help making comparisons against the theory presented in Figure 6.15. By looking at the EDF results, it's clear that the execution order follows the theory both in event sequence and timing, with no deadline misses detected at all. It's also worth noting that even though the theoretical 100% CPU usage limit for single-core EDF is not entirely realistic for real systems,

```
[00:00:05.024,000] <inf> KARVEL: TEST 1: Testing normal execution of tasks under EDF scheduling. Expected fails: 0.
[00:00:05.032,000] <inf> KARVEL: [ 1 ] start    (at 7ms)        (d = 706ms)
[00:00:05.335,000] <inf> KARVEL: [ 1 ] finish   (at 309ms)      (d = 706ms)
[00:00:05.335,000] <inf> KARVEL: [ 2 ] start    (at 310ms)      (d = 906ms)
[00:00:05.565,000] <inf> KARVEL: [ 2 ] finish   (at 540ms)      (d = 906ms)
[00:00:05.565,000] <inf> KARVEL: [ 3 ] start    (at 540ms)      (d = 1005ms)
[00:00:05.865,000] <inf> KARVEL: [ 3 ] finish   (at 840ms)      (d = 1005ms)
[00:00:05.865,000] <inf> KARVEL: [ 1 ] start    (at 840ms)      (d = 1407ms)
[00:00:06.165,000] <inf> KARVEL: [ 1 ] finish   (at 1140ms)     (d = 1407ms)
[00:00:06.166,000] <inf> KARVEL: [ 2 ] start    (at 1140ms)     (d = 1808ms)
[00:00:06.396,000] <inf> KARVEL: [ 2 ] finish   (at 1370ms)     (d = 1808ms)
[00:00:06.396,000] <inf> KARVEL: [ 3 ] start    (at 1370ms)     (d = 2005ms)
[00:00:06.696,000] <inf> KARVEL: [ 3 ] finish   (at 1670ms)     (d = 2005ms)
[00:00:06.696,000] <inf> KARVEL: [ 1 ] start    (at 1670ms)     (d = 2107ms)
[00:00:06.996,000] <inf> KARVEL: [ 1 ] finish   (at 1970ms)     (d = 2107ms)
[00:00:06.996,000] <inf> KARVEL: [ 2 ] start    (at 1970ms)     (d = 2708ms)
[00:00:07.226,000] <inf> KARVEL: [ 2 ] finish   (at 2200ms)     (d = 2708ms)
[00:00:07.227,000] <inf> KARVEL: [ 1 ] start    (at 2201ms)     (d = 2808ms)
[00:00:07.527,000] <inf> KARVEL: [ 1 ] finish   (at 2501ms)     (d = 2808ms)
[00:00:07.527,000] <inf> KARVEL: [ 3 ] start    (at 2501ms)     (d = 3005ms)
[00:00:07.827,000] <inf> KARVEL: [ 3 ] finish   (at 2801ms)     (d = 3005ms)
[00:00:07.827,000] <inf> KARVEL: TEST 1: EDF scheduling sequence is correct.
[00:00:07.827,000] <inf> KARVEL: TEST 1: Passed.
```

Figure 6.16: practical results for EDF scheduling of Figure 6.15.

```
[00:00:08.849,000] <inf> KARVEL: TEST 2: Testing normal execution of tasks under Rate Monotonic (RM) scheduling. Expected fails: 0.
[00:00:08.851,000] <inf> KARVEL: [ 1 ] start    (at 1ms)
[00:00:09.152,000] <inf> KARVEL: [ 1 ] finish   (at 302ms)
[00:00:09.152,000] <inf> KARVEL: [ 2 ] start    (at 302ms)
[00:00:09.382,000] <inf> KARVEL: [ 2 ] finish   (at 532ms)
[00:00:09.382,000] <inf> KARVEL: [ 3 ] start    (at 532ms)
[00:00:09.552,000] <inf> KARVEL: [ 1 ] start    (at 702ms)
[00:00:09.852,000] <inf> KARVEL: [ 1 ] finish   (at 1002ms)
[00:00:09.852,000] <inf> KARVEL: [ 2 ] start    (at 1002ms)
[00:00:10.082,000] <inf> KARVEL: [ 2 ] finish   (at 1232ms)
[00:00:10.202,000] <inf> KARVEL: [ 3 ] finish   (at 1352ms)
[00:00:10.202,000] <err> KARVEL: Rate Monotonic handler 2: Exceeded period by 352ms.
[00:00:10.202,000] <inf> KARVEL: [ 3 ] start    (at 1352ms)
[00:00:10.252,000] <inf> KARVEL: [ 1 ] start    (at 1402ms)
[00:00:10.552,000] <inf> KARVEL: [ 1 ] finish   (at 1702ms)
[00:00:10.652,000] <inf> KARVEL: [ 2 ] start    (at 1802ms)
[00:00:10.882,000] <inf> KARVEL: [ 2 ] finish   (at 2032ms)
[00:00:10.952,000] <inf> KARVEL: [ 1 ] start    (at 2102ms)
[00:00:11.252,000] <inf> KARVEL: [ 1 ] finish   (at 2402ms)
[00:00:11.282,000] <inf> KARVEL: [ 3 ] finish   (at 2432ms)
[00:00:11.282,000] <err> KARVEL: Rate Monotonic handler 2: Exceeded period by 80ms.
[00:00:11.652,000] <err> KARVEL: TEST 2: one or more tasks missed their deadlines.
[00:00:11.653,000] <err> KARVEL: TEST 2: Failed (expected: 0, result: 2)
```

Figure 6.17: practical results for RM scheduling of Figure 6.15.

the fact a 98% task set has been properly scheduled is remarkable.

Additionally, as Figure 6.17 depicts, the practical scheduling result for RM also closely matched the theory depicted in Figure 6.15: the scheduling outcome is the same, and the timing of events is noticeably similar. There is, however, one apparent misalignment between theory and practice that is worth noting: the second error log points a period overrun reports of $80ms$ whereas at this moment in execution the time since the deadline should have been pointed closer to $500ms$. This observation was considered an implementation bug at first, but after further analysis it was noticed this is actually expected and documented. The reason is that the cycle start time resets whenever an overflow is detected, so in the first overflow the cycle start was shifted from $t = 1000ms$ to $t = 1352ms$. Thus, that same cycle end should now be $t = 2352ms$, which is exactly $80ms$ apart from the timestamp of the second overflow

```
[00:00:10.448,000] <inf> KARVEL: TEST 4: Testing detection of EDF deadline misses. Expected fails: 5.
[00:00:10.449,000] <inf> KARVEL: [ 1 ] start    (at 0ms)        (d = 1000ms)
[00:00:11.949,000] <inf> KARVEL: [ 1 ] finish   (at 1501ms)     (d = 1000ms)
[00:00:11.949,000] <err> KARVEL: [ 1 ] missed deadline by 501ms
[00:00:11.949,000] <wrn> KARVEL: EDF handler 0: task executed for longer than its period (took extra 500ms)
[00:00:11.949,000] <inf> KARVEL: [ 1 ] start    (at 1501ms)     (d = 2501ms)
[00:00:13.449,000] <inf> KARVEL: [ 1 ] finish   (at 3001ms)     (d = 2501ms)
[00:00:13.449,000] <err> KARVEL: [ 1 ] missed deadline by 500ms
[00:00:13.449,000] <wrn> KARVEL: EDF handler 0: task executed for longer than its period (took extra 500ms)
[00:00:13.449,000] <inf> KARVEL: [ 1 ] start    (at 3001ms)     (d = 4001ms)
[00:00:14.949,000] <inf> KARVEL: [ 1 ] finish   (at 4501ms)     (d = 4001ms)
[00:00:14.949,000] <err> KARVEL: [ 1 ] missed deadline by 500ms
[00:00:14.949,000] <wrn> KARVEL: EDF handler 0: task executed for longer than its period (took extra 500ms)
[00:00:14.949,000] <inf> KARVEL: [ 1 ] start    (at 4501ms)     (d = 5501ms)
[00:00:16.449,000] <inf> KARVEL: [ 1 ] finish   (at 6001ms)     (d = 5501ms)
[00:00:16.449,000] <err> KARVEL: [ 1 ] missed deadline by 500ms
[00:00:16.449,000] <wrn> KARVEL: EDF handler 0: task executed for longer than its period (took extra 500ms)
[00:00:16.450,000] <inf> KARVEL: [ 1 ] start    (at 6001ms)     (d = 7001ms)
[00:00:17.950,000] <inf> KARVEL: [ 1 ] finish   (at 7501ms)     (d = 7001ms)
[00:00:17.950,000] <err> KARVEL: [ 1 ] missed deadline by 500ms
[00:00:17.950,000] <inf> KARVEL: TEST 4: Passed.
```

Figure 6.18: period overrun detection test for the EDF API, with periods equal to deadlines.

event, $t = 2452ms$ (hence the logged value). It's obviously debatable if such policy of resetting periods is the best strategy towards overruns, however since this RM implementation is intended to follow the same convention as RTEMS, no changes were considered necessary. Furthermore, as the introduced EDF API is aimed at being as interchangeable as possible with RM's, the optional period configuration for EDF will also reset the task's cycle upon detection of overflows.

The last developed test, depicted in Figure 6.18, is precisely about this detection mechanism and follows the same structure as described for RM (which results were shown in Figure 6.13). In both of these tests, the same task used in the aforementioned schedulability assessment attempts to execute five 1500-millisecond cycles with declared periods of 1000 milliseconds (and deadlines once again equal to periods). It comes to no surprise that all attempts should fail, as the execution time clearly exceeds the intended period. Figure 6.18 shows two log messages for each detected overrun: an error log, exhibited by the task right after checking the flag set by the deadline callback, and a warning log, performed by the `TLCH_EDFDeadlineStart` method when it detects the overrun.

Although apparently redundant, these log pairs are generated by different mechanisms and highlight that each is working as expected. For the deadline start method, an overrun is not considered critical, only undesired, hence why it outputs a warning message (it does however return an error status code that can be used by the calling task). As for the task itself, the overrun will be considered an error simply because it has detected a deadline miss, and such is simply not admitted in critical systems.

Thus, the logs only seem redundant because deadlines are equal to periods for this test case. To prove that, a second version of this test dropped the deadline to $850ms$ while keeping the period set as $1000ms$ and the execution time as $1500ms$. In this scenario, by the time of each

```
[00:00:05.009,000] <inf> KARVEL: TEST 1: Testing detection of EDF deadline misses. Expected fails: 5.
[00:00:05.011,000] <inf> KARVEL: [ 1 ] start    (at 1ms)        (d = 851ms)
[00:00:06.512,000] <inf> KARVEL: [ 1 ] finish   (at 1502ms)     (d = 851ms)
[00:00:06.512,000] <err> KARVEL: [ 1 ] missed deadline by 651ms
[00:00:06.512,000] <wrn> KARVEL: EDF handler 0: task executed for longer than its period (took extra 501ms)
[00:00:06.512,000] <inf> KARVEL: [ 1 ] start    (at 1502ms)     (d = 2352ms)
[00:00:08.012,000] <inf> KARVEL: [ 1 ] finish   (at 3002ms)     (d = 2352ms)
[00:00:08.012,000] <err> KARVEL: [ 1 ] missed deadline by 650ms
[00:00:08.012,000] <wrn> KARVEL: EDF handler 0: task executed for longer than its period (took extra 500ms)
[00:00:08.013,000] <inf> KARVEL: [ 1 ] start    (at 3003ms)     (d = 3852ms)
[00:00:09.513,000] <inf> KARVEL: [ 1 ] finish   (at 4503ms)     (d = 3852ms)
[00:00:09.513,000] <err> KARVEL: [ 1 ] missed deadline by 651ms
[00:00:09.513,000] <wrn> KARVEL: EDF handler 0: task executed for longer than its period (took extra 500ms)
[00:00:09.513,000] <inf> KARVEL: [ 1 ] start    (at 4503ms)     (d = 5353ms)
[00:00:11.013,000] <inf> KARVEL: [ 1 ] finish   (at 6003ms)     (d = 5353ms)
[00:00:11.013,000] <err> KARVEL: [ 1 ] missed deadline by 650ms
[00:00:11.013,000] <wrn> KARVEL: EDF handler 0: task executed for longer than its period (took extra 500ms)
[00:00:11.013,000] <inf> KARVEL: [ 1 ] start    (at 6003ms)     (d = 6853ms)
[00:00:12.513,000] <inf> KARVEL: [ 1 ] finish   (at 7503ms)     (d = 6853ms)
[00:00:12.513,000] <err> KARVEL: [ 1 ] missed deadline by 650ms
[00:00:12.514,000] <inf> KARVEL: TEST 1: Passed.
```

Figure 6.19: period overrun detection test for the EDF API, with periods different from deadlines.

cycle end the deadline should be missed by nearly $650ms$ whereas the period should have overflown by $500ms$ - and that's exactly the output depicted in Figure 6.19, which shows the execution of this test variant. With both the deadline callback and period overflow showcasing correct behaviors, the EDF API implementation can be considered successful in all of its aspects.

6.3.2 Constant Bandwidth Server

As explained in Chapter 3, the usual way or implementing the CBS on real-time systems has historically been directed towards a more conservative variant called the "hard" CBS, which effectively suspends the job execution on the occasion of a budget exhaustion. This is the approach taken by RTEMS, Linux (with the `SCHED_DEADLINE` option enabled) and PikeOS (Community 2024, RTEMS Project 2025b and Vanga et al. 2017) to avoid an unbounded interference on other EDF tasks. In these implementations, the deadline is subject to recalculation at *every* context switch in which a CBS job enters the CPU (instead of only when a job is pushed to an idle server, or when a job exhausts the budget), a subtle but significant change that greatly reduces the adaptability of the theoretical model. However, to the best of our knowledge there is currently no implementation of the CBS which closely follows the theory in any commercial or open-source RTOS.

Zephyr, which has a very rudimentary support to EDF in the first place, unsurprisingly lacks the support for scheduling servers of any sort. However, there was a previous work conducted by the author of this Thesis which aimed at porting the CBS according to the theoretical model (Paschoaletto et al. 2025). The implementation consisted of a message queue to store incoming jobs, a dedicated system thread to execute them and a timer to keep track of the budget consumption. In search for the ease of use by application developers and in line with

Zephyr standards, the exposed API consisted of essentially one `K_CBS_INIT` macro for static initialization and a runtime `k_cbs_push_job` to insert jobs in the given CBS queue. The resulting implementation was successful in bringing the theoretical model into practice, and managed to do so having the worst-case measured overhead barely surpassing 150 clock cycles for the RISC-V microcontroller used for tests. The resulting code was turn into a contribution and submitted as a Pull Request, being under analysis by Zephyr reviewers at the time of writing (Paschoaletto 2025a).

However, the original CBS implementation was not suited for direct integration into KARVEL for a few reasons. First, the standalone contribution necessarily configured the CBS in a static manner (at compile time) whereas KARVEL initializes nearly all of its components at runtime. Second, the standalone CBS in Zephyr was tailored for executing arbitrary jobs (materialized as isolated functions) within an internal thread, whereas in KARVEL all executive entities are explicitly defined at application level through either tasks or interrupts. Finally, the original CBS was developed around the very simple EDF implementation contained in Zephyr, which is a much different situation now with the feature-rich API introduced in Section 6.3.1. It therefore made considerably more sense to simply modify the introduced EDF as a solution to enable CBS in KARVEL rather than creating a whole new separate API just for this effect, however the Zephyr contribution would still be able to provide invaluable insights regarding kernel interactions.

The main design change between theory and this new implementation was to make the CBS tailored for the regulation of application tasks rather than arbitrary jobs. In this model, each task opting to be scheduled by EDF would now be able to have one exclusive CBS to modulate its execution, as if each task cycle corresponded to a job release in the theoretical model. A task-centric approach greatly simplifies the implementation and cuts some edge cases that could not be avoided in the contribution made for Zephyr in (Paschoaletto 2025a). If the jobs are now cycles of one same task and cycle starts are necessarily marked with a call to `TLCH_EDFDeadlineStart`, than there is *never* more than one job at once in the system. This means there is no need to check if the CBS was idle by the time a new job comes simply because it will always be finished with the previous cycle by the start of the next, even if it missed a deadline. Thus, the CBS can always be assumed idle when `TLCH_EDFDeadlineStart` is invoked and the existence of a job queue becomes redundant.

The only real change needed for EDF in terms of API was the inclusion of a function to configure the maximum budget Q itself. The CBS period, on the other hand, didn't need a proper configuration method as its value could be inferred from the task's relative deadline (which is mandatorily declared on every call to `TLCH_EDFDeadlineStart`). This makes sense, as in practical terms the CBS period acts more like a relative deadline (dictating the length of the absolute deadline) than an actual period (a behavior which, again, the CBS doesn't require). Moreover, using the EDF period instead of the deadline would make the CBS dependent on another optional parameter to work, which is not desired at all.

The new method for setting the task's budget was named `TLCH_EDFSetCBSBudget`. It was designed to be optional and used as a setter, similarly to the ones created for the period and deadline callback. The actual enforcement of this value only takes effect during `TLCH_EDFDeadlineStart`, which would then track the budget consumption using a dedicated timer (the same approach of the standalone Zephyr contribution described in (Paschoaletto et al. 2025)). The timer interval is configured with the available budget at each activation, and as a result, it expires precisely when the budget is exhausted.

```
void TLCH_CBSTimerCallback(struct k_timer *budgetTimer){
    cbs->currentBudget = cbs->maxBudget;
    oldDeadline = edfHandler->owner->base.prio_deadline;
    newDeadline = oldDeadline + cbs->relativeDeadline;
    ...
    k_timer_start(budgetTimer, K_TICKS(cbs->maxBudget), K_FOREVER);
    k_thread_deadline_set_absolute(
        edfHandler->owner, K_TICKS(newDeadline)
    );

    if(HAS_DEADLINE_CALLBACK(edfHandler)){
        k_timer_start(
            deadlineTimer, K_TICKS(newDeadline - now), K_FOREVER
        );
    }
    ...
}
```

Listing 6.10: CBS budget timer expiry function.

The callback function developed to perform the aforementioned steps has the most relevant parts viewable on Listing 6.10. Note there's an extra step at the end that checks if the deadline miss callback is configured, and which re-starts the deadline timer if such is the case. This is not part of the CBS logic *per se*, but is required since the task's deadline is necessarily postponed at this point. Thus, it becomes obvious the deadline timer needs to follow the postponing in order to keep working as intended. The changes introduced in `TLCH_EDFDeadlineStart`, by their turn, basically consisted of applying extra steps at the start and end of this method to check if the task has a configured CBS, and if so, perform the necessary logic corresponding to a job's end and beginning, respectively. Figure 6.20 builds on top of the original algorithm (Figure 6.14) and highlights the newly included steps in yellow. Note that the core execution flow remains exactly the same as before when the budget is not set.

When the CBS is enabled, the task will check, in the upper path right before setting the new deadline, if the bandwidth condition (Equation 2.4) is met, i.e. if there is still a considerable amount of budget left from a previous execution cycle compared to the current absolute deadline. If true, the deadline is in fact recalculated and the budget is replenished to match the predefined bandwidth. If false, the task will *not* recalculate its deadline (or refill the budget).

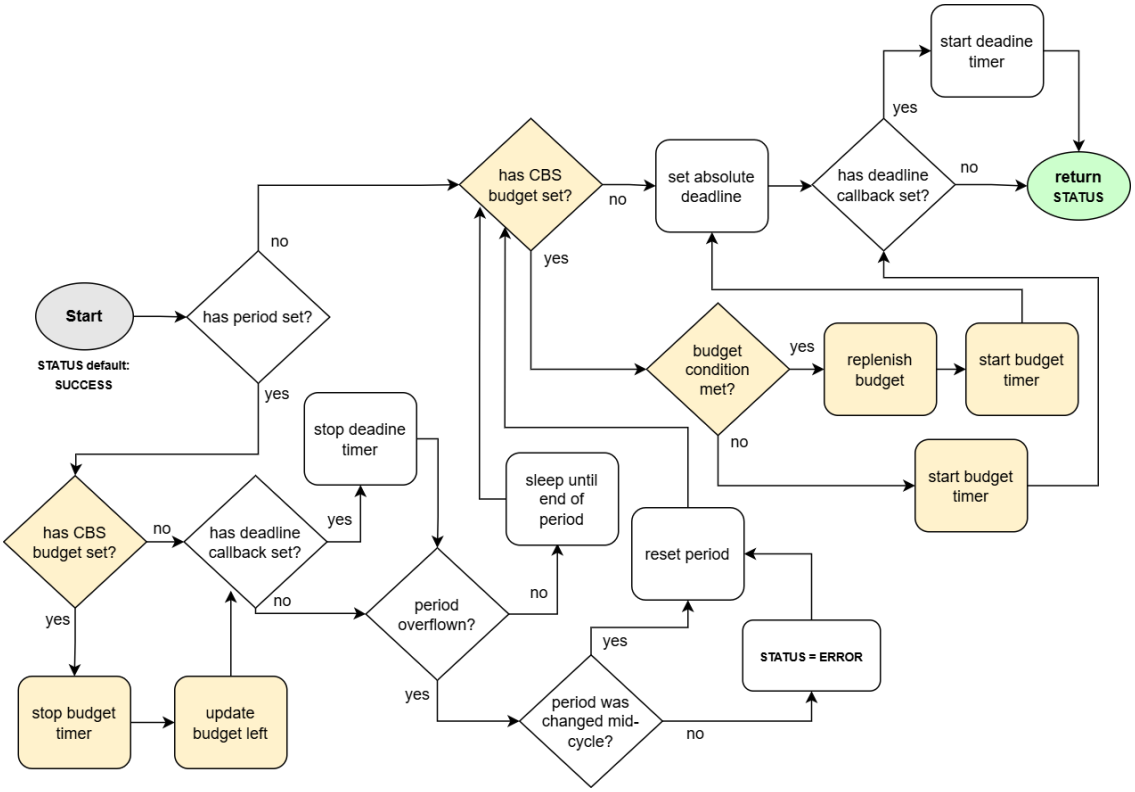


Figure 6.20: TLCH_EDFDeadlineStart algorithm with the CBS extension.

That happens because an unmet condition implies the task can be served with the current deadline without risking schedulability guarantees, thus making the deadline setting unnecessary[4]. Hence, enabling the CBS in `TLCH_EDFDeadlineStart` means it may no longer update the task's deadline. If the task has configured a period, the lower path of Figure 6.20 is traversed, and the CBS both stops the budget timer and updates the budget level to reflect what's been consumed.

Preemptions

The steps added for the CBS in Figure 6.20 are responsible for updating a task's budget based on how much was spent during the previous execution cycle. However, they assume for calculations that the task remained uninterrupted in the CPU between calls to `TLCH_EDFDeadlineStart`, which is not true if a preemption ever happens in between. Some additional procedure was therefore required to suspend the consumption when preemptions take place; in practical terms, this meant introducing some code on context switches to pause the budget timer when a CBS task is forced to leave the CPU, and resume it once it returns. This is exactly the strategy adopted in (Paschoaletto et al. 2025), from where the CBS timer was also taken from.

To that effect, the first step then was to include a reference to the task's CBS in its own control block, thus enabling the kernel to fetch the CBS data from the thread when performing a

[4]This would be the case, for example, of an sporadic task executing for too little time in a previous cycle, thus still having plenty of budget left in the next one.

switch. Then, the methods shown in Listing 6.11 were placed in the relevant context switching paths: `TLCH_CBSSwitchedIn` to update the incoming thread's CBS, and `TLCH_CBSSwitchedOut` to handle the outgoing counterpart. Both took as argument the CBS pointer inserted in the thread control block.

```
void TLCH_CBSSwitchedIn(TLCH_CBS_t *cbs) {
  if (cbs->isActive) {
        k_timer_start(
            &(cbs->budgetTimer),
            K_TICKS(cbs->currentBudget),
            K_NO_WAIT
        );
        cbs->startTick = k_uptime_ticks();
    }
}

void TLCH_CBSSwitchedOut(TLCH_CBS_t *cbs) {
    if (cbs->isActive) {
        k_timer_stop(&(cbs->budgetTimer));
        cbs->currentBudget -= (k_uptime_ticks() - cbs->startTick);
    }
}
```

Listing 6.11: context switch functions required for the CBS budget tracking.

As for the executed logic, it goes as follows. When a CBS-enabled task leaves the CPU, the scheduler checks if the CBS in question was active (consuming budget) or not. If so, it simply stops the budget timer (which will prevent deadline recalculations) and update the budget left by subtracting the delta since the cycle start. Later on, when the task returns to the CPU, the budget timer is re-started with the updated value as expiration interval and the starting tick is also updated for future calculations. These steps guarantee that on the possibility of subsequent preemptions in one same cycle, the consumed budget will always reflect the de facto execution time.

Tests

Before proceeding to develop any new test cases to evaluate the CBS implementation, the original EDF-only schedulability test with the task set of Figure 6.15 was executed once again to make sure that the changes to `TLCH_EDFDeadlineStart` hadn't produced any unexpected side effects in the regular EDF scheduler. Fortunately, the results remained identical to the ones observed in Figure 6.16, therefore no problems were identified on this regard.

Knowing the EDF continued behaving accordingly, two new schedulability tests were appended to the existing suite with new task sets. In the first set, a standard EDF task (in the exact same format as the ones employed in the "pure" EDF tests) attempted to run alongside another which was regulated by a CBS, making a combined CPU usage over 90%. In the second task

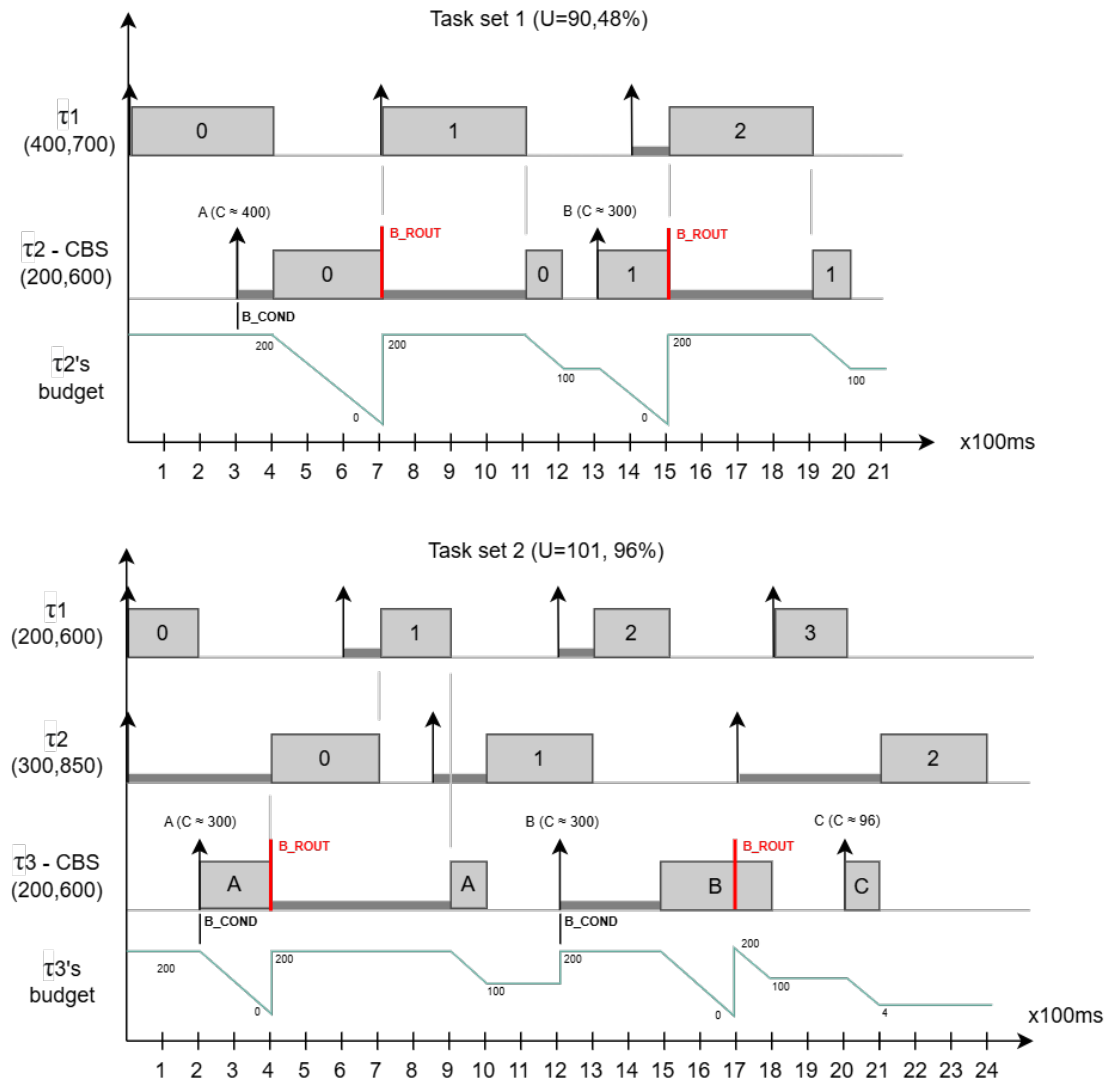


Figure 6.21: Task sets used for the scheduling tests of EDF + CBS.

set, two standard EDF tasks and one CBS task contended for the CPU in a combined utilization of over 101%. Note that the latter is presumably not schedulable on a single-core processor but did manage to produce an acceptable result for the analyzed time window.

The CBS task was built from the EDF peers, but modified to remove the periodic activations in favor of individually configurable intervals between cycles. On task set 2, for instance, this enabled jobs A and B to be activated 1000 milliseconds between each other, whereas jobs B and C stood apart by 800 milliseconds. Further modifications included capturing the absolute deadlines of the task at both job start and end phase to infer if a budget runout event happened in between. This information would then be compared with previously configured data at each cycle end to verify whether or not such event should had really happened at the respective cycle or not.

The expected scheduling outcome for each task set is observable in Figure 6.21. While the

```
[00:00:13.019,000] <inf> KARVEL: TEST 2: Testing execution of EDF + CBS scheduling. Expected fails: 0.
[00:00:13.020,000] <inf> KARVEL: [ 1 ] start    (at 0ms)        (d = 700ms)
[00:00:13.320,000] <inf> KARVEL: [    ] B_COND
[00:00:13.420,000] <inf> KARVEL: [ 1 ] finish   (at 400ms)      (d = 700ms)
[00:00:13.420,000] <inf> KARVEL: [ 2 ] start    (at 400ms)      (d = 1100ms)
[00:00:13.720,000] <wrn> KARVEL: [    ] B_ROUT
[00:00:13.721,000] <inf> KARVEL: [ 1 ] start    (at 701ms)      (d = 1401ms)
[00:00:14.121,000] <inf> KARVEL: [ 1 ] finish   (at 1101ms)     (d = 1401ms)
[00:00:14.201,000] <inf> KARVEL: [ 2 ] finish   (at 1181ms)     (d = 1900ms)
[00:00:14.321,000] <inf> KARVEL: [ 2 ] start    (at 1301ms)     (d = 1900ms)
[00:00:14.539,000] <wrn> KARVEL: [    ] B_ROUT
[00:00:14.540,000] <inf> KARVEL: [ 1 ] start    (at 1521ms)     (d = 2101ms)
[00:00:14.940,000] <inf> KARVEL: [ 1 ] finish   (at 1921ms)     (d = 2101ms)
[00:00:15.015,000] <inf> KARVEL: [ 2 ] finish   (at 1996ms)     (d = 2700ms)
[00:00:15.121,000] <inf> KARVEL: [ 1 ] start    (at 2101ms)     (d = 2801ms)
[00:00:15.521,000] <inf> KARVEL: [ 1 ] finish   (at 2501ms)     (d = 2801ms)
[00:00:15.521,000] <inf> KARVEL: TEST 2: EDF + CBS scheduling sequence is correct.
[00:00:15.521,000] <inf> KARVEL: TEST 2: Passed.

[00:00:16.024,000] <inf> KARVEL: TEST 3: Testing execution of EDF + CBS scheduling. Expected fails: 0.
[00:00:16.025,000] <inf> KARVEL: [ 1 ] start    (at 0ms)        (d = 600ms)
[00:00:16.223,000] <inf> KARVEL: [    ] B_COND
[00:00:16.225,000] <inf> KARVEL: [ 1 ] finish   (at 200ms)      (d = 600ms)
[00:00:16.225,000] <inf> KARVEL: [ 3 ] start    (at 201ms)      (d = 799ms)
[00:00:16.425,000] <wrn> KARVEL: [    ] B_ROUT
[00:00:16.426,000] <inf> KARVEL: [ 2 ] start    (at 401ms)      (d = 850ms)
[00:00:16.726,000] <inf> KARVEL: [ 2 ] finish   (at 701ms)      (d = 850ms)
[00:00:16.726,000] <inf> KARVEL: [ 1 ] start    (at 702ms)      (d = 1201ms)
[00:00:16.926,000] <inf> KARVEL: [ 1 ] finish   (at 902ms)      (d = 1201ms)
[00:00:17.016,000] <inf> KARVEL: [ 3 ] finish   (at 992ms)      (d = 1399ms)
[00:00:17.016,000] <inf> KARVEL: [ 2 ] start    (at 992ms)      (d = 1701ms)
[00:00:17.228,000] <inf> KARVEL: [    ] B_COND
[00:00:17.316,000] <inf> KARVEL: [ 2 ] finish   (at 1292ms)     (d = 1701ms)
[00:00:17.316,000] <inf> KARVEL: [ 1 ] start    (at 1292ms)     (d = 1801ms)
[00:00:17.517,000] <inf> KARVEL: [ 1 ] finish   (at 1492ms)     (d = 1801ms)
[00:00:17.517,000] <inf> KARVEL: [ 3 ] start    (at 1492ms)     (d = 1804ms)
[00:00:17.716,000] <wrn> KARVEL: [    ] B_ROUT
[00:00:17.817,000] <inf> KARVEL: [ 3 ] finish   (at 1792ms)     (d = 2404ms)
[00:00:17.817,000] <inf> KARVEL: [ 2 ] start    (at 1792ms)     (d = 2551ms)
[00:00:17.825,000] <inf> KARVEL: [ 1 ] start    (at 1800ms)     (d = 2400ms)
[00:00:18.025,000] <inf> KARVEL: [ 1 ] finish   (at 2000ms)     (d = 2400ms)
[00:00:18.026,000] <inf> KARVEL: [ 3 ] start    (at 2002ms)     (d = 2404ms)
[00:00:18.122,000] <inf> KARVEL: [ 3 ] finish   (at 2098ms)     (d = 2404ms)
[00:00:18.392,000] <inf> KARVEL: [ 2 ] finish   (at 2368ms)     (d = 2551ms)
[00:00:18.426,000] <inf> KARVEL: [ 1 ] start    (at 2401ms)     (d = 3001ms)
[00:00:18.626,000] <inf> KARVEL: [ 1 ] finish   (at 2601ms)     (d = 3001ms)
[00:00:18.626,000] <inf> KARVEL: TEST 3: EDF + CBS scheduling sequence is correct.
[00:00:18.626,000] <inf> KARVEL: TEST 3: Passed.
```

Figure 6.22: Test results for the EDF + CBS task sets of Figure 6.21.

simpler test case 1 is directly taken from the examples given by CBS authors in (G. C. Buttazzo 2011), the more complex task case 2 allows one to verify the whole set of idiosyncrasies associated with the CBS. At instant $t = 200ms$ of case 2, the budget condition of τ_3 is met simply because the server has yet no deadline set for it, so it is calculated as $d = t + T$ at this point ($200 + 600 = 800ms$). That's why τ_3 manages to begin its execution immediately, surpassing τ_2 (which has $d_{\tau_2} = 850ms$). Shortly after that the circumstances change, as τ_3 exhausts the budget and has its priority updated to $d_{\tau_3} = d_{\tau_3} + T = 1400ms$ to compensate for the replenishment.

When job B comes (the second activation of τ_3), the budget left is $100ms$ and the time remaining until d_{τ_3} is $200ms$. This yields a bandwidth of 50%, which is higher than the configured $Q/T = 33\%$ and therefore calls for a budget replenishment. The new deadline is thus

$d_{\tau_3} = t + T = 1800ms$. When the job finally gets to execute, it ends up exhausting the budget once again at $t = 1700ms$, which is exactly when τ_2 becomes ready for a new cycle. However, the updated deadline for τ_3 is $d_{\tau_3} = 2300ms$, whereas τ_2 receives $d_{\tau_2} = 2550ms$. In other words, τ_3 remained with the highest priority of all even after a deadline recalculation, which explains why it gets to keep executing despite the activation of τ_2.

Finally, the release of job C has a remaining budget of $100ms$ and a time until deadline of $300ms$. This is a bandwidth of 33%, which is equal to Q/T and therefore needless of recalculations. Job C is then served with the same deadline as before, sparing the involvement of the scheduler this time around.

The test results of both task sets are depicted in Figure 6.22; for the sake of clarity, additional logs have been inserted at the CBS keypoints to highlight its main events. As expected, the observed sequence matches the theoretical output of Figure 6.21 with great precision: the tasks showcase the correct execution order and timing in both tests, and the budget condition check and exhaustion procedures are also according to theory. The CBS-regulated tasks have their deadlines altered mid-execution on cycles with budget runouts, and this has the effect of them losing the CPU to awaiting tasks on all occasions but the last occurrence of task set 2. As these results seem to show that the CBS is working accordingly and without jeopardizing regular EDF scheduling, this implementation in KARVEL is considered adequate.

Chapter 7

Test Application

Although satellites were the obvious candidates for an implementation, those were unfortunately not available during the works of this Thesis. Emulation could probably be achieved by coupling microcontrollers and device stubs with some software taken from an actual spacecraft hardware, but it was proved challenging to use too much of an existing commercial product that already utilizes KARVEL without risking problems with Non-Disclosure Agreements (NDAs) between CSW and its business partners. Furthermore, emulation guarantees the same logical output as the original party but may not meet temporal accuracy, which is critical for performing schedulability benchmarks. In the works conducted in (Paschoaletto et al. 2025), for example, the CBS implementation yielded the same scheduling sequence for both QEMU targets and real microcontrollers but found highly inconsistent overhead measurements for context switches in emulation, effectively needing to be discarded for the end analysis.

Having these into consideration, the application chosen for testing the work described in Chapter 6 was the firmware for a robotic arm. Beyond the aforementioned reasons, a few extra ones were considered for making this choice, the main ones being the previous experience of this Thesis' author with robots, the existence of an easily accessible robot with outdated firmware, and the intrinsic nature of a robotic application, which demands both periodic and aperiodic, time triggered and event-triggered, tasks to work accordingly, often demanding high CPU usage in the process. Throughout the remainder of this Chapter, we'll describe the robot used for the tests and the application architecture developed for it.

7.1 The Robot

The machine chosen for the application development was the PegasoV3, a robotic arm entirely developed by this Thesis' author as the final project of the Mechatronics Engineering course; the full design process is described in (Paschoaletto 2022). The arm, viewable in Figure 7.1, features the same articulated topology of industrial manipulators and has 6 independent joints, known in literature as Degrees of Freedom (DOF), all driven by NEMA 17 stepper motors and coupled with planetary gear reductions to amplify their torque at the expense of their speed.

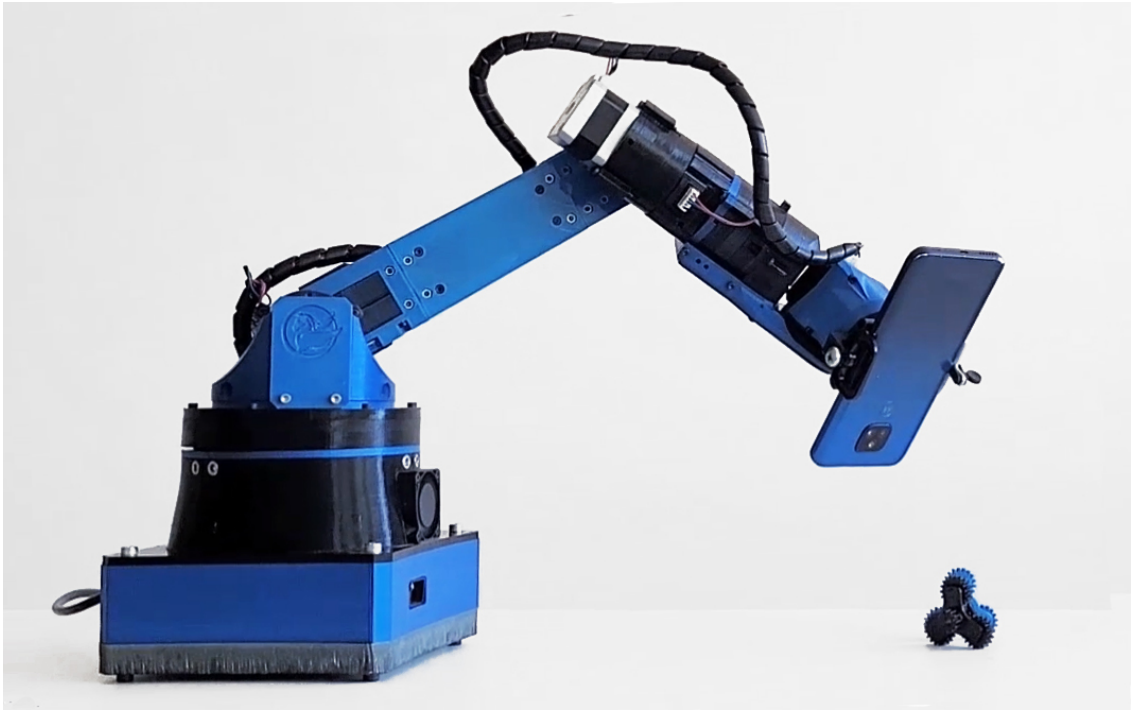


Figure 7.1: PegasoV3 robot.

7.1.1 Software

The software developed originally for the robot was written in Micropython and featured a single-threaded, super-loop programming paradigm. It was therefore inefficient by design, however writing the code in an interpreted language was highly beneficial from the development standpoint. The super-loop routine consisted of polling for inputs from either a UART or USB ports, then parsing the received payload (if any) to derive the commands, then simply executing them accordingly before resuming polling. The UART port was connected to an ESP-01s WiFi module, which displayed a web interface for the robot control, whereas the USB port was directly accessible to the user by plugging in a cable to an external device such as a computer or tablet. The UART responses to command were kept to a minimal and formatted as HTML snippets to be included in the web interface, whereas the USB responses featured additional logs about the robot status for debugging purposes.

Motion control

When a movement was commanded, a dedicated routine proceeded with identifying the motors needed to run, calculating the necessary pulses each selected motor should receive to reach the position, and finally cycling through them, regularly applying pulses (steps) to each motor until all reached the calculated step count. The cycle period was unconventionally independent of the motion length: it was merely calculated as $T = 3200/M_\mu * V_r$, where M_μ was

the motors' microstep resolution[1] and V_r was the robot speed. In other words, V_r only determined the *single step* duration - not the *entire movement* duration. This approach clearly had a disadvantage of unbounded total movement times, but it guarantees the feasibility of the resulting step periods (i.e. they would always be greater than the minimum accepted stepper driver period) and was found adequate at the time of the original work.

```
def move(selected, intervals, acc, T):
    while (var.totalSteps > 0):
            i = acc.index(min(acc))
            var.motors[selected[i]].step()
            acc[i] += intervals[i]
            var.totalSteps -= 1
            utime.sleep_us(T)
            check_interface()
```

Listing 7.1: movement sub-routine.

The sub-routine responsible for making the robot work is illustrated in Listing 7.1, where it's interesting to note two things. First, although this strategy was able to make the robot start and finish the movement of all concerned motors at the same time, it did so disregarding the total movement time and with a single function coordinating all joint movements. For the single-threaded context of the original application this worked, but a completely different solution would likely be needed when refactoring the code for using multiple threads. Second, the same single-threaded processing requires that polling for UART/USB inputs also happens *within* the movement sub-routine, or else the robot would simply ignore emergency stop commands if those ever arrived.

Inverse kinematics

The PegasoV3 had two options of movement commands: by joint angles and by coordinates. The first approach simply invoked the aforementioned flow directly, passing in the desired set of angles for each of the 6 motors, however this sort of control is not very practical because, just like humans won't position their own arms thinking about what individual angles the elbow, wrist and shoulder should assume for the hand to reach certain location, it's quite often the position of the robot's *end effector* (the place where the tools go attached) that matters the most for robot use cases. The mathematical field used to provide relations between a set of coordinates (usually in Euclidean space, i.e. XYZ) and the individual position of each joints is broadly known as *kinematics*. If one has the joint angles and desires to find the corresponding coordinates, he applies the *forward* kinematics (FK) analysis for this effect. Conversely, if one

[1] a *microstep* is a feature provided by stepper motor drivers that enables one to divide the default angle that the motor turns per step (for NEMA17 motors, that's 1.8 degrees) in two or more smaller steps. For the DRV8825 drivers used, a full step can be divided up to 32 times, meaning a more granular step displacement of up to 0.05625 degrees.

has the coordinates and wishes to know the respective joint angles, that's a job for the *inverse* kinematics (IK) analysis.

Just like scheduling algorithms, plenty of solutions were proposed throughout the years to tackle both forward and inverse kinematics. In the original PegasoV3 development, only the IK option was implemented due to its higher importance for the purpose of camera assistant it was made for. The implemented algorithm was heavily inspired in the Triangulation method described in (Muller-Cajar and Mukundan 2007), which had the significative advantage of not being recursive and guaranteeing a solution for any point in space which is known to be within the robot's reach. As a consequence, the second option of movement command was to provide a set of [X, Y, Z] and [Roll, Pitch, Yaw] parameters[2] to the robot and see it internally apply the IK analysis to derive the absolute joint angles that would take it the to those coordinates. These angles were then forwarded to the motion control routine as usual.

7.2 KARVEL-based Application

As seen in the previous sections, the existing application was very limited and inefficient. It was developed in Micropython, and ran entirely based on polling strategies and single-threaded processing for all the work - UART/USB command polling and parsing, non-volatile storage management, kinematics calculations and motor coordination. It also had no operating system aside from the Micropython interpreter, and therefore made no use of kernel primitives such as semaphores, timers and message queues. In fact, with the exception of the IK algorithm and GPIO mapping, nothing else in the original firmware was fit for a KARVEL-based version of it, and as a consequence a completely new application was found needed. We now proceed to describe the resulting implementation, highlighting the role performed by each created task.

7.2.1 Requirements

As the implementation for the purpose of this Thesis was meant more for casting performance comparisons between scheduling algorithms than actually using the robot on its original use case, the requirements for the application had slightly different priorities from the pre-existing Micropython one. For instance, while the previous firmware only sought to provide a coordinated movement between the joints at the motion control routine, the newly developed, Zephyr-based application should instead be able to achieve the same coordination with the usage of a scheduler and a multi-threaded paradigm. The following items were the base requirements outlined for the new application.

1. There should be one task to control each motor, independently.

    1.1. The task should have a message queue to receive commands.

[2] while [X,Y,Z] are the well-known tridimensional space coordinates, the [Roll, Pitch, Yaw] parameters are what specify the object orientation when achieving the XYZ coordinates.

1.2. The message sent to this task's queue should contain the end position of the motor, in degrees, as well as the desired amount of time to perform the full motion and the scheduling algorithm to employ for the motion.

1.3. Once received a command, the task should perform all the necessary calculations related to that single motor and execute the movement afterwards.

2. There should be a synchronization barrier shared between the motor tasks.

2.1. Once a task finishes movement, it should wait indefinitely at the barrier before proceeding to the next command.

3. There should be a semaphore to control motor-related shared data access.

3.1. Changes in motor resolution should be protected by this semaphore.

3.2. Changes in the variable holding the number of active motors should be protected by this semaphore.

3.3. The reset of motors should be protected by this semaphore.

4. There sould be a Motion Control task to receive, parse and distribute motion commands to the respective motor tasks.

4.1. The task should also have a message queue to receive motion commands.

4.2. The message sent to this task's queue should contain all joint's absolute position, in degrees, as well as the time period to reach this position and the scheduling algorithm to use when performing the motion.

4.3. The task should be able to identify if the commanded angle is beyond the limits of any given joint, and block the command in this case.

5. There should be a Kinematics task to perform the IK calculations.

5.1. The task should have a message queue to receive coordinates, as well as the time period to reach the coordinates and the scheduling algorithm to use when performing the movement.

5.2. The task should apply the IK analysis to the received coordinates and convert those to a set of absolute joint angles. The received period and scheduling algorithm should not be processed in any way.

5.3. The conversion result should be sent alongside the period and scheduling algorithm to the Motion Control task.

6. There should be means of collecting runtime scheduling statistics.

6.1. The statistics should include cycles executed, deadline misses and the length of overruns in the deadline misses.

7. There should be an Input task to listen for commands from UART or USB.
   7.1. The commands should be formatted as JSON.
   7.2. The commands should have one of the following types and respective arguments as part of the JSON payload:
      a. **EMERGENCY_STOP** to stop the movement immediately.
      b. **SET_HOME** to reset all motors position to 0.
      c. **SET_RESOLUTION** to set all motors resolution.
         - option[0]: resolution value from range [2, 4, 8, 16, 32].
      d. **GET_STATISTICS** to collect runtime statistics.
      e. **MOVE_ANGLES** to move to a given position specified by absolute joint angles.
         - period: the movement period, in milliseconds.
         - values[0-5]: base, A, B, C, D and E joint angles, in degrees.
         - option[0]: scheduler to run the movement: 0 for RM, 1 for EDF, 2 for EDF + CBS.
      f. **MOVE_COORDINATES** to move to a given position specified by [X, Y, Z] coordinates and [Roll, Pitch, Yaw] orientation of the end effector.
         - period: the movement period, in milliseconds.
         - values[0-2]: X, Y and Z coordinates, in centimeters.
         - values[3-5]: Roll, Pitch and Yaw orientation, in degrees.
         - option[0]: scheduler to run the movement: 0 for RM, 1 for EDF, 2 for EDF + CBS.
   7.3. The received command should be executed immediately if settings-related and be forwarded to the Motion Control task in case of angles movement, and to the Kinematics task in case of coordinates movement.

These requirements were designed having two major goals in mind: making broad use of KARVEL-supported kernel mechanisms and enabling a solid base for the robot control. The system resulting from these requirements is one where all core functions are segmented in self-contained tasks, and which all communicate with one another asynchronously through message queues.

7.2.2 Architecture

The requirements outlined in Section 7.2.1 led to the development of the system illustrated in Figure 7.2. In terms of implementation, every single component - tasks, queues, barriers and

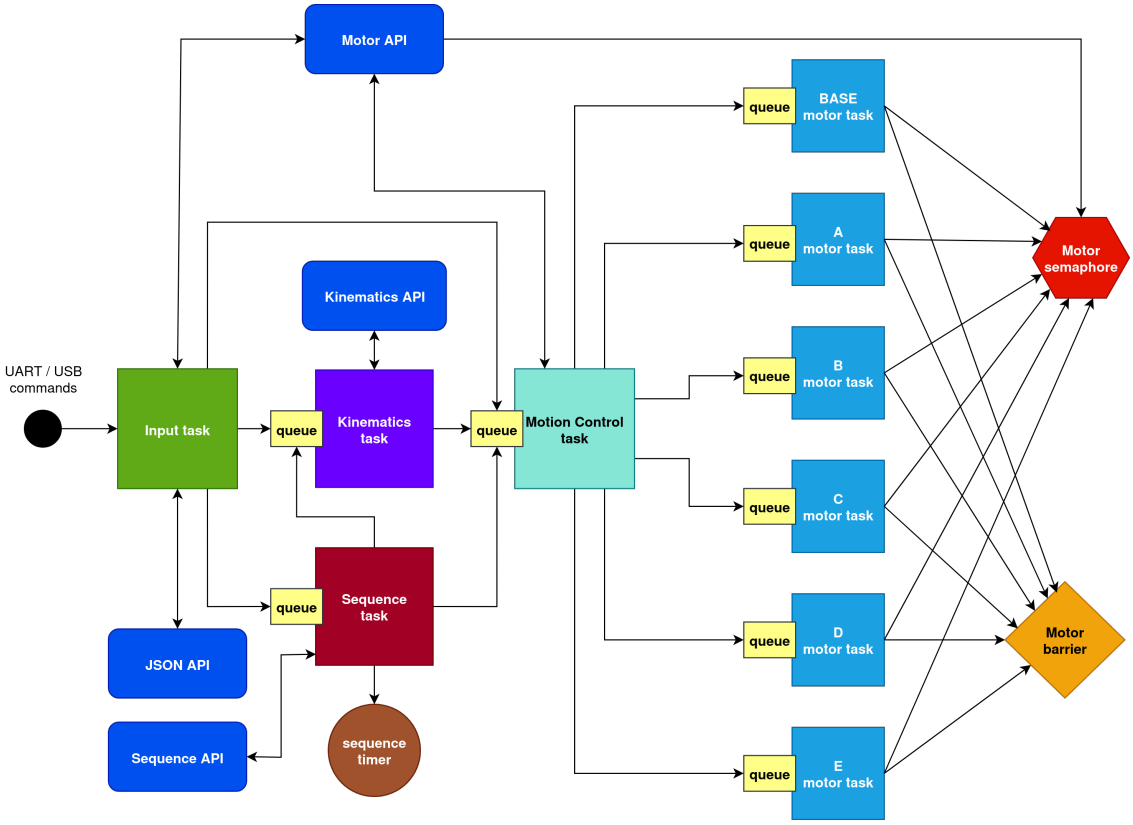


Figure 7.2: Test application architecture.

timers - was developed using the KARVEL API, and as it will be discussed in detail in Chapter 8, the resulting application managed to work as expected. It's interesting to note the whole system is designed to work mostly in one single direction: the input task receives commands, forwards them to another task or API responsible for processing the command, and the process continues until it eventually reaches the motors. There is no flow of information in the opposite direction as one would expect in a fully-fledged firmware, and the responses provided by the APIs to the Input task for the commands it receives are kept to a minimum.

This was all made on purpose, as the goal of this application was not to substitute the original firmware but rather to validate the KARVEL port to Zephyr in a more realistic scenario, also providing comparisons across the implemented scheduling algorithms. As such, the illustrated architecture is already sufficiently complex and able to demonstrate the intended analysis, therefore being considered adequate for this Thesis' scope. The following Sections provide an overview of the tasks and APIs.

Application-level APIs

The application development phase quickly saw the need for an extra set of libraries to manage application-level work. This is because KARVEL provides the necessary interface with the kernel but makes no assumptions regarding peripherals, for example, which are themselves necessary to run the stepper motor drivers and read packets incoming from the UART port.

The Motor API, for example, was one of these application libraries, and was developed specifically to provide all necessary control and configuration tools related to motors. In a similar approach, every work related to JSON parsing was implemented in a JSON API, and every function applied to the kinematics analysis was included in the Kinematics API. There were more libraries implemented in the application; the overview of Figure 7.2 illustrates the most significant ones for conciseness reasons.

```
int32 is_angle_within_motor_limits(stepper_t *motor, float angle)
{
    if((angle < motor->min_angle) || (angle > motor->max_angle)){
        TLCH_Log(LOG_ERROR, "Motor '%s' can't reach angle %.2f", ...);
        return SYSTEM_UNSUCCESS;
    }
    return SYSTEM_SUCCESS;
}

int32 reset_all_stepper_motors(void)
{
    int32 status = SYSTEM_UNSUCCESS;
    TLCH_SemaphoreTake(motor_semaphore);

    if(motors_in_use == 0){
        for(int i = 0; i < 6; i++){
            motors[i].angle = 0.0f;
            motors[i].step_count = 0;
        }
        status = SYSTEM_SUCCESS;
    }

    TLCH_SemaphoreGive(motor_semaphore);

    if(status == SYSTEM_SUCCESS){
        TLCH_Log(LOG_INFO, "robot is home.\n");
    } else {
        TLCH_Log(LOG_ERROR, "Failed to reset home: robot is moving\n");
    }
    return status;
}
```

Listing 7.2: a few Motor API methods.

Listing 7.2 shows two of the many functions developed for the Motor API, and noticeably they follow the same convention as KARVEL by returning either `SYSTEM_SUCCESS` or `SYSTEM_UNSUCCESS` depending on the operation result.

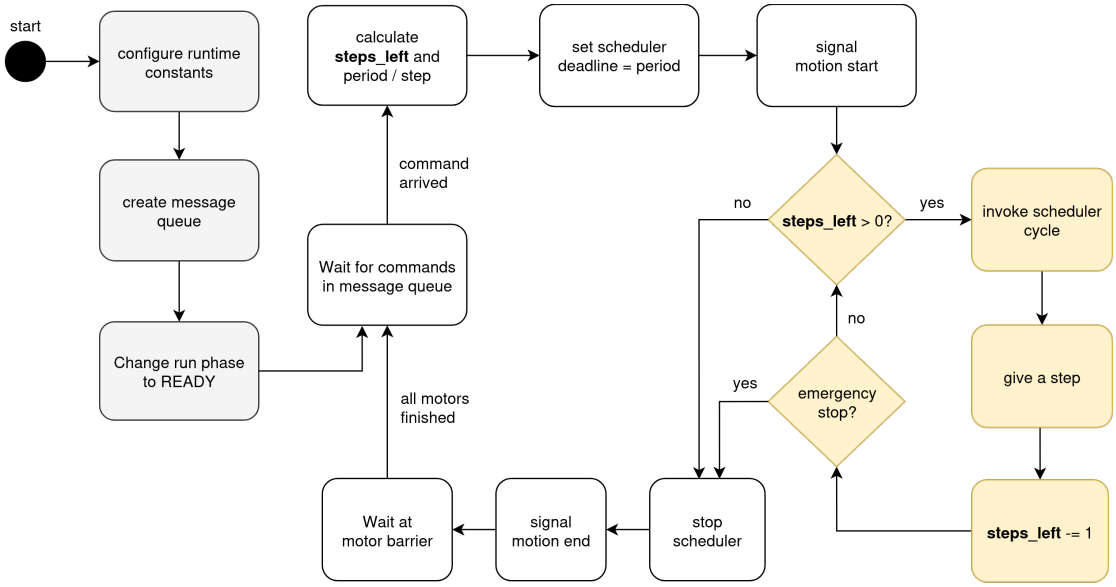


Figure 7.3: Motor task.

Motor task

The most fundamental task in the whole application is the Motor task, of which 6 copies (one for each motor) work concurrently in the system as viewable in Figure 7.2. Each copy has a different setup phase (where the motor GPIO pins and runtime constants such as minimum and maximum joint angles, gearbox reduction and so forth, are declared) but they all share the exact same core routine, which executes forever once the setup is done: wait for a movement command to come, then set the scheduler, then execute the motion. The motion itself is an inner loop which consists of `steps_left` cycles, each with a period per step of `total_period / steps_left`, all regulated by the chosen scheduler's cycle function (`TLCH_RateMonotonicPeriod` for RM and `TLCH_EDFDeadlineStart` for EDF and CBS)[3]. The inner loop is the main point of interest for our analysis, as it is what executes periodically under one of the introduced scheduling algorithms.

Figure 7.3 illustrates the execution flow of the task, where the gray-colored steps on the left side are the setup phase and the yellow-colored steps on the right side are the inner motion execution itself. After each cycle, the emergency flag is verified: if set, it means the robot has been commanded to stop immediately (i.e. `EMERGENCY_STOP` has been issued by the user). Original plans of an emergency stop procedure involved sending a STOP command to the Motor task through its message queue, which would reduce the polling overhead and centralize the means of interacting with the task. Moreover, this strategy could be realized by sending the STOP command with the `TLCH_MsgQueueSendUrgent` API, which would increase the usage of KARVEL APIs. However, this idea was discarded after considering that since the motor only checks the message queue *between* movement commands, it would obviously not stop *during*

[3] It's worth noting that the period per step calculation is fundamentally different from the original algorithm (refer to Section 7.1.1 for details) since it depends on the *total period* length and not on a constant value. This difference is enough to make the new application *way* more suitable for schedulability analysis.

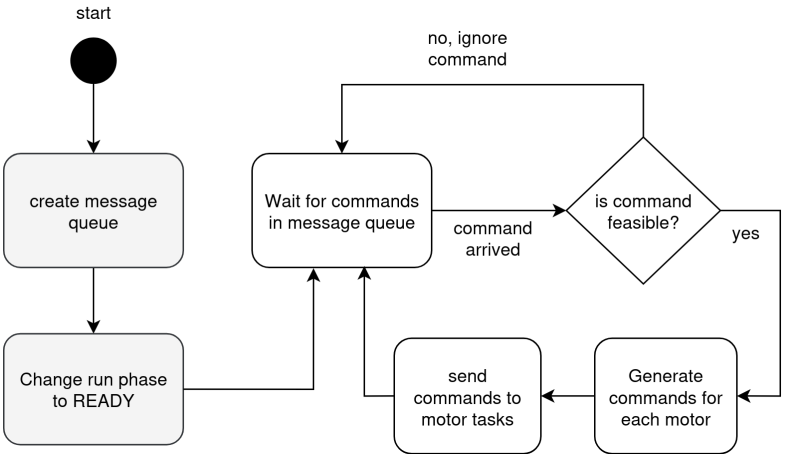


Figure 7.4: Motion Control task.

movement commands, which was simply not a fast enough response for a halting operation. As a result, the Motor API function invoked by the emergency procedure simply takes the Motor semaphore and raises a flag to signal an immediate stop must take place.

Motion Control task

This task performed a rather simple routine and always worked on demand, not being very suitable for RM scheduling but a good candidate for receiving a CBS budget when such scheduler was selected (we recall the CBS provides CPU bandwidth control without demanding a periodic execution for the task). For the test application developed, its reccurent work consisted of simply waiting for a movement command to turn up on its message queue, then check if the command is feasible (i.e. if the absolute angles requested are within the joint limits), then break down the command into 6 single-motor commands before forwarding each one to its respective Motor task.

As a side note, this task main purpose is to provide an unified way of sending commands to the motors, effectively acting as a proxy between the Input task and the Motor tasks. In possible future works derived from the current one, however, this task could easily grow in complexity by introducing features such as non-linear acceleration (e.g. by incorporating the Proportional-Integrative-Derivative (PID) algorithm), adaptative motor resolution setting, and on-the-fly motion command, which would expand the control possibilites of the robot to enable joystick-based actuation. For the scope of this Thesis, however, this task is limited to performing base checks and forwarding commands. The current execution flow is viewable in Figure 7.4.

Kinematics task

This task has a pretty similar working principle as the Motion control task. As illustrated in Figure 7.5, its current procedure is simply to create a message queue in the setup phase and

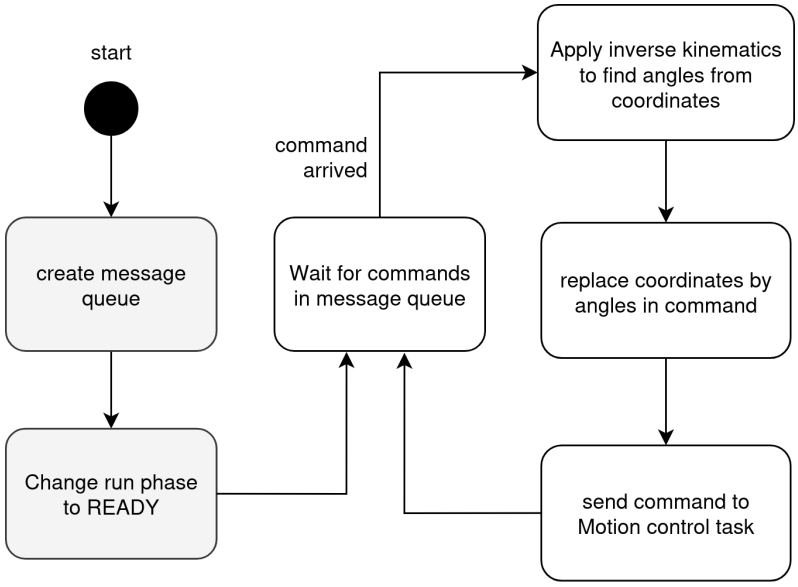


Figure 7.5: Kinematics task.

indefinitely wait for commands coming from it afterwards. A command in this case will feature cartesian (XYZ + RPY) coordinates, and it's up for this task to invoke the Kinematics API and derive the corresponding absolute joint angles from it. The task then replaces the coordinates in the command with the calculated angles, and forwards it to the Motion control task alongside the chosen period and scheduler.

One could argue that since this task only performs work that will be forwarded to the Motion control task, it could as well be blended into it, thus enlarging the scope of the motion control and leading to one less task in the system. However that would be a bad coding practice if one considers the separation of concerns principle: finding the joint angles from coordinates is not part of the movement coordination per se, only necessary to translate a command into a format that is understandable by the motion control task. Another reason is that there is still room for a Forward Kinematics (FK) implementation in future work, and while this has nothing to do with the motion control in any way, it could be easily integrated into the Kinematics task.

7.2.3 Input task

Zephyr has three modes of UART interactions - Polling, Interrupt-Driven and Async - where the first one is simpler to use and the latter is considered more efficient. The Input task leverages the Polling API to listen for both UART and USB packets, and conceptually performs a similar event-driven approach as the other application tasks discussed in previous sections. However, unlike the default message queue behavior in KARVEL, the Polling UART mode is non-blocking by design. Thus, the strategy for making it continuously listen for packets was to make it repeatedly check for packets, and consider a message to have been completely received by the time a newline (`\n`) or carriage return (`\r`) character arrives (or both). At this moment, the resulting buffer is then handled to the JSON API in order to extract the desired payload, and

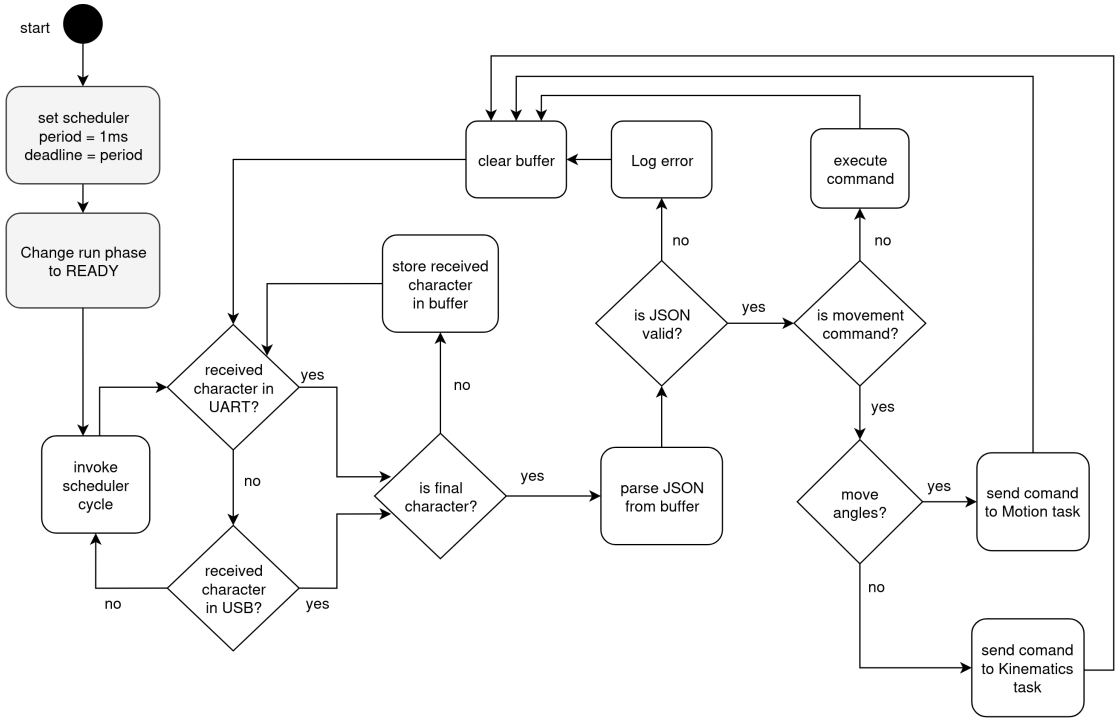


Figure 7.6: Input task.

if not a command that requires further processing (i.e. not related to movement), execute it in place before resuming listening for new packets.

Figure 7.6 illustrates the execution flow performed by this task. The task is statically configured as having a 1ms period to provoke minimal "idle" impact on a system where the most demanding tasks take microseconds as periods. As for the scheduler, it can be configured to run with either a RM, EDF or CBS regulation, but this setting must be defined statically[4]. It's interesting to note, however, that once a packet shows up in either UART or USB, the task will recursively look for any subsequent packets instead of necessarily waiting for the next cycle to get the next packet. This significantly speeds up responsiveness of the task, but as a side effect the execution times of each cycle becomes subject to high variability, which is aggravated by the JSON parsing that might follow when the message is completly delivered. As a consequence, this is one sort of task that would likely take great benefit from the bandwidth reservation feature of the CBS scheduler.

Sequence task

A sequence in the scope of PegasoV3 is a set of positions (either specified by joint angles or cartesian coordinates) that the robot goes to one right after the other. It is the foundation of

[4] it wouldn't be effective to configure the scheduler on-the-fly (as already done for the motor tasks) anyway because this setting would come *during* the task execution, by the time it's parsing the received JSON command. This would lead to the configured scheduler only being actually applied from the following message onwards, which then might as well receive another scheduler setting, which will again only be applied in the next packet and so on.

complex trajectories, which are themselves essential to nearly any application that requires a robot in the first place. In the original robot firmware, sequences could be dynamically created by positioning a robot in a given spot and inserting that position in the sequence. Then, if desired, the sequence could be saved in a non-volatile storage unit such as an SD card for later usage. Sequences in the Micropython application would only contain the positions; the speed to move across them was defined by the robot speed at the time.

The sequence task for the newly developed firmware, however, was not originally listed in the requirements of Section 7.2.1; the need for it was identified during the implementation of the Input task. The original plans for sequential movement execution was to generate those in the web interface provided by the ESP-01s WiFi module (which was the original control interface, as described in Section 7.1.1), however the Zephyr implementation had persistent problems in making the Rapsberry Pi Pico microcontroller communicate with the module[5]. As a solution couldn't be found in time, we opted to proceed with an alternative strategy of creating a task (and the corresponding API) specifically for the sequence management and execution. The requirements developed for this effect are given below.

1. There should be a Sequence task to run movement sequences.
    1.1. Just like the other tasks in the application, the Sequence task should have a message queue to receive commands.
    1.2. The sequence task should run sequences identified by an ID.
    1.3. If a cycle count were not specified, the sequence should run indefinitely.
    1.4. Each position in the sequence should have a default specification of speed and scheduler. Optionally, the user should be able to specify custom values for these properties, in which case the default values should be overriden.
    1.5. For the purpose of schedulability analysis, a speed increase factor should be also optionally specifiable by the user. If so, at every new cycle the speed should increase by the increase factor.
    1.6. The sequence command should be either to start a sequence, gracefully stop a sequence (return to home position after ending the current cycle), or emergency stop a sequence (halt the movement immediately).

And as a consequence of these requirements, the following commands were added to the Input task list of accepted commands.

- **RUN_SEQUENCE** to run a sequence.
    - option[0]: ID of the sequence to run.
    - option[1]: cycles to execute for the sequence.

[5] More specifically, both devices generated the expected responses when tested separately but did not seem to send each other packets when connected directly.

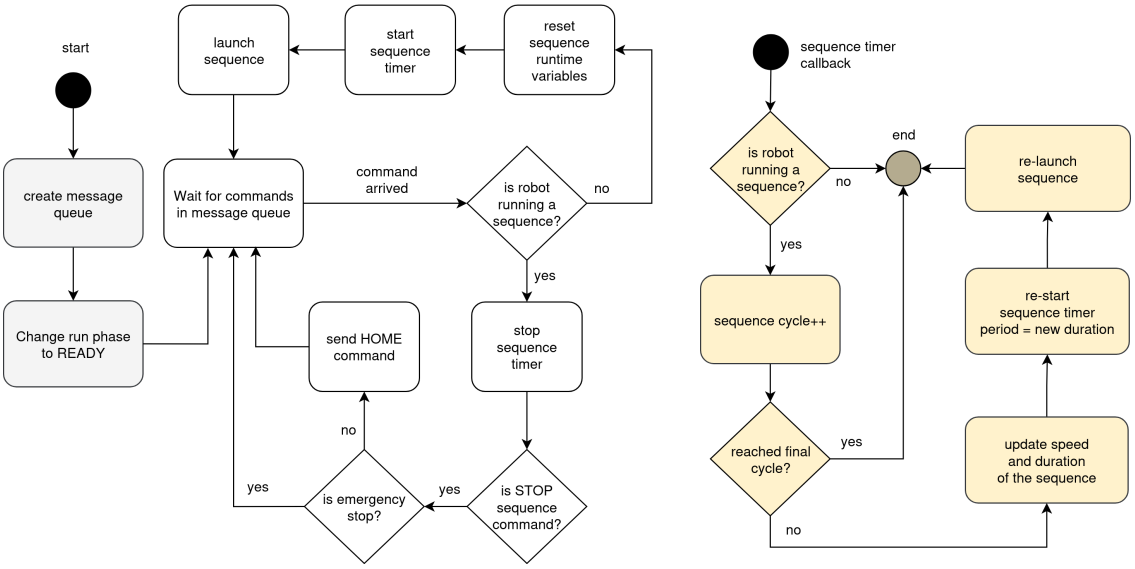


Figure 7.7: Sequence task and timer.

  - option[2]: speed percentage of the default sequence values.
  - option[3]: speed increase percentage for each new cycle.

- **STOP_SEQUENCE** to gracefully stop a sequence.

Although the resulting implementation was made with extensibility in mind, no requirement or command was delineated to dynamically add or remove positions from a sequence as it happened in the original firmware. This was due to the fact that the non-volatile storage was outside the scheduling scope of this Thesis, and a task dedicated for managing it wouldn't happen frequently enough to justify analysing its impact in schedulability anyway. As a consequence, all sequences used for the tests were declared statically as arrays and identifiable by a numeric ID. Also note that according to the established requirements, each element in a sequence would not only contain the desired *position* to reach but also *which scheduler* to use in the movement and *how much time* to take, in milliseconds.

The execution flow for the implemented task, as well as the timer it ended up using, is viewable in Figure 7.7. While the grey steps at the left side denote the setup phase, the yellow flow at the right side refers to the timer callback function. The task working principle is described as follows. First, a sequence would be *launched* by sending all of its positions immediately (one right after the other) to either the Kinematics or Motion control task, depending on the position type. Then, the total sequence execution time was derived by simply making a sum of all position periods. This total execution time estimation would then be set as the timer period right before launching the sequence itself. If the user specifies the optional speed factor when issuing the command, the position periods (and consequently the timer period) would all be pre-adjusted by this factor beforehand.

Then, by the moment the timer callback was invoked, it would check if the sequence should

run for yet another cycle. If so, the sequence would be relaunched and the timer re-started. If the user specified a speed increase factor (i.e. acceleration), the new periods would be iterated once again. For example, if a sequence with a default estimated duration of 20000ms were commanded to run 3 times with a speed factor of 120% and a speed increase factor of 30%, then the first cycle would run at 120% of the default speed (resulting in a total period of 16666ms) and the following ones would run at 150% (period of 13333ms) and 180% (period of 11111ms). The commanded angles/coordinates would remain exactly the same, only the time to run the sequence would change - making this task an important tool for performing scheduling benchmarks.

7.3 Additional contributions to KARVEL

A few extra contributions were made to KARVEL in order to properly implement the robot application using its API. Most notably, both EDF and RM APIs were extended with methods to operate with microsecond resolution for both periods and deadlines (rather than being forced to always work in terms of milliseconds). This was necessary because the motors of the robot are coupled with gearboxes that greatly elevate the required steps to take for any given angle, which as a consequence makes step periods frequently drop to (way) less than a millisecond. For instance, if joint C and its 25:1 reduction were commanded to move 60 degrees in reasonable 5 seconds, a step would have to be given every 0.1875 milliseconds[6] to fulfill it, which is clearly impossible if the minimum configurable period is 1 millisecond. The introduced API is `TLCH_RateMonotonicSetTimeUnit` or `TLCH_EDFSetTimeUnit` depending on the scheduler, and it is completely optional. If the application does not specify a time unit, the API defaults to milliseconds.

Another contribution made because of this test application was enabling the collection of scheduling statistics: as requirement 6 in Section 7.2.1 specifies, there should be some way of collecting the amount of cycles executed, deadline misses and by how much the deadline (or period) was missed. Although the first metric was easily implementable at application level, within the motor tasks, we found that it would be much simpler to track the latter two directly within the periodic functions of each scheduler; thus, all three were moved towards the abstraction layer. The implementation consisted of simply creating per-scheduler-manager variables that would be automatically updated when the concerned event happened, and to extend the API with a method that would copy it elsewhere before resetting. Listing 7.3 illustrates the main modifications introduced in the EDF API for the collection and retrieval of runtime data. The RM API received similar changes.

```
1 // periodic EDF function
2 int32 TLCH_EDFDeadlineStart(uint32 deadline, ...){
```

[6] considering a microstep resolution of 32 and the NEMA 17 default 200 steps per revolution, a full 360-degree turn of joint C takes $200 * 25 * 32 = 160000$ steps. Thus, a 60-degree turn takes 26666 steps. For the given 5-second period, we have $5/26666 = 0.0001875$ seconds, or 0.1875 milliseconds.

```
    ...
    if(periodic){
        edfHandler->statistics.cycles++;
        ...
    }
    if(missed_period){
        ...
        int64 missed = (now - previous_deadline);
        edfHandler->statistics.deadline_miss_count++;
        edfHandler->statistics.deadline_miss_amount += missed;
    }
}

// introduced API for collecting statistics
int32 TLCH_EDFGetStatistics(TLCH_SchedulerStatistics_t *statistics, ...){
    ...
    int64 cycles = edfHandler->statistics.cycles;
    int64 miss_count = edfHandler->statistics.deadline_miss_count;
    int64 miss_amount = edfHandler->statistics.deadline_miss_amount;

    statistics->cycles = cycles;
    statistics->deadline_miss_count = miss_count;
    statistics->deadline_miss_amount = miss_amount;

    edfHandler->statistics.cycles = 0;
    edfHandler->statistics.deadline_miss_count = 0;
    edfHandler->statistics.deadline_miss_amount = 0;
    return SYSTEM_SUCCESS;
}
```

Listing 7.3: scheduling statistics collection method.

When invoked, the introduced `TLCH_EDFGetStatistics` method simply copies its own data to an external variable provided by the application and resets it afterwards. Note it would be impractical to keep records of individual overruns, so the API simply accumulates them as they are detected; still, by knowning the total overrun amount and the number of cycles with deadline misses, one could easily extract the average value and infer the task overall performance.

As a final note, this introduced statistics API was found considerably more useful to the application than the original deadline miss detection mechanism of logging an error message and providing binary return values of `SYSTEM_SUCCESS` or `SYSTEM_UNSUCCESS`. By knowing a the *number* of deadline misses and by *how much* was a miss, the application could react to it in a more precise way (for example, elongating the task's deadline if it found out more than 20% of the cycles executed had missed deadlines). Moreover, just incrementing integer variables likely brings less overhead than throwing error messages, even if such operation has the actual string processing offloaded to a background task.

Chapter 8

Results

We now proceed to show the tests developed for the KARVEL-based application described in Section 7.2, as well as discuss the obtained results.

8.1 Single Motor Tests

Before delving into complex movements and their performance, it was deemed interesting to first analyze the fundamental executive unit - the Motor task - to infer its base scheduling properties. As such, the very first test aimed at discovering the approximate execution times of the motor tasks across various commanded periods, in a way to observe if changing the task's frequency had any impact on how long it would take to move the stepper motors. The results of this tests would likely provide insights on how much budget should be provided for the Motor tasks when applying the CBS scheduler.

```
static inline void step(stepper_t *motor)
{
    uint64_t start = k_cycle_get_64();
    gpio_pin_set_dt(&motor->step_pin, 1);
    k_busy_wait(4);
    gpio_pin_set_dt(&motor->step_pin, 0);
    motor->step_count += motor->direction;
    motor->execution_time += (k_cycle_get_64() - start);
}
```

Listing 8.1: Core motor task routine

Listing 8.1 illustrates the function responsible for actually sending a step pulse to the stepper motor drivers, and which is invoked at every cycle of the periodic segment of Figure 7.3 (the yellow blocks). The step simply consists of toggling the related GPIO high and low, and updating runtime variables afterwards; the `k_busy_wait` method is invoked to ensure a minimum pulse width of 4 microseconds as required by the DRV8825 motor driver. For the purposes of this test, an extra `execution_time` variable was created in the task control block, acting basically as an accumulator of how many hardware cycles have elapsed since the function start. Thus,

	300us		100us		50us		25us		10us	
	RM	EDF	RM	EDF	RM	EDF	RM	EDF	RM	EDF
BASE	0.0% 6.81us	0.0% 6.19us	0.03% 6.84us	0.03% 6.00us	0.01% 6.74us	0.01% 5.92us	18.93% 5.65us	2.40% 7.53us	29.09% 5.83us	48.34% 5.95us
A	0.0% 6.71us	0.0% 6.19us	0.0% 6.51us	0.0% 6.26us	0.01% 5.51us	0.01% 7.71us	16.99% 8.49us	2.49% 5.80us	29.20% 6.22us	48.64% 5.97us
B	0.0% 6.75us	0.0% 6.24us	0.01% 6.25us	0.01% 6.50us	0.02% 7.08us	0.02% 6.20us	17.83% 6.19us	2.56% 6.45us	29.01% 6.20us	48.30% 5.95us
C	0.0% 6.80us	0.0% 6.28us	0.0% 6.36us	0.0% 6.61us	0.03% 6.58us	0.02% 6.08us	10.38% 7.69us	2.39% 5.12us	30.22% 6.57us	48.29% 5.71us
D	0.0% 6.92us	0.0% 6.05us	0.01% 6.77us	0.0% 5.93us	0.02% 6.68us	0.02% 5.97us	10.60% 6.55us	2.50% 6.03us	29.97% 6.30us	48.30% 5.91us
E	0.0% 6.91us	0.0% 6.19us	0.0% 6.90us	0.01% 5.93us	0.02% 6.69us	0.02% 5.98us	10.36% 6.55us	2.39% 6.27us	29.83% 6.26us	48.29% 5.91us

Table 8.1: Average deadline miss percentage and execution times for single motor movements on different schedulers and period lengths.

after n cycles one might divide this measure by n and obtain the average task execution time per cycle. It's worth noting that this data collection method does not account for preemptions, which means that for accurate results each analyzed motor task should run alone in the movement.

Thus, for this test execution a series of single-motor movements were issued to each one of the available Motor tasks (six in total, one for each joint). Each command was tailored to yield step periods of 300, 100, 50, 25 and 10 microseconds and a minimum of 5000 steps per movement. For every period, at least 10 commands were generated, 5 for each scheduler (RM and EDF) - ultimately leading to a total population of at least 25000 cycles per combination of period, scheduler and joint. Finally, after each group of commands were sent and executed, statistics were collected to note the accumulated execution time, total cycles ran, total deadlines missed and accumulated overrun per miss.

Table 8.1 showcases the resulting percentage of deadline misses per total of cycles, and the execution times computed for the step function of Listing 8.1. A few observations might be taken from the obtained values: first, it's intuitive to think that since the measured function is executed the exact same way regardless of the chosen scheduler, this setting should not cast any influence in the task execution at all, and even if it did for some reason, Rate Monotonic (RM) should yield the least execution times due to having less cyclic operations associated with it, and therefore cause less overhead than EDF. However, the observed outcome is precisely the opposite: for almost all combinations of joint and period, the execution times are lower for EDF scheduling than they are for RM scheduling. This might be explained by some sort of compiler optimization during the build phase, but the precise causes deserve further investigation to be clarified.

Second, the execution times for each combination of scheduler and period are consistently close between different joints, which is expected since the only theoretical difference between

them is the GPIO being controlled by each task. Aside from that, there is no clear pattern of any joint taking more microseconds to run than others, which indicates that no special treatment needs to be given to any particular motor (for example, providing more CBS budget than others).

Lastly, it's interesting to note that deadlines are missed even if the CPU usage of the tasks never exceeds 100% for any motor or period. The BASE joint, for example, starts missing deadlines with periods as long as 100 microseconds which, considering the observed execution time, would yield a CPU usage of just 6.84%. It's unlikely that the remaining tasks in the system (Kinematics, Input, Motion control, etc), which are event-driven and were not altered or deactivated for these tests, suddenly demanded the remaining CPU capacity at any given time, however the fact that the deadline miss ratio consistently grows to over 48% for all motors on the most intensive 10us period under EDF suggests two things:

- the worst-case execution times are considerably higher than the average, which would explain the misses on longer periods; and
- since the deadline miss detection is part of the scheduling algorithm API and that the task's execution time itself actually becomes smaller (on average) as the period descends, the RM and EDF API overheads may be significative factors for the deadline misses as well.

The EDF expectedly causes more misses on 10us cycles, but it falls behind RM on the more relaxed 25us period. Considering this behavior was noted across the motor tasks, it's likely no coincidence. It should be noted, however, that the periods chosen for this test were selected based only on the intention of assessing the scheduling behavior of the concerned tasks; in practice, the PegasoV3 robot used as a target for the test application is a real machine, and therefore subject to the laws of physics. It rarely runs with periods below 40us for any given joint to avoid the mechanical stress (or simply stepper motor skipping issues) caused by inertia. As such, the results for periods of 50us and higher are more relevant in practice, and as viewable in Table 8.1, the deadline miss ratio is marginal for both schedulers on these ranges.

8.2 Single Interpolation Tests

If the first test focused on analyzing the motor tasks when running isolated from each other in an attempt to observe the individual characteristics of each one, the next focused on interpolations[1] to verify how the system behaves on a more complex and realistic scenario. Specifically, the goal was to observe how quickly, and by how much, does each scheduler starts missing deadlines when one same movement progressively increases its speed. For the evalution, three interpolations were defined - all starting from the robot's HOME position:

[1] An *interpolation* is a movement where only the start and finish points matter - the specific trajectory taken between them is irrelevant, so the robot may move in any arbitrary manner to fulfill the command. Usually, that means generating a homogeneous period for each of the necessary joints in a way they all start and finish at the same moment.

- **A:** a relatively short interpolation with destination coordinates of [x, y, z, roll, pitch, yaw] = [150, -100, 350, 0, 0, 0] and linear distance from the HOME position of 127.46mm. The joints need to run 1230002 steps in total to fulfill this movement, being distributed as follows: BASE joint: 5641 steps (4.50% of total); A joint: 1246 steps (1.01%); B joint: 15185 steps (12,34%); C joint: 30528 steps (24.82%); D joint: 44239 steps (35.96%); E joint: 26163 steps (21.27%).
- **B:** a movement with destination coordinates of [x, y, z, roll, pitch, yaw] = [380, 100, 200, 0, 0, 0] and a corresponding distance of 274.94mm from the robot's HOME position. It is therefore more than double the length of interpolation A and, at the total step count of 228373, more than 100000 additional steps need to be given by the robot joints. It's noticeable that the *distribution* of these steps is also radically different, where joint A alone responds for over 50% of the total amount: BASE joint: 2104 steps (0.92%); A joint: 142823 steps (62.54%); B joint: 27206 steps (11,91%); C joint: 14975 steps (6.55%); D joint: 28117 steps (12.31%); E joint: 13148 steps (5.76%).
- **C:** the longest of all three interpolations, with a total step count of 330762 steps and a final destination of [x, y, z, roll, pitch, yaw] = [100, 300, 100, -90, -45, 0]. It's located 442.91mm apart from the HOME position the robot will depart from, and purposefully comes with the additional specifications of roll and pitch angles to increase the movement complexity. The steps are distributed between the joints in the following manner: BASE joint: 9705 steps (2.93%); A joint: 150409 steps (45.47%); B joint: 12727 steps (3.85%); C joint: 50901 steps (15.39%); D joint: 43986 steps (13.30%); E joint: 63034 steps (19.06%).

These trajectories were designed with different lengths, complexities and step distributions on purpose, so one could observe if these factors also provoked differences in scheduling behavior beyond just the notion of total CPU usage. The common ground for the analysis was the interpolation duration: all three movements were executed with increasingly smaller periods, in a range that went from 14 seconds down to just 2 seconds. Naturally, the interpolation A would reach lower CPU usages than B or C for the same periods, as it needs to give less steps, in which case the other conditions such as the load balancing between motors should be a likely cause for scheduling differences, if any.

At least three runs were executed for every duration applied to the aformentioned movements. The data collected came from the statistics required in Section 7.2.1 - namely the total cycles executed, total cycles with deadline misses and average overrun per miss - alongside the movement execution time, which is informed at the end of every robot interpolation by default. Figure 8.1 illustrates the USB output when the statistics are requested via the respective command. Note it gives per-scheduler data for each of the joints, then combines everything related to each scheduler into the last two lines. This combined data was the primary source of analysis for the interpolations, however as Figure 8.1 shows it was equally possible to go further and gather per-motor data, if needed.

```
KARVEL: Motor 'BASE': RM:  missed 12/9704 cycles (0.12%).      Avg. overrun/miss: 240445.58 ticks or 480891.17us.
KARVEL: Motor 'BASE': EDF: missed 4438/9704 cycles (45.73%).   Avg. overrun/miss: 10.48 ticks or 20.95us.

KARVEL: Motor 'A': RM:  missed 7462/150409 cycles (4.96%).     Avg. overrun/miss: 4.38 ticks or 8.76us.
KARVEL: Motor 'A': EDF: missed 60457/150409 cycles (40.20%).   Avg. overrun/miss: 9.14 ticks or 18.28us.

KARVEL: Motor 'B': RM:  missed 302/12726 cycles (2.37%).       Avg. overrun/miss: 9295.78 ticks or 18591.56us.
KARVEL: Motor 'B': EDF: missed 7106/12726 cycles (55.84%).     Avg. overrun/miss: 10.50 ticks or 20.99us.

KARVEL: Motor 'C': RM:  missed 16996/50900 cycles (33.39%).    Avg. overrun/miss: 32.27 ticks or 64.55us.
KARVEL: Motor 'C': EDF: missed 20836/50900 cycles (40.94%).    Avg. overrun/miss: 9.36 ticks or 18.72us.

KARVEL: Motor 'D': RM:  missed 7373/43985 cycles (16.76%).     Avg. overrun/miss: 308.10 ticks or 616.20us.
KARVEL: Motor 'D': EDF: missed 17540/43985 cycles (39.88%).    Avg. overrun/miss: 10.43 ticks or 20.85us.

KARVEL: Motor 'E': RM:  missed 2645/63033 cycles (4.20%).      Avg. overrun/miss: 4.46 ticks or 8.92us.
KARVEL: Motor 'E': EDF: missed 25671/63033 cycles (40.73%).    Avg. overrun/miss: 9.10 ticks or 18.21us.

KARVEL: RM:  in total, missed 34790/330757 cycles (10.52%)     average overrun/miss of 245.97 ticks or 491.94us.
KARVEL: EDF: in total, missed 136048/330757 cycles (41.13%)    average overrun/miss of 9.45 ticks or 18.90us.
```

Figure 8.1: Statistics collected after interpolations.

The results collected from all interpolations are showcased in the charts of Figure 8.2. The charts are organized as follows: the leftmost ones are in respect to interpolation A, whereas the center ones are about interpolation B and the rightmost showcase the results for interpolation C. When looking by rows, the top row (bar charts) shows the effective time taken to run the interpolation versus the expected value passed in the command. The center row shows the estimated CPU usage[2] of the movement (red line) per commanded duration, and the observed percentage of deadline misses versus the total cycles executed. Lastly, the bottom row shows the average overruns measured per missed cycle, in microseconds.

Let P be the commanded period which varied from 14 down to 2 seconds. By looking at the charts in the first row, one can see that the robot fulfills the specified P with close precision on the upper range of $P = 14$ to $P = 10$ seconds. However, when $P = 8$ is commanded on interpolation C, the robot finishes in 8.124 seconds (1.55% more) for RM and 8.015 seconds (0.18% more) for EDF. This *breaking point*, where the interpolation as a whole begins showing delays, corresponds to an estimated CPU usage (in the chart right below) of around 25%. Coincidentally, that is the same CPU usage level at which interpolations A and B start showing delays as well: for A that's when $P = 3$, and for B that's when $P = 6$. On A and B, however, RM poses a small edge over EDF on these spots.

Curiously, the scenario changes dramatically in favor of EDF from this processor utilization onwards. When $P = 2$ for interpolation A, the estimated CPU usage U is at 37.87% and the execution with RM is finished at an average of 3.957 seconds (97.85% more), whereas EDF manages to finish in 2.380 seconds (19% more). For interpolation B, U is 37.74% at $P = 4$, and at this point RM finishes considerably later than EDF once again, obtaining 6.293 seconds (57.32% more) versus 4.635 seconds (15.87% more). The same pattern goes for interpolation C, where the estimated usage of 33.96% comes when $P = 6$ seconds: RM takes 11.767

[2] for calculating the usage, we considered an execution time of 6.157 microseconds for motor task, taken by averaging the measurements of Table 8.1. It's likely that the actual values differ in practice due to the reasons discussed in Section 8.1, so this usage is more of a reference than an actual value.

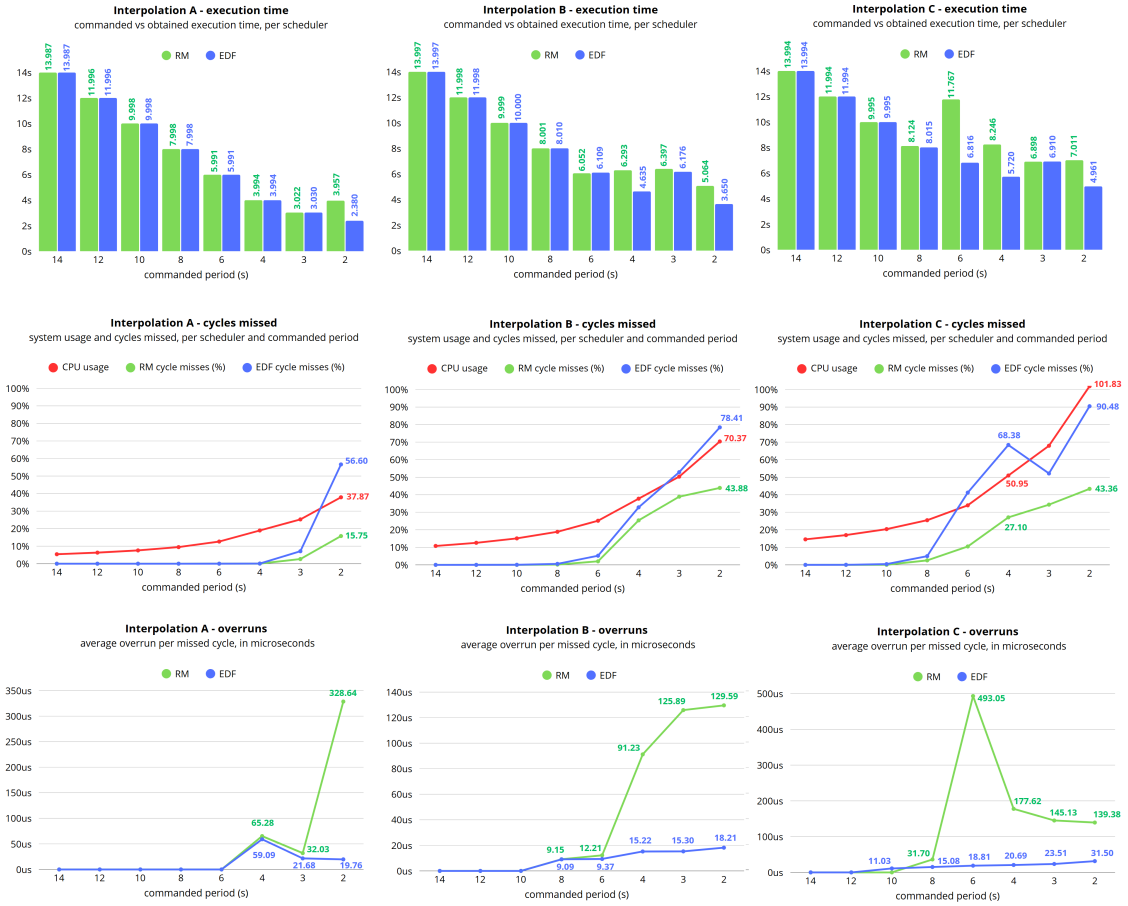


Figure 8.2: Interpolation test results.

seconds (96.12% more), or nearly twice the time provided, to perform the motion, whereas EDF manages to finish with a much lighter penalty of 13.60% (at 6.816 seconds). The performance differences keep varying further as P decreases, however it's widely noticeable that EDF delivers durations closer to the commanded period P than RM.

With that said, it's puzzling to note in the next charts that EDF consistently missed *more* deadlines than RM at a task cycle level. The charts at the center row illustrate the average percentage of cycles missed by the motor tasks when executing the movements for each commanded P, and EDF clearly holds the larger percentages at all points and interpolations. The miss rates rises as high as 56.60% for intepolation A, 78.41% at interpolation B and remarkably 90.48% of the total cycles executed for interpolation C, whereas in RM the peaks for each interpolation are 15.75%, 43.88% and 43.36%, respectively. Judging by this metric, then, the EDF undeniably showcases worse performance than RM when looking at the individual task's execution outcomes. Still, when the interpolation execution times are analyzed as a whole, the EDF scheduled movements manage to yield less delays than RM; so what explains it?

The straightforward explanation lies in the bottom row of charts, which illustrate, on average, how large were the overruns for each of the deadline misses. By combining the information on center and bottom charts, one can easily notice that while RM produces less missed cycles

than EDF, it does produce higher overruns per miss to an extent that ultimately has a greater influence on the perceived loss of performance of the interpolations.

However, there is a deeper analysis to be made regarding the way EDF and RM behave when both are dealing with heavier loads than what they can handle. The theoretical analysis of scheduling algorithms traditionally focuses on the range of CPU usages from 0 to 100% and determining the feasibility of a given task set in this range. It doesn't make sense at first to analyze task sets which demand more than 100% of a single CPU simply because they will necessarily miss deadlines. As a consequence, little to no attention is drawn to the dynamics of scheduling algorithms when deadlines are *guaranteed* to be missed.

We've saw in the single-motor tests of Section 8.1 that motor tasks can miss deadlines even when running one at a time (i.e. in a configuration with presumably small U). That, allied with the fact KARVEL unit tests of Chapter 6 produced logically correct results for both RM and EDF (EDF specially, which were able to operate with no misses on U close to 100%) and that the motor tasks operate in the newly-introduced microsecond range (whereas KARVEL tests were built in milliseconds) lead to the hypothesis that the estimated CPU usage curves showcased on the charts of Figure 8.2 differ significantly from the worst-case scenario, and that it is very likely the system may often or entirely operate over 100% capacity for most commanded values of P. If only looking at the interpolation runs which finished on time, the results suggest that the actual U = 100% is achieved as far back where the displayed estimate is indicating 20%. However there is no way to precisely estimate how often the worst-case executions show up, or if they progress linearly as P decreases, just with the data displayed. A deeper analysis should be conducted in future work.

Assuming that the system is overloaded, it gets fairly easy to understang both why EDF has more deadline misses and RM has higher average overruns. As a static scheduling algorithm, motor tasks operating under RM will have one same priority order throughout the movement they are participating in. Naturally, that means the less frequent tasks will be preempted the most, and by looking at the example in Figure 6.15 it's easy to perceive that multiple missed deadlines in a row tend to accumulate overruns with time. Thus, in RM the missed cycles will be lower because they are bounded to the least frequent tasks, and each registered miss may have the accumulated amount of previously unattended cycles.

For EDF, however, priorities are by definition redefined at every new cycle of any given task. The lowest priority tasks change more often as well, and even the mid-priority ones may be subject to deadline misses. Thus, whereas in RM the overload effects tend to be concentrated in a subset of the tasks, in EDF they might be more evenly distributed and therefore lower in magnitude. This may cause an overall better system response, as noticed in execution times of Figure 8.2, but indeed means more tasks are missing deadlines. The choice of which scheduler performs better in this case is hard to do without taking into consideration the application it's in. For the case of robots such as PegasoV3, a properly coordinated movement between

joints is what makes it operate accordingly, so EDF seems better. However in a spacecraft a concentrated penalty on the lower priority tasks may be preffered.

8.2.1 Increasing the work factor

The aforementioned explanation for why the EDF runs seem to miss more deadlines than RM counterparts from a given P onward might make sense, however it would be interesting to have some results coinciding with the expected in theory as well. The fact that the original scheduling results obtained in the synthetic tests of Sections 6.3.1 and 6.2.5, where busy waits emulated execution times, were right on par with the theoretical outcomes of Figures 6.11 and 6.15, seem to indicate that the determinant factor for the high variability on the motor task cycles are in fact that the synthetic tests ran in terms of milliseconds, whereas the motor tasks need to operate in microseconds.

```
static inline void step(stepper_t *motor)
{
    uint64_t start = k_cycle_get_64();
    for(int i=0; i < WORK_FACTOR; i++){
        gpio_pin_set_dt(&motor->step_pin, 1);
    }
    k_busy_wait(4);
    gpio_pin_set_dt(&motor->step_pin, 0);
    motor->step_count += motor->direction;
    motor->execution_time += (k_cycle_get_64() - start);
}
```

Listing 8.2: Core motor task routine with work factor (WF) included

Thus, maybe the root cause for the observed overload might actually reside in the overheads associated with context switches, timers, message queue handling and logging system outputs, for example, which may no longer being negligible to the scheduling outcome. The easiest way to test this was to increase both the task execution time and period, so two changes were made in this regard. First, the stepper motor resolution, which was set at 32 for the previous tests, was reajusted to 8, which automatically reduced the steps necessary for any given angle by a factor or 4. This would therefore increase every period by the same factor. Second, a *work factor* (WF) was introduced in the core step routine performed by the motor tasks, as viewable in Listing 8.2. Basically, for every step taken the task would now turn the GPIO on WF times before proceeding instead of only one. In logic terms, this changes nothing (only the first call to `gpio_pin_set_dt` actually produces a change in GPIO state), but it introduces more work for the task to do, therefore increasing its execution time.

We then ran Interpolation C once again with $WF = 300$, and the results are viewable in Figure 8.3. The values of P were changed from 10 to 9 seconds, which was the identified range where the changes between RM and EDF started to show up. As observed, from $P = 10$ to

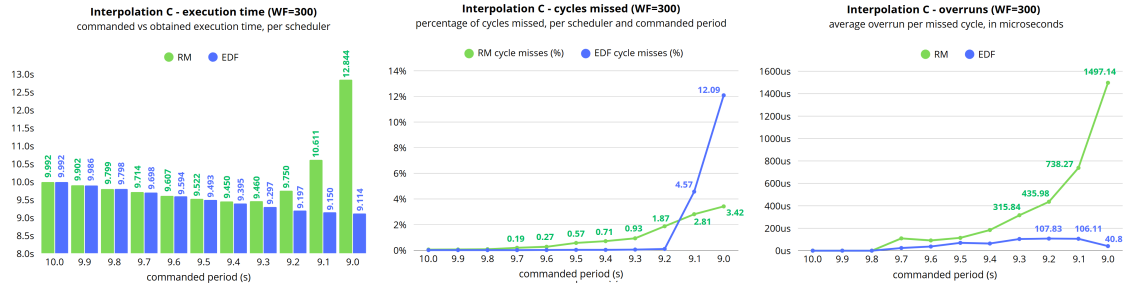


Figure 8.3: Interpolation C test results with $WF = 300$.

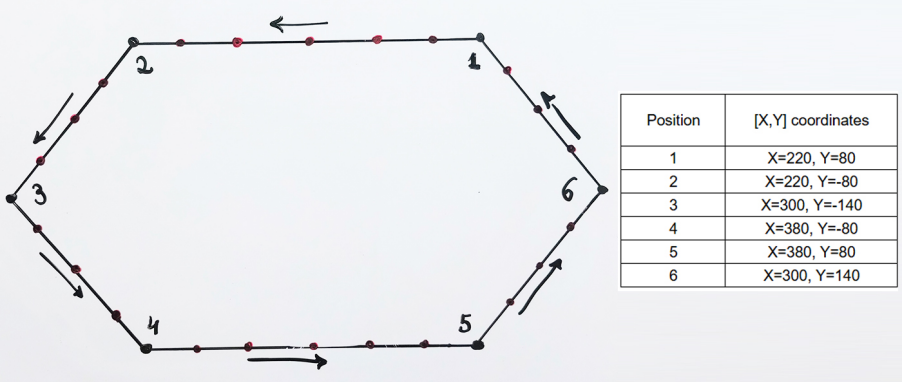
$P = 9.2$ the EDF results seem to match the commanded time and miss almost no cycles, whereas RM progressively increases both the percentage of misses and the average overrun per miss. From $P = 9.1$ onwards both schedulers start having considerable delays, and just as observed in the test before, EDF begins showing a higher percentage of cycles misses. The new data collected seem to indicate the EDF achieves its breaking point somewhere between $P = 9.2$ and $P = 9.1$ seconds, and that RM does it around $P = 10$ and $P = 9.9$. This is a considerable difference, and makes sense according to theory. Although there were deadlines missed for EDF prior to $P = 9.2$, the percentage of cycles it represents is of less than 0.01% and could as well be considered outliars.

8.3 Trajectory tests

The final set of tests aimed at evaluating the robot performance when executing defined *trajectories*. Those differ from interpolations in the sense that while interpolations seek at simply taking the robot from point X to point Y, trajectories also care about the path traversed when performing the movement, and therefore it is critical the robot manages to move in a very specific way. This is the most common way robots work in real-life applications. For example, a robot that interacts with an injection molding machine must enter and leave the molding area to collect the injected part through a constrained route. Attempting to just perform interpolations from the "outside the machine" position to the "inside the machine" position will likely lead to collisions.

To run trajectories for this test, a sequence of 6 key positions forming an hexagon was created as illustrated in Figure 8.4. When launched, the sequence task would send these positions and generate n additional ones in between them in a way to create a linear path for the robot to run. So the robot would still interpolate between any two subsequent positions, but being forced to pass through "checkpoints" in such a way that would ultimately make it respect the provided path. If the movement duration specified between two key positions was of m, the sequence task would set the generated positions to have an interpolation period of $m/(n+1)$, thus making the global trajectory still match the expected duration of m in the end.

The test, then, consisted of attaching a whiteboard marker in the robot's end effector and launching the aforementioned sequence with increasing speed factors for both EDF and RM



Position	[X,Y] coordinates
1	X=220, Y=80
2	X=220, Y=-80
3	X=300, Y=-140
4	X=380, Y=-80
5	X=380, Y=80
6	X=300, Y=140

Figure 8.4: Tested sequence. Numbered positions were specified by the user. Unumbered positions were generated by the sequence task to create a linear path.

schedulers. The robot would then execute the sequence, generating a drawing in a white-board that should roughly[3] match the hexagon of Figure 8.4. The objective was to obtain both qualitative (how consistent was the resulting drawing) and quantitative (how did the sequence execution time compare with the commanded duration) insights on the performance of a more complex work when performed with both EDF and RM. Of all three types of test, this was the closest to a real-world application.

Figure 8.5 shows the results for speed factors ranging from 100% to 500% of the original sequence speed. For convenience, each of the results is identified with the speed factor, the expected sequence duration corresponding to that speed, and the average duration obtained in the robot movement itself. By inspecting the drawings, one can notice EDF and RM are able to produce similar results, and at similar times, between speeds of 100% and 300%. The obtained times in this range appear to be slightly less than configured, however that is likely due to the rounding error that happens when periods, expressed during calculation phase as floating-point numbers, are cast to integers by the schedulers API, which needs to work with system timouts underneath.

When the speed advances to 400% (period of 16750ms), through, the RM scheduler starts falling behind EDF in the movement duration, showcasing a delay of nearly 800 milliseconds (at 17544ms), whereas EDF manages to finish 30ms prior to the configured time (at 16720ms). The breaking point of deadline misses is probably around this speed. At 450% both schedulers start missing deadlines, however RM increases its distance from the expected time to 1365ms (an overrun of 9,16%), while EDF remains in the lead with a less critical 354-millisecond (2,3%) overrun. The resulting drawing made by the robot also shows its first major differences between the two schedulers, where the hexagon made by RM gains two extra vertices near positions 4 and 5. This might be a symptom of starvation for one of the joints (likely the BASE

[3] due to its current mechanical properties (e.g. gearbox backlashes) and intricacies of the IK algorithm, which are not in the scope of this Thesis, the robot is generally not able to reproduce every path commanded with tight precision. However, the paths executed are repeatable and consistent between runs, which enable the analysis anyway.

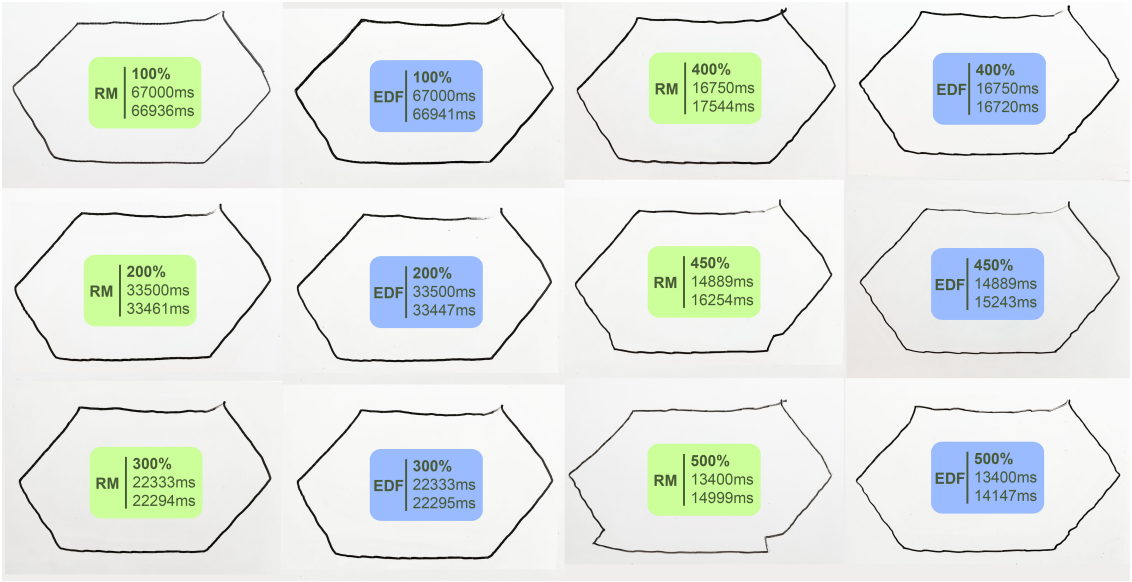


Figure 8.5: Sequence execution result for RM and EDF with different speed factors.

joint, which tends to have higher periods due to its reduction gear ratio, and therefore a lower priority for RM). These vertices become even more noticeable in the following speeds.

The EDF-based drawings, in contrast, also degrade in quality as the speed progresses but remain in the expected hexagonal shape at all moments. This is likely because, as seen in Section 8.2, in overloaded systems EDF is prone to distribute the deadline misses across the tasks rather than concentrating them in the lowest priority ones. This may give one the illusion the system is becoming homogeneously slower, but still keeping concurrent service of all tasks. Still, as seen by the data obtained EDF is consistently capable of delivering lower overruns than RM, which reinforces the idea that it may be able to provide better overall responsiveness than RM when the CPU load becomes abnormal.

Chapter 9

Conclusion

We conclude this document by highlighting the main topics and results discussed throughout the chapters. The first ones provided overviews on the current space industry paradigms and the motivations behind bringing more efficient real-time tools, namely scheduling algorithms such as EDF and CBS, into space applications. We then analyzed the main offerings regarding open-source RTOS, remarking Linux as having recently introduced the latest snippets of the PREEMPT-RT patch into mainline kernel, and undergone an overview of cFS and KARVEL, two software platforms that are purpose-built for spacecraft and enable both reduced development costs by employing a re-usable codebase and a seamless integration with a selection of RTOS, such as FreeRTOS and RTEMS.

Next, we brought KARVEL compatibility into Zephyr, an open-source RTOS backed by a great number of companies and maintained by the Linux Foundation. In the process, three contributions were submitted to the project repository and one has been approved. The introduced schedulers worked as intended in the synthetic tests, where busy-wait routines emulated the effective execution of the tasks, and a few comparisons between each scheduler were delineated. The implementation seemed to have worked with no major challenges encountered.

Finally, we applied the new implementation of KARVEL into a real-world test by developing a robot arm firmware on top of it. The robot had 6 joints driven by stepper motors, and a Raspberry microcontroller (which is officially supported by Zephyr) to coordinate them. The firmware developed used nearly all KARVEL APIs across its tasks, being the most fundamental ones the message queues and schedulers. Each joint was regulated by a dedicated KARVEL task which could switch between EDF and RM as commanded. The resulting software worked as intended, and sequences of movements to test the robot performance were created.

The tests applied to the firmware showcased that EDF and RM are both capable of executing accordingly up until a given CPU usage. For a slight margin after that, EDF keeps scheduling tasks without deadline misses while RM starts falling behind. However, when the average CPU usage reaches its maximum capacity, EDF starts missing *more* deadlines than RM, but each miss has a much smaller magnitude and shows up distributed in the task set, whereas RM concentrates the losses in the lower priority ones. The actual notion of which scheduler handled the system better under overloads is debatable and dependent on the scope of the

application. As the practical tests have shown, however, the robot was better suited for EDF than RM. For future work, we intend to provide real-world performance assessments for CBS as well, and extend KARVEL to support multi-core schedulers in both semi-partitioned and global algorithms.

Bibliography

Abeni, L. and G. Buttazzo (1998). “Integrating multimedia applications in hard real-time systems”. In: *Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No.98CB36279)*, pp. 4–13. doi: 10.1109/REAL.1998.739726.

Akamai (2024). *XZ Utils Backdoor — Everything You Need to Know, and What You Can Do*. available: https://www.akamai.com/blog/security-research/critical-linux-backdoor-xz-utils-discovered-what-to-know.

Apache software Foundation (2023). *About Apache NuttX*. online. Accessed: January 26th, 2025.

ARB, OpenMP (2018). *OpenMP API specification: barrier construct*. available: https://www.openmp.org/spec-html/5.0/openmpsu90.html.

AWS Open Source (2024). *The FreeRTOS Kernel*. Available at: https://www.freertos.org/RTOS.html. Accessed: January 26th, 2025.

Baber, William W. and Arto Ojala (2024). “New Space Era: Characteristics of the New Space Industry Landscape”. In: *Space Business: Emerging Theory and Practice*. Ed. by Arto Ojala and William W. Baber. Singapore: Springer Nature Singapore, pp. 3–26. isbn: 978-981-97-3430-6. doi: 10.1007/978-981-97-3430-6_1. url: https://doi.org/10.1007/978-981-97-3430-6_1.

Bini, E., G.C. Buttazzo, and G.M. Buttazzo (2003). “Rate monotonic analysis: the hyperbolic bound”. In: *IEEE Transactions on Computers* 52.7, pp. 933–942. doi: 10.1109/TC.2003.1214341.

Buttazzo, Giorgio C. (2003). “Rate Monotonic vs. EDF: Judgment Day”. In: *Embedded Software*. Ed. by Rajeev Alur and Insup Lee. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 67–83. isbn: 978-3-540-45212-6.

– (2011). *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. 3rd. Springer New York.

Carlson, Jay (2023). *So you want to build an embedded Linux system? A primer on how embedded Linux systems are developed*. Available: https://jaycarlson.net/embedded-linux/. Accessed: January 26th, 2025.

Community, Linux Kernel Development (2024). *The Linux Kernel - Deadline Task Scheduling*. https://docs.kernel.org/scheduler/sched-deadline.html. Accessed on: September 11th, 2024.

Corbet (2023). *The high-resolution timer API*. online. Accessed: January 26th, 2025.

De Santo, D. et al. (2021). “Exploiting the MIL-STD-1553 avionic data bus with an active cyber device”. In: *Computers & Security* 100, p. 102097. issn: 0167-4048. doi: https://doi.org/10.

1016/j.cose.2020.102097. url: https://www.sciencedirect.com/science/article/pii/S0167404820303709.

Elad, Barry (2023). *15+ Eye-Opening Supercomputer Statistics That You Should Know in 2023*. Accessed: January 26th, 2025.

Ferreira, R. (2025). *NASA cFS version Aquila Software Vulnerability Assessment*. available: https://visionspace.com/nasa-cfs-version-aquila-software-vulnerability-assessment/.

Foundation, Linux (2023). *Zephyr: Project Members*. available: https://zephyrproject.org/project-members/.

IBM (1965). *IBM Operating System/360 - Concepts and Facilities*. page 17. 112 East Post Road, White Plains, N. Y. 10601.

Joerin, Arthur E. (1910). "How to build the famous "Demoiselle" Santos-Dumont's Monoplane". In: *Popular Mechanics* 13.6, pp. 775–78. url: https://archive.org/details/sim_popular-mechanics_popular-mechanics_1910-06_13_6/page/774/mode/2up.

Jones, Mike (Dec. 1997). *What really happened on Mars Rover Pathfinder*. available: https://www.cs.cornell.edu/courses/cs614/1999sp/papers/pathfinder.html. Accessed: September 25th, 2025.

Kruskal, Joseph B. (1969). "Work-scheduling algorithms: A nonprobabilistic queuing study (with possible application to no. 1 ESS)". In: *The Bell System Technical Journal* 48.9, pp. 2963–2974. doi: 10.1002/j.1538-7305.1969.tb01200.x.

Laboratory, National Physical (2025). *What is a leap second?* available: https://www.npl.co.uk/resources/q-a/what-is-a-leap-second. Accessed: January 26th, 2025. Hampton Road, Teddington, Middlesex, TW11 0LW.

Linux Foundation (2015). *Zephyr Project Documentation*. Available at: https://docs.zephyrproject.org/. Accessed: January 26th, 2025.

Liu, C. L. and James W. Layland (Jan. 1973). "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment". In: *J. ACM* 20.1, pp. 46–61. issn: 0004-5411. doi: 10.1145/321738.321743. url: https://doi.org/10.1145/321738.321743.

Mitsis, Peter (2024). *Pull Request 83249: Add k_reschedule*. available: https://github.com/zephyrproject-rtos/zephyr/pull/83249.

Muller-Cajar, R. and Ramakrishnan Mukundan (Jan. 2007). "Triangulation: A new algorithm for Inverse Kinematics". In: *Image and Vision Computing - IVC*.

NASA (2015). *Core Flight System Repository*. available: https://github.com/nasa/cFS. Accessed: January 26th, 2025. Goddard Space Flight Center, Greenbelt, MD.

– (2017). *The Core Flight System (cFS) Scheduler (SCH) application*. available: https://github.com/nasa/SCH/. Accessed: January 26th, 2025. Goddard Space Flight Center, Greenbelt, MD.

– (2021). *Core Flight System Training*. available: https://ntrs.nasa.gov/api/citations/20205011588/downloads/TM%2020205000691%20REV%201.pdf. Goddard Space Flight Center, Greenbelt, MD.

– (2025). *Core Flight Executive User's Guide*.

Nashif, Anas (2022). *The Zephyr Story: How It Became a Self-Sustaining Ecosystem*. available: https://www.intel.com/content/www/us/en/developer/articles/community/

zephyr-story-how-became-self-sustaining-ecosystem.html. Accessed: April 20th, 2025.

O'Callaghan, Jonathan (2020). *SpaceX Makes History With First-Ever Human Rocket Launch For NASA*. Available: https://www.forbes.com/sites/jonathanocallaghan/2020/05/30/spacex-makes-history-with-first-ever-human-rocket-launch-for-nasa/. Accessed: April 20th, 2025.

Paschoaletto, Alexander P. (2022). "Desenvolvimento de um braço robótico de 6 graus de liberdade para fins cinematográficos". MA thesis. Universidade de Brasília.

– (2024). *Issue 79361: EDF updates don't work when made from ISR context*. available: https://github.com/zephyrproject-rtos/zephyr/pull/83249.

– (2025a). *Pull Request 83601: Adding Support for the Constant Bandwidth Server (CBS)*. available: https://github.com/zephyrproject-rtos/zephyr/pull/83601.

– (2025b). *Pull Request 87314: Adding k_thread_deadline_set_absolute*. available: https://github.com/zephyrproject-rtos/zephyr/pull/87314.

– (2025c). *Pull Request 92865: Adding support for k_msgq_put_front and sample code*. available: https://github.com/zephyrproject-rtos/zephyr/pull/92865.

Paschoaletto, Alexander P. et al. (2025). "Supporting Soft Real-Time Tasks in Zephyr with Constant Bandwidth Servers". In: *Proceedings of the 28th International Symposium On Real-Time Distributed Computing*. unplished. IEEE. Toulouse, FR.

Project, RTEMS (2017). *RTEMS 4.11.2 C User Manual - Timer Manager*. available: https://docs.rtems.org/docs/4.11.2/c-user/timer_manager.html.

Project, Zephyr (2025). *Threads - Zephyr Project Documentation*. available: https://docs.zephyrproject.org/latest/kernel/services/threads/index.html.

Prokop, Lorraine (2014). *NASA's Core Flight Software - a Reusable Real-Time Framework*. available: https://ntrs.nasa.gov/api/citations/20140017040/downloads/20140017040.pdf. Accessed: January 26th, 2025. Goddard Space Flight Center, Greenbelt, MD.

PUSopen (2025). *What is CCSDS and ECSS PUS?* Available: https://pusopen.com/ecss-pus. Accessed: April 20th, 2025.

Red Hat, Inc. (2024). *CVE-2024-3094 - Xz: malicious code in distributed source*. available: https://www.cve.org/CVERecord?id=CVE-2024-3094.

Reghenzani, Federico, Giuseppe Massari, and William Fornaciari (Feb. 2019). "The real-time linux kernel: A survey on PreemptRT". In: *ACM Computing Surveys* 52, pp. 1–36. doi: 10.1145/3297714.

RTEMS Project (2025a). *RTEMS 6.17*. Available at https://docs.rtems.org/. Accessed: January 26th, 2025.

– (2025b). *RTEMS Classic API Guide: Scheduling Concepts*. Available at https://docs.rtems.org/. Accessed: January 26th, 2025.

Software, Critical (2023). *European Space Agency Trusts Critical Software with Astronauts' Accommodation Safety*. Available: https://criticalsoftware.com/en/news/esa-trusts-critical-software-with-astronauts-accomodation-safety. Accessed: April 20th, 2025.

Software, Critical (2025). *Karvel: a one-stop shop software platform for satellites*. Available: https://criticalsoftware.com/multimedia/critical/en/D4bv6WF7T-CSW_Space_Flyer_Karvel.pdf. Accessed: April 20th, 2025.

Source, AWS Open (2024a). *FreeRTOS Kernel Customization*. available: https://www.freertos.org/Documentation/02-Kernel/03-Supported-devices/02-Customization.

– (2024b). *FreeRTOS Software Timers*. available: https://www.freertos.org/Documentation/02-Kernel/02-Kernel-features/05-Software-timers/01-Software-timers.

Sprunt, Brinkley, Lui Sha, and John Lehoczky (June 1989). "Aperiodic task scheduling for Hard-Real-Time systems". English (US). In: *Real-Time Systems: The International Journal of Time-Critical Computing Systems* 1.1, pp. 27–60. issn: 0922-6443. doi: 10.1007/BF02341920.

Tipaldi, Massimo et al. (Aug. 2015). "Spacecraft autonomy and reliability in MTG satellite via On-board Control Procedures". In: pp. 155–159. doi: 10.1109/MetroAeroSpace.2015.7180645.

Ultimaker (2023). *Ultimaker Cura: Releases*. available: https://github.com/Ultimaker/Cura/releases/.

Vanga, Manohar et al. (2017). "Supporting low-latency, low-criticality tasks in a certified mixed-criticality OS". In: *Proceedings of the 25th International Conference on Real-Time Networks and Systems*. RTNS '17. Grenoble, France: Association for Computing Machinery, pp. 227–236. isbn: 9781450352864. doi: 10.1145/3139258.3139274. url: https://doi.org/10.1145/3139258.3139274.

Vaughan-Nichols, Steven (2024). *20 years later, real-time Linux makes it to the kernel - really*. online. Accessed: January 26th, 2025.

Wilhelm, Reinhard et al. (May 2008). "The worst-case execution-time problem—overview of methods and survey of tools". In: *ACM Trans. Embed. Comput. Syst.* 7.3. issn: 1539-9087. doi: 10.1145/1347375.1347389. url: https://doi.org/10.1145/1347375.1347389.

Wind River Systems, Inc. (2024). *VxWorks Datasheet*. Available at: https://www.windriver.com/resource/vxworks-datasheet.

Wu, TaiJu (2024). *Pull Request 71195: Message queue support LIFO*. available: https://github.com/zephyrproject-rtos/zephyr/pull/71195.

Yamcs (2025). *CCSDS File Delivery Protocol (CFDP)*. available: https://docs.yamcs.org/yamcs-server-manual/services/instance/cfdp/. Accessed: April 20th, 2025.