FEDERAL UNIVERSITY OF SANTA CATARINA

DEPARTMENT OF AUTOMATION AND SYSTEMS, CENTER OF TECHNOLOGY

POSTGRADUATE PROGRAM IN AUTOMATION AND SYSTEMS ENGINEERING

Elton Ferreira Broering

RUNTIME MONITORING LIBRARY FOR THE FREERTOS

Florianópolis

2023

Elton Ferreira Broering

RUNTIME MONITORING LIBRARY FOR THE FREERTOS

Monograph submitted to the Postgraduate Program in Automation and Systems Engineering of Federal University of Santa Catarina for degree acquirement in Master in Automation and Systems Engineering.

Supervisor: Prof. Leandro Buss Becker, Dr.

Florianópolis

2023

Cataloging at source by the University Library of the Federal University of Santa Catarina.

File compiled at 14:13h of the day Monday 17th July, 2023.

Elton Ferreira Broering
Runtime Monitoring Library for the FreeRTOS / Elton Ferreira Broering; Supervisor, Prof. Leandro Buss Becker, Dr. – Florianópolis, 14:13, 23 of April of 2023.
86 p.

Monograph – Federal University of Santa Catarina, Department of Automation and Systems, Center of Technology, Postgraduate Program in Automation and Systems Engineering.

Includes references

1. Real-Time Operating System. 2. Runtime Verification. 3. UAV. I. Prof. Leandro Buss Becker, Dr. II. Postgraduate Program in Automation and Systems Engineering III. Runtime Monitoring Library for the FreeRTOS

Elton Ferreira Broering

RUNTIME MONITORING LIBRARY FOR THE FREERTOS

This Monograph was considered appropriate to get the Master in Automation and Systems Engineering, and it was approved by the Postgraduate Program in Automation and Systems Engineering of Department of Automation and Systems, Center of Technology of Federal University of Santa Catarina.

Prof. Leandro Buss Becker, Dr.
Federal University of Santa Catarina

Certificamos que esta é a **versão original e final** do trabalho de conclusão que foi julgado adequado para obtenção do título de Mestre em Engenharia de Automação e Sistemas.

ICP-Edu
Documento assinado digitalmente
Julio Elias Normey Rico
Data: 17/07/2023 15:58:44-0300
CPF: ***.840.859-**
Verifique as assinaturas em https://v.ufsc.br

Prof. Julio Elias Normey Rico, Dr.
Coordinator of Postgraduate Program in
Automation and Systems Engineering

ICP-Edu
Documento assinado digitalmente
Leandro Buss Becker
Data: 17/07/2023 15:53:21-0300
CPF: ***.357.870-**
Verifique as assinaturas em https://v.ufsc.br

Prof. Leandro Buss Becker, Dr.
Supervisor
Federal University of Santa Catarina

Florianópolis, 23 of April of 2023.

This work is dedicated to all humanity, I hope that somehow the knowledge produced in this project will contribute to science and the world.

ACKNOWLEDGEMENTS

I am grateful to the Federal University of Santa Catarina and to the Graduate Program in Automation and Systems Engineering, to the lecturing and administrative staff for the opportunity to take the master's course. In particular, I am grateful to my advisor, Leandro B. Becker for his support, guidance, and dedication. I also thank my family, girlfriend and friends who supported me and were by my side during my journey up to this moment. I also extend my thanks to everyone who somehow passed through my life and contributed to me becoming the person I have become. My sincere thanks!

RESUMO

Runtime Verification (RV) é uma técnica leve e dinâmica que verifica a execução atual do sistema por meio de estruturas chamadas monitores e produz um veredicto sobre se essa execução satisfaz ou não uma propriedade específica de correção do sistema. Atualmente, há alguns trabalhos que fornecem bibliotecas para suporte à RV. No entanto, uma característica comum é o suporte apenas a sistemas POSIX, e muitos Real-Time Operating Systems (RTOS) não possuem suporte a POSIX, como, por exemplo, o FreeRTOS. Neste trabalho, foi desenvolvida uma biblioteca de suporte ao RunTime Monitoring (RM), que são os monitores para suporte a RunTime Verification com suporte a sistemas não-POSIX. Um dos requisitos era o desenvolvimento de uma biblioteca altamente desacoplada da arquitetura do Real-Time Operating System (RTOS), com suporte ao monitoramento dos Deadlines e Execution Time das Tasks e suporte às Tasks Periódicas e Não Periódicas. O trabalho também possui dois modos de operação: um modo de operação online e um modo de operação offline. O modo de operação offline é mais simples e leve, buscando impactar menos o sistema sendo monitorado, enquanto o modo de operação online faz todas as verificações dos monitores no sistema em execução e também possui suporte a alguns recursos adicionais. Foi desenvolvida uma aplicação exemplo para demonstrar o funcionamento da biblioteca, a partir de um modelo proveniente do Projeto ProVANT. A biblioteca deu suporte ao monitoramento e validação do modelo proveniente do ProVANT e também demonstrou como utilizar a biblioteca e todos os recursos que ela suporta.

Palavras-chaves: Sist. Operacionais de Tempo Real. Verificação em Tempo de Execução. VANT.

RESUMO EXPANDIDO

INTRODUÇÃO

Muitas abordagens verificam a precisão do sistema e as propriedades de segurança em sistemas críticos. No entanto, a técnica de verificação de modelos é normalmente utilizada, destacando sua eficiência no processo de verificação formal. No caso de abordagens dinâmicas, o *Runtime Verification* (RV) é considerado relevante devido a pesquisas e estudos recentes que o apontam como complementar à análise estática, além de tratar diretamente da aplicação principal, que cobre lacunas como a dificuldade de modelar o sistema de destino.

O trabalho em (PEDRO et al., 2018) mostra que as técnicas para impor o isolamento do tempo são geralmente delegadas às capacidades do *Real Time Operating Systems* (RTOS) (MEIER; HONEGGER; POLLEFEYS, 2015), verificados não formalmente. Autores afirmam que o RV pode aumentar a confiabilidade desses sistemas, principalmente quando associado a métodos de verificação estáticos. O RV pode aumentar a cobertura geral do sistema, garantindo a correção do tempo de execução em partes onde as abordagens estáticas falham em fornecer cobertura global.

Ele apresenta uma estrutura para executar RV em *Real Time Systems* (RTS) para resolver tais problemas. A estrutura consiste na ferramenta *rmtld3synthcpp* para sintetizar automaticamente monitores de destino C++11, a ferramenta *rmtld3synthocaml* para Ocaml e a biblioteca de tempo de execução *RTMLib* para oferecer suporte a monitores de acoplamento em placas de metal simples.

A estrutura proposta pode lidar com tempo e durações explícitos, dois conceitos essenciais para detecção de anomalias em sistemas de tempo real rígidos. No entanto, uma limitação do *RTMLib* desenvolvido é que ele segue estritamente o padrão *Portable Operating System Interface* (POSIX), como por exemplo sistemas Linux. Sistemas POSIX (Portable Operating System Interface) são um conjunto de padrões e especificações para sistemas operacionais que visam fornecer compatibilidade e interoperabilidade entre diferentes sistemas UNIX.

No entanto, nos estudos feitos por (PEDRO et al., 2018), ele é executado usando o NuttX, pois é um RTOS que segue o padrão POSIX. No entanto, muitos aplicativos de sistema embarcado *bare-metal* em tempo real não oferecem suporte a essa arquitetura, como aqueles que usam o FreeRTOS. O trabalho aqui proposto tenta resolver o fato de que o *RTMLib* é exclusivo para uso em sistemas que seguem o padrão POSIX.

OBJETIVOS

O principal objetivo desta dissertação de MSc. é desenvolver uma Biblioteca de Suporte ao Monitoramento em Tempo de Execução para o FreeRTOS que está implementando um projeto similar ao RTMLib.

METODOLOGIA

Como sub-objetivos, é possível destacar o seguinte:

1. Biblioteca de monitoramento em tempo de execução para FreeRTOS.
2. Modos de Operação Online e Offline, para ter um modo de operação mais completo e um modo mais leve.

3. Desacoplar implementações RTOS específicas.

4. Suporte a Tarefas Periódicas e Aperiódicas, Suporte ao monitoramento de Deadline, Suporte ao monitoramento *Worst Case Execution Time* (WCET), em Tarefas Não Preemptivas.

RESULTADOS E DISCUSSÕES

É apresentada uma modelagem de escalonamento de tempo real proveniente do Projeto ProVANT, que demonstra o diagrama de blocos das comunicações das tarefas e seu comportamento esperado no sistema, bem como seus períodos e o tempo máximo de execução conhecido como WCET.

Foi desenvolvido um projeto de exemplo para testar a utilização da biblioteca desenvolvida, que utiliza como base um modelo de escalonamento apresentado, e valida em um hardware embarcado, verificando o modelo de escalonamento por meio do suporte da biblioteca no monitoramento do projeto.

São testados três cenários diferentes no exemplo, buscando extrapolar e estressar o modelo de escalonamento apresentado, a fim de buscar eventuais falhas e também explorar os recursos disponíveis na biblioteca para o monitoramento em tempo real deste modelo em execução.

CONSIDERAÇÕES FINAIS

Um projeto de exemplo foi desenvolvido para validar o funcionamento e comportamento da biblioteca em uso prático. A validação foi realizada considerando um cenário real do projeto ProVANT.

Os resultados foram analisados para avaliar satisfatoriamente o comportamento do modelo de escala proposto no projeto ProVANT, implementado no projeto exemplo. Foi possível observar as facilidades e recursos que a biblioteca pode agregar no processo de desenvolvimento e monitoramento de projetos de software embarcado em tempo real.

A biblioteca *Runtime Monitoring* (RM) desenvolvida é capaz de coletar eventos de interesse e analisá-los. No entanto, o monitor desenvolvido é atualmente limitado, pois só pode raciocinar sobre prazos perdidos e cumprimento do WCET. Conforme discutido em Pedro et al. (2018), torná-lo mais genérico fica para trabalho futuro.

Uma característica importante para futuras implementações é o desenvolvimento de suporte a System Hooks para identificar algumas operações do RTOS, como inserir a tarefa na fila de tarefas disponíveis para execução e preemptar tarefas para agendamento. A implementação dessas novas funcionalidades tornará a coleta de informações mais "precisa"e suportará a monitorização de tarefas não periódicas e preemptivas para a biblioteca RM.

No entanto, essa implementação está altamente ligada à arquitetura do RTOS, portanto seria necessário implementar o suporte da biblioteca RM para esses recursos e também implementar Hooks com o FreeRTOS para validar os recursos da biblioteca e do RM no exemplo do projeto.

Outro recurso importante para o ciclo completo de RV é a síntese dos monitores para integrar a biblioteca RM com sistemas de verificação que, a partir de modelos de entrada, sintetizam automaticamente os monitores a serem verificados em RunTime.

Palavras-chaves: Sist. Operacionais de Tempo Real. Verificação em Tempo de Execução. VANT.

ABSTRACT

Runtime Verification (RV) is a lightweight and dynamic technique that checks the current running of the system through structures called monitors and produces a verdict on whether or not this run satisfies a specific property of system correctness. Currently there are some works that provide libraries to support RV, however a common feature is the support only for POSIX systems like Linux, and many *Real Time Operating Systems* (RTOS) do not support POSIX, such as FreeRTOS. In this work, a support library for *Runtime Monitoring* (RM) is developed, which are the monitors to support RunTime Verification with support for FreeRTOS. One of the requirements was the development of a library highly decoupled from the RTOS architecture, with support for monitoring Deadlines and Execution Time of Tasks and support for Periodic and Non-Periodic Tasks. There are also two operation modes, an online operation mode and an offline operation mode, the offline operation mode is simpler and lighter, seeking to impact less the system being monitored, the online operation mode does all the checks of the monitors on the running system and also supports some additional features. An example application was developed to demonstrate how the library works, based on a model from the ProVANT Project. The library supported the monitoring and validation of the model from ProVANT and also demonstrated how to use the library and all the features it supports.

Keywords: Real-Time Operating System. Runtime Verification. UAV.

LIST OF FIGURES

LIST OF TABLES

LISTINGS

LIST OF ACRONYMS

AADL	*Architecture Analysis and Description Language*
AI	*Artificial Intelligence*
ARM	*Advanced RISC Machine*
CPS	*Cyber Physical Systems*
CPU	*Central Processing Unit*
CTL	*Computation Tree Logic*
ESC	*Electronic Speed Controllers*
FV	*Formal Verification*
FPU	*Float Point Unit*
HLC	*High-Level Command*
I2C	*Inter-Integrated Circuit*
JSON	*JavaScript Object Notation*
LTL	*Linear Temporal Logic*
ML	*Machine Learning*
MC	*Model Checking*
MDE	*Model-Driven Engineering*
MMU	*Memory Management Unit*
PFET	*Past Finite Execution Trace*
POSIX	*Portable Operating System Interface*
RISC	*Reduced Instruction Set Computer*
RT	*Real Time*
RTOS	*Real Time Operating Systems*
RTS	*Real Time Systems*
RM	*Runtime Monitoring*
RV	*Runtime Verification*
SCS	*Safety-Critical Systems*
SPI	*Serial Peripheral Interface*
UAVs	*Unmanned Aerial Vehicles*
USB	*Universal Serial Bus*
WCET	*Worst Case Execution Time*

CONTENTS

1 INTRODUCTION

Recent advances in various technologies have enabled the creation of systems in which computer programs embedded in the most diverse devices combined with sensors and actuators, act on the physical environment. Systems with this characteristic are often referred to in the literature as *Cyber Physical Systems* (CPS) (RAJKUMAR et al., 2010).

CPS can be complex, bringing together a large number of devices that act in an interconnected way (BROY; CENGARLE; GEISBERGER, 2012). Examples of this type of system are *smart grids*, autonomous vehicles, intelligent transport systems, and *smart cities* (STOJMENOVIC, 2014). These systems can be considered the intersection of *Real Time Operating Systems* (RTOS), embedded systems, and distributed sensing and control systems (RAJKUMAR et al., 2010).

The day-to-day modern society is dependent on small programmable electronic systems that are commonly unknown to their users. They are present in automobiles, planes, toys, and hospitals, among others, and guarantee our comfort, safety, leisure, food, and supplies. They are called embedded systems and consist of devices that perform specific functions (BARR, 1999). Because it is highly specialized, an embedded system is optimized for power, code size, execution time, weight, and dimensions.

There is an important class of embedded systems, which must function without errors, otherwise, it may result in loss of life, significant property damage or environmental damage, they are called *Safety-Critical Systems* (SCS) (KNIGHT, 2002). Security critical applications require deterministic behavior during their operation. To deal with them, two important features must be explored: reliability and temporal determinism.

A *Real Time Systems* (RTS) of a computer system that must respond to a stimulus correctly and before a given time constraint, otherwise undesirable consequences may occur (STANKOVIC, 1988). The most important feature of this system is the required temporal determinism and, according to the consequence of its failure, it can be classified into two types of systems: hard real-time and soft real-time. The first ones can cause injuries, deaths, and property/environmental damages, while the second ones can generate a maximum loss of application performance.

In a multitasking operating system, a real-time scheduler is responsible for coordinating processor access to ensure that all real-time tasks in the system do their work on time. In real-time scheduling theory, tasks are described by mathematical variables, which are used in analytical models to prove the scalability of the system (OLIVEIRA et al., 2020).

Rigorous verification strategies are especially vital in the realm of security-critical embedded real-time systems, where systems often need not only to meet a set of functional requirements but also - equally important - strict time constraints. The correct behavior of these systems is defined by the sequence of data they produce, either internally or in their physical outputs, complemented by their temporal behavior.

Formal Verification (FV) is the act of proving or disproving the correctness of a system about a given specification or formal property, where the possible behavior of the system is verified against the desired behavior, this consists of a static analysis of the set of possible scenarios represented by an abstract mathematical model and providing a formal proof based on the pre-defined requirements (ONEM; GURDAG; CAGLAYAN, 2008).

The key idea behind FV techniques such as model verification is to exhaustively verify all implementations of a framework that are related to an implementation and its environment against given requirements. Exhaustive program analysis, however, often suffers from practical infeasibility (due to state space explosion) and/or theoretical impossibility (due to undecidability results) (REINBACHER; FUGGER; BRAUER, 2014).

Runtime Verification (RV) is a lightweight and dynamic technique that checks the current execution of the system through structures called monitors and produces a verdict on whether or not this execution satisfies a specific property of the system correction (FALCONE; HAVELUND; REGER, 2013). It has its origins in model checking. The main difference between the two techniques, in addition to their static and dynamic nature, is that model checking deals with infinite traces, i.e., all possible runs of a given system. RV deals with finite traces, i.e., the current execution status of a target system (LEUCKER; SCHALLHART, 2009).

In RV, observers are synthesized to automatically evaluate the current execution of a system under test, usually from a formal specification into a logic suitable for covering certain forms of specification of the system in the real world. The instantaneous nature of the runtime check can be associated with some overhead mitigated by reducing instrumentation points (REINBACHER; FUGGER; BRAUER, 2014). To evaluate the specifications, RV depends on observations of the system's state under analysis. These observations are called events and serve as input to the observer. However, the state of the system is usually not directly observable.

A classically adopted approach to obtain observations toward performing RV is instrumenting the source code. This technique has proven viable for many high-level implementation languages such as C and C++. Instrumentation can be done manually or automatically by examining programs for assignments and function calls at the implementation language level and then inserting connection functions that emit relevant events to an observer (REINBACHER; FUGGER; BRAUER, 2014).

Another problem faced by developers of real-time embedded systems is the overload of tasks (MIZOTANI et al., 2015). This problem is commonly addressed through fault-tolerant mechanisms, which can be used to recover degraded systems in order to ensure liveliness. These mechanisms are not formally verified due to their inherent complexity and concurrency restrictions. Coupling synthesized monitors automatically and together with the target application allows transient overloads to be checked.

1.1 MOTIVATION

Many approaches verify system accuracy and security properties on critical systems. However, the model verification technique is usually used, highlighting its efficiency in the formal verification process. A method for the design of CPS was developed within the research group that the present author is part of (see (GONÇALVES, 2018)). Such a method uses *Model-Driven Engineering* (MDE) techniques. It deals with the design of models in an automated way, ensuring the maintenance of application characteristics throughout the development process for CPS, especially for *Unmanned Aerial Vehicles* (UAVs), allowing the integration of related models and helping in the validation of system properties.

It supports analyzing the system under design in the functional and architectural modeling phases. Regarding the architectural model, the proposal uses the *Model Checking* (MC) technique as a formal verification approach to verify the desired properties of the system. To support the evaluation of MC, the system's behavior was extracted from a model in the *Architecture Analysis and Description Language* (AADL) (FEILER; GLUCH; HUDAK, 2006) and transformed into a set of timed automata.

The work from (MISSON et al., 2019) extends the previous one ((GONÇALVES, 2018)), proposing a verification process that provides means to confront the developed system model, which represents the system's functionalities and characteristics about the specifications and project requirements.

In the case of dynamic approaches, RV is considered relevant due to recent research and studies that point to it as complementary to static analysis, in addition to dealing directly with the main application, which covers gaps such as the difficulty of faithfully modeling the target system.

The work in (PEDRO et al., 2018) shows that the techniques to enforce isolation of time are generally delegated to the capabilities of the non-formally verified RTOS (MEIER; HONEGGER; POLLEFEYS, 2015). Authors state that RV can increase these systems' reliability, mainly when associated with static verification methods. RV can increase overall system coverage by ensuring runtime correction in parts where static approaches fail to provide global coverage.

It presents a framework for performing RV of bare-metal (LAL, 2013) RTS to tackle such problems. The framework consists of the *rmtld3synthcpp* tool for automatic synthesizing of C++11 targeted monitors, the *rmtld3synthocaml* tool for Ocaml, and the *RTMLib* runtime library to support docking monitors on bare metal boards.

The proposed framework can handle explicit time and durations, two essential concepts for anomaly detection in hard real-time systems. However, a limitation of the developed *RTMLib* is that it strictly follows the *Portable Operating System Interface* (POSIX) standard. However, in the studies carried out by (PEDRO et al., 2018) it is executed using NuttX, as it is an RTOS that follows the POSIX standard. However, many real-time bare-metal embedded system applications do not support this architecture, such as those using FreeRTOS. The work proposed here tries to solve the fact that *RTMLib* is exclusive for use in systems that follow the POSIX standard.

1.2 GOALS

The main goal of this MSc. Thesis is to develop a Library to Support Runtime Monitoring for the FreeRTOS that is implementing a project similar to RTMLib.

As specific goals, it is possible to highlight the following:

1. Runtime Monitoring Library for FreeRTOS.
2. Online and Offline Operation Modes, to have a more complete operation mode and a lighter mode.
3. Decouple specific RTOS implementations.
4. Support Tasks Periodics and Aperiodics, Support to monitoring Deadline, Support to monitoring WCET, in Tasks Non Preemptives.

1.3 THESIS OUTCOMES

So far this M.Sc. Thesis has produced two important outcomes, as follows:

1. An open source Non Posix Runtime Monitoring Library, which available at https://github.com/EltonBroering/RMLib
2. A full paper in the SBESC 2022 conference (BROERING; BECKER, 2022).

1.4 DOCUMENT ORGANIZATION

The parts of this work are organized as follows. Chapter 2 provides technical background for what is discussed in this work and also includes a literature review; Chapter 3 describes the project, some design aspects, and implementation details; Chapter 4 details the conducted experimentation; finally, Chapter 5 outlines the conclusions and the future works directions.

2 RELATED TECHNOLOGIES AND LITERATURE REVIEW

This chapter reviews the main concepts addressed in this work, which includes Embedded Systems, *Real Time Systems* (RTS), *Formal Verification* (FV), and *Runtime Verification* (RV). The chapter also presents the related research project, ProVANT, and finishes addressing relevant related works.

2.1 EMBEDDED SYSTEMS

A *Cyber Physical Systems* (CPS) is an integration of computation with physical processes whose behavior is defined by both cyber and physical parts of the system. Embedded computers and networks monitor and control the physical processes, usually with feedback loops where physical processes affect computations and vice versa. They are systems that combine computational and physical elements, such as sensors and actuators, to create intelligent systems that can sense and control the physical world. This field is growing rapidly, with new applications being developed in areas such as transportation, healthcare, and manufacturing (LEE; SESHIA, 2016). The field of embedded systems is also closely related to the field of CPS.

Embedded systems are information processing systems embedded into enclosing products, are computer systems that are integrated into other devices or products to perform specific functions. These systems can be found in a wide range of devices, such as smartphones, cars, medical equipment, and industrial machines. The key characteristic of embedded systems is that they are designed to perform specific tasks, as opposed to general-purpose computers, which are designed to perform a wide range of tasks (MARWEDEL, 2021).

In recent years, there has been a growing interest in the use of open-source software in embedded systems. The use of open-source software can help to reduce the costs associated with the development of embedded systems, as well as increase the flexibility and adaptability of the systems. There are a number of open-source software platforms and tools that are commonly used in the development of embedded systems, such as Linux, Arduino, and Raspberry Pi for example.

One of the key concepts in the theoretical foundation of embedded systems is real-time computing. This refers to the ability of an embedded system to respond to external events within a specific time frame. Real-time computing is essential for embedded systems that operate in safety-critical or time-sensitive environments, such as aviation or medical equipment (TAKADA, 2001).

Important concept is embedded systems architecture. This refers to the way in which different components of an embedded system are organized and interact with each other. There are several common architectures used in embedded systems, including Von Neumann, Harvard, and RISC (Reduced Instruction Set Computing). Embedded systems often have limited resources, such as memory, processing power, and power consumption, these are a resource constraints. This means that the design of embedded systems must take these constraints into account in order to ensure that the system can perform its intended function (LEE; SESHIA, 2016).

A key aspect of embedded systems design is the use of micro-controllers or microprocessors. These are small, low-power computer systems that can be integrated into embedded systems to control various aspects of their operation. Micro-controllers are often used in conjunction with sensors and actuators to create closed-loop control systems that can respond to external inputs and adjust system behavior accordingly. Are also heavily dependent on the availability of sensors and actuators. Sensors are devices that can detect changes in the environment and provide input to the system. Actuators are devices that can perform actions based on the input received from the sensors. These devices must be carefully selected and integrated into the system to ensure that they can provide the required level of accuracy and reliability (SICILIANO; KHATIB; KRÖGER, 2008).

The design of embedded systems is a complex task that requires a deep understanding of both hardware and software. On the hardware side, embedded systems designers must consider issues such as power consumption, signal processing, and communication protocols. On the software side, they must consider issues such as real-time constraints, memory management, and software architecture (MARWEDEL, 2021). A key challenge in the field of embedded systems is the design and development of software for these systems. The software must be designed to make efficient use of the limited resources available and must also be reliable and robust. There are a number of approaches to the design and development of software for embedded systems, including the use of real-time operating systems and the use of formal methods for the verification of the software. Often use specific programming languages and development tools that are optimized for the specific requirements of embedded systems. Examples of common programming languages used in embedded systems include C, C++, and Python (S TANENBAUM; BOS, 2015).

The use of communication protocols is also crucial for embedded systems. These protocols allow different components of an embedded system to communicate with each other and with external devices. Common communication protocols used in embedded systems include *Serial Peripheral Interface* (SPI), *Inter-Integrated Circuit* (I2C), *Universal Serial Bus* (USB) and wireless protocols like Wifi, 802.15.4 and LoRa (ZURAWSKI, 2018).

Another important aspect of embedded systems is power management. Embedded systems often operate in battery-powered or energy-constrained environments, and therefore, power management is essential to ensure that the system can operate for as long as possible without needing to be recharged or replaced. This includes techniques such as power gating and dynamic voltage and frequency scaling (MARWEDEL, 2021).

One of the most important considerations in the design of embedded systems is security. Embedded systems are often connected to networks and the Internet, and therefore, they are vulnerable to cyber-attacks. This includes techniques such as encryption and authentication to ensure that only authorized individuals can access the system (KOCHER et al., 2004).

In recent years, the field of embedded systems has seen significant advancements in the use of *Artificial Intelligence* (AI) and *Machine Learning* (ML) techniques. These techniques allow embedded systems to make more accurate predictions and decisions based on data inputs. This is particularly useful in applications such as predictive maintenance and self-driving cars (HAIGH et al., 2015).

One of the key challenges facing embedded systems is the issue of scalability. As embedded systems become more complex and interconnected, it becomes increasingly difficult to manage and maintain them. Embedded systems design is the need to ensure that the system is reliable and robust, they are often used in critical applications, such as medical devices or industrial automation, where a failure could have serious consequences.

This requires a thorough understanding of the system architecture, as well as the use of robust software development practices and testing methods (LEE; SESHIA, 2016).

2.2 REAL-TIME SYSTEMS

Real Time Systems (RTS) refer to systems that are constrained by both timing and resource limitations, which means that their accuracy depends on both functional and temporal considerations, with resource constraints being a potential factor. Burns and Wellings distinguish RTS from other types of systems by their ability to avoid negative consequences resulting from delayed or absent responses to stimuli (DAVIS; BURNS, 2011). According to Mall, RTS are characterized by their ability to respond quickly (MALL, 2009), although the significance of this attribute may vary among different authors.

Numerous definitions of RTS exist, with each author presenting a distinct one. However, a critical interpretation in real-time computing is that its accuracy depends not just on the logical outcome of its computation, but also on the timing of the results it generates (STANKOVIC, 1988).

RTS are often classified into two categories, namely soft real-time systems and hard real-time systems. In soft real-time systems, missing a deadline could impact the system's performance to some extent, but it may not result in catastrophic consequences. For example, in an online gaming system, a delay in the response time could result in reduced user experience, but it may not necessarily cause the game to crash.

On the other hand, in hard RTS, meeting deadlines is critical and cannot be compromised under any circumstances. For instance, in a pacemaker device, missing a deadline could be life-threatening for the patient. Similarly, in an autonomous driving system, a delay in processing information could lead to an accident, resulting in serious harm to people's lives. In such cases, meeting the deadline is of utmost importance, and even the slightest delay could have severe consequences (MARWEDEL, 2021).

RTS cover a broad spectrum of applications, including process control systems, manufacturing systems, embedded systems, and even multimedia systems (such as a system for streaming video). They differentiate from general-purpose systems in the following aspects:

- Time constraints are essential to meet deadlines, ensure timely execution, and prevent delays. For example, deadlines specify the moment by which a process must be completed.
- The correctness criterion applies to both real-time and non-real-time systems, but it has a different meaning in the context of RTS. In such systems, correctness encompasses both functional and temporal correctness, and a result that meets functional requirements but is delivered after the deadline is still considered incorrect.
- Numerical computation support is necessary for hybrid systems that combine control activities, such as power plant management systems. RTS are typically dynamic and require timing constraints at discrete points in time, while their behavior is a mixture of discrete and continuous dynamics.
- Safety-criticality refers to the combination of safety and reliability requirements in a system. In non-real-time systems, safety and reliability are often treated separately.

A system is considered safe if it does not cause harm even in case of failure, while reliability refers to the ability of a system to operate without failures over a long period of time.

- Large and complex systems pose significant challenges compared to small programs, which are often simpler in nature and do not encounter the same issues.

RTS face a significant challenge in managing multiple interacting activities, also known as tasks, while ensuring that their timing constraints are met. This challenge is known as real-time scheduling and is an area of ongoing research. Tasks can be conceptualized as abstract entities that represent code components with specific execution requirements. They are triggered by events, such as pressing a power off button or detecting and avoiding a hazardous situation by a kitchen robot. These timing constraints impose limits on the execution of code. The subsequent paragraphs will provide a classification of tasks and timing constraints.

Multiple instances of a task are commonly referred to as jobs. Real-time jobs can have relative or absolute deadlines, such as being based on the arrival of a stimulus event or the system's execution start time. The response time of a job is the duration between when it is released and when the task completes its execution. Scheduling algorithms are designed for uni-processor, multiprocessor, and distributed systems, which have significant differences in terms of delays and spatial positioning. The transition between uni-processor and multiprocessor systems to distributed systems involves several major abstractions.

The following passage will highlight the main terms and characteristics of real-time systems below, to briefly facilitate their understanding (OLIVEIRA, 2020).

- **Task**: The term task is used more specifically for the execution of a code segment that has some of its temporal attributes or restriction, such as a period or a deadline.
- **Task Periodicity**: Tasks are executed periodically at predetermined intervals to achieve a specific timing requirement, the period of a task is the amount of time that elapses between two consecutive instances of the task's execution. The periodicity of tasks is critical in real-time systems, where tasks must be completed within specific deadlines, where different tasks are scheduled and executed based on their priority and periodicity.
  - **Periodic Tasks**: Periodic tasks are those that execute at regular intervals of time. These tasks are typically used for repetitive and time-critical operations.
  - **Aperiodic Tasks**: Aperiodic tasks are those that do not have a fixed periodicity. These tasks are triggered by external events, such as user input or sensor data, and require immediate attention.
  - **Sporadic Tasks**: Sporadic tasks are similar to aperiodic tasks in that they do not have a fixed periodicity. However, sporadic tasks are less frequent and less critical than aperiodic tasks.
  - **Idle Tasks**: Idle tasks are those that execute when no other tasks are ready to run. These tasks are typically used to perform low-priority operations.
- **Task Period**: Interval time for periodic tasks, those that run at regular time intervals.

- **Deadline**: Deadline for the execution of the task, and it can have no relative deadline, when it is defined in relation to the arrival of the task, or absolute deadline, when it is defined in relation to UTC time.

- **Execution Time**: Execution time is defined as the time that the task needs a given resource to execute, most of the time of the resource in question is the processor.

- **Worst Case Execution Time (WCET)**: It is the maximum execution time that can be observed for a task on a computer, this time is usually estimated based on the tasks project.

- **Task Preemption**: task preemption refers to the ability of the operating system to temporarily suspend the execution of a running task in order to allow a higher-priority task to run.

- **Task Release**: is the moment in which the task is included in the capable queue and starts to be considered for execution by the operating system kernel.

- **Response Time**: the time interval between the arrival of the task and its completion. Within the response time, there are several other times, such as the execution time of the task, the execution time of other tasks that execute while the task waits, and the release jitter.

- **Arrival Time**: The arrival time corresponds to the initial time from which the task could be executed. From this moment on, the task can be executed. For periodic tasks, the beginning of each period is considered as the arrival time. For aperiodic tasks, arrival is caused by the event that triggers task execution.

2.3 FORMAL VERIFICATION

Formal Verification (FV) is the process of using mathematical techniques to prove the correctness of software and hardware systems. It has been widely adopted in the field of computer science and engineering as a way to ensure the reliability and safety of complex systems. The theoretical foundation of formal verification can be traced back to the work of Alan Turing and Alonzo Church in the 1930s, who laid the foundations for the theory of computability (BJESSE, 2005).

The theoretical foundation of formal verification is rooted in mathematical logic and automata theory. FV methods are based on the use of formal languages, such as predicate logic and temporal logic, to express the properties of a system. These properties are then verified using automated proof systems, such as model checkers and theorem provers. One of the key concepts in formal verification is that of a formal system, which is a set of axioms, inference rules, and a set of theorems that can be derived from the axioms using the inference rules. A formal proof is a sequence of deductions, each of which is made using the inference rules, that shows that a theorem is a logical consequence of the axioms (DRECHSLER et al., 2004).

The most widely used formality in computer science is the first-order logic, which is a type of predicate logic that is used to express the properties of software and hardware systems. First-order logic allows for the definition of predicates and functions, as well as the use of quantifiers to express properties of sets of objects.

MC is a technique that is used in formal verification to automatically check the validity of a system against a set of properties. The properties are typically specified in temporal logic, such as *Linear Temporal Logic* (LTL) or *Computation Tree Logic* (CTL). The model checker then constructs a finite state model of the system, and uses algorithms such as BDDs or SAT solvers to check whether the properties hold for all possible states of the system (CORTÉS; ELES; PENG, 2003).

Another important technique in formal verification is theorem proving, which is the process of constructing a formal proof of the correctness of a system. Theorem provers use a formal proof system, such as natural deduction or resolution, to construct the proof.

One of the most widely used theorem provers is the Boyer-Moore theorem prover, which is based on the resolution principle and is able to automatically generate proofs for a wide range of problems. Other popular theorem provers include the Coq system, which is based on type theory and the Isabelle system, which is based on higher-order logic.

Formal verification also plays a crucial role in the design and implementation of hardware systems, as it allows for the automatic checking of the correctness of digital circuits. The most widely used formal system for hardware verification is temporal logic, such as LTL and CTL, which can be used to specify properties of hardware systems (DRECHSLER et al., 2004).

One of the main challenges in hardware verification is the state explosion problem, which is caused by the large number of possible states that a hardware system can be in. This can be addressed by using techniques such as symbolic model checking, which uses Boolean satisfiability solvers to check the properties of a system.

In recent years, there has been a growing interest in the application of formal verification to CPS. These systems are typically subject to strict safety and reliability requirements, and formal verification can be used to ensure that they meet these requirements (LEE; SESHIA, 2016).

FV has also been used in the field of embedded systems, which are systems that include both hardware and software components. In embedded systems, FV is used to ensure that the interactions between the hardware and software components are correct. Overall, FV is a powerful method for ensuring the correctness of systems, and it has been widely used in a variety of fields. It is based on a solid theoretical foundation in mathematical logic and automata theory, and it has been successfully applied to a wide range of systems (ALUR, 2011).

2.4 RUNTIME VERIFICATION

As *Real Time Systems* (RTS) continue to increase in magnitude and complexity, they are no longer small, standalone applications. Instead, they are often embedded in a larger context with multiple interacting components and systems. This interconnectedness makes these systems more susceptible to errors, which can grow exponentially with the number of system components. Concurrency and non-determinism, which are essential to modeling RTS, are particularly challenging to manage using standard techniques (BAIER; KATOEN, 2008).

In this section it is focused on model-based verification, particularly using timed temporal logic, which is well-suited for modeling RTS. This is because RTS are complex and dependent on temporal constraints, making many static approaches unfeasible. Reachability becomes undecidable due to the time clock operations in these systems, such as addition and subtraction by a constant. As a result, there is a growing need for reliable and safe development alternatives to static approaches, especially for hard real-time systems. While research on dynamic approaches has been increasing for soft real-time systems, it has been slower for hard real-time systems, with a greater emphasis on functional aspects (PEDRO et al., 2018).

The technique of RV is used to monitor the behavior of a system to ensure that it adheres to a desired set of logical properties. It should be noted that much of the RV literature focuses on event-triggered solutions, which can introduce significant overhead and unpredictability to the system being monitored.

Runtime monitoring, also known as monitoring at execution time, involves automatically synthesizing monitors (dedicated blocks of source code) from formal specifications. This technique can be used offline for debugging or online for dynamic checking of properties during execution. Offline monitoring involves collecting a program trace, which is then analyzed to determine compliance with the specification. This technique can be used to capture operations such as system calls, interrupts, context switches, and state variables, and is useful for replay and analysis of the scheduling process.

Online monitoring, on the other hand, involves checking properties during execution. For example, inline monitoring involves inserting monitoring as annotations (e.g., assertions) into the execution code, while outline monitoring involves executing the monitor as a separate concurrent process. Outline monitors can be implemented in hardware, synthesized from high-level formal specifications, and executed on FPGAs to result in zero runtime overhead on the system's *Central Processing Unit* (CPU). However, RV often incurs a significant time penalty during system execution, so some authors recommend using multi-processor systems when a hardware monitoring approach is not used. This allows the monitoring process to be performed concurrently on a different processor, without delays to the system under monitoring (GOODLOE; PIKE, 2010).

A runtime monitor is defined as a process that examines a *Past Finite Execution Trace* (PFET) and produces a verdict during runtime. The verdict represents a truth value from a truth domain that can range from simple true and false, to a three-valued true, false, and unknown, or even a probabilistic interval within the range of 0 to 1.

In essence, the mathematical problem of RV can be boiled down to solving the word problem, which is essentially the problem of determining whether a given word belongs to a particular language. Denoting the set of valid executions that satisfy a certain property Φ as $J\Phi K$, then the word inclusion problem becomes checking whether the execution w is a member of $J\Phi K$. However, the language inclusion problem is generally more complicated and undecidable, as is the case with classical timed automata (ALUR; DILL, 1992).

Dealing with the complexity of modern software systems is a major challenge in RV. Many of these systems are distributed and concurrent, making it difficult to ensure their correctness. Moreover, changing requirements and environments further complicate the task of creating accurate specifications.

Another significant challenge in RV is coping with uncertainty. Programs are frequently executed in environments that are not fully known or understood, making it hard to anticipate their behavior. Additionally, changes or updates to programs can impact their behavior. To overcome these challenges, a range of techniques has been developed to handle uncertainty, including probabilistic methods, fuzzy logic, and Bayesian networks.

In addition to uncertainty, scalability is also a crucial aspect of RV when dealing with software systems. As systems become larger and more intricate, verifying their correctness becomes increasingly challenging. To tackle this challenge, researchers have devised various techniques for scaling RV, such as using distributed and parallel algorithms and specialized hardware (HAMLET, 2010).

To address these challenges, researchers have developed a variety of techniques for RV, including dynamic analysis, formal methods, and machine learning. Dynamic analysis techniques, such as profiling and tracing, are used to observe the program's execution and gather information about its behavior. Formal methods, such as model checking, are used to formally verify that a program conforms to a set of specifications. Machine learning techniques are used to learn properties of the program's execution and use this knowledge to improve the accuracy of RV.

In recent years, there has been a growing interest in the use of RV for the verification of autonomous systems, such as self-driving cars and drones. These systems are subject to a wide range of safety and reliability constraints, which makes RV a critical component of their development. Additionally, as the use of autonomous systems is expected to increase in the future, there is a growing need for techniques that can ensure their safety and reliability.

2.5 RELATED WORKS

The ProVant [1] project, whose objective concerns the design of autonomous UAVs, was created in 2012 by the Federal University of Santa Catarina (UFSC) in partnership with the Federal University of Minas Gerais (UFMG). Since its inception, different studies and research have been done, covering different phases of UAVs design, such as the design of complex control systems, long-distance communication, artificial intelligence, and the design of complex embedded systems.

About the design of embedded systems, especially UAVs projects, a design method was created to guide the different design teams during the construction of the (GONÇALVES; RAFFO; BECKER, 2016) project. This method is composed of a set of phases and activities and aims to adequately cover the representation of the required characteristics of the UAVs. It is based on MDE and provides for the construction of complementary representations to map the application characteristics.

To this end, tools were developed within the scope of the project to support the integration of stages throughout the life cycle, such as model transformation tools that aim to automate the generation of new representations from their input data.

Regarding method activities, system verification and validation is defined as an activity that aims to ensure the evaluation and validation of system requirements. Thus, studies have been carried out to evaluate the recommended approach to be applied in this process. There are three very interesting works for the context of this project, "Project Of The Embedded Software Architecture Of An Unmanned Aerial Vehicle" (GONÇALVES et al., 2014), "Applying formal verification techniques in Embedded Software in the design of a UAV" (MISSON et al., 2019) and "*Dynamic contracts* for checking and applying properties in real-time systems" (MATOS PEDRO, 2018).

Many different approaches are used to verify system correctness and security properties on critical systems. However, the model verification technique is normally used,

[1] http://provant.paginas.ufsc.br

highlighting its efficiency in the formal verification process.

A method for CPS design was developed within the present research group, this method is proposed by (GONÇALVES, 2018). The proposed method is based on DEM and deals with the design of models in an automated way, ensuring the maintenance of application characteristics throughout the development process for CPS, especially for UAVs, allowing the integration of related models and helping in the validation of properties of the system.

For this, important phases are contemplated from the definition of requirements, modeling of the dynamics of the aircraft in the functional model, modeling of the system architecture, and transformation process that generates a suitable model for static verification. In addition, it seeks to integrate all these processes, contributing to the design of the embedded system, as well as the interface with the set of sensors and actuators.

The main contributions of the thesis improve the development process of CPS, in which the tools for transforming related models stand out, allowing the analysis of the system in the functional and architectural modeling phases.

Regarding the architectural model, the proposal uses the MC technique as a formal verification approach to verify the desired properties of the system. To support the MC evaluation, the system behavior was extracted from the AADL model and transformed into a set of timed automata. MC is a static, automatic, and exhaustive FV technique, where the abstract model of the system is checked against properties written in a formal language (CLARKE; EMERSON, 1981).

To support the evaluation process, two distinct phases were adopted, so that the properties are properly represented and allow the validation of the system characteristics. The first phase describes the analysis of isolated tasks coupled to the set of devices that interface with them. This phase aims to investigate possible failures in related devices.

The second phase evaluates the system's set of *threads* according to a defined scheduling policy and also considers possible system failure conditions. This was done as an initial programming approach, where timing information such as running time was estimated.

The proposed method provided an integrated solution for the phases and activities of the UAVs design process, in which the use of MC was designed as a formal verification technique. As the treated system is considered critical and its design is complex, it is recommended to use more than one method to verify the accuracy of the system.

The work "Applying formal verification techniques in Embedded Software in the design of a UAV" (MISSON et al., 2019) proposes a verification process that provides means to confront the developed system model, which represents the functionalities and characteristics of the system about the specifications and project requirements.

Some verification approaches are known and widely applied, such as static verification, which does not require the existence of executable code, this is a model verification technique. Static verification is based on an abstract system model, while the desired properties are written in formal language. A verification tool is used to apply algorithms and output verdicts of satisfaction or non-satisfaction of the specified properties against the model (GRUMBERG; CLARKE; PELED, 1999).

Although it is a widely used technique, it has some limitations, such as state space explosion, which in many cases makes it impossible for the model verifier to perform the verification. This occurs when the model is not very simplistic so that the possible combinations of its execution grow exponentially (infinite traces). However, other approaches and methods can be used as a complement to guarantee the correctness of the system.

The proposed verification process aims to fill the gaps with unsatisfied or undefined properties, in addition to the state explosion found in previous project approaches, associated with the challenge of modeling the application in a way that portrays its execution closer to the real one. The Flow involves dynamic, static, and *tracing* analysis, restricted to steps 1 to 4 (red rectangles), highlighted in figure 2.1.

In step 1, static verification (1st round) is performed, and model verification is performed as a first verification approach. It uses an abstract model of systems that can be represented, for example, as Finite State Machines, Petri Nets, or Timed Automata. In the end, the model check tool checks the model against the specifications.

In step 2 the analysis of *tracing* is done, once the code is generated, the tracing tools can be used to perform a complete analysis of the actual execution of the application. This is very important to allow an understanding of software behavior, as well as to obtain more accurate timing information of system tasks, including preemption times, mutexes, and CPU usage.

In step 3, static verification is carried out (2nd round), this step suggests carrying out another static verification, representing a refinement and update of the verification performed in the first step. Overall, it has the same goal, which is to automatically and exhaustively check the model against formal properties.

Step 4 is Dynamic Verification, the dynamic verification step deals with the run-time execution of the actual application code observed and verified by an entity called a monitor.

The relationship between this step and the previous one is closely linked because if the model designed in the previous approach does not satisfy an important property or even if this property cannot be verified, due to the limitations of the formal static verification technique, the guarantee of correctness and System reliability can be done using the dynamic method.

In the case of the dynamic approach, the run-time verification technique was considered due to recent research and studies that point to it as a complement to static analysis, in addition to dealing directly with the main application, which covers gaps such as the difficulty of faithfully modeling the target system.

The work in (PEDRO et al., 2018) shows that even when formal methods of synthesis and verification are used, the techniques to impose isolation are generally discarded and delegated to the capabilities of non-formally verified RTOS (MEIER; HONEGGER; POLLEFEYS, 2015). Ideally, RV whould be used in collaboration with static verification methods. Together, these techniques can increase overall system coverage by ensuring runtime correctness in those parts where static approaches fail to give overall coverage.

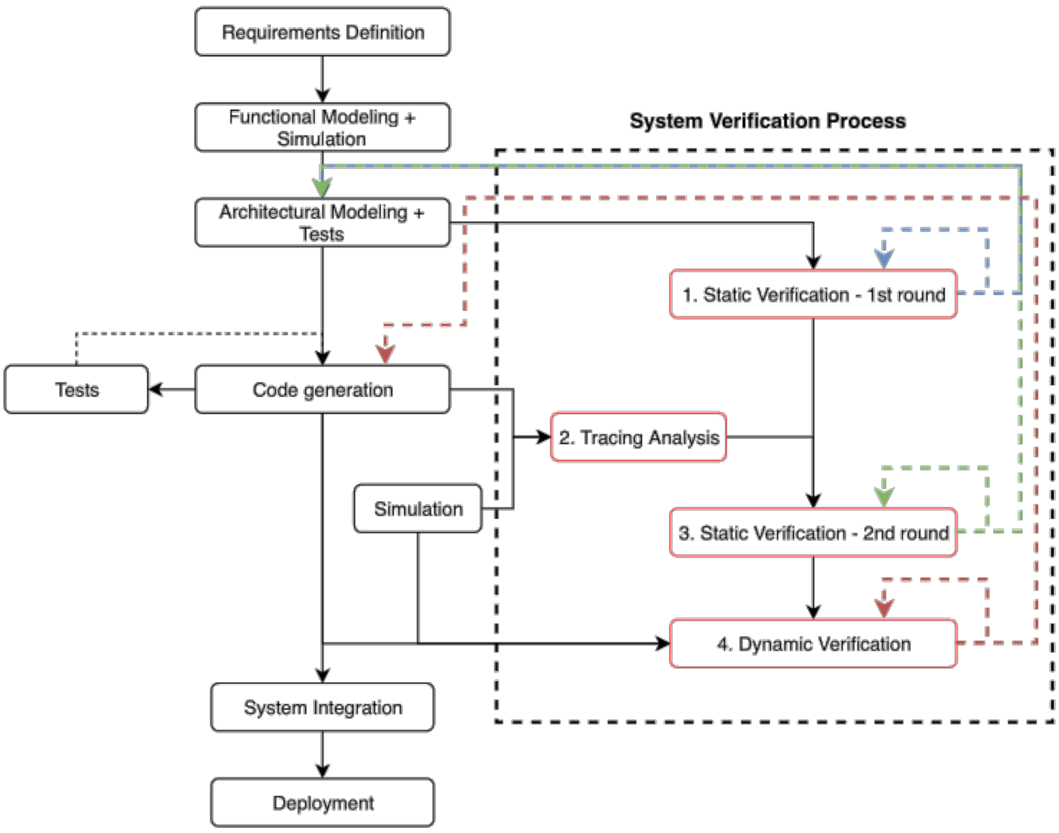
Requirements Definition
Functional Modeling + Simulation
Architectural Modeling + Tests
System Verification Process
1. Static Verification - 1st round
Tests
Code generation
2. Tracing Analysis
Simulation
3. Static Verification - 2nd round
4. Dynamic Verification
System Integration
Deployment

Figure 2.1 – Proposed design process flow (MISSON et al., 2019).

There is yet another problem faced by developers of real-time embedded systems, task overload in RTOS. This problem is commonly handled through fault-tolerant mechanisms, which can be used to recover degraded systems to guarantee liveliness. Most of these mechanisms are not formally verified, due to their inherent complexity and concurrency constraints. Coupling automatically synthesized monitors together with the target application allows transient overloads to be checked for (MIZOTANI et al., 2015) safety.

A framework for performing RV of bare-metal embedded systems of RTS is presented in (PEDRO et al., 2018). The framework consists of the `rmtld3synthcpp` tool for automatically synthesizing monitors targeting `C++11`, the `rmtld3synthocaml` tool for Ocaml, and the `RTMLib` runtime library for supporting docking monitors on bare-metal boards. This RV framework is capable of dealing with explicit time and durations, two of the essential concepts for detecting anomalies in hard real-time systems.

2.6 FINAL REMARKS

The RV framework developed for Andre Pedro is capable of dealing with explicit time and durations, two of the essential concepts for detecting anomalies in hard real-time systems. However, a limitation of this tool is that it follows the POSIX standard.

POSIX systems are a set of standards and specifications for operating systems that aim to provide compatibility and interoperability among different UNIX-like systems. The idea behind POSIX is to create a common interface that allows applications written for one POSIX operating system to run on any other POSIX operating system without the need for significant modification.

The tool has a dependency on using PThread, for example, and in the studies carried out by (PEDRO et al., 2018), it is executed using `NuttX`, as it is an RTOS that follows the POSIX standard. However, many real-time bare-metal embedded systems applications do not support this architecture, such as those using FreeRTOS.

The present work aims to complement the analysis previously developed by implementing the dynamic analysis proposed in the work developed by (MISSON et al., 2019). This implementation will use the modeling developed in steps 1 to 3, deepen the study and implementation of step 4, and compare it with the results obtained in the other verification approaches carried out. The objective is to work on the *Runtime Verification* process, adapting the work proposed by (PEDRO et al., 2018) for use in non-POSIX RTOS, such as the FreeRTOS used in this work.

We also explored several other RunTime Verification libraries, but their RunTime Monitoring support had restrictions when it comes to using FreeRTOS, as well as RTM-Lib. Feather-Trace is a minimal event tracing toolkit primarily designed for tracing the Linux Kernel, and the Kernel Runtime Verification Subsystem provides support for Kernel Runtime verification (THE LINUX KERNEL ORGANIZATION, 2023). However, neither of these options is compatible with FreeRTOS. Another library, R2U2 (ROZIER; SCHUMANN, 2017), operates by monitoring a system trace and "filtering messages". However, in more constrained systems that utilize FreeRTOS (EDITION, n.d.), implementing a file system or managing memory through an *Memory Management Unit* (MMU) is not supported.

3 RUNTIME MONITORING LIBRARY

The aim of this project is to develop a *Runtime Monitoring* (RM) library that supports the FreeRTOS operating system, with a focus on being architecture-independent and compatible with multiple platforms. The goal is to provide a lightweight and non-intrusive solution for implementing RM in systems, while also allowing for customization based on specific architecture characteristics.

The initial implementation of the library focuses on the essential functions and minimal storage structures required for RM. The intention is to provide a generic interface for system-specific implementations, enabling designers to adapt the code to their target architecture. The current implementation primarily focuses on instrumenting the code for RM but does not include automated monitor generation.

One of the key features of the proposed RM library is the support for two operation modes: online verification and offline verification. In offline verification mode, the monitors run concurrently with the application code, collecting timestamps at specific points of interest. These timestamps are then stored for later exportation and evaluation of monitored constraints using external software. Offline verification is less intrusive and imposes less overhead on the system, but it offers fewer features for data analysis. In Figure 3.1, it's possible to observe the use case of the Offline Mode of the proposed library.

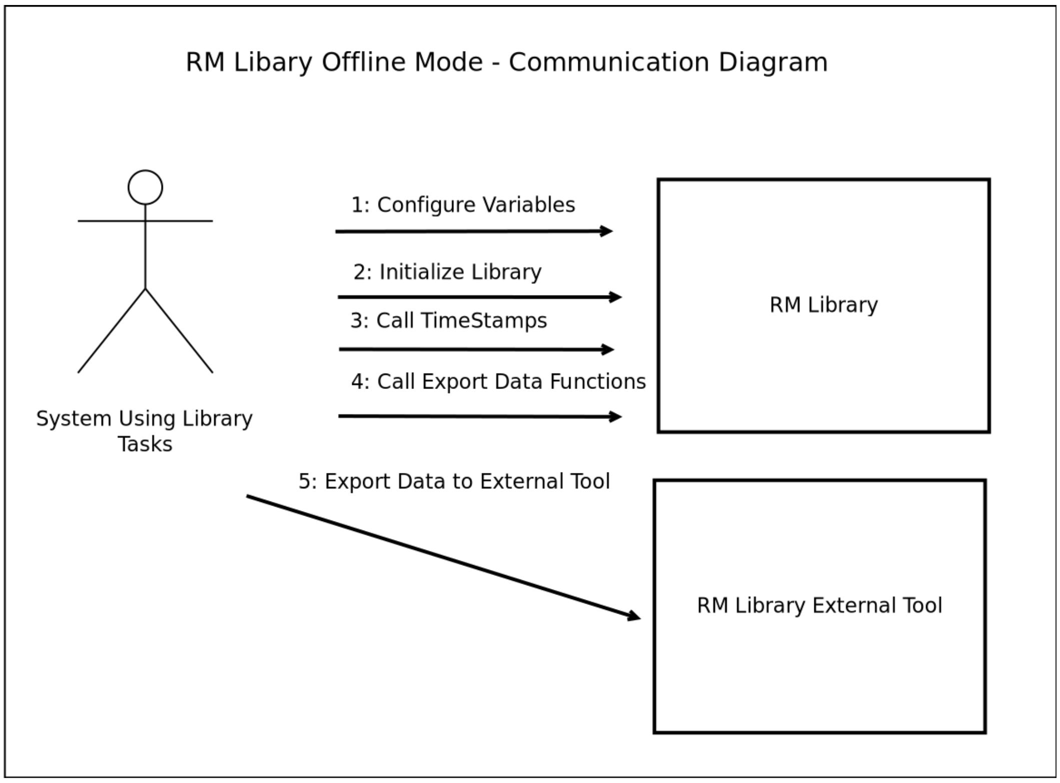
RM Libary Offline Mode - Communication Diagram
System Using Library Tasks
1: Configure Variables
2: Initialize Library
3: Call TimeStamps
4: Call Export Data Functions
5: Export Data to External Tool
RM Library
RM Library External Tool

Figure 3.1 – RM Library Offline Mode Communication Diagram.

On the other hand, online verification mode operates similarly to offline verification in terms of monitoring and timestamp collection. However, the verification of system restrictions occurs online within the target system being monitored, and only the verification results are exported. Online verification is independent of external tools and can be used as a debugging tool. It offers additional features such as timestamp analysis, exporting only cases of *Real Time* (RT) violations, and exporting system information when violations occur. However, it imposes higher overhead on the system and can significantly impact system performance. In Figure 3.2, it's possible to observe the use case of the Online Mode of the proposed library.

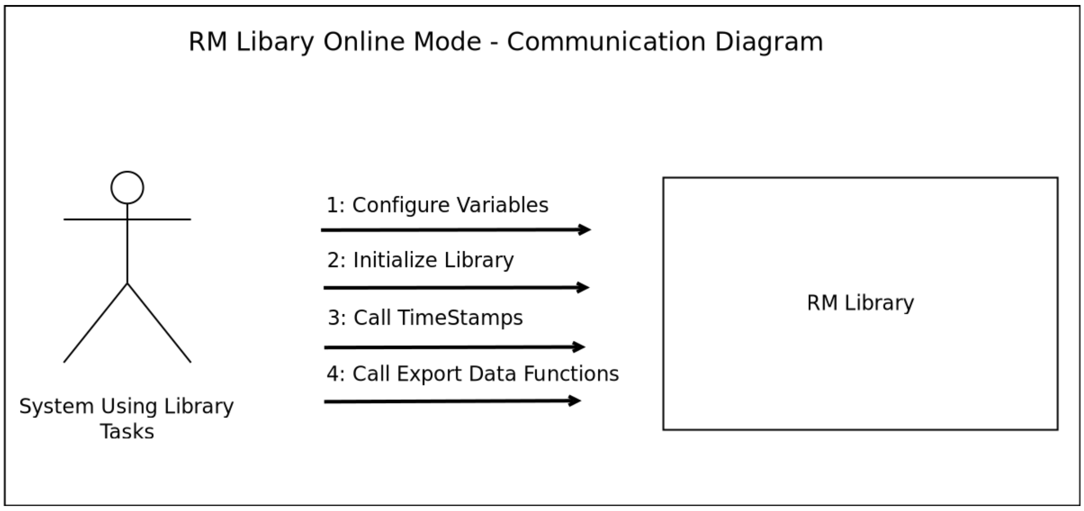
RM Libary Online Mode - Communication Diagram
1: Configure Variables
2: Initialize Library
3: Call TimeStamps
4: Call Export Data Functions
System Using Library Tasks
RM Library

Figure 3.2 – RM Library Online Mode Communication Diagram.

In the online mode, more resources are available to provide a comprehensive analysis of the system. The flag `EXPORT_ONLY_RTOS_ERRORS` optimizes data storage by exporting only cases with RT violations, making it easier to analyze these cases. Another feature is the `EXPORT_DUMP_STATUS_TASKS` flag, which allows for exporting the status of tasks by calling a specific function. When RT violations occur, the data export function is provided by the library user, as it is specific to each system.

The complete library project can be accessed on the author's GitHub page (BROERING, 2022).

3.1 OFFLINE MODE

The following diagram of use 3.3 seeks to demonstrate in a simple way the mode of use proposed for using the library. The objective of the diagram is to present a sequence of operations to use the Library.

The diagram shows the necessary initialization of the system, timers of the system and RM Library, the creation of the tasks, and the execution sequence of the tasks using the library. The process of tasks using the library basically involves calling the library at the beginning and end of the task execution. Additionally, there are communication tasks that must export the data to be analyzed by the external tool of the RM Library.

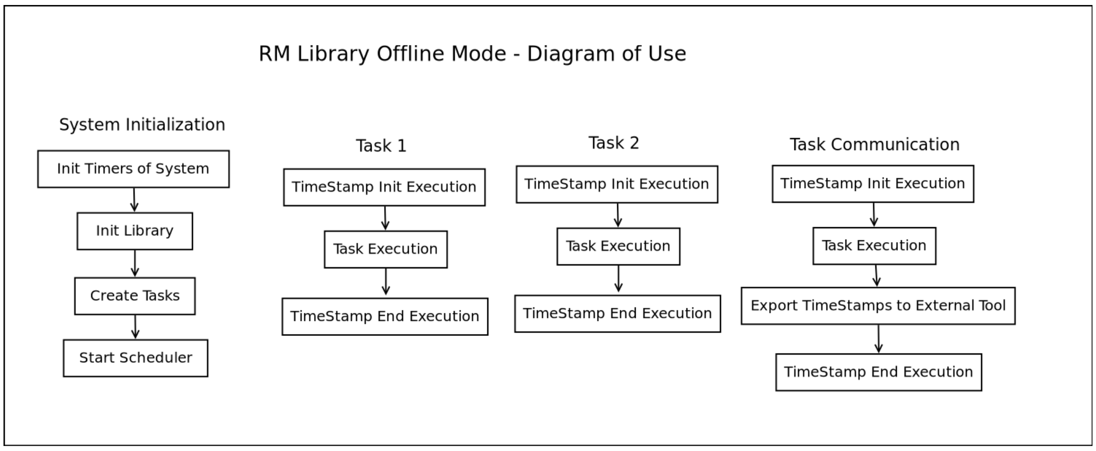

Figure 3.3 – RM Library Offline Mode Sequence Diagram.

The implemented library is based on the work by Pedro et al. (2018) (PEDRO et al., 2018). Its fundamental component is the *Event structure*, which is responsible for storing the individual data of each event. The structure includes the following elements:

- *TimeStamp*: Represents the timestamp of the event, indicating when it occurred.
- *Counter Task*: Identifies the execution cycle of a task, allowing for proper event ordering.
- *Task Identifier*: Provides a unique identifier for the task associated with the event.
- *Task State*: Indicates whether the event represents the start or end of a task.

The structure is designed to respect 32-bit multiples to ensure proper memory alignment on architectures that require it. You can refer to Listing 3.1 to see the structure's representation.

For a more detailed understanding of the implementation, you can consult Listing A.1 in the library, which provides the corresponding code.

```
32 bits TimeStamp
25 bits Counter Task
5 bits  Identifier of Task
2 bits  State of Task
```

Listing 3.1 – Data Structure used to store events in Offline Mode.

In order to configure the RunTime Monitoring library, it is necessary to define the parameter *NUMBER TASKS RUNTIME MONITORING*, which determines the number of tasks to be monitored by the library. This configuration is present in the library's header file. You can see the configuration definition in Listing 3.2.

To further understand the implementation of this concept, you can refer to Listing A.2 in the library, which provides the corresponding code.

```
NUMBER_TASKS_RUNTIME_MONITORING        X
```

Listing 3.2 – Definition of Number Tasks of RM library in Offline Mode.

The initialization of the library in Offline Mode involves initializing the circular queue structures used to store the system's events. The size of the storage queue is defined in the library header, and the memory areas for storing these structures are statically allocated, which is a recommended practice for embedded systems to avoid dynamic allocation. The initialization process is demonstrated in Listing 3.3.

For a closer look at the implementation of this concept, you can refer to Listing A.3 in the library, which provides the corresponding code.

```
SIZE RUN TIME BUFFER QUEUE  X
QueueEventsBuffer [SIZE_RUN_TIME_BUFFER_QUEUE]

rmlib init
    init queue
```

Listing 3.3 – Initialization function of the RM library in Offline Mode.

The proposed library includes a function to read the system time in milliseconds, which is crucial for registering the time (TimeStamps) of the system's events. The function definition is presented in Listing 3.4. It's important to note that the implementation of this function needs to be provided by the user of the library since it is closely tied to the specific system where the library is utilized.

To gain a better understanding of how this function can be implemented, you can refer to Listing A.4 in the library, which showcases the code for this particular concept.

```
32 bits ReadCounterMiliSeconds
```

Listing 3.4 – Definition of ReadCounter Function in MilliSeconds.

The TimeStamp function is responsible for registering events and takes two inputs: the Task Identifier and the Task State, which indicates whether it's a task start or end monitoring point. In offline mode, the function reads the system milliseconds and stores the Task's Event, as illustrated in the pseudocode presented in Listing 3.5.

For a more detailed understanding of the implementation, you can refer to Listing A.5 in the library, which demonstrates the actual code for this concept.

```
timestamp runtime (task_identifier , task_state)
    store parameter in variable Event
    ReadCounterMiliSeconds
    store Event
```

Listing 3.5 – Function to collect TimeStamps in RunTime while in Offline Mode.

The data export function in the library removes events from the events queue (using the "pop" action) and provides a way to convert the collected events into a text format that can be interpreted by the offline checker. Both functions are shown in Listing 3.6.

The exported data format for the offline checker is a string in JSON format, containing the fields present in the Event structure. The function demonstrates the implementation of COMPACT EXPORT DATA, which offers a more compressed solution for data export, reducing the utilization of system resources. In Offline Working Mode, this is crucial due to the large amount of data exported for analysis using the external tool. The external tool supports both data export formats.

Please note that the specific details of the implementation can be observed in Listing A.6 in the library.

```
rmlib export data (event_rmlib)
    pop(event_rmlib)

rmlib export data string (event_rmlib)
    IF compact export data
        export data compact string
    ELSE
        export data string
```

Listing 3.6 – Functions to export Events while in Offline Mode.

The offline analysis tool, which checks the exported data and monitors events in offline mode, collects the events, verifies the input and output timestamps of tasks, and checks if they comply with the defined time requirements. It then presents a verdict for each task. The algorithm developed for this purpose in the example scenario is shown in Listing 3.7. Additionally, in Listing A.6, you can observe the implementation of this concept in the library.

```
ANALYZE_DATA(DATA):
  ANALYZE STATE OF TIMESTAMP
  GROUPING MESSAGES USING TASK IDENTIFIER
  IF NUMBER_OF_TIMESTAMPS_TASK_IDENTIFIER > 1:
    ANALYZE_TASK_TIMESTAMPS
      IF DELTA_TIMESTAMP <= TIME_RESTRICTIONS:
        TASK CONDITION: OK
      ELSE:
        TASK CONDITION: NOK
  ELSE:
    STORE_TIMESTAMP
 WHILE:
    DATA = RECEIVE_MESSAGES_EVENTS
    ANALYZE_DATA(DATA)
```

Listing 3.7 – Pseudo code from their RM Offline Analyzer.

To use the RunTime Monitoring (RM) offline analysis tool, you need to configure certain fields in the ReaderSerial tool. These fields include the deadline, periods, and WCET vectors, which are reference parameters passed for analyzing the received data. Additionally, the serial_interface needs to be defined to specify the communication interface through which the tool will receive the data. These configurations are shown in Listing 3.8. Currently, the tool is limited to receiving data through the serial interface.

```
vector_deadlines     = [X...Y]
vector_period        = [X...Y]
vector_wcet          = [X...Y]
serial_interface     = intarface name
```

Listing 3.8 – RM Offline Analyzer configurations.

For a better understanding of how to use the RMLib library in Offline Mode, we will present a pseudo-code that demonstrates the step-by-step process of initialization, configuration, and usage. This simplified guide will show the necessary steps to get started.

First, in the initialization process, the system timer is initialized. Then, the library itself is initialized with the appropriate parameters. After that, the system tasks are created, and finally, the system scheduler is initialized. The code snippet in Listing 3.9 illustrates this initialization process.

The tasks implemented in the system have the following characteristics: at the beginning of each execution cycle, the task executes the timestamp function with the starting identifier, and at the end of the execution cycle, the timestamp function is executed with the ending identifier. This ensures that the events are properly registered. The code snippet in Listing 3.9 demonstrates this implementation.

When using the RMLib library in offline mode, an important feature is the export of timestamps for verification by an external tool for later analysis. This allows for in-depth analysis of the collected data. The code snippet in Listing 3.9 showcases this export functionality.

```
system initilization:
    init timer of system
    init library
    create tasks
    start scheduler

task 1:
    task loop execution:
        timestamp runtime(task1,init of taks)
        taks execution
        timestamp runtime(task1,end of taks)
task 2:
    task loop execution:
        timestamp runtime(task2,init of taks)
        taks execution
        timestamp runtime(task2,end of taks)

task communication:
    task loop execution:
        timestamp runtime(task3,init of taks)
        taks execution
            export timestamps to Extern Tool
        timestamp runtime(task3,end of taks)
```

Listing 3.9 – Pseudo code Example Case of Offline Mode.

3.2 ONLINE MODE

The following sequence diagram 3.4 seeks to demonstrate in a simple way the proposed mode of use for the library.

The diagram illustrates the necessary system initialization, timers of the system and library, tasks with their parameters for verification, creation of tasks, and the execution of tasks utilizing the library. The process of tasks using the library basically involves calling the library at the beginning and end of task execution. Additionally, it includes communication tasks that export the verdicts of the RM Library system.

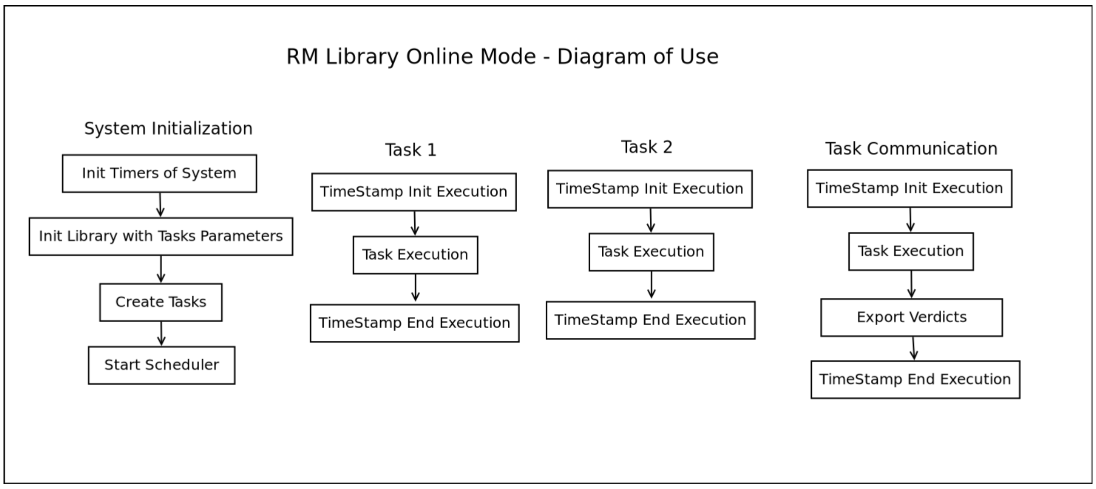

Figure 3.4 – RM Library Online Mode Sequence Diagram.

For the online operation mode, in addition to the basic event structure shown in Listing 3.1, where events that have not yet been processed are stored due to the task's execution cycle not ending, the data structure called *TimeStampVerdict* is also created. It serves as the fundamental system structure for storing the verdicts of each execution cycle of the tasks after processing in the online operation mode. This structure is shown in Listing 3.10. Additionally, in Listing B.1, you can observe the implementation of this concept in the library.

The *TimeStampVerdict* structure consists of the following elements: TimeStamp (which stores the initial timestamp of this verdict), CounterTask (task execution cycle counter), ExecutionTime (task cycle execution time), Identifier of Task, Status of WCET Task (indicates the status of the task's worst-case execution time (WCET) in the task execution cycle, indicating whether the WCET constraint was fulfilled correctly or not), and Status of Deadline Task (indicates if the task's deadline status fulfilled the task execution cycle deadline constraint).

```
16 bits     TimeStamp
16 bits     Counter Task
16 bits     Execution Time
5 bits      Identifier of Task
1 bit       Status of WCET Task
1 bit       Status of DeadLine Task
9 bits      Reserved Bits
```

Listing 3.10 – Data Structure used to store TimeStamp Veredicts.

One of the requirements for the RunTime Monitoring library is the definition of NUMBER TASKS RUNTIME MONITORING, which configures the number of Tasks to be monitored by the library, this library configuration definition is present in the library header file. This configuration can be observed in Listing 3.11. Additionally, in Listing B.2, it is possible to observe the implementation of this concept in the Library.

```
NUMBER TASKS RUNTIME MONITORING      X
```

Listing 3.11 – Definition of Number Tasks of RM library in Online Mode.

Initializing the library for Online Verification is similar to Initializing the library in Offline mode showed in Listing 3.1, conceptually initializing the queues is the same. However, TimeStampVeredicts queues are initialized and at initialization, vectors are also passed with the parameters to be verified according to RT restrictions, such as TaskIdentifiers, DeadLines, Period and WCET of the Tasks. Its initialization is presented in Listing 3.12. Additionally, in Listing B.3, it is possible to observe the implementation of this concept in the Library.

```
SIZE RUN TIME BUFFER QUEUE   X
QueueEventsBuffer  [SIZE_RUN_TIME_BUFFER_QUEUE]

rmlib init (RT_paratemeters_verification)
    init queue
    init variables with RT_paratemeters_verification
```

Listing 3.12 – Initialization function of the RV library in Online Mode.

The proposed library requires a function to read the system time (in milliseconds). This basic function is needed to register the time (TimeStamps) of the system's events. The function definition is presented in Listing 3.13. In this case the function definition is the same used in Offline Mode Operation. Additionally, in Listing B.4, it is possible to observe the implementation of this concept in the Library.

```
32 bits ReadCounterMiliSeconds
```

Listing 3.13 – Definition of Function ReadCounter in MilliSeconds.

The TimeStamp function implements the registration of the events. Its input is the *Task Identifier* and the *Task State* (task start or end monitoring point). In the online mode the function reads the system milliseconds and stores the Task's Event, but if the TaskState entry in function is TASK END EXECUTION, the function make the processing of Task Cycle, and verify RT Constraints to make a verdict, and then save this verdict, this resources is presented and explained in Chapter 3. It's possible to observe some implementations of extra resources, how EXPORT_ONLY_RTOS_ERRORS and EXPORT_DUMP_STATUS_TASKS. As shown in the source code presented in Listing 3.14. Additionally, in Listing B.5, it is possible to observe the implementation of this concept in the Library.

```
timestamp runtime (task_identifier , task_state)
    store parameter in variable Event
    ReadCounterMiliSeconds
    IF TASK END EXECUTION
```

```
        analyze Events and store Results in Verdict
        store Verdict
    ELSE
        save Event
```

Listing 3.14 – Function to collect TimeStamps in RunTime while in Onine Mode.

The function that exports the data in the online mode removes the TimeStampVerdicts (pop action) from the TimeStampVerdicts' queue. The library also implements a function to convert the set of events collected to the text format to export, to turn easier to integrate data exported to others, this data is exported in *JavaScript Object Notation* (JSON) format. Both functions are shown in Listing 3.15. In this function is possible to observe the COMPACT EXPORT DATA implementation, this additional resource implements a better solution to export data more compressed and using minor resources of the system. Additionally, in Listing B.6, it is possible to observe the implementation of this concept in the Library.

```
rmlib export data (verdict_rmlib)
    pop(verdict_rmlib)

rmlib export data string (verdict_rmlib)
    IF compact export data
        export data compact string
    ELSE
        export data string
```

Listing 3.15 – Functions to export Events Online Mode.

For a better understanding of the use of the library, we will present pseudo code to demonstrate how to use the library RMLib in Offline Mode in a simplified way, showing the step-by-step guide for initialization, configuration, and use. First, in the initialization process, the System Timer is initialized, the library is initialized, the System Tasks are created, and finally, the system scheduler is initialized, as can be seen in Listing 3.16.

The implementation of Tasks has the following characteristics: at the beginning of the execution cycle, the Task must execute the timestamp function with the beginning identifier, and at the end of the execution cycle, the timestamp function must be executed with the identifier of the end of the execution cycle, as shown in Listing 3.16.

When using the RMLib library in offline mode, an important feature is the export of TimeStamps for verification by an external tool for later analysis, as shown in Listing 3.16.

```
system initilization:
    init timer of system
    init library with Tasks Parameters
    create tasks
    start scheduler

task 1:
    task loop execution:
        timestamp runtime(task1,init of taks)
        taks execution
        timestamp runtime(task1,end of taks)
task 2:
    task loop execution:
        timestamp runtime(task2,init of taks)
        taks execution
        timestamp runtime(task2,end of taks)

task communication:
    task loop execution:
        timestamp runtime(task3,init of taks)
        taks execution
            export Verdicts
        timestamp runtime(task3,end of taks)
```

Listing 3.16 – Pseudo code Example Case of Offline Mode.

3.3 CONCLUDING REMARKS

In this chapter the implementation of the RM library was presented, initially the basic concepts of the library and its resources, and in detail the two implementations, the operation mode with Online Verification and the operation mode with Offline Verification. In the next Chapter 4, an example implementation of the library will be presented, seeking to demonstrate how to use it, through the example and how to monitor the system, using the resources of the developed library.

4 EVALUATION

This chapter describes the application of the developed *Runtime Monitoring* (RM) library within the context of the ProVant project, presented in Chapter 2.5, which aims to develop *Unmanned Aerial Vehicles* (UAVs). This study intends to demonstrate the usage of the developed RM library to support the design of a safe continuous-control architecture, as further detailed.

The experiments will be conducted on top of a UAV control scenario, exploring situations in which the time constraints are not met, aiming to observe how the system will behave. This also allows us to explore some additional features of the developed library to facilitate fault detection and analysis of these cases.

In order to identify anomalies, the feature of exporting only the faulty situations (temporal faults) is used. This facilitates the identification of faults and the use of *DumpTaskStatus* to improve the analysis of the causes of these faults, as well as to properly observe the remaining system threads when faults/violations occur.

4.1 ENVIRONMENT SETUP

To perform the proposed experimentation, all implementations are carried out using a 32-bit *Advanced RISC Machine* (ARM) Cortex-M4 *Reduced Instruction Set Computer* (RISC) processor platform, specifically on the SAM4E Xplained Pro development kit (MICROCHIP, 2022). Atmel Studio was used as the development environment, and the operating system is FreeRTOS version 10.0.0. The implementation of this study is available for download at (BROERING, 2022).

The first step in performing experiments using RMLib in our experiment is the initialization of the chosen hardware for the example. Initially, the initialization of all pins and peripherals that will be used in the system must be performed, along with the initialization of the USB driver to serve as serial communication, allowing the system to export the collected data. Finally, the initialization of the Timer Counter hardware must be performed, using the Timer Counter peripheral of the ARM processor to accurately and correctly measure time. Additionally, there is the configuration of a hardware button for use in the experiment, which will be used in one of the experiments to manually call a Non-Periodic Task. This initialization can be observed in Listing 4.1, located in the *main()* function of the *main.c* file.

```
/* main(): Initialize the SAM system */
sysclk_init();
board_init();
stdio_usb_init();
Timer_init();
#ifdef ASYNCHRONOUS_TASK
configure_button();
#endif
...
```

Listing 4.1 – Function *main()* in *main.c*: board and peripherals initialization.

It must be highlighted that the library must implement a *ReadCounterMs* function, which should return the number of milliseconds of the system. To do so, the System Time Counter peripheral is used. The peripheral is initialized, and the time interruption intervals are defined according to the defined time scale, which in this case is 1 ms. The configured interrupt Handler function is implemented, along with the *ReadCounterMs* function, which returns the number of milliseconds since the peripheral initialization. These implementations are shown in Listing4.2, located in the file *Timings.c*.

```
void Timer_init(){
uint32_t ul_sysclk = sysclk_get_cpu_hz();
pmc_enable_periph_clk(ID_TC0);
tc_find_mck_divisor(SYSTEM_TIMER_COUNT_SCALE, ul_sysclk, &ul_div
    , &ul_tcclks, ul_sysclk);
tc_init(TC0, 0, ul_tcclks | TC_CMR_CPCTRG);
tc_write_rc(TC0, 0, (ul_sysclk / ul_div) /
    SYSTEM_TIMER_COUNT_SCALE);
NVIC_EnableIRQ((IRQn_Type) ID_TC0);
tc_enable_interrupt(TC0, 0, TC_IER_CPCS);
tc_start(TC0, 0);
}
void TC0_Handler(void){
ul_dummy = tc_get_status(TC0, 0);
g_ul_ms_ticks++;
}
uint32_t ReadCounterMiliSeconds(void){
  return g_ul_ms_ticks;
}
```

Listing 4.2 – Auxiliary functions for retrieving timming information (file *Timings.c*).

After initializing the RMLib according to the Online_Verification or Offline_-Verification usage mode, initializing the library according to the proper operation mode requires changing some parameters as explained earlier in Chapter 3. In this case, when initializing the RM library, vectors are passed with the parameters of the system's Tasks, such as the Tasks identifier Vector (the number that identifies the Tasks), the Tasks Deadlines vector, the Periods of the Tasks, and the WCET vector of the Tasks to be monitored. All of them should be in the same order as the Tasks identifiers. These parameters are being configured according to the information present in Table 4.1, which includes information from the project's Tasks and serves as an example of using the library.
After this stage, still in the *main()* function of *main.c*, the creation of system tasks and the initialization of the task scheduler must be done, as shown in Listing 4.3.

```
...
#ifdef ONLINE_VERIFICATION
init_buffer_tasks_runtime_verification_online();
rmlib_init(&Identifiers_Tasks,&Vector_Deadline_Tasks,&
    Vector_Period_Tasks,&Vector_WCET_Tasks);
#endif
#ifdef OFFLINE_VERIFICATION
rmlib_init();
#endif
if(xTaskCreate(task_controller, "Controller",
    TASK_CONTROLLER_STACK_SIZE, NULL, TASK_CONTROLLER_PRIORITY,&
    TaskHandle_Controller) != pdPASS){
    printf("Failed to create Controller task\r\n");
}
if(xTaskCreate(task_communication, "Communication",
    TASK_COMMUNICATION_STACK_SIZE, NULL,
    TASK_COMMUNICATION_PRIORITY,&TaskHandle_Communication) !=
    pdPASS){
    printf("Failed to create coomunication task\r\n");
}
if(xTaskCreate(task_led_hlc, "Led", TASK_LED_HLC_STACK_SIZE,
    NULL, TASK_LED_HLC_PRIORITY, &TaskHandle_LedHLC) != pdPASS){
    printf("Failed to create led task\r\n");
}
if(xTaskCreate(task_dummy_sensing, "Dummy Sensing",
    TASK_DUMMY_SENSING_STACK_SIZE, NULL,
    TASK_DUMMY_SENSING_PRIORITY, &TaskHandle_DummySensing) !=
    pdPASS){
    printf("Failed to create dummy task\r\n");
}
if(xTaskCreate(task_dummy_actuation, "Dummy Actuation",
    TASK_DUMMY_ACTUATION_STACK_SIZE, NULL,
    TASK_DUMMY_ACTUATION_PRIORITY, &TaskHandle_DummyActuation) !=
     pdPASS){
    printf("Failed to create dummy task\r\n");
}
#ifdef ASYNCHRONOUS_TASK
if(xTaskCreate(task_asynchronous, "Asynchronous",
    TASK_ASYNCHRONOUS_STACK_SIZE, NULL,TASK_ASYNCHRONOUS_PRIORITY
    , &TaskHandle_Asynchronous) != pdPASS){
    printf("Failed to create asynchronous task\r\n");
}
#endif
/* Start the scheduler. */
vTaskStartScheduler();
```

Listing 4.3 – Code of initialization of RMLib and Tasks of System.

4.2 EXPERIMENTAL WORKLOAD DESCRIPTION

The analyzed continuous-control architecture consists of five concurrent processes/threads, as illustrated in Figure 4.1. The High-Level Control (*High-Level Command* (HLC)) is a non-preemptive thread responsible for interfacing with higher-level control entities, such as a separate cognitive agent deployed on separate hardware. The HLC task is critical because it must execute before the control thread within the control cycle to provide input data for the control algorithm. The Sensing thread is a non-preemptive thread responsible for interfacing with the sensors, collecting data, and performing pre-processing to prepare the data for use as input by the control algorithm. The Control thread is a non-preemptive thread responsible for executing the continuous-control algorithm. It utilizes input sensor data generated by the Sensing thread and commands from the HLC thread. The output of the Control thread consists of actuation values that need to be sent to the Actuation thread. The Actuation thread is a non-preemptive thread responsible for interfacing with the actuation components, such as the two Electronic Speed Controllers (*Electronic Speed Controllers* (ESC)) connected to the brushless motors.

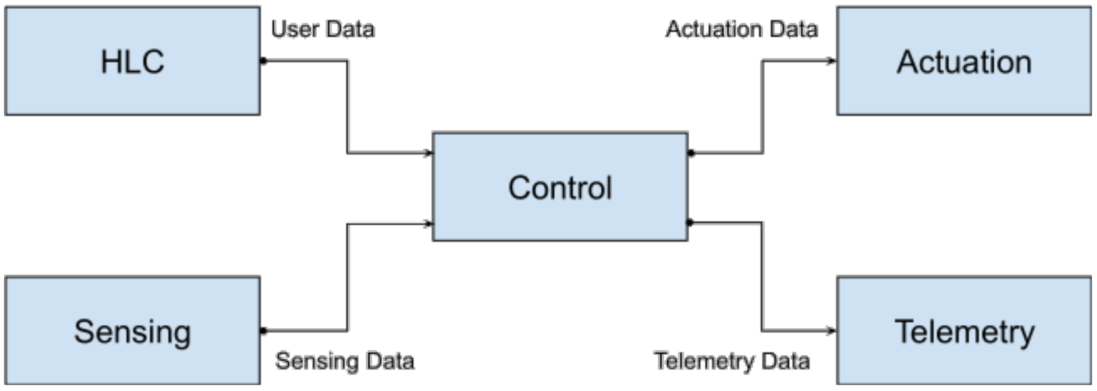

Figure 4.1 – Model of the UAV continuous-control architecture.

Given that the Electronic Speed Controllers (ESC) require periodic commands, the Actuation thread becomes critical as its slack time needs to be very small (close to zero). On the other hand, the Telemetry thread is a preemptive thread responsible for transmitting monitoring data to a remote base station. The amount of data transmitted by this thread is determined and configured by the UAV's operation team, which can result in high execution times. In this thesis, the highest possible execution time for this thread was established to ensure that the overall CPU utilization approaches 100%. Table 4.1 provides a summary of the timing and scheduling properties of the adopted continuous-control thread architecture.

Task	Period	Deadline	WCET	Utilization	Priority	Preemptive
T1-Actuation	12	12	2	0.166	5	NO
T2-Sensing	12	12	3	0.250	4	NO
T3-Control	12	12	4	0.333	2	NO
T4-HLC	120	120	4	0.033	3	NO
T5-Telemetry	600	600	130	0.216	1	YES

Table 4.1 – Thread-set of the proposed continuous-control architecture.

The defined WCET takes into account the maximum possible execution time of each task, considering the restrictions imposed by the task period, rather than estimating the actual WCET values. Therefore, the WCET is considered as the maximum time constraint for each task, defined by its period. This model is proposed by Misson in (MISSON et al., 2019).

The scheduling model that will be used is based on a priority-based scheduling system with both preemptive and non-preemptive tasks.

Regardless of the Telemetry thread's execution time and the resulting CPU utilization factor, the CPU is fully utilized during the "critical instant." In this study, the critical instant lasts 12ms and occurs every 120ms, as illustrated in the Gantt diagram in Fig. 4.2. Please note that the threads in the diagram are presented in the same order as in Table 4.1, with T1 (Actuation) at the top and T5 (Telemetry) at the bottom. In this model, the critical instant occurs every 120ms, i.e., whenever the HLC (T4) thread is ready for execution, so it must be executed before the Control (T3) thread. It is important to pay attention to the 12ms timing windows immediately before (starting at 108ms) and immediately after (starting at 132ms) the critical instant. These windows are used to execute the Telemetry thread, which is the only preemptable thread. Finally, it should be noted that the Actuation thread should consistently execute without jitter, i.e., with a slack time close to zero.

Typical UAVs missions take several minutes or even hours to complete. However, the current experiment aims to validate the developed RM infrastructure rather than prove the success of a specific mission. Therefore, the focus has been on a minimal time window of approximately 120ms. This hyper-period corresponds to the task set presented in Table 4.1.

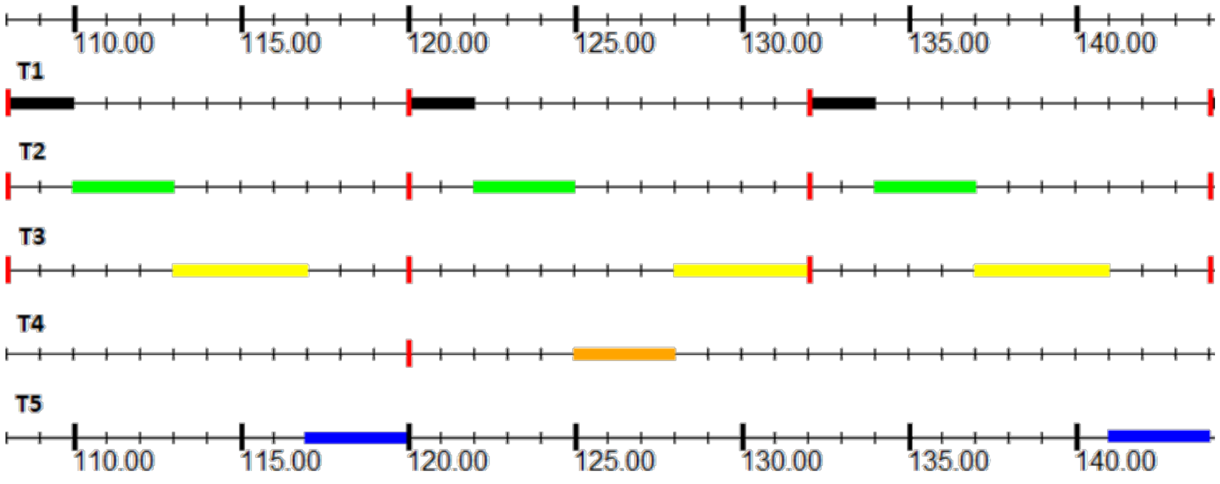

Figure 4.2 – Gantt diagram of the adoped thread-set execution, with the critical instant in the center (120ms - 132ms).

Another important aspect to highlight is that the execution loops of the five tasks comprising the continuous-control architecture are not executing the actual application source code. Instead, they are executing synthetic code that has equivalent execution times. In other words, the execution time in this case represents the WCET of the respective tasks.

Regarding the behavior of the Control Task, during its initialization, it sets up the control function to use the *Float Point Unit* (FPU). Upon entering the cyclic execution loop, it first utilizes a function to define an atomic region where preemption is disabled. Then, it calls the TimeStamp function with the task identifier and the state

TASK_INIT_EXECUTION. Subsequently, it invokes the Control LQR Controller function (DONADEL et al., 2015), which represents the control algorithm modeled in the ProVANT project. Finally, it records the end of task execution timestamp and introduces a delay to ensure compliance with its execution period. You can refer to Listing 4.4 for the implementation of the Task Controller function.

```
static void task_controller(void *pvParameters){
c_control_lqrArthur_init();
while(true){
     vTaskSuspendAll();
     timestamp_runtime(TASK_IDENTIFIER_CONTROLLER,
    TASK_INIT_EXECUTION);
     c_control_lqr_controller(&controller_input,&controller_ouput
    );
     timestamp_runtime(TASK_IDENTIFIER_CONTROLLER,
    TASK_END_EXECUTION);
     counter_tasks_runtime[TASK_IDENTIFIER_CONTROLLER-1]++;
     uint32_t MileSecondsTask = ReadCounterMiliSeconds();
     uint32_t ValueTaskDelay;
     if((TASK_CONTROLLER_PERIOD*counter_tasks_runtime[
    TASK_IDENTIFIER_CONTROLLER-1]) > MileSecondsTask){
          ValueTaskDelay = (TASK_CONTROLLER_PERIOD*
    counter_tasks_runtime[TASK_IDENTIFIER_CONTROLLER-1] -
    MileSecondsTask);
     }
     else{
          ValueTaskDelay = 0;
     }
     vTaskDelay(ValueTaskDelay);
     xTaskResumeAll();
}}
```

Listing 4.4 – Function Task Controller.

The Sensing and Actuation tasks indeed share a similar implementation. They do not have a specific initialization step, and in their main execution loop, they are non-preemptive. They utilize signals to indicate the start and end of the task to the RM library, and an idle execution period occurs between these signals, ensuring adherence to their respective WCET and the period between executions. You can refer to Listing 4.5 for the implementation of the Task Actuation function. The Sensing task follows the same structure, with the only difference being the values of WCET, task period, and task identifier passed to the TimeStamp function.

```
static void task_dummy_actuation(void *pvParameters){
while(true){
    vTaskSuspendAll();
    timestamp_runtime(TASK_IDENTIFIER_DUMMY_ACTUATION,
   TASK_INIT_EXECUTION);
    uint32_t count_tmp = 0;
    while(count_tmp <  (TASK_DUMMY_ACTUATION_WORST_CASE*
   MS_COUNTS_DUMMY)){
        count_tmp++;
    }
    timestamp_runtime(TASK_IDENTIFIER_DUMMY_ACTUATION,
   TASK_END_EXECUTION);
    counter_tasks_runtime[TASK_IDENTIFIER_DUMMY_ACTUATION-1]++;
    uint32_t MileSecondsTask = ReadCounterMiliSeconds();
    uint32_t ValueTaskDelay;
    if(TASK_DUMMY_ACTUATION_PERIOD*counter_tasks_runtime[
   TASK_IDENTIFIER_DUMMY_ACTUATION-1] > MileSecondsTask){
        ValueTaskDelay = (TASK_DUMMY_ACTUATION_PERIOD*
   counter_tasks_runtime[TASK_IDENTIFIER_DUMMY_ACTUATION-1] -
   MileSecondsTask);
    }
    else{
        ValueTaskDelay = 0;
    }
    vTaskDelay(ValueTaskDelay);
    xTaskResumeAll();
}}
```

Listing 4.5 – Function Task Actuation.

The HLC task has a slight difference in that it includes a request to blink an LED. This serves as a watchdog, indicating to the developer that the system is still running. You can refer to the implementation of the Task Actuation function in Listing 4.5 for a reference on how this is implemented.

The last task is the system communication task, which is solely responsible for system telemetry and is the only task in the system that can be preempted. It also has a much longer execution period and the lowest priority among the tasks. In this example scenario, it is used to export data from the RM library to the offline verifier. It's worth noting that the RM library also monitors this task. In other words, the RM library includes RM monitors at the beginning and end of the task execution. You can see the implementation of the Task Communication function in Listing 4.6.

```
static void task_communication(void *pvParameters){
while(true)
{
if(ReadCounterMiliSeconds() < MS_INIT_USB_MILISECOND){
    continue;
}
timestamp_runtime(TASK_IDENTIFIER_COMMUNICATION,
    TASK_INIT_EXECUTION);
while(rmlib_export_data(&QueueTimeStampsBufferDumped) ==
    COMMAND_OK){
    rmlib_export_data_string(&QueueTimeStampsBufferDumped);
}
#ifdef EXPORT_DUMP_STATUS_TASKS
char * token = strtok(szList, "\n");
while(token != NULL){
    printf("%s\n", token); //printing each token
    token = strtok(NULL, "\n");
}
memset(&szList[0],0,size_buffer_export);
#endif
timestamp_runtime(TASK_IDENTIFIER_COMMUNICATION,
    TASK_END_EXECUTION);
counter_tasks_runtime[TASK_IDENTIFIER_COMMUNICATION-1]++;
#ifdef TASK_COMMUNCATION_PERIODIC
uint32_t MileSecondsTask = ReadCounterMiliSeconds();
uint32_t ValueTaskDelay;
if(TASK_COMMUNICATION_PERIOD*counter_tasks_runtime[
    TASK_IDENTIFIER_COMMUNICATION-1] > MileSecondsTask){
    ValueTaskDelay = ((TASK_COMMUNICATION_PERIOD*
    counter_tasks_runtime[TASK_IDENTIFIER_COMMUNICATION-1]) -
    MileSecondsTask);
}
else{
    ValueTaskDelay = 0;
}
vTaskDelay(ValueTaskDelay);
#else
vTaskDelay(10);
#endif
}}
```

Listing 4.6 – Function Task Communication.

In the experiment, two different execution configurations were used for the Communication Task. In one configuration, it executes periodically like the other tasks, allowing idle time for the processor. In the other configuration, it runs continuously in a preemptable manner, utilizing all the free time of the highest priority non-preemptive tasks for the Communication Task.

The aim of this experiment was to explore the system's behavior in scenarios where there is time slack for execution and scenarios where the Communication Task occupies all the free time as a preemptive task. By analyzing these scenarios, it was possible to observe how the system's priority tasks are affected when the Communication Task does not meet its deadline.

Furthermore, another experiment was conducted to introduce a new non-periodic task into the system. This task is triggered by pressing a button on the board and has a higher priority than the existing tasks in the system. This task represents a potential emergency or error situation that may occur during system execution, and it needs to be executed immediately. This experiment provided an opportunity to investigate the usage and monitoring of non-periodic tasks using the developed library.

The experiments conducted aimed to address the following issues:

1. Ensure that all tasks in the system are executed without missing their deadlines. This is important to assess the real-time performance of the system and validate the scheduling and timing properties.
2. Measure and analyze the execution times of the tasks. This helps in identifying any potential bottlenecks or performance issues and provides insights into the system's timing behavior.
3. Validate the two execution modes of the library: Online and Offline modes. This ensures that the library functions correctly and produces accurate results in both modes of operation.
4. Validate the scaling model of the system by examining the behavior of the Communication Task in periodic or continuous operation modes. This allows for the evaluation of the impact of different execution configurations on the system's overall performance and responsiveness.
5. Validate the system's behavior in the presence of an additional asynchronous task, which serves as a "disturbance" to the system. By using the library to analyze this scenario, it becomes possible to assess the system's robustness and its ability to handle unexpected events or task interactions.

By addressing these issues through the experiments, a comprehensive evaluation of the system's performance, timing properties, and response to various scenarios can be achieved.

4.3 RESULTS AND DISCUSSION

4.3.1 Offline Mode - Continuous Task Communication

In Appendix C.1, the monitors' analysis and related events with TimeStamps are presented for the critical instants that occur approximately every 120 ms.

The analysis involves examining the system events and checking the TimeStamps of task executions. The tasks are validated to ensure that they were executed within their WCET and deadline restrictions.

The verdict is determined based on the following criteria:

- If the difference (Delta) between Task Finish and Task Release Time is less than or equal to the Deadline, the verdict is True (T).
- If Delta exceeds the Deadline, the verdict is False (F).

Another Delta is calculated as the difference between task start and end times. If it is less than or equal to the WCET, the verdict is True (T). If it exceeds the WCET, the verdict is False (F).

In the experiment, the RM Offline Analysis tool receives the events through the serial interface and groups them based on TaskIdentifier. The events are stored in tuples of start and end TaskState events. Once both events are present, the verdict for that time interval is determined. The analysis involves evaluating the TimeStamps of the received events to measure the WCET and analyze the deadline.

In the offline verification mode, these calculations are performed externally to the system being analyzed. This is done to reduce the overhead of RM in the analyzed system, as searching for event couplings and performing calculations, even though seemingly simple, can consume significant processor execution time and system memory. For example, a complete cycle time window in the evaluation scenario generated approximately 200 verdicts, requiring around 400 events to be recorded.

In offline verification, the analysis of verdicts is performed later, and in the monitored system, only the storage of events takes place. This approach minimizes the additional execution time impact due to the use of the RM library. Tests were conducted to run a complete scheduling window of the system's time and verify the library's functionality and proper system scaling. Additionally, some tests were conducted to intentionally reproduce scenarios with adverse verdicts to validate the library's behavior under challenging conditions.

In the experiment, the Communication Task was executed in continuous mode, meaning it runs whenever other higher-priority and non-preemptive tasks are not executing. The Reporting Task is used in this case to utilize all of the processor's free time without affecting the proper functioning of higher-priority tasks. In cases where the Communication Task exceeds its WCET real-time constraints, it does not impact the higher-priority tasks.

Using the offline operation mode in this way is advantageous because it reduces the utilization of processors and memory for storing events. This mode handles the large amount of data generated by the extensive message export in a more efficient manner.

4.3.2 Online Mode - Task Communication Periodic

In Appendix D.1, the monitors' analysis and related events with TimeStamp Verdicts are presented for the critical instants that occur approximately every 120 ms.

The verdicts are determined by analyzing the system events and checking the TimeStamps of task executions. The tasks are validated to ensure that they were executed within their WCET and deadline restrictions.

The verdict is determined based on the following criteria:

- If the difference (Delta) between Task Finish and Task Release Time is less than or equal to the Deadline, the verdict is True (T).

- If Delta exceeds the Deadline, the verdict is False (F).

Another Delta is calculated as the difference between task start and end times. If it is less than or equal to the WCET, the verdict is True (T). If it exceeds the WCET, the verdict is False (F).

In this experiment using the RMLib Online, events are stored using TaskIdentifier in pairs of events (start and end TaskState events). Once both events are present, the verdict for that time interval is determined and stored in the verdicts queue. The analysis involves evaluating the TimeStamps of the events to measure the WCET and analyze the deadline.

Tests were conducted to run a complete scheduling window of the system's time to verify the library's functionality and proper system scaling. Additionally, some tests were conducted to intentionally reproduce scenarios with adverse verdicts for library validation.

In the experiment, the Communication Task was executed in periodic mode, where the task executed up to its WCET and started at intervals corresponding to its period. This execution model, introduced in the ProVANT project, has proven to be highly effective for the Online Verification Mode. In this mode, only a minimal amount of data needs to be exported, with the reports being the only exported data in this case.

The schedule modeling proposed in this project works well, and within a cycle, there are no violations of time constraints.

It is observed that the proposed scheduling model works correctly with the Communication Task being executed continuously or periodically. Due to its preemption configuration, the fact that this task is preemptive does not significantly affect the operation of the most critical tasks with high priorities in the system.

An interesting point of analysis is the Actuation Task, given the demand for periodic receipt of a command by the ESC. This task is critical because its time off must be very small (close to zero). By monitoring the system and analyzing the TimeStamps of the beginning and end of the task's execution in each execution cycle within its period (every 12 ms in the analysis interval of 120 ms), it can be determined if the characteristic of low time off for processing is respected. By analyzing Appendix D, it is observed that in most cases, the task is executed exactly at its release time due to its highest priority. In the worst case, there is a maximum blocking time of 3 ms, allowing the task to execute comfortably while respecting its WCET and deadline.

4.3.3 Online Mode Task Non-Periodic

In the test scenario being described, the goal is to thoroughly examine the operation of the proposed scaling model and assess its performance when faced with disturbances. The authors aim to create scenarios where temporal violations occur in the tasks modeled in the system due to an unforeseen and higher priority non-periodic task manually triggered by the user.

To achieve this, an extra non-periodic task is added to the system with a higher priority than all other tasks. This task has a TaskIdentifier of 6, a WCET of 8 ms, and a Deadline of 15 ms. It is executed periodically whenever a critical external event occurs, serving as a safeguard to prevent system crashes.

This test scenario serves multiple purposes. Firstly, it validates the monitoring of non-periodic tasks, which was not initially included in the proposed scheduling model. Secondly, it helps validate additional features developed in the RM library, such as the Online Verification feature of exporting only cases with RT constraints and the ability to export the status of tasks from the system.

In the provided Listing 4.7, the system output for the asynchronous task can be observed, despite the violation of temporal restrictions. The implementation includes the use of the ExportTaskStatus function, which utilizes the dumpTaskGetSystemState function of FreeRTOS [1] . This function populates a TaskStatus structure for each task in the system, containing information such as the task handle, task name, task priority, task state, and the total amount of run time consumed by the task.

By analyzing the output, it becomes apparent that the asynchronous task executed despite the temporal violation. This highlights the ability of the RM library to capture and analyze such cases.

```
{"TaskIdentifier":1,"TimeStamp":41442,"ExecutionTime":9,"
    CounterTask":8916,"StatusWCET":0,"StatusDeadline":1}
Dummy Actuation R 4 223 5
Asynchronous    Dummy Sensing   R 4 223 4
IDLE            R 0 237 7
Led             B 3 223 3
Communication   B 1 220 2
Controller      B 2 223 1
{"TaskIdentifier":1,"TimeStamp":55317,"ExecutionTime":7,"
    CounterTask":15535,"StatusWCET":0,"StatusDeadline":1}
Asynchronous    IDLE            R 0 237 7
Dummy Sensing   B 4 223 4
Dummy Actuation B 4 223 5
Led             B 3 223 3
Communication   B 1 166 2
Controller      B 2 83  1
```

Listing 4.7 – Data Exported in Temporal Restrictions Violation

[1] https://www.freertos.org/uxTaskGetSystemState.html

5 CONCLUSIONS

The main objective of this thesis was to develop a Library to Support Runtime Monitoring for the FreeRTOS, this work implemented this library seeking to have some characteristics that were considered important, as follows:

1. Runtime Monitoring Library for FreeRTOS.
2. Online and Offline Operation Modes, to have a more complete operation mode and a lighter mode.
3. Decouple specific RTOS implementations.
4. Support Tasks Periodics and Aperiodics, Support to monitoring Deadline, Support to monitoring WCET, in Tasks Non Preemptives.

It was also developed an example project to validate the operation and behavior of the library in practical use. The validation was carried out considering a real example scenario from the ProVANT project.

Analyzing the results, it was possible to satisfactorily evaluate the behavior of the scaling model proposed in the ProVANT project that was implemented in the example project. It was also possible to observe the facilities and resources that the library can add in the process of developing and monitoring embedded software projects real-time.

5.1 FUTURE WORKS

The developed *Runtime Monitoring* library was able to collect events of interest and analyze them. However, currently, the developed monitor is limited as it can only reason about missed deadlines and WCET compliance. As discussed in (PEDRO et al., 2018), making it more generic is for future work.

An important feature for future implementations is the development of support for System Hooks, to identify some RTOS operations, such as inserting the task in the queue of available tasks for execution and preempting tasks for scheduling. The implementation of these new functions will make the collection of information more "accurate" and will support adding the monitoring of non-periodic and preemptive tasks to the RM library.

However, this implementation is highly linked to the RTOS architecture, so it would be necessary to implement the RM library's support for these features and also to implement Hooks, with FreeRTOS, to validate the library's features and RM in the example of the project.

Another important resource for the complete RV cycle is the monitors' synthesis, to integrate the RM library with verification systems, which from input models automatically synthesized the monitors to be verified in runtime.

REFERENCES

ALUR, Rajeev. Formal verification of hybrid systems. In: PROCEEDINGS of the ninth ACM international conference on Embedded software. [S.l.: s.n.], 2011. P. 273–278. Cit. on p. 34.

ALUR, Rajeev; DILL, David. The theory of timed automata. In: SPRINGER. REAL-TIME: Theory in Practice: REX Workshop Mook, The Netherlands, June 3–7, 1991 Proceedings. [S.l.: s.n.], 1992. P. 45–73. Cit. on p. 35.

BAIER, Christel; KATOEN, Joost-Pieter. **Principles of model checking**. [S.l.]: MIT press, 2008. Cit. on p. 34.

BARR, Michael. **Programming embedded systems in C and C++**. [S.l.]: " O'Reilly Media, Inc.", 1999. Cit. on p. 25.

BJESSE, Per. What is formal verification? **ACM SIGDA Newsletter**, ACM New York, NY, USA, v. 35, n. 24, 1–es, 2005. Cit. on p. 33.

BROERING, Elton Ferreira. **RunTIme Verification Library - RTMLib**. RTMLib. Aug. 2022. Available from: <`https://github.com/EltonBroering/RTMLib`>. Cit. on pp. 42, 51.

BROERING, Elton Ferreira; BECKER, Leandro Buss. Applying Runtime Verification in Real-Time Systems with FreeRTOS. In: IEEE. 2022 XII Brazilian Symposium on Computing Systems Engineering (SBESC). [S.l.: s.n.], 2022. P. 1–6. Cit. on p. 28.

BROY, Manfred; CENGARLE, Maria Victoria; GEISBERGER, Eva. Cyber-physical systems: imminent challenges. In: SPRINGER. MONTEREY workshop. [S.l.: s.n.], 2012. P. 1–28. Cit. on p. 25.

CLARKE, Edmund M; EMERSON, E Allen. Design and synthesis of synchronization skeletons using branching time temporal logic. In: SPRINGER. WORKSHOP on logic of programs. [S.l.: s.n.], 1981. P. 52–71. Cit. on p. 37.

CORTÉS, Luis Alejandro; ELES, Petru; PENG, Zebo. Modeling and formal verification of embedded systems based on a Petri net representation. **Journal of Systems Architecture**, Elsevier, v. 49, n. 12-15, p. 571–598, 2003. Cit. on p. 34.

DAVIS, Robert I; BURNS, Alan. A survey of hard real-time scheduling for multiprocessor systems. **ACM computing surveys (CSUR)**, ACM New York, NY, USA, v. 43, n. 4, p. 1–44, 2011. Cit. on p. 31.

DONADEL, Rodrigo et al. Modeling and control of a tiltrotor unmanned aerial vehicle for path tracking, 2015. Cit. on p. 56.

DRECHSLER, Rolf et al. **Advanced formal verification**. [S.l.]: Springer, 2004. v. 122. Cit. on pp. 33, 34.

EDITION, NXP LPC17xx. Using the FreeRTOS™ Real Time Kernel. Cit. on p. 40.

FALCONE, Yliés; HAVELUND, Klaus; REGER, Giles. A tutorial on runtime verification. **Engineering dependable software systems**, IOS Press, p. 141–175, 2013. Cit. on p. 26.

FEILER, Peter H; GLUCH, David P; HUDAK, John J. **The architecture analysis & design language (AADL): An introduction**. [S.l.], 2006. Cit. on p. 27.

GONÇALVES, Fernando. **Integrated method for designing complex cyber-physical systems**. 2018. PhD thesis – Universidade Federal de Santa Catarina. Cit. on pp. 26, 27, 37.

GONÇALVES, Fernando S; RAFFO, Guilherme V; BECKER, Leandro B. Managing cps complexity: Design method for unmanned aerial vehicles. **IFAC-PapersOnLine**, Elsevier, v. 49, n. 32, p. 141–146, 2016. Cit. on p. 36.

GONÇALVES, Fernando Silvano et al. Projeto da arquitetura de software embarcado de um veículo aéreo não tripulado, 2014. Cit. on p. 36.

GOODLOE, Alwyn E; PIKE, Lee. **Monitoring distributed real-time systems: A survey and future directions**. [S.l.], 2010. Cit. on p. 35.

GRUMBERG, Orna; CLARKE, EM; PELED, DA. **Model checking**. [S.l.]: MIT press Cambridge, 1999. Cit. on p. 37.

HAIGH, Karen Zita et al. Machine learning for embedded systems: A case study. **BBN Technologies: Cambridge, MA, USA**, v. 8571, p. 1–12, 2015. Cit. on p. 30.

HAMLET, Dick. **Composing Software Components: A Software-testing Perspective**. [S.l.]: Springer Science & Business Media, 2010. Cit. on p. 36.

KNIGHT, John C. Safety critical systems: challenges and directions. In: PROCEEDINGS of the 24th international conference on software engineering. [S.l.: s.n.], 2002. P. 547–550. Cit. on p. 25.

KOCHER, Paul et al. Security as a new dimension in embedded system design. In: PROCEEDINGS of the 41st annual design automation conference. [S.l.: s.n.], 2004. P. 753–760. Cit. on p. 30.

LAL, Sanjay. Bare-Metal Systems. In: REAL World Multicore Embedded Systems. [S.l.]: Elsevier, 2013. P. 517–560. Cit. on p. 27.

LEE, Edward Ashford; SESHIA, Sanjit Arunkumar. **Introduction to embedded systems: A cyber-physical systems approach**. [S.l.]: Mit Press, 2016. Cit. on pp. 29, 31, 34.

LEUCKER, Martin; SCHALLHART, Christian. A brief account of runtime verification. **The Journal of Logic and Algebraic Programming**, Elsevier, v. 78, n. 5, p. 293–303, 2009. Cit. on p. 26.

MALL, Rajib. **Real-time systems: theory and practice**. [S.l.]: Pearson Education India, 2009. Cit. on p. 31.

MARWEDEL, Peter. **Embedded system design: embedded systems foundations of cyber-physical systems, and the internet of things**. [S.l.]: Springer Nature, 2021. Cit. on pp. 29–31.

MATOS PEDRO, André de. **Dynamic contracts for verification and enforcement of real-time systems properties**. 2018. PhD thesis – Citeseer. Cit. on p. 36.

MEIER, Lorenz; HONEGGER, Dominik; POLLEFEYS, Marc. PX4: A node-based multithreaded open source robotics framework for deeply embedded platforms. In: IEEE. 2015 IEEE international conference on robotics and automation (ICRA). [S.l.: s.n.], 2015. P. 6235–6240. Cit. on pp. 11, 27, 38.

MICROCHIP. **SAM4E XPLAINED PRO**. Part Number: ATSAM4E-XPRO SAM4E XPLAINED PRO. Aug. 2022. Available from: <https://www.microchip.com/en-us/development-tool/ATSAM4E-XPRO>. Cit. on p. 51.

MISSON, Henrique Amaral et al. Applying formal verification techniques to embedded software in UAV design, 2019. Cit. on pp. 27, 36, 37, 39, 40, 55.

MIZOTANI, Keigo et al. An integration of imprecise computation model and real-time voltage and frequency scaling. In: PROCEEDINGS of the 30th International Conference on Computers and Their Applications (CATA'15). [S.l.: s.n.], 2015. P. 63–70. Cit. on pp. 26, 39.

OLIVEIRA, Daniel Bristot de et al. Demystifying the Real-Time Linux Scheduling Latency. In: SCHLOSS DAGSTUHL-LEIBNIZ-ZENTRUM FUR INFORMATIK. 32ND Euromicro Conference on Real-Time Systems (ECRTS 2020). [S.l.: s.n.], 2020. Cit. on p. 25.

OLIVEIRA, Rômulo Silva de. **Fundamentos dos Sistemas de Tempo Real**. [S.l.]: Edição do Autor, 2020. Cit. on p. 32.

ONEM, Evren; GURDAG, A Burak; CAGLAYAN, M Ufuk. Secure routing in ad hoc networks and model checking? In: TRAFFORD PUBLISHING. PROCEEDINGS of the First International Conference on Security of Information and Networks. [S.l.: s.n.], 2008. P. 346. Cit. on p. 25.

PEDRO, André et al. Runtime verification of autopilot systems using a fragment of MTL-$\int$. **International Journal on Software Tools for Technology Transfer**, Springer Berlin Heidelberg, v. 20, n. 4, p. 379–395, 2018. Cit. on pp. 11, 27, 35, 38–40, 43, 63.

RAJKUMAR, Ragunathan et al. Cyber-physical systems: the next computing revolution. In: IEEE. DESIGN automation conference. [S.l.: s.n.], 2010. P. 731–736. Cit. on p. 25.

REINBACHER, Thomas; FUGGER, Matthias; BRAUER, Jorg. Runtime verification of embedded real-time systems. **Formal methods in system design**, Springer, v. 44, n. 3, p. 203–239, 2014. Cit. on p. 26.

ROZIER, Kristin Y; SCHUMANN, Johann. R2U2: tool overview. In: ARC-E-DAA-TN48849. INTERNATIONAL Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools. [S.l.: s.n.], 2017. Cit. on p. 40.

S TANENBAUM, ANDREW; BOS, Herbert. **Modern operating systems**. [S.l.: s.n.], 2015. Cit. on p. 30.

SICILIANO, Bruno; KHATIB, Oussama; KRÖGER, Torsten. **Springer handbook of robotics**. [S.l.]: Springer, 2008. v. 200. Cit. on p. 30.

STANKOVIC, John A. Misconceptions about real-time computing: A serious problem for next-generation systems. **Computer**, IEEE, v. 21, n. 10, p. 10–19, 1988. Cit. on pp. 25, 31.

STOJMENOVIC, Ivan. Machine-to-machine communications with in-network data aggregation, processing, and actuation for large-scale cyber-physical systems. **IEEE Internet of Things Journal**, IEEE, v. 1, n. 2, p. 122–128, 2014. Cit. on p. 25.

TAKADA, H. Real-time operating system for embedded systems. **Tutorial**, 2001. Cit. on p. 29.

THE LINUX KERNEL ORGANIZATION. **Runtime Verification Documentation**. 2023. Available from: <https://docs.kernel.org/trace/rv/runtime-verification.html>. Visited on: 23 June 2023. Cit. on p. 40.

ZURAWSKI, Richard. **Embedded Systems Handbook 2-Volume Set**. [S.l.]: CRC press, 2018. Cit. on p. 30.

Appendix

APPENDIX A – OFFLINE MODE - IMPLEMENTATION OF RMLIB

```c
typedef struct PACKED
{
     uint32_t     TimeStamp;
     uint32_t     CounterTask : 25;
     uint8_t      Identifier _of _Task : 5;
     uint8_t      State _of _Task : 2;
} EventTimeStamp_t;
```

Listing A.1 – Data Structure used to store events in Offline Mode.

```c
#define NUMBER_TASKS_RUNTIME_MONITORING 5
```

Listing A.2 – Definition of Number Tasks of RM library in Offline Mode.

```c
#define SIZE_RUN_TIME_BUFFER_QUEUE 1000
Event_t QueueEventsBuffer[SIZE_RUN_TIME_BUFFER_QUEUE];
void rmlib_init()
{
     cb_init(&QueueEvents,&QueueEventsBuffer[0],(size_t)
    SIZE_RUN_TIME_BUFFER_QUEUE,(size_t)sizeof(Event_t));
}
```

Listing A.3 – Initialization function of the RM library in Offline Mode.

```c
extern uint32_t ReadCounterMiliSeconds(void);
```

Listing A.4 – Definition of ReadCounter Function in MilliSeconds.

```c
int8_t timestamp_runtime(uint32_t task_identifier,uint16_t
    task_state)
{
     EventInsert.Identifier_of_Task = task_identifier;
     EventInsert.State_of_Task = task_state;
     EventInsert.TimeStamp = ReadCounterMiliSeconds();
     return cb_push_back(&QueueEvents,&EventInsert);
}
```

Listing A.5 – Function to collect TimeStamps in RunTime while in Offline Mode.

```c
int8_t rmlib_export_data(Event_t * buffer_rmlib)
{
  return cb_pop_front(&QueueEvents, buffer_rmlib);
}
const char rmlib_export_data_string(EventTimeStamp_t * buffer_rmlib)
{
  #ifdef COMPACT_EXPORT_DATA
   printf("I%d-S%d-T%d-C%d\n", buffer_rmlib->Identifier_of_Task, buffer_rmlib->State_of_Task, buffer_rmlib->TimeStamp, buffer_rmlib->CounterTask);
  #else
   printf("{\"TaskIdentifier\" : %d,\"TaskState\" : %d,\"TimeStamp\" : %d,\"TaskCounter\" : %d}\n", buffer_rmlib->Identifier_of_Task, buffer_rmlib->State_of_Task, buffer_rmlib->TimeStamp, buffer_rmlib->CounterTask);
  #endif
  return;
}
```

Listing A.6 – Functions to export Events while in Offline Mode.

```python
import serial
import json
import time

while True:
    try:
        ser = serial.Serial(serial_interface,576000,timeout=30,rtscts=1) # open serial port
        print("Interface Serial : " + ser.name)          # check which port was really used
        break
    except:
        print("Error in open serial")
        time.sleep(0.01)

vector_deadlines = [12,12,12,120,600]
vector_period = [12,12,12,120,600]
vector_wcet = [2,3,4,4,135]
vector_itens_analyzed = {}
serial_interface    = '/dev/ttyACM0';

def analyze_data(item):
  if item["TaskState"] == 1:
    if item["TaskIdentifier"] in vector_itens_analyzed:
      vector_itens_analyzed[item["TaskIdentifier"]].append(item)
  else:
    vector_itens_analyzed[item["TaskIdentifier"]] = [item]
  if item["TaskIdentifier"] in vector_itens_analyzed:
    if len(vector_itens_analyzed[item["TaskIdentifier"]]) == 2:
      it = vector_itens_analyzed.pop(item["TaskIdentifier"])
      if(len(it) > 1):
        print(it)
        if (it[1]["TimeStamp"] - it[0]["TimeStamp"]) <= vector_wcet[it[0]["TaskIdentifier"]-1]:
          print("Task " + str(it[0]["TaskIdentifier"]) + " Condition WCET: T - WCET Time: " + str((it[1]["TimeStamp"] - it[0]["TimeStamp"])))
        else:
          print("Task " + str(it[0]["TaskIdentifier"]) + " Condition WCET: F - WCET Time: " + str((it[1]["TimeStamp"] - it[0]["TimeStamp"])))
        if (vector_period[it[0]["TaskIdentifier"]-1] == 0):
          if((vector_wcet[it[1]["TaskIdentifier"]-1]) >= it[1]["TimeStamp"]):
            print("Task " + str(it[0]["TaskIdentifier"]) + " Condition DeadLine: T")
          else:
            print("Task " + str(it[0]["TaskIdentifier"]) + " Condition DeadLine: F")
        else:
          if ((it[1]["TaskCounter"]*vector_period[it[1]["TaskIdentifier"]-1]) >= it[1]["TimeStamp"]):
            print("Task " + str(it[0]["TaskIdentifier"]) + " Condition DeadLine: T")
          else:
            print("Task " + str(it[0]["TaskIdentifier"]) + " Condition DeadLine: F")
        print("Task " + str(it[0]["TaskIdentifier"]) + " TimeStamp: " + str(it[1]["TimeStamp"]) + " DeadLine: " + str(it[1]["TaskCounter"]*vector_deadlines[it[1]["TaskIdentifier"]-1]))

def process_compact_data(item):
  dict_tmp = {}
  items = item.strip().split("-")
  dict_tmp["TaskIdentifier"]  = int(items[0].split("I")[1])
  dict_tmp["TaskState"]       = int(items[1].split("S")[1])
  dict_tmp["TimeStamp"]       = int(items[2].split("T")[1])
  dict_tmp["TaskCounter"]     = int(items[3].split("C")[1])
  return dict_tmp

def process_data(item):
  if(item[0] == "I"):
    return process_compact_data(item)
  else:
    return json.loads(item)

```

```
while True:
  try:
    line = str(ser.readline(),"ISO-8859-1")
    item = process_data(line)
    analyze_data(item)
  except json.decoder.JSONDecodeError:
    print("Error in json decode -> " + str(line))
```

Listing A.7 – RV Offline Analyzer

APPENDIX B – ONLINE MODE - IMPLEMENTATION OF RMLIB

```c
//TimeStamp Veredict of system RunTime Verification
typedef struct PACKED {
  uint16_t     TimeStamp;
  uint16_t     CounterTask;
  uint16_t     ExecutionTime;
  uint8_t      Identifier_of_Task : 5;
  uint8_t      Status_of_WCET_Task : 2;
  uint8_t      Status_of_DeadLine_Task : 2;
  uint8_t      ReservedBits : 7;
} TimeStampVeredict_t;
```

Listing B.1 – Data Structure used to store TimeStamp Veredicts.

```c
#define NUMBER_TASKS_RUNTIME_MONITORING 5
```

Listing B.2 – Definition of Number Tasks of RM library in Online Mode.

```c
#define SIZE_RUN_TIME_BUFFER_QUEUE 1000
TimeStampVeredict_t       QueueTimeStampVerdictsBuffer[
    SIZE_RUN_TIME_BUFFER_QUEUE];
void rmlib_init(uint32_t * tasks_identifiers,uint32_t *
     deadlines_service,uint32_t * period_service,uint32_t *
     wcet_service){
  cb_init(&QueueTimeStamps,&QueueTimeStampVerdictsBuffer[0],(
     size_t)SIZE_RUN_TIME_BUFFER_QUEUE,(size_t)sizeof(
     TimeStampVeredict_t));
  memset(&counter_tasks_runtime_verification,0x00,sizeof(
     uint16_t)*NUMBER_TASKS_RUNTIME_VERIFICATION);
  memcpy(&Identifiers_Tasks,tasks_identifiers,
    NUMBER_TASKS_RUNTIME_VERIFICATION * sizeof(uint32_t));
  memcpy(&Vector_WCET_Tasks,wcet_service,
    NUMBER_TASKS_RUNTIME_VERIFICATION * sizeof(uint32_t));
  memcpy(&Vector_Deadline_Tasks,deadlines_service,
    NUMBER_TASKS_RUNTIME_VERIFICATION * sizeof(uint32_t));
  memcpy(&Vector_Period_Tasks,period_service,
    NUMBER_TASKS_RUNTIME_VERIFICATION * sizeof(uint32_t));
}
```

Listing B.3 – Initialization function of the RV library in Online Mode.

```c
extern uint32_t ReadCounterMiliSeconds(void);
```

Listing B.4 – Definition of Function ReadCounter in MilliSeconds.

```c
int8_t timestamp_runtime(uint32_t task_identifier,uint16_t task_state) {
  uint8_t task_index = 0;
  for(uint8_t number_task = 0;  number_task <
    NUMBER_TASKS_RUNTIME_VERIFICATION; number_task++) {
    if(Identifiers_Tasks[number_task] == task_identifier) {
      task_index = number_task;
      break;
    }
  }
  TimeStampsBufferProcessing[task_index][task_state].Identifier_of_Task
    = task_identifier;
  TimeStampsBufferProcessing[task_index][task_state].State_of_Task =
    task_state;
  TimeStampsBufferProcessing[task_index][task_state].TimeStamp =
    ReadCounterMiliSeconds();
  if(task_state == TASK_END_EXECUTION) {
    counter_tasks_runtime_verification[task_index]++;
  }
  TimeStampsBufferProcessing[task_index][task_state].CounterTask =
    counter_tasks_runtime_verification[task_index];
  if(task_state == TASK_END_EXECUTION) {
    TimeStampInsert.TimeStamp = TimeStampsBufferProcessing[task_index][
    TASK_INIT_EXECUTION].TimeStamp;
    TimeStampInsert.ExecutionTime = (TimeStampsBufferProcessing[
    task_index][TASK_END_EXECUTION].TimeStamp -
    TimeStampsBufferProcessing[task_index][TASK_INIT_EXECUTION].TimeStamp
    );
    TimeStampInsert.CounterTask = TimeStampsBufferProcessing[task_index
    ][TASK_END_EXECUTION].CounterTask;
    TimeStampInsert.Identifier_of_Task = Identifiers_Tasks[task_index];
    if(TimeStampInsert.ExecutionTime <= Vector_WCET_Tasks[task_index]) {
      TimeStampInsert.Status_of_WCET_Task = true;
    }
    else {
      TimeStampInsert.Status_of_WCET_Task = false;
    }
    if(Vector_Period_Tasks[task_index] == 0){
      if((TimeStampInsert.ExecutionTime + TimeStampInsert.TimeStamp) <=
    Vector_Deadline_Tasks[task_index]){
        TimeStampInsert.Status_of_DeadLine_Task = true;
      }
      else{
        TimeStampInsert.Status_of_DeadLine_Task = false;
      }
    }
    else{
      if((TimeStampInsert.ExecutionTime + TimeStampInsert.TimeStamp) <=
    (Vector_Period_Tasks[task_index] * TimeStampInsert.CounterTask)){
        TimeStampInsert.Status_of_DeadLine_Task = true;
      }
      else{
        TimeStampInsert.Status_of_DeadLine_Task = false;
        }
      }
      #ifdef EXPORT_ONLY_RTOS_ERRORS
      if(!TimeStampInsert.Status_of_DeadLine_Task || !TimeStampInsert.
     Status_of_WCET_Task) {
        #ifdef EXPORT_DUMP_STATUS_TASKS
        DumpStatusTasks();
        #endif
        return cb_push_back(&QueueTimeStamps,&TimeStampInsert);
      }
      else {
        return COMMAND_OK;
      }
      #else
      return cb_push_back(&QueueTimeStamps,&TimeStampInsert);
      #endif
    }
    else {
      return COMMAND_OK;
    }
}
```

Listing B.5 – Function to collect TimeStamps in RunTime while in Onine Mode.

```
int8_t rmlib_export_data(TimeStampVeredict_t * buffer_rmlib) {
  return cb_pop_front(&QueueTimeStamps,buffer_rmlib);
}
const char rmlib_export_data_string(TimeStampVeredict_t *
    buffer_rmlib) {
  #ifdef COMPACT_EXPORT_DATA
  printf("IT%d-TS%d-ET%d-CT%d-SW%d-SD%d\n", buffer_rmlib->
    Identifier_of_Task, buffer_rmlib->TimeStamp, buffer_rmlib->
    ExecutionTime, buffer_rmlib->CounterTask, buffer_rmlib->
    Status_of_WCET_Task, buffer_rmlib->Status_of_DeadLine_Task);
  #else
  printf("{\"TaskIdentifier\":%d,\"TimeStamp\":%d,\"
    ExecutionTime\":%d,\"CounterTask\":%d,\"StatusWCET\":%d,\"
    StatusDeadline\":%d}\n", buffer_rmlib->Identifier_of_Task,
    buffer_rmlib->TimeStamp, buffer_rmlib->ExecutionTime,
    buffer_rmlib->CounterTask, buffer_rmlib->Status_of_WCET_Task,
     buffer_rmlib->Status_of_DeadLine_Task);
  #endif
  return;
}
```

Listing B.6 – Functions to export Events Online Mode.

APPENDIX C – OUTPUT DATA RTMLIB OFFLINE MODE TASK COMMUNICATION CONTINUOUSLY

```
[{'TaskIdentifier': 1, 'TaskState': 0, 'TimeStamp': 0, 'TaskCounter': 0}, {'TaskIdentifier': 1, 'TaskState': 1, 'TimeStamp': 1, 'TaskCounter': 1}]
Task 1 Condition WCET: T - WCET Time: 1
Task 1 Condition DeadLine: T
Task 1 TimeStamp: 1 DeadLine: 12
[{'TaskIdentifier': 2, 'TaskState': 0, 'TimeStamp': 1, 'TaskCounter': 0}, {'TaskIdentifier': 2, 'TaskState': 1, 'TimeStamp': 4, 'TaskCounter': 1}]
Task 2 Condition WCET: T - WCET Time: 3
Task 2 Condition DeadLine: T
Task 2 TimeStamp: 4 DeadLine: 12
[{'TaskIdentifier': 4, 'TaskState': 0, 'TimeStamp': 4, 'TaskCounter': 0}, {'TaskIdentifier': 4, 'TaskState': 1, 'TimeStamp': 7, 'TaskCounter': 1}]
Task 4 Condition WCET: T - WCET Time: 3
Task 4 Condition DeadLine: T
Task 4 TimeStamp: 7 DeadLine: 120
[{'TaskIdentifier': 3, 'TaskState': 0, 'TimeStamp': 7, 'TaskCounter': 0}, {'TaskIdentifier': 3, 'TaskState': 1, 'TimeStamp': 11, 'TaskCounter': 1}]
Task 3 Condition WCET: T - WCET Time: 4
Task 3 Condition DeadLine: T
Task 3 TimeStamp: 11 DeadLine: 12
[{'TaskIdentifier': 2, 'TaskState': 0, 'TimeStamp': 11, 'TaskCounter': 1}, {'TaskIdentifier': 2, 'TaskState': 1, 'TimeStamp': 13, 'TaskCounter': 2}]
Task 2 Condition WCET: T - WCET Time: 2
Task 2 Condition DeadLine: T
Task 2 TimeStamp: 13 DeadLine: 24
[{'TaskIdentifier': 1, 'TaskState': 0, 'TimeStamp': 13, 'TaskCounter': 1}, {'TaskIdentifier': 1, 'TaskState': 1, 'TimeStamp': 15, 'TaskCounter': 2}]
Task 1 Condition WCET: T - WCET Time: 2
Task 1 Condition DeadLine: T
Task 1 TimeStamp: 15 DeadLine: 24
[{'TaskIdentifier': 3, 'TaskState': 0, 'TimeStamp': 15, 'TaskCounter': 1}, {'TaskIdentifier': 3, 'TaskState': 1, 'TimeStamp': 18, 'TaskCounter': 2}]
Task 3 Condition WCET: T - WCET Time: 3
Task 3 Condition DeadLine: T
Task 3 TimeStamp: 18 DeadLine: 24
[{'TaskIdentifier': 3, 'TaskState': 0, 'TimeStamp': 20, 'TaskCounter': 2}, {'TaskIdentifier': 3, 'TaskState': 1, 'TimeStamp': 23, 'TaskCounter': 3}]
```

```
Task 3 Condition WCET: T — WCET Time: 3
Task 3 Condition DeadLine: T
Task 3 TimeStamp: 23 DeadLine: 36
[{'TaskIdentifier': 2, 'TaskState': 0, 'TimeStamp': 23, 'TaskCounter':
    2}, {'TaskIdentifier': 2, 'TaskState': 1, 'TimeStamp': 26, '
    TaskCounter': 3}]
Task 2 Condition WCET: T — WCET Time: 3
Task 2 Condition DeadLine: T
Task 2 TimeStamp: 26 DeadLine: 36
[{'TaskIdentifier': 1, 'TaskState': 0, 'TimeStamp': 26, 'TaskCounter':
    2}, {'TaskIdentifier': 1, 'TaskState': 1, 'TimeStamp': 27, '
    TaskCounter': 3}]
Task 1 Condition WCET: T — WCET Time: 1
Task 1 Condition DeadLine: T
Task 1 TimeStamp: 27 DeadLine: 36
[{'TaskIdentifier': 2, 'TaskState': 0, 'TimeStamp': 33, 'TaskCounter':
    3}, {'TaskIdentifier': 2, 'TaskState': 1, 'TimeStamp': 35, '
    TaskCounter': 4}]
Task 2 Condition WCET: T — WCET Time: 2
Task 2 Condition DeadLine: T
Task 2 TimeStamp: 35 DeadLine: 48
[{'TaskIdentifier': 1, 'TaskState': 0, 'TimeStamp': 35, 'TaskCounter':
    3}, {'TaskIdentifier': 1, 'TaskState': 1, 'TimeStamp': 37, '
    TaskCounter': 4}]
Task 1 Condition WCET: T — WCET Time: 2
Task 1 Condition DeadLine: T
Task 1 TimeStamp: 37 DeadLine: 48
[{'TaskIdentifier': 3, 'TaskState': 0, 'TimeStamp': 37, 'TaskCounter':
    3}, {'TaskIdentifier': 3, 'TaskState': 1, 'TimeStamp': 40, '
    TaskCounter': 4}]
Task 3 Condition WCET: T — WCET Time: 3
Task 3 Condition DeadLine: T
Task 3 TimeStamp: 40 DeadLine: 48
[{'TaskIdentifier': 3, 'TaskState': 0, 'TimeStamp': 45, 'TaskCounter':
    4}, {'TaskIdentifier': 3, 'TaskState': 1, 'TimeStamp': 48, '
    TaskCounter': 5}]
Task 3 Condition WCET: T — WCET Time: 3
Task 3 Condition DeadLine: T
Task 3 TimeStamp: 48 DeadLine: 60
[{'TaskIdentifier': 2, 'TaskState': 0, 'TimeStamp': 48, 'TaskCounter':
    4}, {'TaskIdentifier': 2, 'TaskState': 1, 'TimeStamp': 51, '
    TaskCounter': 5}]
Task 2 Condition WCET: T — WCET Time: 3
Task 2 Condition DeadLine: T
Task 2 TimeStamp: 51 DeadLine: 60
[{'TaskIdentifier': 1, 'TaskState': 0, 'TimeStamp': 51, 'TaskCounter':
    4}, {'TaskIdentifier': 1, 'TaskState': 1, 'TimeStamp': 52, '
    TaskCounter': 5}]
Task 1 Condition WCET: T — WCET Time: 1
Task 1 Condition DeadLine: T
Task 1 TimeStamp: 52 DeadLine: 60
[{'TaskIdentifier': 2, 'TaskState': 0, 'TimeStamp': 57, 'TaskCounter':
    5}, {'TaskIdentifier': 2, 'TaskState': 1, 'TimeStamp': 59, '
    TaskCounter': 6}]
Task 2 Condition WCET: T - WCET Time: 2
Task 2 Condition DeadLine: T
Task 2 TimeStamp: 59 DeadLine: 72
[{'TaskIdentifier': 1, 'TaskState': 0, 'TimeStamp': 59, 'TaskCounter':
    5}, {'TaskIdentifier': 1, 'TaskState': 1, 'TimeStamp': 61, '
    TaskCounter': 6}]
Task 1 Condition WCET: T - WCET Time: 2
Task 1 Condition DeadLine: T
Task 1 TimeStamp: 61 DeadLine: 72
[{'TaskIdentifier': 3, 'TaskState': 0, 'TimeStamp': 61, 'TaskCounter':
    5}, {'TaskIdentifier': 3, 'TaskState': 1, 'TimeStamp': 64, '
    TaskCounter': 6}]
Task 3 Condition WCET: T - WCET Time: 3
Task 3 Condition DeadLine: T
Task 3 TimeStamp: 64 DeadLine: 72
[{'TaskIdentifier': 3, 'TaskState': 0, 'TimeStamp': 69, 'TaskCounter':
    6}, {'TaskIdentifier': 3, 'TaskState': 1, 'TimeStamp': 72, '
    TaskCounter': 7}]
Task 3 Condition WCET: T - WCET Time: 3
Task 3 Condition DeadLine: T
Task 3 TimeStamp: 72 DeadLine: 84
[{'TaskIdentifier': 2, 'TaskState': 0, 'TimeStamp': 72, 'TaskCounter':
    6}, {'TaskIdentifier': 2, 'TaskState': 1, 'TimeStamp': 75, '
    TaskCounter': 7}]
Task 2 Condition WCET: T - WCET Time: 3
Task 2 Condition DeadLine: T
Task 2 TimeStamp: 75 DeadLine: 84
[{'TaskIdentifier': 1, 'TaskState': 0, 'TimeStamp': 75, 'TaskCounter':
    6}, {'TaskIdentifier': 1, 'TaskState': 1, 'TimeStamp': 76, '
    TaskCounter': 7}]
Task 1 Condition WCET: T - WCET Time: 1
Task 1 Condition DeadLine: T
Task 1 TimeStamp: 76 DeadLine: 84
[{'TaskIdentifier': 2, 'TaskState': 0, 'TimeStamp': 81, 'TaskCounter':
    7}, {'TaskIdentifier': 2, 'TaskState': 1, 'TimeStamp': 83, '
    TaskCounter': 8}]
Task 2 Condition WCET: T - WCET Time: 2
Task 2 Condition DeadLine: T
Task 2 TimeStamp: 83 DeadLine: 96
[{'TaskIdentifier': 1, 'TaskState': 0, 'TimeStamp': 83, 'TaskCounter':
    7}, {'TaskIdentifier': 1, 'TaskState': 1, 'TimeStamp': 85, '
    TaskCounter': 8}]
Task 1 Condition WCET: T - WCET Time: 2
Task 1 Condition DeadLine: T
Task 1 TimeStamp: 85 DeadLine: 96
[{'TaskIdentifier': 3, 'TaskState': 0, 'TimeStamp': 85, 'TaskCounter':
    7}, {'TaskIdentifier': 3, 'TaskState': 1, 'TimeStamp': 88, '
    TaskCounter': 8}]
Task 3 Condition WCET: T — WCET Time: 3
Task 3 Condition DeadLine: T
Task 3 TimeStamp: 88 DeadLine: 96
[{'TaskIdentifier': 3, 'TaskState': 0, 'TimeStamp': 93, 'TaskCounter':
    8}, {'TaskIdentifier': 3, 'TaskState': 1, 'TimeStamp': 96, '
    TaskCounter': 9}]
Task 3 Condition WCET: T — WCET Time: 3
Task 3 Condition DeadLine: T
Task 3 TimeStamp: 96 DeadLine: 108
[{'TaskIdentifier': 2, 'TaskState': 0, 'TimeStamp': 96, 'TaskCounter':
    8}, {'TaskIdentifier': 2, 'TaskState': 1, 'TimeStamp': 99, '
    TaskCounter': 9}]
Task 2 Condition WCET: T — WCET Time: 3
Task 2 Condition DeadLine: T
Task 2 TimeStamp: 99 DeadLine: 108
[{'TaskIdentifier': 1, 'TaskState': 0, 'TimeStamp': 99, 'TaskCounter':
    8}, {'TaskIdentifier': 1, 'TaskState': 1, 'TimeStamp': 100, '
    TaskCounter': 9}]
Task 1 Condition WCET: T — WCET Time: 1
Task 1 Condition DeadLine: T
Task 1 TimeStamp: 100 DeadLine: 108
[{'TaskIdentifier': 2, 'TaskState': 0, 'TimeStamp': 105, 'TaskCounter':
    9}, {'TaskIdentifier': 2, 'TaskState': 1, 'TimeStamp': 107, '
    TaskCounter': 10}]
Task 2 Condition WCET: T — WCET Time: 2
Task 2 Condition DeadLine: T
Task 2 TimeStamp: 107 DeadLine: 120
[{'TaskIdentifier': 1, 'TaskState': 0, 'TimeStamp': 107, 'TaskCounter':
    9}, {'TaskIdentifier': 1, 'TaskState': 1, 'TimeStamp': 109, '
    TaskCounter': 10}]
Task 1 Condition WCET: T — WCET Time: 2
Task 1 Condition DeadLine: T
Task 1 TimeStamp: 109 DeadLine: 120
[{'TaskIdentifier': 3, 'TaskState': 0, 'TimeStamp': 109, 'TaskCounter':
    9}, {'TaskIdentifier': 3, 'TaskState': 1, 'TimeStamp': 112, '
    TaskCounter': 10}]
Task 3 Condition WCET: T — WCET Time: 3
Task 3 Condition DeadLine: T
Task 3 TimeStamp: 112 DeadLine: 120
[{'TaskIdentifier': 4, 'TaskState': 0, 'TimeStamp': 117, 'TaskCounter':
    1}, {'TaskIdentifier': 4, 'TaskState': 1, 'TimeStamp': 120, '
    TaskCounter': 2}]
Task 4 Condition WCET: T — WCET Time: 3
Task 4 Condition DeadLine: T
Task 4 TimeStamp: 120 DeadLine: 240
```

```
[{'TaskIdentifier': 2, 'TaskState': 0, 'TimeStamp': 120, 'TaskCounter':
    10}, {'TaskIdentifier': 2, 'TaskState': 1, 'TimeStamp': 123, '
    TaskCounter': 11}]
Task 2 Condition WCET: T - WCET Time: 3
Task 2 Condition DeadLine: T
Task 2 TimeStamp: 123 DeadLine: 132
[{'TaskIdentifier': 1, 'TaskState': 0, 'TimeStamp': 123, 'TaskCounter':
    10}, {'TaskIdentifier': 1, 'TaskState': 1, 'TimeStamp': 124, '
    TaskCounter': 11}]
Task 1 Condition WCET: T - WCET Time: 1
Task 1 Condition DeadLine: T
Task 1 TimeStamp: 124 DeadLine: 132
```

Listing C.1 – Monitor with Events critical moment and end of minimal time-window

APPENDIX D – OUTPUT DATA RTMLIB ONLINE MODE TASK COMMUNICATION PERIODIC

```
{"TaskIdentifier":1,"TimeStamp":0,"ExecutionTime":1,"CounterTask":1,"StatusWCET":1,"StatusDeadline":1}
{"TaskIdentifier":2,"TimeStamp":1,"ExecutionTime":3,"CounterTask":1,"StatusWCET":1,"StatusDeadline":1}
{"TaskIdentifier":4,"TimeStamp":4,"ExecutionTime":3,"CounterTask":1,"StatusWCET":1,"StatusDeadline":1}
{"TaskIdentifier":3,"TimeStamp":7,"ExecutionTime":4,"CounterTask":1,"StatusWCET":1,"StatusDeadline":1}
{"TaskIdentifier":2,"TimeStamp":11,"ExecutionTime":2,"CounterTask":2,"StatusWCET":1,"StatusDeadline":1}
{"TaskIdentifier":1,"TimeStamp":13,"ExecutionTime":2,"CounterTask":2,"StatusWCET":1,"StatusDeadline":1}
{"TaskIdentifier":3,"TimeStamp":15,"ExecutionTime":3,"CounterTask":2,"StatusWCET":1,"StatusDeadline":1}
{"TaskIdentifier":3,"TimeStamp":20,"ExecutionTime":3,"CounterTask":3,"StatusWCET":1,"StatusDeadline":1}
{"TaskIdentifier":2,"TimeStamp":23,"ExecutionTime":3,"CounterTask":3,"StatusWCET":1,"StatusDeadline":1}
{"TaskIdentifier":1,"TimeStamp":26,"ExecutionTime":1,"CounterTask":3,"StatusWCET":1,"StatusDeadline":1}
{"TaskIdentifier":2,"TimeStamp":33,"ExecutionTime":2,"CounterTask":4,"StatusWCET":1,"StatusDeadline":1}
{"TaskIdentifier":1,"TimeStamp":35,"ExecutionTime":2,"CounterTask":4,"StatusWCET":1,"StatusDeadline":1}
{"TaskIdentifier":3,"TimeStamp":37,"ExecutionTime":3,"CounterTask":4,"StatusWCET":1,"StatusDeadline":1}
{"TaskIdentifier":3,"TimeStamp":45,"ExecutionTime":3,"CounterTask":5,"StatusWCET":1,"StatusDeadline":1}
{"TaskIdentifier":2,"TimeStamp":48,"ExecutionTime":3,"CounterTask":5,"StatusWCET":1,"StatusDeadline":1}
{"TaskIdentifier":1,"TimeStamp":51,"ExecutionTime":1,"CounterTask":5,"StatusWCET":1,"StatusDeadline":1}
{"TaskIdentifier":2,"TimeStamp":57,"ExecutionTime":2,"CounterTask":6,"StatusWCET":1,"StatusDeadline":1}
{"TaskIdentifier":1,"TimeStamp":59,"ExecutionTime":2,"CounterTask":6,"StatusWCET":1,"StatusDeadline":1}
{"TaskIdentifier":3,"TimeStamp":61,"ExecutionTime":3,"CounterTask":6,"StatusWCET":1,"StatusDeadline":1}
{"TaskIdentifier":3,"TimeStamp":69,"ExecutionTime":3,"CounterTask":7,"StatusWCET":1,"StatusDeadline":1}
{"TaskIdentifier":2,"TimeStamp":72,"ExecutionTime":3,"CounterTask":7,"StatusWCET":1,"StatusDeadline":1}
{"TaskIdentifier":1,"TimeStamp":75,"ExecutionTime":1,"CounterTask":7,"StatusWCET":1,"StatusDeadline":1}
```

```
{"TaskIdentifier":2,"TimeStamp":81,"ExecutionTime":2,"CounterTask":8,"StatusWCET":1,"StatusDeadline":1}
{"TaskIdentifier":1,"TimeStamp":83,"ExecutionTime":2,"CounterTask":8,"StatusWCET":1,"StatusDeadline":1}
{"TaskIdentifier":3,"TimeStamp":85,"ExecutionTime":3,"CounterTask":8,"StatusWCET":1,"StatusDeadline":1}
{"TaskIdentifier":3,"TimeStamp":93,"ExecutionTime":3,"CounterTask":9,"StatusWCET":1,"StatusDeadline":1}
{"TaskIdentifier":2,"TimeStamp":96,"ExecutionTime":3,"CounterTask":9,"StatusWCET":1,"StatusDeadline":1}
{"TaskIdentifier":1,"TimeStamp":99,"ExecutionTime":1,"CounterTask":9,"StatusWCET":1,"StatusDeadline":1}
{"TaskIdentifier":2,"TimeStamp":105,"ExecutionTime":2,"CounterTask":10,"StatusWCET":1,"StatusDeadline":1}
{"TaskIdentifier":1,"TimeStamp":107,"ExecutionTime":2,"CounterTask":10,"StatusWCET":1,"StatusDeadline":1}
{"TaskIdentifier":3,"TimeStamp":109,"ExecutionTime":3,"CounterTask":10,"StatusWCET":1,"StatusDeadline":1}
{"TaskIdentifier":4,"TimeStamp":117,"ExecutionTime":3,"CounterTask":2,"StatusWCET":1,"StatusDeadline":1}
{"TaskIdentifier":2,"TimeStamp":120,"ExecutionTime":3,"CounterTask":11,"StatusWCET":1,"StatusDeadline":1}
{"TaskIdentifier":1,"TimeStamp":123,"ExecutionTime":1,"CounterTask":11,"StatusWCET":1,"StatusDeadline":1}
```

Listing D.1 – Monitor with Events critical moment and end of minimal time-window