FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Adaptation of the IMC protocol to microcontrollers

João Nuno Pereira Bogas

Mestrado em Engenharia Eletrotécnica e de Computadores

Supervisor: João Tasso de Figueiredo Borges de Sousa

October 16, 2024

Resumo

Esta dissertação aborda a implementação de protocolos de comunicação para sistemas embarcados basedos em microcontroladores, com particular foco na melhoria dos sistemas computacionais de veículos subaquáticos autónomos (AUVs). O desafio central reside em permitir arquiteturas computacionais que garantam a interação sem falhas entre os vários componentes do sistema.

O foco principal da pesquisa é a adaptação e implementação do protocolo Inter-Module Communication (IMC) em sistemas baseados em microcontroladores. Este protocolo foi selecionado pelo seu design como um protocolo orientado a mensagens, ideal para construir sistemas interligados de veículos, sensores e operadores humanos que trocam informação em tempo real e procuram objetivos comuns. A natureza abstrata à camada de transporte do protocolo IMC permite a sua integração em diferentes protocolos de comunicação, promovendo modularidade e escalabilidade.

Este trabalho fornece uma revisão da literatura que descreve a estratégia de pesquisa, identificando campos-chave e extraindo palavras-chave relevantes. Depois, analisa as relações entre os estudos existentes, discutindo a sua importância para a investigação atual e fornecendo contexto essencial. Esta revisão conclui identificando lacunas na literatura que este estudo pretende abordar.

Para atingir isto, a dissertação descreve os requisitos de hardware e software necessários para uma implementação bem-sucedida. A abordagem envolve a configuração de uma arquitetura de software em camadas onde as tarefas comunicam através de mensagens IMC. Na base desta arquitetura está um sistema operativo de tempo real (RTOS), que gere o agendamento de tarefas e a alocação de recursos, onde a separação das tarefas em unidades independentes promove a capacidade de resposta do sistema e a tolerância a falhas. Acima do RTOS, a camada Core trata das tarefas do utilizador, fornecendo a configuração de tarefas, bem como a comunicação com dispositivos externos e entre tarefas.

O capítulo de implementação inclui descrições detalhadas dos componentes de hardware e software, com foco na distribuição de mensagens, sincronização de relógios e recuperação de falhas. Isto abrange falhas de hardware, como exceções de CPU e falhas de software, como erros em tarefas. O processo de validação envolve testes rigorosos em relação aos requisitos funcionais e não funcionais para garantir o comportamento e o desempenho esperados do sistema.

Os resultados demonstram que o protocolo IMC implementado nos microcontroladores melhora a interoperabilidade e simplifica a integração de novos componentes num veículo, através da uniformização das comunicações. Permite ainda a criação de uma interface de comunicação para microcontroladores que abstrai a camada de transporte, possibilitando o isolamento de dispositivos computacionais em módulos. Isto é conseguido encapsulando cada dispositivo dentro de limites claros e criando ligações bem definidas, possibilitadas pela utilização de uma linguagem de comunicação comum. Por fim, melhora a eficiência do sistema computacional dos veículos, distribuindo a carga de trabalho pelas unidades de processamento, aliviando assim a CPU principal do trabalho excessivo.

Este estudo contribui para o campo da robótica ao fornecer uma estrutura nova para a comunicação de microcontroladores, abrindo caminho para uma exploração autónoma mais avançada.

Abstract

This dissertation addresses the implementation of communication protocols in microcontroller based embedded systems, with a particular focus on enhancing the computational systems of autonomous underwater vehicles (AUVs). The central challenge lies in enabling computational architectures that ensure seamless interaction among various system components.

The primary focus of the research is on the adaptation and implementation of the Inter-Module Communication (IMC) protocol within microcontroller-based systems. This protocol was selected for its design as a message-oriented protocol, ideal for building interconnected systems of vehicles, sensors, and human operators that exchange real-time information and pursue common goals. The IMC protocol's transport layer abstract nature allows for its integration across different communication protocols, promoting modularity and scalability.

This work provides a literature review that describes the research strategy by identifying key fields and extracting relevant keywords. After, it analyzes the relationships between existing studies, discussing their significance to the current research and providing essential context. This review concludes by identifying gaps in the literature that this study aims to address.

To achieve this, the dissertation outlines the hardware and software requirements necessary for successful implementation. The approach involves setting up a layered software architecture where tasks communicate using IMC messages. At the base of this architecture is a real-time operating system (RTOS), which manages task scheduling and resource allocation, where separating tasks into independent units promotes the system's responsiveness and fault tolerance. Above the RTOS, the Core layer handles user tasks, providing task configuration as well as communication with external devices and between tasks.

The implementation chapter involves providing detailed descriptions of the hardware and software components, with a focus on message distribution, clock synchronization, and fault recovery. This encompasses hardware faults like CPU exceptions and software faults such as task errors. The validation process involves rigorous testing against both functional and non-functional requirements to ensure the system's expected behavior and performance.

The results demonstrate that the IMC protocol implemented in microcontrollers improves interoperability and simplifies the integration of new devices in a vehicle by standardizing the communications. It also allows for the creation of a communication interface for microcontrollers that abstracts the transport layer, enabling the isolation of computational devices into modules. This is achieved by encapsulating each device within clear boundaries and creating well-defined connections, made possible using a common communication language. Finally, it improves the computational system efficiency of vehicles by distributing the workload across processing units, thereby relieving the main CPU of excessive work.

This study contributes to the field of robotics by providing a new framework for microcontroller communication, paving the way for more advanced and reliable autonomous exploration.

United Nations Sustainable Development Goals

The Sustainable Development Goals (SDGs) are a set of global objectives adopted by United Nations member states as part of the 2030 Agenda for Sustainable Development. They encompass 17 interconnected goals that address global challenges to achieve a more sustainable and equitable world by 2030. Each goal includes specific targets and indicators to measure progress and guide international efforts toward a more prosperous and resilient future for the people and the planet.

The following table 1 illustrates how this project aligns with and contributes to several SDGs and outlines possible performance indicators and metrics to measure the contribution. By focusing on these specific SDGs, the project aims to make meaningful contributions towards improving global well-being, protecting the environment, and fostering inclusive and sustainable development.

ODS	Goal	Contribution	Performance Indicators and Metrics
9	9.5	Enhances scientific research and technological capabilities in underwater robotics.	Measure the number of research projects and publications the enhanced underwater systems enable.
13	13.1	Contributes to climate resilience through improved underwater monitoring.	Contributions to disaster preparedness via underwater data collection.
14	14.1	Supports the reduction of marine pollution through improved monitoring and data collection.	Monitor changes in pollution levels and marine ecosystem health improvements using underwater robotic systems' data.
14	14.2	Supports sustainable marine ecosystem management through advanced AUV technology.	Number of successful deployments and collected data for ecosystem management.
14	14.a	Enhances scientific research and technological capabilities in AUVs.	Measure the number of research projects and publications made possible with new AUVs.

Table 1: Project Contributions to SDGs

Acknowledgements

I would like to express my deeply gratitude to my supervisor, Prof. João de Borges Sousa, for his invaluable guidance, mentorship, and support throughout this research journey. His expertise and insightful feedback have been instrumental in shaping this work.

I am also extremely grateful to the dedicated team at the LSTS for providing me with the invaluable opportunity to collaborate and work alongside them. The experience gained from working with LSTS, particularly in exploring the intricacies of the IMC protocol and the DUNE framework, has significantly enriched my research and understanding in this field.

Lastly, I extend my thanks to my family and friends for their continuous encouragement, understanding, and patience during the challenging phases of this project. Their unwavering support has been a constant source of motivation.

João Nuno Pereira Bogas

"You should be glad that bridge fell down.
I was planning to build thirteen more to that same design"

Isambard Kingdom Brunel

Contents

1 Introduction **1**
- 1.1 Motivation . . . 2
- 1.2 Problem . . . 3
- 1.3 Work Proposal and Objectives . . . 3
- 1.4 Document Structure . . . 4
- 1.5 Publications . . . 5

2 Background **7**
- 2.1 Embedded Systems . . . 7
- 2.2 RTOS . . . 8
- 2.3 OSI model . . . 11
- 2.4 Intra-vehicular Communication . . . 12
  - 2.4.1 RS-232 . . . 12
  - 2.4.2 Ethernet . . . 13
  - 2.4.3 SPI . . . 14
  - 2.4.4 CAN . . . 15
- 2.5 LSTS Vehicle System . . . 15
  - 2.5.1 Unmanned Vehicles . . . 17
  - 2.5.2 Payload . . . 19
  - 2.5.3 Ground Station . . . 20
- 2.6 LSTS Software Toolchain . . . 21
  - 2.6.1 DUNE . . . 21
  - 2.6.2 IMC . . . 22

3 Literature Review **25**
- 3.1 Research strategy . . . 25
- 3.2 Selected Publications . . . 27
- 3.3 Analysis of Publications . . . 29
- 3.4 Related Work . . . 31

4 Approach **33**
- 4.1 Project Requirements . . . 33
  - 4.1.1 Functional Requirements . . . 33
  - 4.1.2 Non-Functional Requirements . . . 34
- 4.2 Hardware board . . . 34
- 4.3 Software Architecture . . . 36
  - 4.3.1 Overview . . . 36
  - 4.3.2 Components . . . 37

4.3.3 Communications 40

5 Implementation **43**
5.1 Hardware Description 43
5.1.1 Arm Cortex-M7 43
5.1.2 STM32F767ZI 45
5.2 RTOS Implementation 46
5.2.1 Thread Scheduling 46
5.2.2 Synchronization mechanisms 48
5.2.3 Fault Recovery 48
5.3 Core Layer 49
5.3.1 Task Parameter Setup 49
5.3.2 Communication 50
5.3.3 IMC Entities configuration 51
5.3.4 Message Distribution 51
5.3.5 Clock Synchronization 52
5.3.6 Fault Recovery 53
5.4 Tasks 54

6 Verification and Validation **55**
6.1 Verification 55
6.1.1 RTOS layer 56
6.1.2 Core layer 57
6.1.3 User Tasks 58
6.1.4 Conclusion 58
6.2 Validation 59
6.2.1 Functional Requirements 59
6.2.2 Non-Functional Requirements 63

7 Conclusion **69**
7.1 Results and Discussion 69
7.2 Future Work 71

References **73**

List of Figures

2.1 Priority Inheritance . . . 10
2.2 Priority Inversion . . . 10
2.3 OSI Model [5] . . . 11
2.4 RS 232 frame of ASCII character '\n' with 1 stop bit and 0 parity bits. . . . 13
2.5 IEEE 802.3 frame . . . 13
2.6 SPI master connected to multiple slaves [14]. . . . 14
2.7 CAN bus . . . 15
2.8 LSTS Vehicle System . . . 16
2.9 LSTS LAUV [16] . . . 17
2.10 Breakdown structure of an LAUV. Designed by LSTS and OceanScan-MST. . . . 20
2.11 IMC message layout . . . 23
2.12 Plan Specification Message . . . 24

3.1 Relevant fields . . . 26
3.2 Micro-ROS Architecture . . . 32

4.1 STM32 Nucleo-144 board [34] . . . 35
4.2 System Architecture . . . 36

5.1 Polling Tick delay . . . 47
5.2 Timer delayed callback . . . 48
5.3 Flash Parameter Serialization . . . 50
5.4 IMC configuration sequence . . . 51
5.5 ClockControl Message . . . 52
5.6 Synchronization Timestamps . . . 53

List of Tables

1 Project Contributions to SDGs v

2.1 Field types in IMC messages 24

4.1 Functional Requirements 34
4.2 Non Functional Requirements 34
4.3 Required RTOS Functionalities 38
4.4 Required Core Functionalities 39
4.5 Required Task Functionalities 40

6.1 Unit Tests for Thread 56
6.2 Unit Tests for Mutex 56
6.3 Unit Tests for Semaphore 57
6.4 Unit Tests for Message Queue 57
6.5 Unit Tests for Entity Configuration 57
6.6 Unit Tests for Parameter Management 57
6.7 Unit Tests for Task Recovery 57
6.8 Unit Tests for Message Distribution 58
6.9 Unit Tests for Background Task 58
6.10 Unit Tests for Task Parameters 58
6.11 Unit Tests for Task Entity State 58
6.12 Unit Tests for Task Message Handling 58
6.13 Validation Table for Functional Requirement R1 60
6.14 Validation Table for Functional Requirement R2 60
6.15 Validation Table for Functional Requirement R3 60
6.16 Validation Table for Functional Requirement R4 60
6.17 Validation Table for Functional Requirement R5 61
6.18 Validation Table for Functional Requirement R6 61
6.19 Validation Table for Functional Requirement R7 61
6.20 Validation Table for Functional Requirement R8 61
6.21 Validation Table for Functional Requirement R9 62
6.22 Validation Table for Functional Requirement R10 62
6.23 Validation Table for Functional Requirement R11 62
6.24 Functional Requirements Results 63
6.25 Validation Table for Non-Functional Requirement R1 64
6.26 Validation Table for Non-Functional Requirement R2 64
6.27 Validation Table for Non-Functional Requirement R3 65
6.28 Validation Table for Non-Functional Requirement R4 65
6.29 Validation Table for Non-Functional Requirement R5 65

6.30 Validation Table for Non-Functional Requirement R6 66
6.31 Validation Table for Non-Functional Requirement R7 66
6.32 Non-Functional Requirements Results 67

Abbreviations and Symbols

ASV Autonomous Surface Vehicle. 20, 21

AUV Autonomous Underwater Vehicle. 1–3, 8, 15, 19–21, 35, 36

CAN Controller Area Network. 3, 7, 15, 34, 35, 45, 71

CISC Complex Instruction Set Computing. 7, 8

CPU Central Processing Unit. 2, 3, 5, 7, 8, 12, 19, 34, 35, 37, 43–50, 70

DTN Disruptive Tolerant Networking. 16

DUNE Unified Navigation Environment. 2, 3, 7, 16, 18, 21, 22, 31, 32, 38, 39, 50, 52, 53, 59, 60, 64, 69–71

GPOS General-Purpose Operating System. 8, 9

GPS Global Positioning System. 1, 18

I2C Inter-Integrated Circuit. 3, 45, 71

IMC Inter-Module Communication. 2–5, 7, 16, 17, 21–23, 27–31, 33, 34, 36–40, 46, 49–51, 54, 55, 57, 58, 60, 63, 67, 69–71

IMU Inertial Measurement Unit. 3, 18

ISO International Organization for Standardization. 11

LAUV Light Autonomous Underwater Vehicle. 2, 3

LSTS Laboratório de Sistemas e Tecnologias Subaquáticas. 2–4, 7, 15–17, 19, 20, 22

MCU Microcontroller Unit. 33, 52, 53, 60

OSI Open Systems Interconnection. 4, 7, 11, 22

RISC Reduced Instruction Set Computing. 7, 8

ROV Remotely Operated Vehicle. 20, 21

RS-232 Recommended Standard 232. 3, 7, 12, 41, 43, 45, 50, 59, 64–67, 69

RTOS Real-Time Operating System. 7–9, 11, 29, 30, 36–39, 43, 44, 46, 48, 49, 51, 56, 57, 70

SPI Serial Peripheral Interface. 3, 7, 14, 34, 35, 45, 71

UAV Unmanned Aerial Vehicle. 20, 21

UDP User Datagram Protocol. 17, 50

Chapter 1

Introduction

As the use of Autonomous Underwater Vehicles (AUVs) increases, the interest and demand for more efficient systems grows. AUVs, designed for autonomous navigation beneath the ocean's surface, encounter a spectrum of challenges unique to their submerged environment. These challenges include navigating intricate underwater topography, adapting to varying ocean conditions, dealing with limited communication bandwidth and energy storage, navigation without Global Positioning System (GPS), and the need for real-time decision-making in dynamically changing conditions. Furthermore, sensing, communications, and computing technologies have been evolving rapidly over the last decade. This is in line with the law of accelerating returns [13]. Moreover, according to Martec's law[1], technology is evolving exponentially, while organizations are evolving logarithmically. This significant and growing gap between the rapid pace of technological advancements and the much slower rate at which organizations adapt implies that organizations struggle to keep up with the latest technological developments. This can potentially lead to inefficiencies and missed opportunities. As a result, organizations need to accelerate their evolution, including adopting new technologies and updating infrastructure to better align with the rapid pace of technological innovation. This work is especially focused on evolving computational environments onboard these vehicles, in particular in what regards to power consumption, modularity, and integration of new components. This cannot be done without novel work on microprocessors and microcontrollers, as well as on software development.

Traditionally, AUVs have used a hierarchical computational architecture featuring a microprocessor for central computing and microcontrollers dedicated to peripheral tasks, sensor acquisition, and actuation. This integration of a microprocessor alongside microcontrollers is crucial for the requirements of underwater exploration. It allows for a better allocation of functionalities to each computational device. Microprocessors are better suited for complex computations, decision-making navigation algorithms, and data analysis. Microcontrollers are better equipped for real-time functionalities, actuator control, sensor acquisition, and quick responses to external events.

[1] https://chiefmartec.com/2016/11/martecs-law-great-management-challenge-21st-century/

In contrast, this work proposes a more distributed computational architecture. In this new architecture, microcontrollers not only acquire data but also process it, translating it into the common language Inter-Module Communication (IMC). This approach decentralizes processing tasks, allowing microcontrollers to handle more of the workload independently of the central microprocessor. Moreover, offloading tasks suitable for microcontrollers reduces the demands from the main microprocessor, creating a more efficient computational system where the main Central Processing Unit (CPU) is not overworked. Furthermore, this approach results in a modular and scalable system, as the distributed architecture allows for easy upgrades and integration of new technologies without rebuilding the entire system.

1.1 Motivation

The work detailed in this document was conducted at the Laboratório de Sistemas e Tecnologias Subaquáticas (LSTS), a laboratory dedicated to the research and development of technology for underwater systems, particularly AUVs and related marine robotics, such as its Light Autonomous Underwater Vehicle (LAUV) [33]. LSTS focuses on creating technologies that enable multiple unmanned vehicles to collaborate effectively. This involves developing software frameworks and communication protocols that facilitate coordination and data sharing among networked vehicles. A critical aspect of LSTS research is enabling communication in environments with intermittent or disrupted connectivity. By enhancing vehicle communication systems, LSTS aims to ensure reliable data transmission for critical missions in remote or harsh maritime environments where traditional networking may be unreliable. The advancements achieved by LSTS in previous projects, such as in developing its LAUV and its software toolchain, particularly with the Unified Navigation Environment (DUNE), the onboard software controlling the vehicle, and IMC, the language LSTS systems use to communicate, have laid the groundwork for the current research. This work addresses several key motivations.

Currently, microcontrollers are unable to communicate using the IMC protocol. This limitation prevents the full utilization of this component, as they are not being used to their full potential. Maximizing the functionality of these devices is essential for developing more efficient and capable underwater vehicles. Improving computational system communications, specifically intra-vehicle communications, is another primary goal. By enhancing the ability of vehicles to communicate and share data, the overall coordination and effectiveness of networked vehicle systems can be significantly improved. This includes creating technologies that enable multiple unmanned vehicles to collaborate effectively through the development of software frameworks and communication protocols.

The ultimate purpose of this research is to standardize protocol communications so that every component of an LAUV system uses the IMC protocol to communicate. This uniformity is essential for enabling a more modular approach, allowing new technologies to be easily implemented and integrated into the existing systems.

1.2 Problem

The LSTS LAUV is designed for autonomous operation in challenging marine environments [20]. It is equipped with a main CPU running the DUNE software, which is responsible for high-level control, decision-making, and mission planning. This main CPU, housed in a PC/104 form factor, primarily communicates with a few microcontrollers via the Recommended Standard 232 (RS-232) communication protocol, but also is capable of using other protocols such as Serial Peripheral Interface (SPI), Inter-Integrated Circuit (I2C), Controller Area Network (CAN), and Ethernet. These microcontrollers enable DUNE to manage motor actuation and process sensor data to determine the LAUV's position, orientation, and depth. Then, the DUNE software calculates navigation and path-planning algorithms, enabling the LAUV to navigate through the underwater environment and execute mission objectives autonomously. To facilitate communication between the LAUV system and with external nodes, the LAUV utilizes the IMC protocol. IMC abstracts the communication layer and provides support in environments where communication is often challenged, ensuring reliable data exchange even under difficult conditions.

The role of microcontrollers within LAUV systems is essential for a successful operation. These microcontrollers acquire sensor data from various sources, such as sonar systems, Inertial Measurement Unit (IMU)s, and pressure sensors. They then transmit this data to the DUNE software for processing. DUNE analyzes the data and generates specific commands for control. These commands are then transmitted to the actuation microcontrollers, which decode them into control signals.

However, this process is inefficient because microcontrollers do not process the data themselves before transmitting it to DUNE. Ideally, data could be processed at the source by the microcontrollers themselves. They could then directly communicate processed data to the actuation microcontrollers, enabling them to generate control signals promptly. In the scenario, DUNE would only oversee the operation rather than directly process every piece of data.

The communication methods between the central microprocessor and microcontrollers in an LAUV play a crucial role in orchestrating coordination and the intricacies of data exchange within the system. However, the diversity in these communication protocols, such as RS-232, SPI, I2C, CAN, and Ethernet can pose challenges. One of the downsides of diverse communication protocols is the complexity it introduces into the design and implementation of Autonomous Underwater Vehicle (AUV) systems. Each protocol has its own set of rules, timings, and requirements, making it necessary for engineers to have a deep understanding of multiple standards. This can lead to increased development time and potential compatibility issues between components that use distinct protocols.

1.3 Work Proposal and Objectives

This thesis proposes to address the need for standardized communication protocols within embedded systems, particularly focusing on microcontrollers. The proposed work is to design and

implement a software framework for microcontrollers using the IMC protocol. The first step is the investigation of several key topics, such as multitasking environments and communication transport abstraction design in microcontrollers. Then several requirements are identified and refined based on validation and development according to LSTS standards. The subsequent steps involved developing the approach, evaluating the design, and assessing the outcomes achieved. By standardizing protocol communications with the IMC protocol and leveraging microcontrollers, this work aims to advance the modularity, efficiency, and effectiveness of underwater vehicle systems, enabling easier implementation of new technologies and pushing the boundaries of marine robotics.

The objectives of this work are multifaceted. Firstly, the goal is to enhance computational efficiency by distributing computational load more effectively. This involves allocating the workload across multiple processing units, thus reducing the excess burden of high-power consumption components, and ultimately promoting the computational system's overall efficiency.

Another key objective is to establish IMC as a communication protocol that can be utilized by every computational device, providing a uniform interface for tasks running in both microprocessors and microcontrollers. This standardization will simplify the development and integration of processing devices. Promoting modularity is also a significant objective. This will be achieved by organizing each processing unit, of the computational system, as a module that can be independently tested and improved, thus enhancing maintainability and flexibility. Scalability is a crucial consideration, as the system must be able to expand its number of computational devices easily and handle varying communication requirements without necessitating extensive modifications. Ensuring fault tolerance is another priority, to guarantee reliable operation under various conditions. Compatibility with existing hardware and software architectures is essential to facilitate seamless integration into current systems. Finally, a comprehensive understanding of potential limitations, such as resource constraints, communication reliability in real-world environments, and bandwidth limitations in high data transfer applications, is vital.

1.4 Document Structure

The structure of this dissertation is organized into several chapters, each detailing a specific aspect of the research, development, and conclusions.

The Background chapter provides a description of fundamental concepts and technologies related to embedded systems, real-time operating systems, the Open Systems Interconnection (OSI) model, and inter-component communication protocols. It also covers the LSTS vehicle structure and software toolchain.

Following this, the Literature Review presents the literature research strategy and the related work in the field. It analyzes the relationships between different studies and discusses their relevance to the current research, providing a context for the problem being addressed. It finalizes with the discussion of related works.

The Approach chapter outlines the project requirements, both functional and non-functional, required to establish a framework for implementing the IMC protocol on microcontroller-based systems. Then it details the hardware utilized, highlighting the use of the STM32 Nucleo-144 board. Following this, the chapter presents the approach employed to address the problem. This includes designing a layered software architecture for enabling IMC communications and providing an overview of the components involved, their structure, and how they interact.

In the Implementation chapter describes the hardware elements, such as the Arm Cortex-M7 CPU and STM32F767ZI microcontroller. It then covers the implementation of the previously described software architecture by detailing each component and its functionalities. Lastly, it describes the implementation of each protocol communication it supports.

The Verification and Validation chapter is focused on the verification and validation processes, detailing the tests conducted to ensure the system meets all requirements. It presents the results of these tests and discusses the system's reliability and performance.

Finally, the Conclusion chapter summarizes the results and discusses their implications. It highlights the contributions of the research to the field of underwater robotics and suggests areas for future work to further enhance the IMC protocol implementation and its applications in real-world scenarios.

1.5 Publications

An abstract titled 'IMC messaging protocol for microcontrollers: enabling low-level interoperability in robotic systems' has been accepted for participation in the OCEANS 2024 Halifax Technical Paper Program.

Chapter 2

Background

This section starts to introduce fundamental concepts and aspects related to this work. It begins by exploring Embedded Systems, Real-Time Operating System (RTOS), and their applicability in ensuring a multitasking environment in microcontroller devices. The OSI model is introduced to provide a theoretical layered structure for understanding communication protocols, which enable inter-component communication within these systems. Then it presents relevant communication protocols used for inter-component communication within these systems, including RS-232, SPI, CAN, and Ethernet.

Additionally, this chapter details the LSTS Vehicle System, highlighting its major components such as the Unmanned Vehicle, Payload, and Ground Station. Lastly, the LSTS Software Toolchain is introduced, with the components of the DUNE framework and the IMC protocol, and their role in shaping the architecture and operation of unmanned vehicles.

2.1 Embedded Systems

"Embedded systems are information processing systems that are embedded into a larger product" [22]. At the heart of many embedded systems lies the microprocessor. The microprocessor is a CPU incorporated into a single integrated circuit, that performs essential functions such as arithmetic and logic operations, all on a single chip [36].

Microprocessors can be implemented using two main architectures: Complex Instruction Set Computing (CISC) and Reduced Instruction Set Computing (RISC). CISC processors have a larger and more complex set of instructions, allowing them to perform multiple operations within a single instruction. On the other hand, RISC processors are designed with a small set of simple instructions, each of which can be executed within a single clock cycle. This leads to greater efficiency and higher performance for simple tasks. Both architectures are valid for implementing microprocessors. While performance, power consumption, and efficiency can be optimized to yield similar results, RISC architectures tend to be optimized for higher efficiency and lower power consumption due to their reduced instruction overhead. In contrast, CISC architectures are often optimized for higher performance, although with increased power consumption [3].

In addition, Embedded systems may utilize microcontrollers, which are tailored for specific tasks within an embedded system. Microcontrollers are electronic devices that contain a processor core, memory, and input/output peripherals all integrated into a single chip. Since these components are designed to serve particular applications they require very low power consumption and support real-time operations. The microcontroller's efficiency lies in its ability to manage input/output operations and interface with sensors and actuators while executing tasks with precise timing requirements. Unlike microprocessors, microcontrollers are designed with separate memory spaces for programs and data, following a RISC architecture since the overhead of CISC instructions would hinder system performance [3]. Given their specialization, microcontrollers are employed in systems such as AUVs to interface with sensors and actuators that require more precise timing.

In systems comprising both these components, integrating microprocessors and microcontrollers is crucial for optimizing performance, efficiency, and functionality. The main CPU, usually a powerful microprocessor, handles complex processing tasks, high-level algorithms, and extensive data operations. Additional microcontrollers manage real-time hardware control, interfacing with sensors and actuators, ensuring precise timing with low power consumption. Consequently, a hierarchical relationship emerges, where the main CPU addresses generalized problems, oversees overall system operations, and delegates specific tasks to microcontrollers. This approach harnesses the computational power of the microprocessor and the efficiency of the microcontroller.

2.2 RTOS

The most common type of operating system is the General-Purpose Operating System (GPOS). A GPOS is an operating system designed to manage a wide range of applications and tasks in a flexible context, where deadlines and time constraints are not always guaranteed to be satisfied. GPOS are optimized for versatility and usability, supporting a broad spectrum of software and hardware configurations.

In contrast, embedded systems often require precise control and timely responses, which is where RTOS come into play. An RTOS is a type of Operating System designed specifically for scheduling task execution with precise time constraints. Unlike GPOS, RTOS prioritizes tasks based on their urgency and seeks to guarantee that critical tasks are executed within their specific deadlines/time constraints. Some key characteristics of an RTOS include minimal latency, high reliability, and the ability to manage multiple high-priority tasks simultaneously [15].

Despite these differences, RTOS and GPOS share some core functional similarities, such as:

- Multitasking.
- Software and hardware resource management.
- Provision of underlying OS services to applications.
- Abstracting the hardware from the software application.

However, the unique requirements of embedded systems highlight the functional differences that set RTOS apart from GPOS. These differences are crucial in determining the suitability of an RTOS for specific real-time applications. Some core functional differences that distinguish RTOS from GPOS include [15]:

- Better reliability in embedded application contexts.
- The ability to scale up or down to meet application needs.
- Faster performance.
- Reduced memory requirements.
- Scheduling policies tailored for real-time embedded systems
- Support for diskless embedded systems by allowing executables to boot and run from Read Only Memory (ROM) or Random-Access Memory (RAM)
- Better portability to different hardware platforms

These distinctions are essential for understanding why an RTOS is often the preferred choice for embedded systems that require stringent timing and reliability guarantees. Commonly in microcontrollers, RTOS comprises only a kernel which is the core supervisory software that provides minimal logic, scheduling, and resource management algorithms [15]. Implementing an RTOS in a microcontroller has several benefits in enhancing its functionality. It allows for the scheduling and management of multiple tasks to be executed concurrently, ensuring efficient use of resources. Additionally, an RTOS guarantees deterministic behavior, which is crucial in real-time applications because it executes critical and higher-priority tasks within their designated time frames. Moreover, it protects resources from concurrent access or execution, and it facilitates inter-task synchronization and communication, enabling different tasks to interact and synchronize effectively.

In an RTOS, common synchronization structures include Mutexes, Semaphores, and Message Queues. A Mutex (mutual exclusion) is a synchronization primitive that prevents multiple tasks from accessing a shared resource simultaneously. Mutexes have an ownership feature, meaning they can only be released by the task that acquired them. This ensures the resource remains protected until the task is finished using it. Many RTOSes implement priority inheritance 2.1 with Mutexes to prevent priority inversion 2.2, where a lower-priority task holds a Mutex needed by a higher-priority task. Tasks lock a Mutex before accessing a shared resource and unlock it afterward. If a Mutex is owned, subsequent tasks attempting to lock it will be blocked until it becomes available.

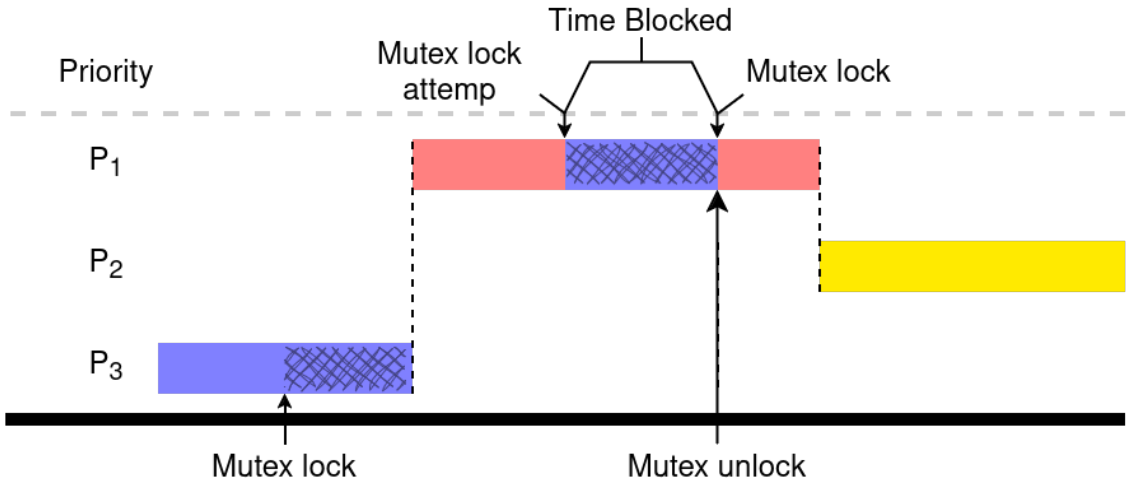

Figure 2.1: Priority Inheritance

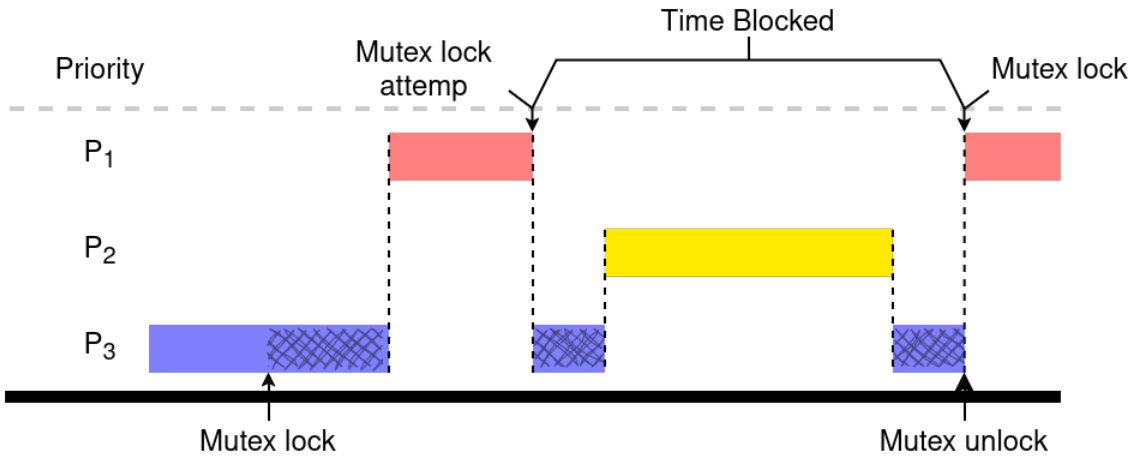

Figure 2.2: Priority Inversion

Semaphores are signaling mechanisms to control access to shared resources or synchronize tasks. They come in two types: binary semaphores and counting semaphores. Binary semaphores can take only two values while counting semaphores can take any integer value and are used to manage a resource pool with a finite number of instances. A Semaphore is signaled (incremented) when a resource becomes available, and a task waits (decrements) a Semaphore before accessing the resource. If the Semaphore count is zero, the task is blocked until it becomes positive. Message Queues are used for communication and synchronization between tasks by sending and receiving messages, allowing tasks to exchange data in a thread-safe manner. Tasks can send messages to the queue and will block if the queue is full. They can also receive messages from the queue

and block if empty. Some RTOSes support priority message queues, where messages can be prioritized. These synchronization structures are essential for managing task coordination and resource sharing in real-time systems, ensuring predictable and reliable operation.

2.3 OSI model

The OSI model is a conceptual framework used to understand and implement standard protocols in network communications. Developed by the International Organization for Standardization (ISO), it divides network communication into seven distinct layers, each with specific functions and protocols. This layered approach aids in the design and troubleshooting of complex network systems by isolating different networking tasks [5].

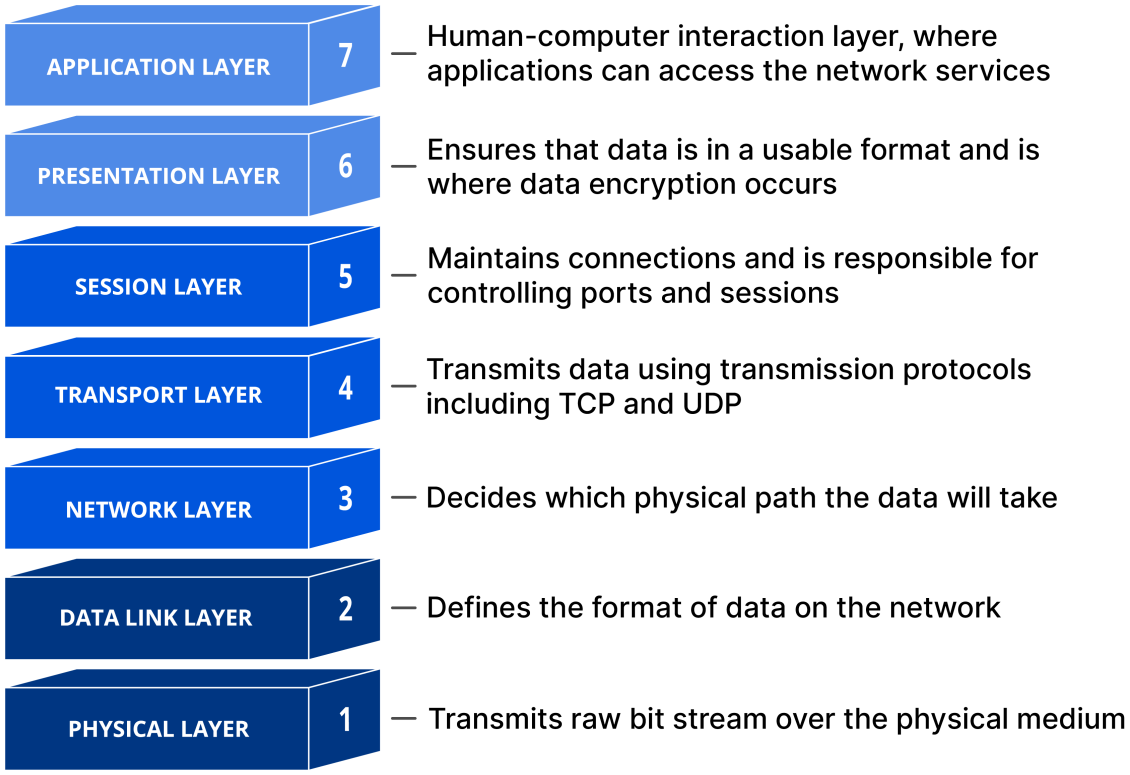

Figure 2.3: OSI Model [5]

This standard networking model is composed of seven layers.

7. Application layer - Provides network services directly to end-users and is responsible for interacting with user data. It is responsible for the protocols and data manipulation that the user software is built upon.

6. Presentation layer - Translates data between the application layer and the network. It handles data encoding, encryption, and compression to ensure that data sent by one application layer is readable by another.

5. Session layer - Manages sessions between applications. It establishes, maintains, and terminates connections and synchronizes data exchange.

4. Transport layer - Ensures end-to-end communication, error recovery, and flow control. It breaks data into segments before sending it to the next layer and reassembles segments on the receiving end.

3. Network layer - Manages data routing, forwarding, and addressing, facilitating data transfer between different networks. It breaks up segments into packets and finds the best path for data to reach its destination.

2. Data Link layer - Ensures data transfer across the physical link by breaking packets into smaller frames. It also manages flow control and error control in the physical layer.

1. Physical layer - Manages the physical connection between devices, defining the specifications for activating, maintaining, and deactivating the physical link. It converts data into a stream of bits.

2.4 Intra-vehicular Communication

Communication between the main CPU and peripheral microcontrollers in autonomous systems can be achieved using several protocols. Here are some common protocols used for such communications, with their physical characteristics and transmission rates:

2.4.1 RS-232

RS-232 is a widely recognized standard for serial communication, primarily designed for point-to-point communication between Data Terminal Equipment (DTE), such as computers, and Data Circuit-terminating Equipment (DCE), like modems.

This standard employs an unbalanced (single-ended) voltage mode, transmitting signals over a single wire relative to a common ground. RS-232 supports both synchronous and asynchronous communication, allowing for flexibility in data transmission methods. It can operate in simplex, half-duplex, and full-duplex modes however, its effectiveness is limited to short distances and it supports relatively low transmission rates, which can be further constrained by cable length and quality. RS-232 baud rates define the speed of symbol changes per second in serial communication. Although this protocol does impose baud rates, they typically range from 300 to 115,200 bps, although their speed capabilities are dependent on the hardware they are implemented on.

The RS-232 standard includes features such as Tx (transmit) and Rx (receive) bit interfaces, flow control mechanisms, and provisions for interfacing with DCEs using control signals like RTS (Request to Send) and CTS (Clear to Send). It also supports self-check testing to ensure the integrity of the communication line and connected devices.

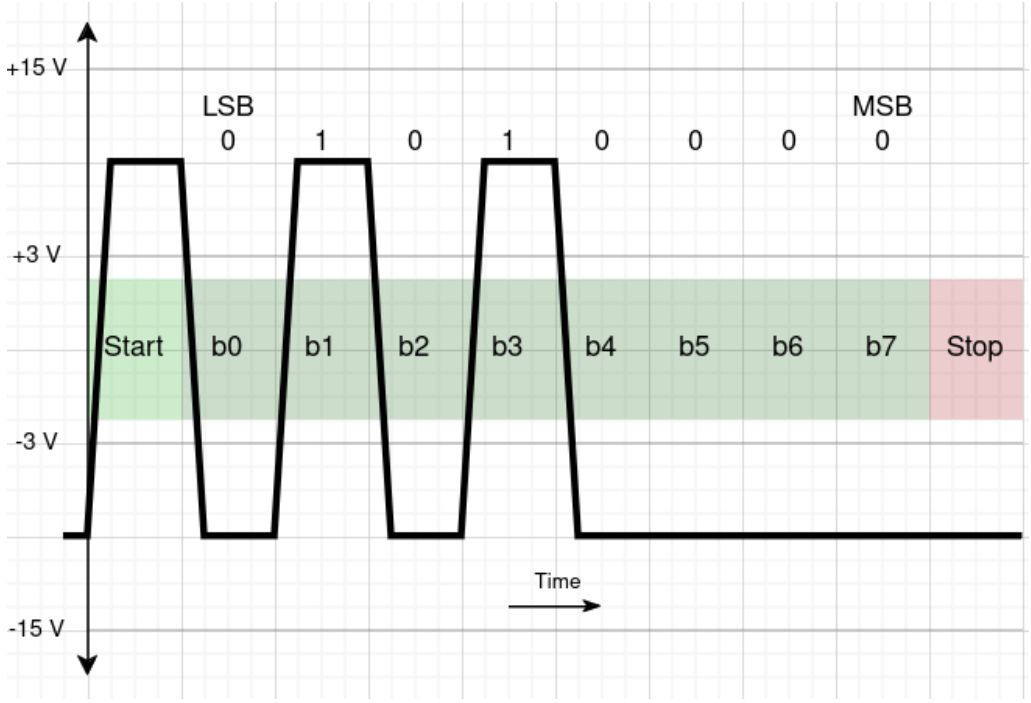

Figure 2.4: RS 232 frame of ASCII character '\n' with 1 stop bit and 0 parity bits.

2.4.2 Ethernet

Ethernet is a commonly used communication technology for Local Area Networks (LANs) that facilitates data exchange between various internal subsystems or external systems. It offers high data transfer rates, making it ideal for applications that require substantial data requirements, ranging from 10 Mbps to 100 Gbps.

Ethernet is based on the IEEE 802.3 standard and uses packet switching to transmit data across the network. While Ethernet is not inherently real-time like some specialized protocols, it can support real-time applications through the use of technologies such as Real-Time Ethernet or by implementing prioritization mechanisms [32].

Ethernet can be implemented using different media types, including twisted pair cables such as Cat6, as well as fiber optic cables like single-mode and multi-mode fibers. Ethernet networks can be structured in different physical topologies, including star topology where each device is connected to a central hub or switch, bus topology with devices connected in a linear sequence, and ring topology where devices are connected in a circular sequence.

Frame	Preamble	Start Frame Delimiter	MAC Destination	MAC Source	Lenght	Payload	Frame Check Sequence (CRC)	Interpacket gap
Lenght (bytes)	7	1	6	6	2	42 - 1500	4	12

Figure 2.5: IEEE 802.3 frame

Ethernet frames, according to IEEE 802.3 standard, include a preamble for synchronization, a start frame delimiter, destination and source Media Access Control (MAC) addresses, length, payload, and Frame Check Sequence (FCS) using a cyclic redundancy check for error detection.

2.4.3 SPI

SPI is a synchronous serial communication interface specification primarily used for short-distance communication in embedded systems. It operates in a master-slave architecture with a single master and one or more slaves. SPI uses a clock signal (SCK) to synchronize data transmission between the master and slave devices. This protocol supports a wide range of clock frequencies, ranging from a few kHz to tens of MHz or even higher, depending on the specific hardware capabilities of the devices involved. The bit transfer rate in SPI is directly proportional to the clock speed, as such, it enables transmission rates ranging from a few kilobits per second (Kbps) to tens of megabits per second (Mbps). It also allows for full-duplex data transfer, meaning that data can be simultaneously transmitted and received through separate lines.

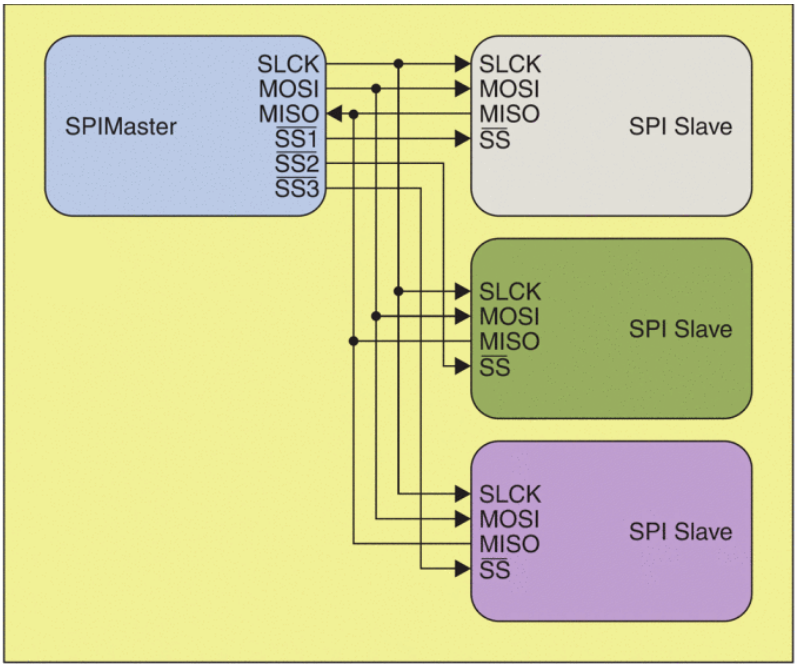

Figure 2.6: SPI master connected to multiple slaves [14].

In SPI, communication typically occurs over four lines: MOSI (Master Out Slave In) for sending data from the master to the slave, MISO (Master In Slave Out) for sending data from the slave to the master, SCK for synchronizing data transmission, and SS (Slave Select), also known as chip select (CS), which the master uses to select the slave device. One of the key advantages of SPI is its ability to operate at high speeds, making it suitable for high-speed data transfers [14].

2.4.4 CAN

CAN bus is a commonly used communication protocol, especially for low-level control and coordination of subsystems. It enables distributed communication among various components or nodes within the AUV, allowing for coordinated control and monitoring. CAN bus is known for its real-time communication capabilities, supporting deterministic communication with low latency, which is essential for time-critical applications within AUVs. The low power consumption of the CAN bus makes it suitable for battery-powered systems, such as AUVs. Additionally, CAN bus uses a priority-based message arbitration mechanism, where messages with lower identifiers have higher priority. This mechanism ensures that critical messages, such as those related to safety or control, take precedence over less critical messages [6].

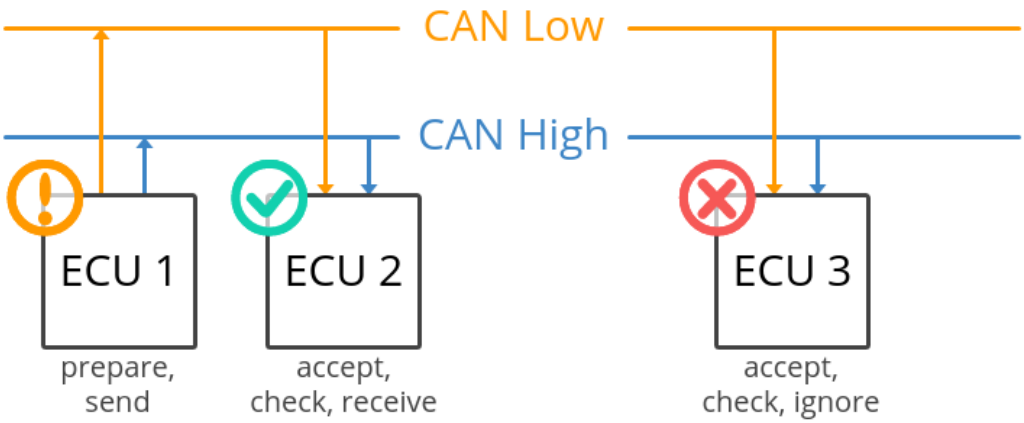

Figure 2.7: CAN bus [10]

Key specifications of the CAN bus include a speed of up to 1 Mbps for Classical CAN and up to 5 Mbps for CAN FD (Flexible Data rate). The bus line can reach up to 40 meters at 1 Mbps, with longer lengths possible at lower speeds. Message length is 8 bytes for Classical CAN and up to 64 bytes for CAN FD. The node capacity can accommodate up to 128 nodes, although the practical limit depends on the application and network load. The CAN bus features sophisticated error detection with Cyclic Redundancy Check (CRC) and acknowledgment mechanisms and uses twisted-pair cable for differential signaling. Voltage levels on the CAN bus include Dominant (0V) and Recessive (2.5V).

2.5 LSTS Vehicle System

LSTS is a research laboratory established in 1997 with researchers from fields such as Electrical, Mechanical, and Computer Engineering and Computer Science. LSTS specializes in designing, constructing, and operating unmanned underwater, surface, and air vehicles and developing tools and technologies for deploying networked vehicle systems.

Over the last 20 years, researchers from LSTS have successfully deployed unmanned air, ground, surface, and underwater vehicles in the Atlantic and Pacific oceans and the Mediterranean

Sea. These vehicle systems implement the LSTS control architecture with the help of the LSTS Neptus-IMC-DUNE software toolchain [16]. This software toolchain is a framework for mixed-initiative control (humans in the planning and control loops) of unmanned ocean and air vehicles operating in communications-challenged environments with support for Disruptive Tolerant Networking (DTN) protocols [11].

Underwater exploration is a challenging and exciting field that relies heavily on autonomous robotics devices to navigate the harsh conditions of the ocean. The core devices inside these systems are microprocessors and microcontrollers which help autonomous underwater vehicles move precisely and make informed decisions. However, communication between these devices is critical and must overcome the challenges of the underwater environment [20].

LSTS employs a well-defined structure to facilitate the development and operation of its autonomous systems. This structure identifies two primary players: the ground station and the vehicle [28].

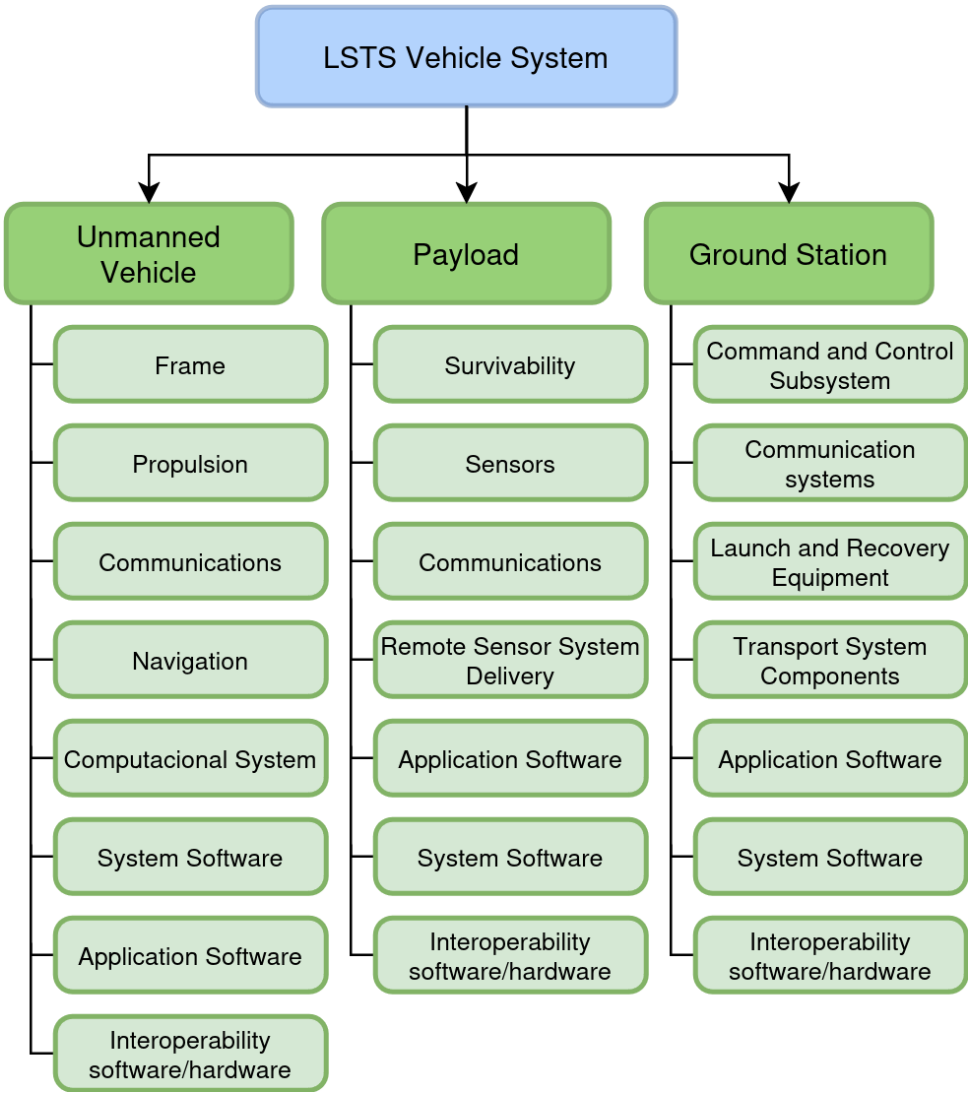

Figure 2.8: LSTS Vehicle System

2.5.1 Unmanned Vehicles

LSTS vehicles comprise several subsystems as described in figure 2.8. These vehicles utilize the IMC protocol for internal, inter-task, and external communications, between nodes within a network. IMC is unique in that it was designed to accommodate unreliable networks with various transport methods. In these types of networks, each node employs one or more transports to communicate with other nodes. Currently, IMC facilitates communication between nodes utilizing Internet Protocol (IP) networking, User Datagram Protocol (UDP) or Transmission Control Protocol (TCP), acoustic modems, telemetry radios, Global System for Mobile (GSM), and Satellite, such as Iridium. Availability of communication means changes based on the environmental conditions of the vehicles. With IMC, a vehicle can alternate between communication methods or even use multiple simultaneously. IMC's addressing mechanism allows for a communication session transfer from one method to another while maintaining the same end peers [21].

Figure 2.9: LSTS LAUV [16]

The unmanned vehicle is a component within the LSTS Structure, designed to perform a variety of missions autonomously. It consists of several subcomponents, each playing a vital role in its operation and functionality:

- Frame - The structural framework of the unmanned vehicle, providing the necessary support and housing for all other components. It is designed to withstand environmental stresses and

impacts while maintaining a lightweight profile for efficient operation.

- Propulsion - This subcomponent includes the engines, motors, and related systems responsible for propelling the vehicle.
- Communications - Focuses on the interfaces and protocols that enable the subcomponents, within an unmanned vehicle, to communicate.
- Navigation - Encompasses the sensors and algorithms required for the vehicle to determine its position, plan its route, and navigate to its destination. This typically includes GPS, IMUs, and other positioning systems.
- Computational System - The onboard computing architecture and devices that process data from sensors, run control algorithms, and manage all operational aspects of the vehicle.
- System Software - The fundamental software infrastructure that supports the operation of all hardware components. This includes the operating system, device drivers, middleware, and other essential software that ensure the vehicle's functionality.
- Application Software - Specific software applications tailored to the mission requirements of the vehicle. These applications process data, control mission-specific equipment, and enable the vehicle to perform complex tasks autonomously.
- Interoperability - Ensures integration and communication between the vehicle's hardware and software components. This involves adhering to standardized protocols and interfaces to enable cohesive operation and interaction.

This work will focus more particularly on the computational system of these vehicles.

2.5.1.1 Computational System

Traditionally, unmanned vehicles utilize a hierarchical computational architecture that integrates microprocessors and microcontrollers to optimize performance and efficiency. The microprocessor serves as the central computing unit, running the DUNE software framework responsible for high-level control, decision-making, and mission planning. It processes data from various sensors, such as GPS, sonar, IMUs, and depth sensors, to determine the vehicle's position, orientation, and depth. The DUNE software then calculates navigation and path planning algorithms to autonomously navigate through the environment and execute tasks to achieve mission objectives [33]. This data, which is essential for DUNE to make decisions, is usually collected by microcontrollers. These are responsible for low-level actuation and control of peripheral devices. Microcontrollers interface with a variety of peripheral devices, including thrusters, propellers, sonar transducers, and other actuators, and each microcontroller is dedicated to a specific subset of devices for efficient and specialized control. They manage tasks such as adjusting thruster speeds or activating/deactivating specific sensors. This architecture allows for the efficient allocation of tasks, microprocessors handle complex computations, navigation algorithms, and data

analysis, while microcontrollers manage time-sensitive operations and quick responses to external events. By offloading tasks to microcontrollers, power management is enhanced, reducing the energy demands of the main CPU.

For each microcontroller to communicate with the main processor, custom message parsers must be implemented to facilitate communication on an application basis. The developer is responsible for designing each message layout and implementing these custom parsers. Then, it is up to each microcontroller and the main CPU to interpret these messages to initiate appropriate actions. These commands allow the main CPU to operate by interacting with microcontrollers, ensuring that low-level control tasks are performed, while also contributing to fault tolerance and redundancy to improve the reliability and safety of AUV operations.

2.5.2 Payload

The payload component within the LSTS Vehicle Structure determines the specific functionalities and capabilities of the unmanned vehicle for different missions. The payload is comprised of several possible subcomponents, each contributing to the vehicle's overall effectiveness and adaptability:

- Survivability - Ensures the payload and the vehicle can withstand harsh environmental conditions and potential threats during missions. This includes protective measures and redundant systems to maintain functionality.
- Sensors - A diverse range of sensors can be integrated to gather data relevant to the mission. These are tailored to the specific requirements of each mission and can range from cameras, sonar, radar, and environmental sensors, among others.
- Communications - Encapsulates communication protocols employed to manage and control the payload-associated components.
- Remote Sensor System Delivery - Enables the deployment and management of remote sensors, which can be placed or dropped in specific locations to extend the range and capabilities of the vehicle's sensing apparatus.
- Application Software - Custom software applications designed to process sensor data, manage mission parameters, and provide the vehicle with the intelligence needed to execute complex tasks autonomously.
- System Software - The underlying software infrastructure that supports the operation of all hardware components (relative to the payload), ensuring they work together. This includes operating systems, device drivers, and additional middleware.
- Interoperability - Ensures the integration between hardware and software within the payload component. This involves adhering to standards and protocols to enable communication and coordination between various payload subcomponents.

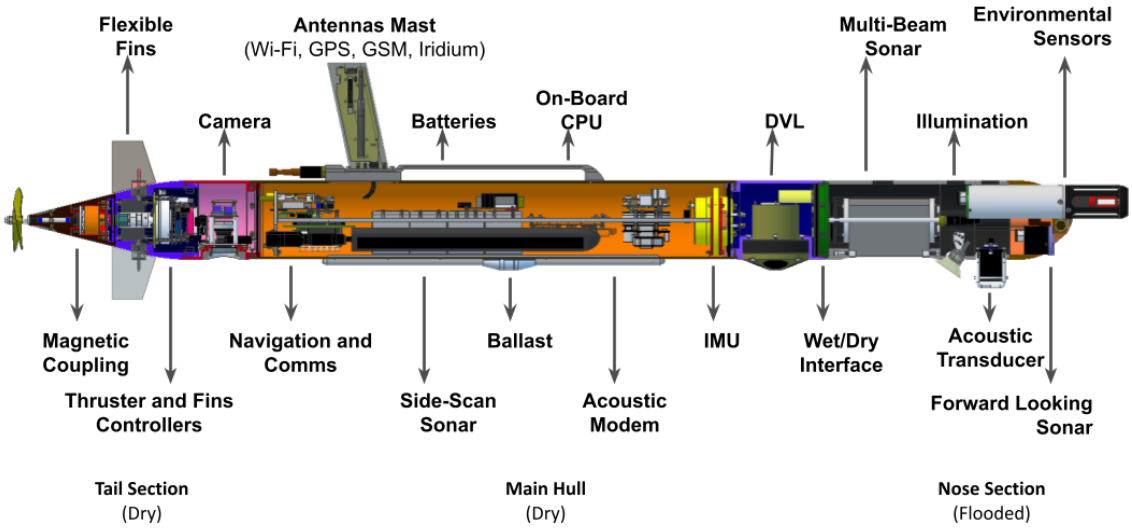

Figure 2.10: Breakdown structure of an LAUV. Designed by LSTS and OceanScan-MST.

Each vehicle is designed to support multiple applications, such as surveillance, environmental monitoring, search and rescue, and scientific research. For each application, a compatible payload must be considered to meet the mission-specific requirements. This involves selecting and configuring the appropriate subcomponents to optimize the vehicle's performance and ensure mission success [20].

2.5.3 Ground Station

The LSTS Ground Station plays an integral role as the central command and control hub for overseeing and managing missions performed by unmanned vehicles, such as AUVs, UAVs, ROVs, and ASVs. It allows human operators to effectively plan, supervise, and monitor missions in real-time, ensuring the successful execution of tasks [7]. Some of its key functions are:

- Mission planning and Execution - Enables operators to plan missions based on objectives, environmental conditions, and operational constraints.
- Runtime Supervision and Monitoring - Allows operators to monitor vehicle telemetry, including position, status, sensor data, and operational parameters, providing visualization tools for vehicle trajectories, sensor readings, and environmental data.
- Mission Revision and Adaptation - Operators can revise and adapt missions by adjusting waypoints, modifying tasks, or changing mission priorities, in response to changing circumstances.
- Data and Status Updates - Provides continuous updates on sensor data, vehicle health metrics, mission progress, and alerts, supplying operators with comprehensive and up-to-date information to make informed decisions.

Additionally, the Ground Station supports the operation of multiple vehicles across diverse communication interfaces:

- Satellite Communications - Facilitates long-range capabilities.
- WiFi - Enables local area network connectivity.
- 4G - Provides reliable communication for coastal operations.
- SMS - Supports basic text-based communication.
- Acoustic Communications - Essential for underwater operations.

This versatility allows the Ground Station to communicate effectively and coordinate multiple vehicles using various technologies. It features a scalable architecture capable of managing and coordinating operations for multiple vehicles simultaneously.

2.6 LSTS Software Toolchain

2.6.1 DUNE

DUNE, Unified Navigation Environment is a sophisticated embedded software designed to be the central nervous system of autonomous vehicles, such as ASVs, ROVs, AUVs, UAVs, and even Manta communication gateways. At its core, DUNE acts as the onboard software responsible for managing all aspects of the vehicle's operation, from interacting with sensors and actuators to handling communication, navigation, control, maneuvering, plan execution, and overall vehicle supervision.

This framework is based on a modular architecture, where each module, called a Task, is responsible for a unique function, much like the UNIX philosophy, where each task should do one job well, and the output of one task may be the input of another, possibly unknown, task. This modularity allows for flexibility and ease of building complex systems by assembling simple tasks. Each task follows a common life cycle and has method handlers for all messages it consumes. Furthermore, each task can register one or more computational entities that can be used to address it uniquely inside a networked vehicle system [29].

Tasks communicate with each other asynchronously using a message bus, which forwards IMC messages from the producer to all registered receivers. DUNE uses a layered control architecture with actuation tasks in close contact with hardware, followed by guidance tasks that translate a desired path into actuation, and then maneuver tasks that execute a higher level of behavior by controlling the desired path of the vehicle. In an autonomous plan, a series of sequential maneuvers form a plan that is requested by a planning task, and this is where IMC nested messages support comes into play.

DUNE boasts a small memory footprint, not exceeding 16 Megabytes. This efficiency is crucial for resource-constrained environments in autonomous vehicles, where optimizing memory

usage is paramount. Additionally, the DUNE source code includes a comprehensive set of sensor drivers, covering a wide array of sensors relevant to the navigation and control of unmanned systems. As such, DUNE stands out as a versatile and modular embedded software solution, capable of powering a diverse range of autonomous vehicles. Its efficient communication framework, small memory footprint, and inclusive sensor drivers make it a valuable tool for developers and engineers working on unmanned systems across various domains [17].

2.6.2 IMC

The Inter-Module Communication protocol, developed at LSTS, aims to create interconnected systems involving vehicles, sensors, and human operators. These systems can collaboratively pursue shared goals by exchanging real-time information and updated objectives. IMC simplifies the complexities of hardware and communication diversity by offering a common set of messages that can be serialized and transmitted through various channels [18]. This protocol has several key features that make it a powerful and versatile communication tool in various systems:

- Common Language Set - IMC serves as a unifying language set embraced by diverse systems, consoles, robots, servers, and other components. This commonality ensures seamless communication and interoperability across the entire system architecture.

- Message-oriented - Every conceptual entity within the system is represented by one or more messages. This includes commands, replies, system states, configurations, and various other entities, enabling a clear and modular approach to communication.

- Nested Messages - Supports the nesting of messages, allowing for hierarchical representation of information, providing where high-level messages encapsulate in a single message multiple low-level messages.

- Transport Agnostic - IMC defines mechanisms addressing, serialization, and discovery remaining independent of the transport method used.

IMC is a protocol that is agnostic to its transport layer, this flexibility is achieved due to its characteristics. This protocol establishes methods for addressing through unique identifiers specific to the system in which it is utilized. It also allows for the creation of entities, within a system, with unique identity numbers. IMC also establishes mechanisms for serialization that are independent of different protocols. This ensures that each IMC message is consistently serialized into a stream of bits, making it adaptable to various protocols. If the data stream exceeds the allowed size for a particular protocol, IMC permits message fragmentation, or a special serialization for that protocol can be implemented. Lastly, IMC establishes mechanisms for discovery, allowing a system to be identified within a network and announce its entities. Since IMC is a protocol agnostic to the transport utilized it is positioned in the application layer of the OSI model 2.3.

Header		
Synchronization Number	2 bytes	Marks the beginning of a packet.
Message Identifacation Number	2 bytes	The identification number of the message contained in the packet. This field is used for correct message interpretation and deserialization.
Message size	2 bytes	The size of the message data in the packet
Timestamp	2 bytes	The time when the packet was sent, as seen by the packet dispatcher, in seconds since Unix epoch
Source Address	2 bytes	The Source IMC system ID
Source Entity	1 bytes	The entity generating this message at the source address
Destination Address	2 bytes	The Destination IMC system ID
Destination Entity	2 bytes	The entity that should process this message at the destination address
Payload		
Varies according to message definition		
Footer		
Check Sum	2 bytes	Computed using the CRC-16-IBM. The data contributing for the CRC includes all preceding header and message bytes.

Figure 2.11: IMC message layout

The IMC protocol defines its structure and specifications in an Extensible Markup Language (XML) document, providing a clear and standardized representation of the protocol's elements. The XML implementation serves multiple purposes, such as documentation, serialization rules, and aiding in continuous development and testing. Each message and its fields are defined within the XML document. Fields within a message must have at least a name, abbreviation (used for code generation), and type. Optionally, units and a range of permissible values can be defined. This structured definition in XML enables a clear representation of message structures and facilitates documentation [21].

The payload field differentiates the various IMC messages, and it is up to the developer to define its variables and each variable type. This protocol is extremely versatile in this field and allows multiple field types such as basic variables and more complex types with nested messages. Within these basic types, some can also be assigned special meanings with the use of enumerations, which allow integer values to have a more symbolic value, and bitfields, which allow integer types to associate with multiple flags at the same time. Some basic variable types such as:

Name	Size	Description
int8_t	1	8 bit signed integer.
uint8_t	1	8 bit unsigned integer.
int16_t	2	16 bit signed integer.
uint16_t	2	16 bit unsigned integer.
int32_t	4	32 bit signed integer.
uint32_t	4	32 bit unsigned integer.
int64_t	8	A 64 bit signed integer
fp32_t	4	32 bit single precision floating point number in IEEE 754 format.
fp64_t	8	64 bit double precision floating point number in IEEE 754 format.
rawdata	n/a	Variable length byte stream.
plaintext	n/a	Variable length ASCII character stream.
message	n/a	An inline message. Useful for encapsulating other messages.
message-list	n/a	A list of messages.

Table 2.1: Field types in IMC messages

This protocol also supports nested messages, which refer to the capability of including one or more messages within another message. This functionality enhances the protocol's flexibility, allowing for a more efficient and organized exchange of information between modules within a system. With nested messages, a single overarching message can encapsulate multiple sub-messages, each carrying specific data or commands. This hierarchical structure enables a more streamlined communication process, as modules can cohesively transmit complex information. The ability to nest messages is especially beneficial in scenarios where interrelated data needs to be conveyed simultaneously, reducing the overhead associated with managing multiple independent messages.

Name	Abbreviation	Unit	Type	Description
Plan ID	plan_id	-	plaintext	The plan's identifier.
Plan Description	description	-	plaintext	Verbose text description of plan.
Namespace	vnamespace	-	plaintext	Namespace for plan variables.
Plan Variables	variables	-	message-list (Plan Variable)	Plan variables.
Starting maneuver	start_man_id	-	plaintext	Indicates the id of the starting maneuver for this plan.
Maneuvers	maneuvers	-	message-list (Plan Maneuver)	List of maneuver specifications.
Transitions	transitions	-	message-list (Plan Transition)	List of maneuver specifications.
Start Actions	start_actions	-	message-list	Contains an optionally defined 'MessageList' for actions fired on plan activation.
End Actions	end_actions	-	message-list	Contains an optionally defined 'MessageList' for actions fired on plan termination.

Figure 2.12: Plan Specification Message [8]

Chapter 3

Literature Review

The field of microcontroller applications has been rapidly advancing, showcasing the capabilities of microcontrollers in various applications. These innovations are crucial for enhancing the performance, efficiency, and versatility of embedded systems used across different industries This chapter provides a literature review focused on implementing a communication protocol in microcontrollers. It outlines the research strategy and examines key studies, highlighting their problems and their methodologies. It then analyzes the relationships between various studies and discusses their relevance to the current research context.

By exploring related work, the chapter provides a foundation for the problem being addressed, identifying critical frameworks and evaluating methodologies. The discussion concludes by summarizing the contributions and limitations of existing research, setting the stage for further exploration in subsequent chapters.

3.1 Research strategy

To perform a literature review, it is essential to analyze the fields relevant to this work. The research strategy was structured to cover these relevant fields, starting from a broad spectrum of communication-related domains and narrowing down to the specific area of microcontroller communications.

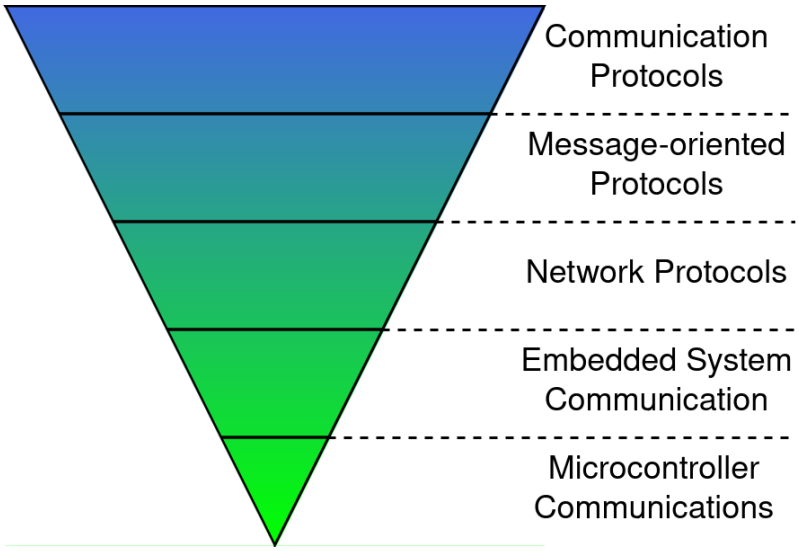

Figure 3.1: Relevant fields

1. **Communication Protocols**: This foundational field sets the stage for understanding how data and messages are exchanged between systems. It encompasses all protocols designed to enable communication between different systems or devices, including everything from high-level internet protocols to low-level data link protocols. Understanding this broad field provides the foundational knowledge necessary to grasp the more specific communication mechanisms used in microcontrollers. All subsequent fields are subsets of this broad area, providing a comprehensive understanding of various communication mechanisms.

2. **Message-oriented Protocols**: This subset focuses on the specific nature of message passing. It involves protocols designed to send messages between systems, ensuring that messages are correctly sent, received, and interpreted by different nodes in a network. This is crucial for reliable communication and helps in understanding how data integrity and synchronization are maintained.

3. **Network Protocols**: This field is crucial for understanding how communication protocols function over different types of networks. It examines protocols that manage and facilitate communication over both wired and wireless networks, involving aspects like routing, data integrity, and security. This knowledge is vital for implementing communication protocols in microcontrollers used in networked environments, where efficient and secure data transmission is paramount.

4. **Embedded System Communication**: This area delves into the specific challenges and requirements of communication within embedded systems. Embedded systems often have resource constraints and specific real-time communication needs, making this field critical for developing efficient and reliable communication protocols. It highlights the importance of resource efficiency and real-time communication, providing a context for developing protocols tailored to embedded environments.

5. **Microcontroller Communications**: This is the most specific category, handling the unique requirements and methods necessary for microcontroller communication. It focuses on how communication protocols are implemented and utilized within microcontrollers, involving understanding the hardware and software constraints specific to microcontrollers. This ensures the development of efficient and effective communication solutions, including considerations of hardware constraints, power consumption, and real-time processing capabilities.

Based on these fields, relevant keywords were extracted to guide the literature research for this work. These keywords are:

- Embedded Systems
- Real-time Communication
- Protocol Implementation
- Communication Protocols
- Robotic Middleware
- Network Protocols
- Microcontroller

3.2 Selected Publications

This research [37] is focused on message-oriented protocols for embedded system communications in the context of wireless sensor networks (WSNs). This is directly connected with this work since IMC is a message-oriented protocol and wireless sensors are commonly implemented with microcontrollers.

The primary issue addressed in this work is the propagation of information within WSNs, which have limited energy resources for sensor nodes. The research aims to fill the gap left by traditional middleware systems, which are unsuitable for WSNs due to their lack of consideration for energy consumption. Additionally, the request/response communication mechanisms of traditional systems are not ideal for the event-driven nature of WSN applications.

In response to these challenges, the study proposes the design of a message-oriented middleware system named WMOS (Wireless Message-Oriented System), which is developed on TinyOS. WMOS is designed to be scalable and fully distributed, using a modular approach for practical implementations. It includes a Self-Adaptive Multilevel QoS Service aimed at saving power and a Topic/Content Double Mode Based on XML that ensures more effective query information management compared to other middleware.

The research on WMOS aligns closely with the principles of IMC protocols in microcontrollers. Both WMOS and IMC emphasize a message-oriented approach, scalability, and modularity, enabling decentralized and efficient communication.

The PX4 framework [23], designed for microcontroller applications, provides valuable insights for implementing IMC protocols in microcontrollers. The use of a multithreaded, publish-subscribe design pattern executed by a micro object request broker (μORB) ensures inter-process communication (IPC) and real-time data distribution, which are critical aspects of the IMC protocol.

The main problem addressed by this research paper is the complexity and lack of flexibility in existing robotics software frameworks for deeply embedded systems. This aligns with the goals of this IMC implementation to create adaptable and maintainable communication mechanisms. As robotics research advances the software complexity increases, making it challenging for individual research groups to develop comprehensive systems. Current systems often lack modularity, efficient inter-process communication, and integration with higher-level systems, hindering the development and deployment of sophisticated robotic applications.

The integration of PX4 with higher-level systems, demonstrates the potential for IMC protocols to interface with both low-level and high-level systems, enabling complete and distributed system designs. By understanding the principles of PX4, the IMC protocol can achieve efficient data exchange, real-time performance, and ideal integration, thus enhancing the development and deployment of more sophisticated microcontroller-based applications.

MIRA [9] is a robotic middleware designed to facilitate applications in both research and real-world environments. Many existing middlewares rely on a central device to manage communication. However, this central point of failure can cause the entire system to halt if the device fails, which is particularly problematic in scenarios involving multiple robots, as each robot should be capable of independent operation.

This work proposes a decentralized communication structure with two primary communication techniques: message passing and remote procedure calls (RPC). In MIRA, named channels enable message passing and follow a publisher-subscriber communication model. Channels are strongly typed and ensure type safety. MIRA manages concurrent multi-threaded access to channels automatically since each channel is implemented as a queue of data slots, allowing multiple readers and writers to access data simultaneously without interference. As for RPC, MIRA automatically selects the optimal communication method based on the context. Additionally, MIRA supports exception passing in RPC, transporting errors on the service side back to the caller for more robust error handling.

The decentralized communication structure and robust error handling in MIRA align closely with the goals of this dissertation. This IMC adaptation could benefit from the decentralized computational model and the publisher-subscriber communication interaction. The ability to select different communication methods and handle exceptions also is a key factor for the IMC implementation since it improves the reliability and flexibility of the protocol, providing fault-tolerant communication in microcontroller-based systems.

In [31], the primary problem addressed by this paper is the inefficiency of cloud computing in handling the vast amounts of data generated at the edge of the network. Cloud computing's limitations, such as latency, bandwidth constraints, and privacy concerns, make it unsuitable for many

real-time and sensitive applications. Edge computing aims to mitigate these issues by enabling data processing at the edge, closer to where data is generated.

The methodology involves a thorough analysis of why edge computing is necessary, followed by a detailed definition and vision of edge computing. The paper provides case studies, including cloud offloading and smart environments like homes and cities, to illustrate the practical applications of edge computing.

The paper concludes that edge computing is essential for the future of data processing due to its ability to provide shorter response times, better reliability, and bandwidth savings. Edge computing transforms the role of network edges from mere data consumers to both data producers and consumers. The study emphasizes that while edge computing presents significant benefits, it also introduces challenges such as maintaining network reliability, managing data privacy, and optimizing energy consumption.

Edge computing's emphasis on localized data processing at the acquisition point is a key motivation for implementing IMC in microcontrollers. By leveraging edge computing principles such as reduced latency and enhanced data management, IMC can optimize communication pathways, ensuring faster data exchange and better resource utilization.

3.3 Analysis of Publications

The following analysis explores the interconnections between research papers in the domains of embedded systems, distributed sensor networks, and robotics. By examining the relationships between these works, a deeper understanding emerges of how their methodologies and findings converge to address common challenges in microcontroller hardware and embedded systems. The key points of energy efficiency, scalability, communication mechanisms, decentralization, and integration with higher-level systems serve as the foundation for this comparative analysis, revealing shared strategies and complementary solutions across the different research domains.

Energy Efficiency

The papers *WMOS* [37] and *Edge Computing* [31] both stress the importance of processing data efficiently closer to the data source. *WMOS* specifically looks at minimizing energy consumption in wireless sensor networks through scalable and distributed middleware. At the same time, edge computing tackles similar issues by processing data at the network's edge to address latency and bandwidth concerns.

Real-Time Performance

The work *PX4* [23] highlights the critical need for real-time performance, achieved through an RTOS, the NuttX RTOS. The PX4 framework leverages this RTOS to enable a multithreaded environment on microcontrollers and employs a publish-subscribe design pattern. This setup ensures the required multitasking capabilities, allowing various tasks to run concurrently. Additionally,

the RTOS manages software and hardware resources, providing the essential operating system services to the framework applications. It also abstracts the hardware details from the software application, enhancing portability across different hardware platforms.

Traditional RTOS concepts, such as basic scheduling and resource management, have been upheld throughout the evolution of these operating systems. The key driver for RTOS evolution has been the advancement of hardware capabilities. For instance, the introduction of multicore architectures [26] originated enhancements in task scheduling and parallel processing. Then the rise of IoT led to the integration of low-power and connectivity features [38], while recently security limitations [4] have driven advancements in RTOS systems.

Scalability and Modularity

The works *PX4* [23], *MIRA*, and *WMOS* [37] present methodologies to develop modular and scalable frameworks. They emphasize modularity, facilitating adaptation to various applications and integration of new technologies.

Communication Mechanisms

Additionally the same works, *PX4* [23], *MIRA* [9], and *WMOS* [37], utilize a publish-subscribe design pattern. WMOS incorporates this pattern to support its Topic/Content Double Mode Based on XML. PX4 implements the publish-subscribe mechanism through μORB. MIRA uses named channels with the same design pattern for message passing. These publish-subscribe mechanisms highlight a shared strategy for handling data distribution.

Decentralizated Computing

MIRA [9] addresses the issue of fault tolerance in a centralized system by proposing a decentralized communication structure. This decentralization is mirrored in *Edge Computing*'s [31] approach to processing data at the edge of the network. This strategy reduces latency and alleviates strain on central components. [37], with its distributed design, inherently supports strategy, which aligns with *MIRA*'s objectives.

Integration with Higher-Level Systems

PX4 [23] is implemented to integrate with higher-level systems for creating a distributed system that leverages real-time performance. This integration is similar to *Edge Computing*'s [31] aim of transforming network edges into both data producers and consumers, emphasizing cohesive data management and processing across different layers of the network.

To conclude the literature review and discuss the importance of the key points for implementing IMC in microcontrollers, it is evident from the reviewed papers that energy efficiency, real-time operation, scalability and modularity, communication mechanisms, computational decentralization, and integration with higher-level systems are crucial considerations.

Despite these advancements, there exists a gap in the literature. Current works focus only on some individual aspects like energy efficiency and communication protocols but do not fully address all issues simultaneously. Developing a middleware tailored to distribute computational loads across microcontroller devices while ensuring real-time operation could bridge this gap. This design based on the IMC protocol would enable this protocol to interface with both high-level and low-level systems, making it ideal for developing scalable and modular systems, since IMC is agnostic to the transport layer. Additionally, implementing this in microcontrollers would reduce power consumption by optimizing resource allocation and operational efficiency.

This dissertation aims to address all these essential subjects. It seeks to facilitate interaction with high-level systems, such as DUNE, which perform functions like planning and navigation. Moreover, it explores the decentralization of computational systems through peripheral components that enable data processing during collection. Scalability and modularity are also emphasized to adapt to various applications and integrate new technologies.

3.4 Related Work

In terms of related work to DUNE-IMC, there is a framework called ROS which can be implemented in microcontrollers with micro-ROS.

The Robotic Operating System is an open-source middleware framework that contrary to its name, is not an operating system but rather an abstraction layer that facilitates communication between various components in a robotic system. Its primary goals include providing a flexible and modular framework for building robotic software, fostering collaboration through open-source development, and offering a standardized platform-independent environment. ROS supports a publish/subscribe communication model, allowing nodes to communicate seamlessly through ROS topics. Its use cases span a wide range of applications, from industrial robots to autonomous vehicles and drones [19].

When comparing IMC to ROS topics, we can see that ROS allows for both synchronous and asynchronous communication, topics can be added easily, the implementation depends on the method used, and communications are centralized on the "master" node. On the other hand, IMC allows for asynchronous communication, messages can be added to local (private) protocols or merged with centralized definitions, there is a stable set of messages for multiple vehicles since 2014, the implementation is transport-agnostic, and there are mechanisms for discovery and broadcast. While IMC is specifically designed for the marine domain, addressing unique challenges in that field, ROS topics are general-purpose and widely used in various robotic applications beyond just marine systems.

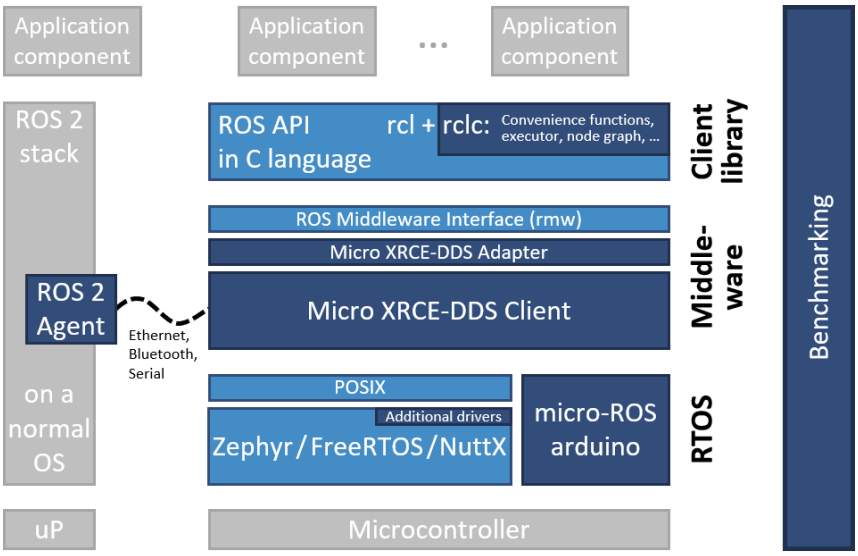

Figure 3.2: Micro-ROS Architecture [25]

Micro-ROS is an open-source project that extends the capabilities of the ROS to microcontrollers. Similar to DUNE, ROS is a flexible framework for writing robot software, providing tools and libraries for various tasks such as hardware abstraction, device drivers, communication between processes, and more. Micro-ROS specifically focuses on enabling ROS functionality on resource-constrained microcontroller platforms. This framework provides real-time capabilities for microcontroller applications, especially for tasks that require precise timing and control. Micro-ROS is designed to work seamlessly with ROS 2, ensuring compatibility and allowing for communication between microcontroller-based systems and more powerful computing platforms in a ROS 2 ecosystem [25].

Micro-ROS utilizes the Data Distribution Service as its communication middleware, which is a protocol for real-time and scalable communication that is commonly used in the ROS 2 ecosystem. It provides a standardized way for different components of a system to communicate with each other seamlessly. DDS follows a publish-subscribe model, where data publishers send information to specific topics, and data subscribers receive the information they are interested in without direct connections between them [27].

To act as a bridge between the microcontroller and the ROS 2 ecosystem, the Micro-ROS Agent is responsible for translating messages between the microcontroller and the ROS 2 communication middleware. This component acts as a server between the DDS Network and micro-ROS nodes inside the MCU. It receives and sends messages from Micro-ROS nodes, and keeps track of the Micro-ROS nodes exposing them to the ROS 2 network. The node interacts with DDS Global Data Space on behalf of the Micro-ROS nodes [24].

Micro-ROS also provides a client library that runs on the microcontroller. This API enables the creation of ROS 2 nodes on microcontroller platforms, which are software entities that perform specific tasks. By publishing and subscribing to topics using DDS, micro-ROS nodes on microcontrollers can participate in the ROS 2 communication ecosystem.

Chapter 4

Approach

This chapter presents the approach employed in this study to develop and evaluate the implementation of a communication protocol in microcontroller-based systems. The work problem addressed in this thesis is the challenge of data exchange between components within unmanned vehicles. Ensuring cohesive communication among various vehicle components is critical for achieving efficient control and adaptability in dynamic underwater environments.

This work proposes implementing a standardized communication protocol in microcontrollers to enhance computational efficiency by distributing computational load more effectively. As such the protocol must be abstract to the specific communication protocol, ensuring seamless integration and interoperability between various devices of the computational system architecture. Standardizing protocol communications promotes modularity and scalability, allowing for easier upgrades and expansions without extensive modifications to the existing computational architecture.

4.1 Project Requirements

In developing a framework for implementing the IMC protocol on microcontroller-based systems, clearly defining both functional and non-functional requirements is essential. These requirements guide the design, development, and testing phases, ensuring the final system meets the intended objectives and user needs. This section outlines the specific functional and non-functional requirements identified for the project.

4.1.1 Functional Requirements

The functional requirements focus on what the system must do, providing detailed descriptions of the necessary features and capabilities. These requirements are essential for ensuring that the system performs its core functions effectively. The key functional requirements for the Microcontroller Unit (MCU) based IMC protocol implementation are as follows:

ID	Requirement Description	Priority
R1	The system shall send and receive IMC packets.	High
R2	The system shall reserve IMC entity IDs for communication.	High
R3	Each task shall be associated with a unique name and ID.	High
R4	Tasks shall be able to communicate with each other.	High
R5	The system shall detect if tasks are stalled.	Medium
R6	The system shall restart stalled tasks.	Medium
R7	The system shall continue to operate if a task fails.	Medium
R8	The system shall be able to switch communication interfaces at runtime.	Medium
R9	The system shall be able to synchronize its clock.	Medium
R10	The system shall be able to save Task parameters to Flash memory.	Low
R11	The system shall be able to retrieve Task parameters from Flash memory.	Low

Table 4.1: Functional Requirements

4.1.2 Non-Functional Requirements

In addition to the functional requirements, non-functional requirements are also critical. These requirements define the system's operational attributes and constraints, ensuring that it performs reliably, efficiently, and within acceptable parameters. The non-functional requirements for this project include:

ID	Requirement Description	Priority
R1	The system shall handle up to 200 messages per second.	Medium
R2	The system shall use less than 5KB of RAM.	Medium
R3	The system shall use less than 300KB of flash memory.	Medium
R4	The system shall correct any clock drift every 60 seconds.	Medium
R5	The system shall enter low-power standby mode while no task is running.	Low
R6	The system shall operate using RS-232 in a point-to-point external network.	Low
R7	The system shall operate using Ethernet in a multipoint external network.	Low

Table 4.2: Non Functional Requirements

4.2 Hardware board

The Hardware layer serves as the foundation for the software architecture, built around the STM32 Nucleo-144 board, which features an STM32F767ZI microcontroller featuring one Arm Cortex-M7 core [35]. Its abundant Input/Output (I/O) capabilities, which include Ethernet, CAN, SPI, and Universal Asynchronous Receiver-Transmitter (UART), coupled with sufficient RAM and a fast, efficient CPU, provide the necessary computational power and connectivity required for this work.

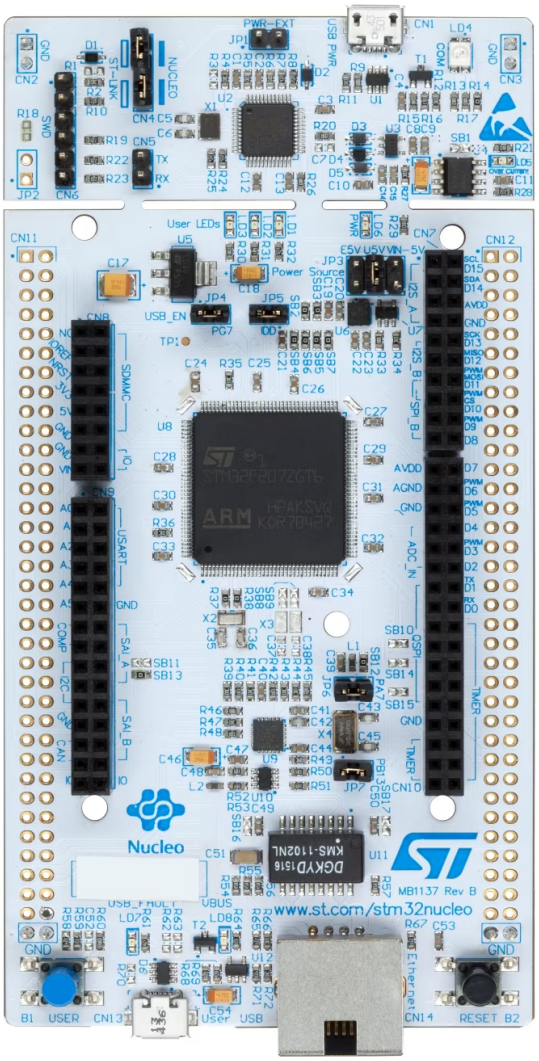

Figure 4.1: STM32 Nucleo-144 board [34]

The inclusion of Ethernet, CAN, SPI, and UART interfaces enhances its versatility, enabling an ideal integration with a wide range of devices and systems commonly used in AUV applications. Additionally, the ample RAM (Random-access memory) ensures efficient data handling, while the high-speed CPU enables rapid computation and response times, needed for real-time operation in dynamic underwater environments.

4.3 Software Architecture

4.3.1 Overview

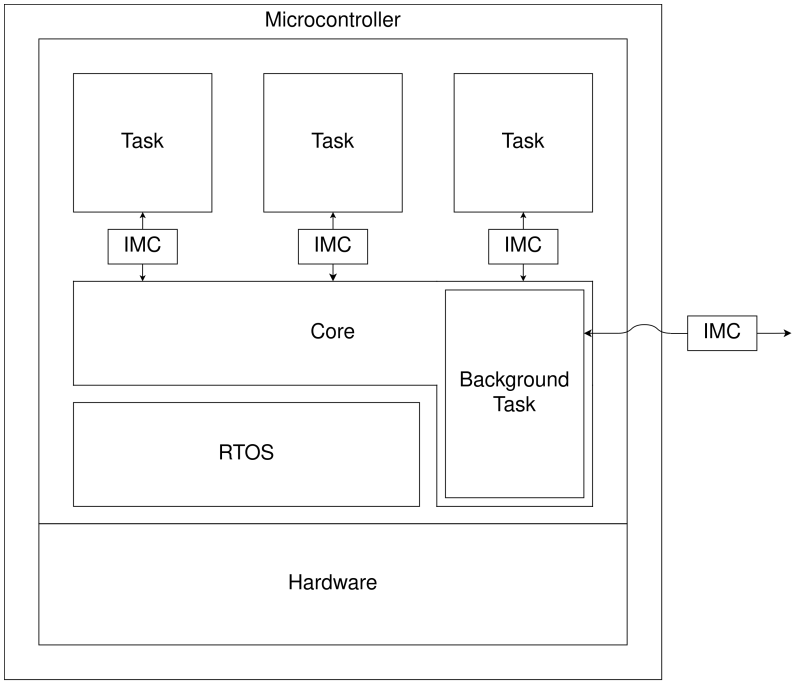

Figure 4.2: System Architecture

A hardware layer consisting of microcontrollers is the software system foundation. Above this is located, an RTOS, responsible for managing thread concurrency, resource allocation, and exclusion to ensure efficient and timely execution of tasks. At the core of the architecture sits the "Core" application layer, which manages the system's functions. It initializes tasks, interfaces with the microcontroller components and the RTOS, and configures tasks based on stored configurations in memory.

The main works of this system are its multiple tasks, each serving specific functions as dictated by the system's requirements. These tasks communicate with each other via the IMC protocol, facilitating seamless data exchange and coordination. Additionally, they communicate externally using IMC, enabling interaction with other systems or components beyond the AUV. This modular approach allows tasks to execute diverse functions based on real-time needs, contributing to the system's adaptability and efficiency.

The software architecture for this system will be developed in C++ using PlatformIO as the development environment. PlatformIO is an open-source ecosystem for embedded development providing a cross-platform build system, library manager, and support for a wide range of development boards and frameworks. The implementation utilizes the Arduino framework and is based on the STM32 platform, guaranteeing compatibility with the target hardware. Additionally, the LwIP library, which has been ported to this RTOS implementation, will handle IP networking, providing robust and reliable communication capabilities within the system.

4.3.2 Components

4.3.2.1 RTOS

The RTOS component of this software system serves as a critical layer responsible for scheduling task execution with precise time constraints. Implementing an RTOS in a microcontroller has several benefits in enhancing its functionality since it allows for the scheduling and management of multiple tasks to be executed concurrently, ensuring efficient use of resources. Moreover, it protects resources from concurrent access or execution and facilitates inter-task synchronization and communication, enabling different tasks to communicate and synchronize. RTOS also guarantees deterministic behavior, crucial in real-time applications as it executes critical and higher-priority tasks within their designated time frames.

Choosing to develop a custom RTOS for this implementation offers several advantages. A custom RTOS provides complete control over the system's behavior, allowing every aspect of thread scheduling, interrupt handling, and resource management to be configured specifically for this implementation. This control ensures that critical tasks are executed within precise time constraints when needed. A custom RTOS can be tailored to meet the specific requirements of the IMC protocol implementation. Developing a custom RTOS allows the incorporation of unique features and functionalities. For instance, we can implement specialized thread management such as mechanisms for restarting threads by reutilizing the pre-allocated stack and additional resource management functionalities. These features ensure that the system addresses the specific challenges and requirements of the IMC protocol.

While choosing to develop a custom RTOS offers several advantages, it also enforces some limitations. One significant limitation is that the RTOS will need to be implemented using Assembly language for thread context switching, which demands managing CPU registers. This is only possible using low-level languages closely tied to the hardware architecture of the CPU. As a result, this ties the RTOS to the specific CPU architecture it is based on. Any change in the hardware architecture would require implementing context switching for the new hardware architecture. While the use of Assembly language will be reduced, it's important to note that developing in Assembly is more complex and error-prone, thus increasing development time. As such, this implementation scope will only be the STM32F767ZI CPU's architecture, ARMv7-M [2].

In essence, the RTOS component in this software system is designed as a kernel that seeks to enable a multitasking environment and includes essential resource management primitives. The RTOS will incorporate a thread component, which facilitates multitasking by allowing the concurrent execution of multiple tasks. Additionally, it will include a mutex component to manage resource allocation and ensure mutual exclusion, preventing multiple threads from accessing the same resource simultaneously. Furthermore, a semaphore component will be integrated to control resource management by regulating the number of accesses to a common resource. Lastly, the message queue component will enable inter-thread communication within the system.

Functionality	**Description**
Thread	Enabling the concurrent execution of multiple threads, and executing them following a priority-based scheduling.
Mutex	Provide exclusive access to a shared resource by allowing only one thread to own the mutex at a time
Semaphore	Control access to shared resources by regulating the number of concurrent accesses.
Message Queue	Enables inter-thread communication by allowing threads to send and receive messages.

Table 4.3: Required RTOS Functionalities

4.3.2.2 Core

The Core layer is situated directly above the RTOS, linking the hardware-focused RTOS and higher-level application-specific tasks. Its main function is to ensure proper operation and communication within the system and with external entities using the IMC protocol.

This layer has several key responsibilities. Firstly, it is in charge of announcing the presence of the microcontroller in the computational system and registering each IMC entity to ensure they are recognized and prepared for interaction. Communicating using IMC requires the system to have its IMC entities properly configured, these are necessary to transmit and receive IMC messages. To create an IMC entity, there must be an association between a unique name and a unique ID since each IMC message is dispatched with the owner's ID as its source Entity field 2.11. This association must be known by the entire system, so this approach proposes that a set of labels is sent to DUNE during the announce sequence. DUNE will then be in charge of associating each label with a unique ID, ensuring that all entities have a unique ID. This process guarantees consistent and accurate communication across the system. Additionally, each IMC entity is configured within a Task and implemented as a thread. Encapsulating each IMC entity of the application with a dedicated Task ensures system integrity by preventing one Task from interfering with another's. Associating parameters and states with an IMC entity allows the operator to change the task behavior at runtime without reconfiguring the application. This flexibility enables more diverse operations and easier adjustments to task functionality as needed.

In addition to its system announcement role, the Core layer is vital for Task parameters management and fault recovery. It is responsible for storing, retrieving, and modifying Task parameters in Flash memory, given that these devices lack a dedicated file system. By serializing and storing parameters in Flash memory, the Core layer ensures their preservation after a system reset. In the event of system resets, the Core layer promptly reconfigures tasks using these stored parameters to restore functionality and minimize downtime. Additionally, this layer will periodically check if each task is blocked. If a Task is detected to be stalled, this layer will attempt to recover it by restarting the associated thread.

Furthermore, this layer is responsible for handling external communications. This is done by creating a Background Task that processes incoming packets and transmits the necessary packets to external devices. This Task selects the communication interface based on its parameters and performs the conditions to interact with the selected interface. Once the entity configuration is complete, this Background Task will notify DUNE of the messages each Task is subscribed to.

This layer closely interacts with the RTOS to manage task scheduling and resource allocation, ensuring that Tasks are scheduled efficiently and system resources are utilized optimally, thereby maintaining system performance and responsiveness. This layer is responsible for message distribution following a publisher-subscriber pattern, where each IMC message is shared to reduce memory usage and, with the help of the RTOS, provide resource exclusion. Lastly, this component is responsible for clock synchronization between devices to use an IMC timestamp with the same time reference point of DUNE. This synchronization ensures that all communications and operations within the system are accurately timed.

Functionality	**Description**
Entity Configuration	Manages the configuration and reservation of IMC entities IDs. Ensures periodic announcements and proper sequence configuration.
Parameter Management	Handles the modification of task parameters, in Flash memory. Also must notify tasks of parameters changed.
Task Recovery	Detects and attempts to recover unresponsive tasks to maintain system stability and functionality.
Background Task	Oversees the creation, setup, and operation of background tasks. Ensures communication interfaces are configured and can be modified at runtime.
Message Distribution	Facilitates the distribution of messages in a publisher-subscriber pattern. Ensures messages are deleted once all subscribers have processed them.

Table 4.4: Required Core Functionalities

4.3.2.3 Tasks

The Tasks layer is a vital part of the system, situated above the Core application, and represents the modular aspect of the system architecture. Within this layer, tasks are associated with IMC entities, each with its parameters and state. These tasks are designed to perform specific operations and be abstract to each other. Tasks within this layer communicate asynchronously with each other using the IMC protocol. Each task is equipped with a mailbox to receive IMC messages, where these messages are shared objects that tasks use to interact and synchronize their activities. This mailbox system ensures that tasks can handle incoming data and commands, maintaining the system's responsiveness and operational integrity. The flexibility and adaptability of the Tasks layer are key to the system's performance. By allowing tasks to change behavior based on their parameters and state, the system can dynamically adjust to varying operational conditions and requirements. This

modular approach not only enhances the overall system's ability to perform complex missions but also ensures robustness and scalability in handling diverse underwater scenarios.

In this system architecture, various types of tasks should be employed to ensure comprehensive functionality.

- Actuator Tasks - These tasks control actuators directly connected to the microcontroller, for example, PWM modules, such as motors or LEDs. Microcontrollers are well-suited for managing these components.
- Monitor Tasks - These tasks monitor sensor inputs, system states, or environmental conditions, triggering appropriate responses based on predefined thresholds or conditions.
- Control Tasks: Basic control tasks, such as small PID controllers, can be implemented on microcontrollers to manage system behavior.
- Supervisor Tasks: These tasks supervise system behavior, monitor sensor inputs, and enforce real-time safety protocols or operational constraints.

When developing applications for microcontrollers, it is essential to minimize the number of IMC messages to only those necessary. Microcontrollers have limited processing power and memory, so reducing the number of messages helps conserve these resources, ensuring efficient task handling without overwhelming the microcontroller. To achieve this, it is crucial to consider the size of some messages. The IMC protocol supports nested messages, which can have heavy payloads. These messages, often related to high-level control tasks like maneuvers and plan supervision, should be avoided on microcontrollers due to resource constraints. Running such tasks on a microcontroller may not be feasible because of its limited resources. By focusing on essential, lightweight messages, the system can maintain efficient communication and processing, aligning with the microcontroller's capabilities and ensuring optimal performance.

Functionality	**Description**
Task Parameters	Capability to create, update, and monitor changes to all parameters associated with a task.
Entity State	Associate each task with a unique state identifier, allowing for updates and status reporting as the task progresses through different execution phases.
Message Handling	Enable tasks to subscribe to any IMC message, dispatch IMC messages with correct source information, and consume received messages.

Table 4.5: Required Task Functionalities

4.3.3 Communications

For implementing the IMC protocol on microcontroller-based systems, it's important to support various communication protocols that are commonly used in robotic embedded systems. Some of the key protocols that are commonly used on these circuits:

- RS-232 - One of the more common standards for serial data transmission. This protocol is widely used due to its simple implementation and reliability for point-to-point communication.
- Ethernet - Standardized wired computer networking protocol typically used in local area networks (LAN). Provides high-speed data transfers, error detection and correction mechanisms, and scalability.

The implementation of these protocols was prioritized based on the specific requirements for this project 4.2.

Chapter 5

Implementation

This chapter aims to provide a detailed account of the practical implementation of the system described in previous chapters. The focus is on integrating hardware and software components to achieve the specified project requirements 4.1.

The implementation architecture is structured around a RTOS, core layer, and individual tasks. The RTOS manages thread scheduling and synchronization mechanisms crucial for real-time performance. The core layer oversees task execution and facilitates inter-task communication, alongside external communications. Tasks are modular components responsible for specific functionalities within the application.

5.1 Hardware Description

This section provides an overview of the hardware components utilized in the implementation, including the Arm Cortex-M7 core and its STM32F767ZI microcontroller for their ample connectivity options, compatible with RS-232 and Ethernet, high performance, extensive RAM, and widespread use in robotic applications [30].

5.1.1 Arm Cortex-M7

The Cortex-M CPU line is a family of 32-bit microcontroller cores designed by ARM. These cores are specifically optimized for embedded systems where low power consumption, and real-time processing, are critical. The Cortex-M7 CPU is a microcontroller processor based on the ARMv7-M architecture, which provides a balance between performance and energy efficiency suitable for embedded applications.

The Cortex-M7 core comes with key features that enable implementing an RTOS. One important feature is the use of dual stack pointers, including the Main Stack Pointer (MSP) for system tasks and the Process Stack Pointer (PSP) for application tasks. This separation of stacks enhances security and stability, preventing stack overflows in user applications from affecting system tasks. It also simplifies context switching, facilitating the switching between tasks by managing separate

stacks for system and user contexts. The MSP is used by interrupt handlers, providing system integrity during interrupts, while the PSP is used within a thread context, allowing user applications to run independently. This distinction between MSP and PSP usage ensures errors are not propagated between each usage, resulting in robust and efficient task management, and safeguarding critical system operations from user-level errors.

Another crucial component is the Nested Vectored Interrupt Controller (NVIC), which allows control over interrupt prioritization and interrupt preemption. This means that higher priority interrupts can preempt lower priority interrupts. This component manages both external interrupts triggered by peripheral devices, such as general-purpose input/output (GPIO), and Timer interrupts, and internal interrupts generated internally by the CPU. Internal interrupts can be fault exceptions (HardFault, MemFault), system timer interrupts (SysTick), and software-triggered interrupts like SVC (Supervisor Call) and PendSV (Pended SVC). The Cortex-M7 core includes several fault exceptions critical for handling errors and ensuring stability:

- HardFault - Triggered by severe error such as system exceptions that cannot be managed by other fault handlers or if a fail occores during interrupt handler execution. It acts as a catch-all for all unhandled faults.
- MemManage - Memory Management Fault occurs when a memory protection violation is detected, such as accessing restricted areas of memory. It prevents illegal memory accesses, which could corrupt data or cause unexpected behavior.
- BusFault - Triggered by memory-related errors during data transfer, such as during exception stacking or instruction prefetching.
- UsageFault - Occurs due to execution errors, such as divide-by-zero operations, unaligned memory access, attempt to execute undefined instructions or stack overflow.

In addition to fault exceptions, internal interrupts are essential for system operations, time management, and task switching in an RTOS. The SysTick generates periodic interrupts at a fixed interval, commonly set to 1 millisecond, and is used for system timekeeping, task scheduling, and implementing delays. In an RTOS, it maintains the system tick count, driving the scheduler to manage task execution times efficiently. The PendSV is a software-triggered interrupt designed to facilitate task switching in multitasking environments. When a context switch is required, PendSV is set pending, and the interrupt service routine handles saving the current task's state and restoring the next task's state, ensuring seamless transitions between tasks. Moreover, the SVC interrupt can be triggered by an instruction and allows a user application to request privileged operations from the kernel. These interrupts provide the basic functionalities for multitasking, so that an RTOS can be implemented, and provide basic error handling.

Lastly, the Cortex-M7 core supports various power modes through the use of WFI (Wait For Interrupt) and WFE (Wait For Event) instructions. These power-saving instructions allow the processor to enter low-power states while waiting for interrupts or events to occur, which can significantly reduce power consumption in embedded systems. The WFI instruction halts the CPU

until an interrupt occurs, while the WFE instruction halts the CPU until an event or interrupt occurs. These instructions are useful in battery-powered and energy-efficient applications, enabling the system to conserve power when full processing power is not required.

5.1.2 STM32F767ZI

The STM32F767ZI is a microcontroller from STMicroelectronics based on the ARM Cortex-M7 core. It's part of the STM32 family, providing high performance, and an extensive peripheral set, and is suitable for a wide range of applications.

This microcontroller offers diverse connectivity interfaces such as:

- Ethernet Connectivity - Includes an Ethernet Media Access Control (MAC) layer, which provides the foundation for high-speed network communications. It also includes a dedicated Direct Memory Access (DMA) controller for Ethernet that enhances data transfer efficiency, allowing for smooth and rapid data handling without overloading the CPU.
- CAN Interfaces - Includes two CAN interfaces, making it suitable for automotive and industrial applications where reliable, real-time data exchange is essential. CAN is widely used for its robustness in noisy environments and efficient communication.
- I2C Interfaces - Offers several I2C interfaces, enabling communication with a variety of peripheral devices.
- SPI Interfaces - Equipped with several SPI channels, allowing for high-speed communication with peripherals.
- UART/USART Interfaces - The device features numerous Universal Asynchronous Receiver/Transmitter (UART) and Universal Synchronous/Asynchronous Receiver/Transmitter (USART) interfaces. These ports enable serial communication such as RS-232.

DMA (Direct Memory Access) is a feature on the STM32F7 microcontroller that enhances data transfer between peripherals and memory without involving the CPU directly. DMA allows peripherals on the STM32F7 microcontroller to transfer data directly to or from memory without CPU intervention. It operates independently once configured, freeing up the CPU to perform other tasks while data transfers occur in the background. DMA can generate interrupts upon half-transfer, transfer completion, or error conditions, allowing the CPU to respond promptly to events without continuously polling peripheral status.

This microcontroller has a Real-Time Clock (RTC) that offers some key functionalities. It can maintain accurate time and date information using separate registers for seconds, minutes, hours, day of the week, day of the month, month, and year, which can be preserved during power loss, as long as a backup power source is connected. The Real-Time Clock supports up to two programmable alarms (Alarm A and Alarm B) that can be set to trigger on specific times or dates. Furthermore, it can continue operating in low-power modes with minimal power consumption.

Lastly, the Real-Time Clock is capable of synchronizing with an external clock source, which enhances its accuracy and reliability.

In addition to its standard operation, this device offers additional low-power modes.

- Sleep mode - the CPU is stopped while peripherals, timers, and interrupt controllers continue to operate. This mode reduces power consumption while still allowing for a quick return to full operation upon an interrupt or event.
- Stop mode - Halts the CPU and most peripherals, except for essential components like RTC, NVIC, and wakeup pins, achieving significant power savings suitable for applications requiring extended battery life.
- Standby mode - Shuts down the main oscillators and most internal clocks, leaving only the RTC, backup registers, and wakeup pins active. It provides the lowest power consumption among the low-power modes and is ideal for scenarios where the device needs to operate intermittently with long periods of inactivity.

5.2 RTOS Implementation

In this implementation, using an RTOS is essential to managing multiple IMC entities. These entities can operate independently by associating each IMC entity with a separate thread. This means that if one entity encounters an error and becomes blocked, it will not affect the operation of the other entities. The RTOS significantly enhances the system's ability to handle errors gracefully, and in the event of a fault, the RTOS can isolate the problem to the specific task where the error occurred, preventing it from impacting the entire system. Additionally, the RTOS also provides a prioritized scheduling mechanism, ensuring that higher-priority tasks are executed promptly. This is crucial for meeting real-time constraints, as it allows critical tasks to be prioritized and executed without delay.

5.2.1 Thread Scheduling

The microcontroller implementation RTOS utilizes preemptive scheduling based on thread priority, allowing higher priority threads to preempt lower priority threads, thereby ensuring the execution of critical tasks within their time frames. Threads can be assigned priority levels ranging from 0 to 254, with 0 being the highest priority and 254 representing the lowest priority, and the value 255 is reserved for the idle thread priority.

Scheduling operates on a 1-millisecond tick rate, enabled by the Cortex-M7's SysTick interrupt, as used in the Arduino framework. This eliminates the need for additional OS-specific interrupts. However, It increases the tick overhead since the SysTick interrupt only starts counting after its handler completes.

Threads are managed in two queues, both implemented using a Fibonacci heap structure [12] for efficient minimum element access and heap merging. The run queue holds threads ready to

execute, while the sleep queue contains threads waiting until the next tick. Each tick merges the sleep queue into the run queue and selects the highest-priority thread for execution. If this thread has a higher or equal priority compared to the currently running thread, a context switch is triggered, ensuring high-priority tasks are executed promptly. Context switches occur within the PendSV interrupt, set to pending when a context switch is needed. This scheduler also has an additional idle thread, that will be scheduled if no additional threads are available. During the execution period of the idle thread, the CPU will enter a low-power standby state while it waits for an interrupt for wake-up. If a thread wants to yield execution for a determined amount of time. There are two available options.

The first method is a polling tick yield, where the current thread is added to the sleep queue and won't execute for the remainder of the tick, and a runnable thread is retrieved from the run queue. On the next tick, the sleep queue merges into the run queue. If the scheduler selects the yielded thread, it checks the tick delay and continues yielding until the delay is met. This delay may not represent an accurate time delay. If a thread is delayed by 1 millisecond but this delay occurred near the end of the tick, it will be delayed for a very short time, since in the next tick the thread is considered runnable. If no threads are runnable, execution will enter a low-power mode, using WFI instruction, until an interrupt occurs. This design implementation is as defined by the Arduino framework.

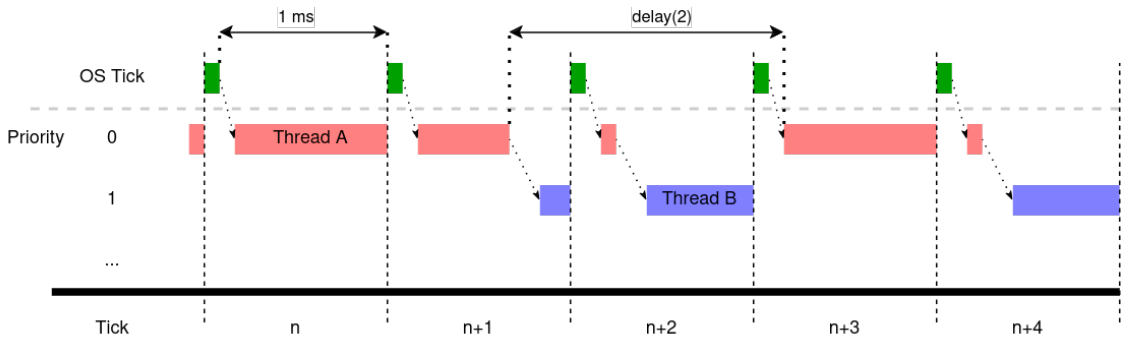

Figure 5.1: Polling Tick delay

The other available method involves a delayed timer interrupt callback. In this case, the thread is yielded but is not added to any scheduler queue, instead is created a delayed timer callback in 'x' milliseconds, where the timer used contains a microsecond resolution. Once the time delay has elapsed, a timer interrupt is triggered to dispatch the thread, setting it to run if it has higher priority than the current thread or adding it to the scheduler run queue. This method offers better resolution and accuracy since it does not rely on the OS tick and uses a timer with microsecond resolution.

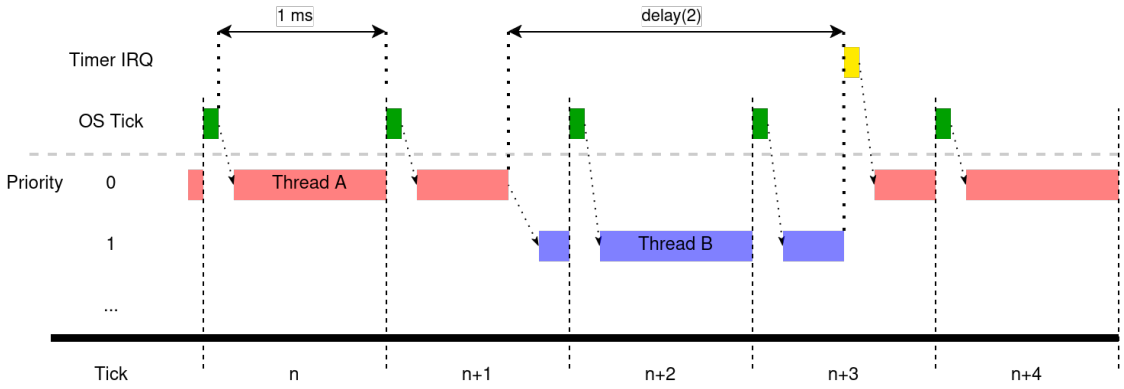

Figure 5.2: Timer delayed callback

5.2.2 Synchronization mechanisms

The RTOS provides several synchronization mechanisms to manage thread interactions and resource sharing. The mutex implementation is recursive, allowing a thread to acquire it multiple times, and incorporates priority inheritance 2.1. If a mutex is already owned, the acquiring thread yields execution and is not added to any scheduler queue and instead is saved on the mutex blocked list. If the owner has a lower priority, it inherits the higher priority of the blocked thread. Upon releasing the mutex, the owner must release it as many times as it was acquired, and the highest-priority blocked thread is dispatched, with the owner's priority restored if it had been elevated.

This layer also provided a counting semaphore synchronization primitive. This semaphore decreases its count when acquired, but if the count is zero, the thread is blocked until the count increases. Signaling the semaphore increases the count and wakes up any blocked threads, though it lacks priority inheritance. Both these methods are available with a timeout version.

Lastly, this layer provides a message queue. The message queue operates as a First In, First Out (FIFO) queue, initialized with a specified length. Messages can be enqueued with both blocking and timeout methods. If the queue is full, not allowing a message to be pushed, the thread yields execution until space is available. Dequeuing messages follow similar logic with both blocking and timeout methods. These mechanisms allow for a multitasking environment and resource management by preventing conflicts and prioritizing critical tasks.

5.2.3 Fault Recovery

This RTOS also provides mechanisms for fault recovery to handle failures. One key feature is the ability to restart any thread without deleting it. It can be applied to both running and sleeping threads but requires privileged access while the kernel is using the CPU Main Stack Pointer (MSP).

Additionally, the kernel can handle fatal errors by invoking an error function. This function signals a critical issue, stores any relevant information, and performs a software reset to restart the microcontroller. During system initialization, the kernel checks the cause of the system reset. Potential causes include:

- Supply Voltage Drop - Supply voltage drops below the operating threshold.
- Low Power Reset - The system exits low-power mode, such as Standby or Stop mode.
- Internal Watchdog Reset - Occurs when the internal watchdog timer is not periodically reset.
- Power-On Reset - Occurs when the microcontroller is initially powered up.
- External Reset Pin - Triggered by an external signal applied to the reset pin on the microcontroller.
- Software Reset - Triggered by the microcontroller's software.

Upon restart, the microcontroller can retrieve both the reset cause and any stored additional information, aiding in diagnosing and addressing the issue. These fault recovery features ensure the system can recover from errors gracefully, maintaining stability and reliability. Lastly, the kernel implements a handler for each internal CPU exception, that reports it as a critical error and restarts the microcontroller.

5.3 Core Layer

The Core layer, as showcased in Figure 4.2, is an integral part of the system architecture, positioned directly above the RTOS, and providing a link between the RTOS and the higher-level application-specific tasks. Its primary function is to enable operating and communicating within the system and with external entities through the IMC protocol. To achieve this each IMC entity is configured within a task, implemented as a thread. This layer works as an abstraction layer so that tasks are not required to know the specific behavior of RTOS. In this layer system integrity and performance are maintained, by supervising each IMC entity's behavior, and not allowing it to halt execution.

When the Core layer starts up, it performs several initializations. Firstly, it creates tasks with respective IMC labels and sets up a background task. Then, it configures task parameters based on available data or defaults if no specific parameters are provided in its memory.

5.3.1 Task Parameter Setup

After all tasks are created, this layer searches in Flash memory for saved task parameters. The location in the flash where parameters are stored is defined by the user at compile time. Since there is no file system in this design, all task parameters are stored in a serialized array of bytes.

The serialization of each task parameter starts with a byte value of 11, followed by the length of the entity label and the string of the entity label. Then comes the task parameters serialization length and the task parameters serialization itself, along with a CRC checksum to validate the integrity of the parameter serialization. Each Task parameter associates a string label, that corresponds to the parameter name with a value in a string format. After the last task checksum, the end

symbol is represented in the final byte with the value 255 to indicate the end of the task parameters serialization.

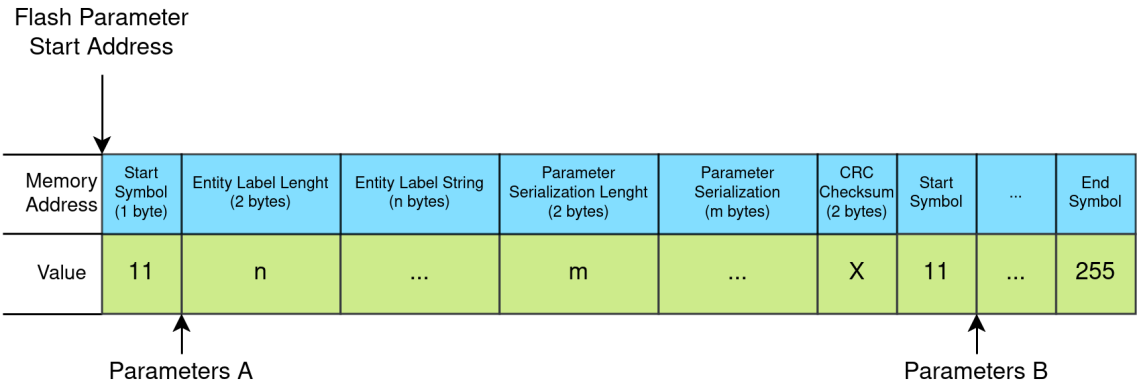

Figure 5.3: Flash Parameter Serialization

5.3.2 Communication

This implementation requires communication with external devices to allow its tasks to perform their functions. This communication is managed by a dedicated background task that handles the data exchange. The communication interface is dynamically selected based on a configurable parameter, allowing flexibility in choosing the appropriate transport method even after deploying the application onto the microcontroller. The system supports two main communication interfaces: RS-232 and Ethernet, which implement data transfer using DMA control blocks to reduce CPU usage.

RS-232 is configured as a point-to-point communication using Full Duplex communication. It offers the option to enable hardware flow control, while software flow control is not supported. Each data frame transmitted over RS-232 is terminated with the characters '\r\n' to aid in framing and synchronization. The interface's IO parameters include settings for flow control flags, Tx Pin, Rx Pin, and baud rate.

Ethernet utilizes the UDP/IP protocol and relies on the LwIP library for networking functionality. To operate correctly, Ethernet must be pre-configured with a static IP address. The microcontroller communicates using port 6001 for transmitting packets, which is the port used by DUNE to announce itself on the network. Ethernet's I/O parameters are configured with settings for destination IP and destination port to receive data frames according to the utilized protocol.

For tasks to communicate with external devices, the configuration of the IMC entities must be completed. Until this is done, the background task will not transmit any messages to external devices because the messages are not properly set up with the right IMC entity identifiers in the source and source entities fields.

5.3.3 IMC Entities configuration

These tasks begin execution without IMC entity IDs, which are required for communication with external devices. An *IMC::Announce* message is periodically sent to configure the system and IMC entities. This message serves the purpose of announcing this microcontroller component to the computational system. The announcement message also reports the cause of the system reset, an RTOS cause, or an application-specific reset. After sending the announcement message, the microcontroller will wait for the *IMC::EntityList* query and respond with all its entity names for reservation. Subsequently, it will wait for a response that associates each IMC Entity Label (unique task name) with a unique IMC Entity ID. Finally, it reports all the IMC messages it intends to receive.

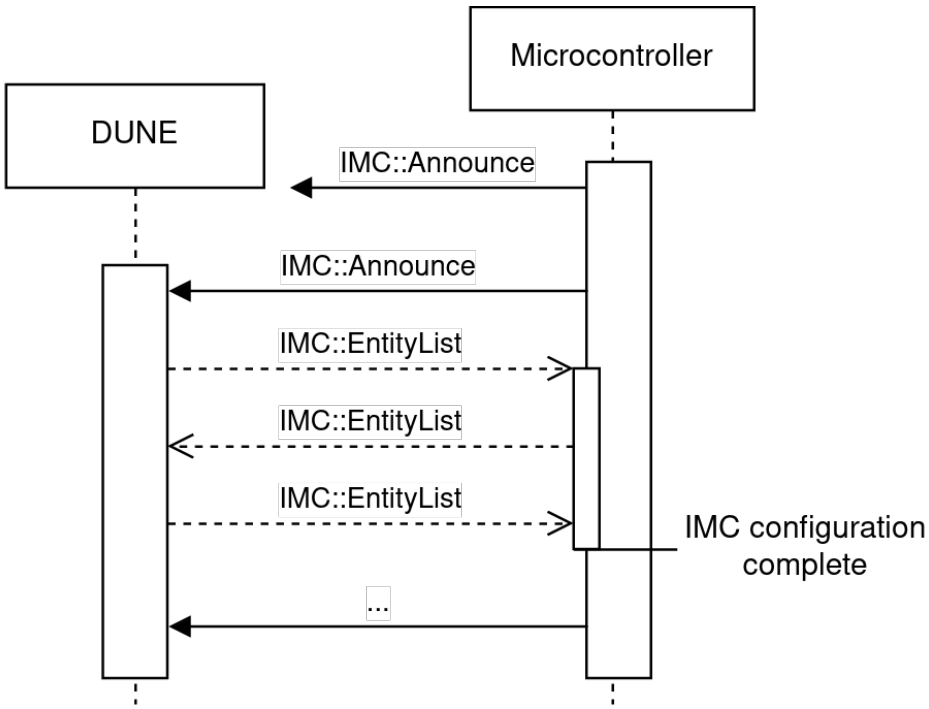

Figure 5.4: IMC configuration sequence

After system configuration, if the microcontroller receives an *IMC::EntityList* query, it will restart the IMC configuration process again, improving fault tolerance if the external component restarted. After the system configuration is completed, the CORE layer will set the Microcontroller RTC epoch according to the last timestamp of the message received.

5.3.4 Message Distribution

In this multitasking environment, messages need to be distributed to multiple tasks. Since Microcontrollers often operate in resource-constrained environments with limited memory, the distribution mechanism is based on shared memory access to minimize resource usage. When a message is sent, the sending task gives up ownership, meaning it can no longer modify it. Instead,

a data structure that encapsulates the message is distributed to the receiving tasks. Each task must treat the message as a read-only object. This approach offers several benefits. Firstly, it ensures efficiency by minimizing memory usage, as multiple copies are not created. It also provides synchronization to ensure that all tasks have a consistent view of the message, as they all read from the same object. Additionally, it optimizes resource management by deleting the message only when it is no longer needed.

Furthermore, the distribution mechanism follows a Publish-Subscribe Model. This is a messaging pattern where publishers send messages to topics, and the subscribers, in turn, receive messages about the topics they are interested in.

5.3.5 Clock Synchronization

The microcontroller will synchronize its RTC with the message *IMC::ClockControl* 5.5.

Clock Control

Clock control.

- Abbreviation: ClockControl
- Identification Number: 106
- Payload Size: 10 bytes
- Message Size: 32 bytes

Name	Abbreviation	Unit	Type	Description	Range
Operation	op	*Enumerated* (Enum Operation)	uint8_t	Operation to perform.	Same as field type
Clock	clock	*s*	fp64_t	Clock value (Epoch time).	Same as field type
Timezone	tz	-	int8_t	Timezone.	min=-23, max=23

Figure 5.5: ClockControl Message

The clock synchronization process starts with DUNE sending a *IMC::ClockControl* request message to the MCU. The timestamp at which DUNE sends this message is recorded as T1. The MCU then receives the *IMC::ClockControl* request message and records the timestamp as T2. After receiving the request, the MCU prepares and sends a response message, recording the timestamp as T3. DUNE then receives the response message from the MCU, recording the timestamp as T4.

The round-trip delay (Δ_{RT}) is calculated as the difference between T4 and T1 minus the difference between T3 and T2.

$$\Delta_{RT} = (T_4 - T_1) - (T_3 - T_2)$$

This accounts for the time it takes for the messages to travel back and forth between DUNE and the MCU, excluding the processing time at the MCU.

The clock offset (Δ_{offset}) is calculated as half of the difference between T2 and T1 added to the difference between T3 and T4.

$$\Delta_{offset} = \frac{(T_2 - T_1) - (T_4 - T_3)}{2}$$

This formula calculates the difference between the DUNE and the MCU clocks and divides it by two to find the average offset. Finally, the MCU adjusts its clock by the calculated offset (Δ_{offset}) to synchronize with the DUNE clock.

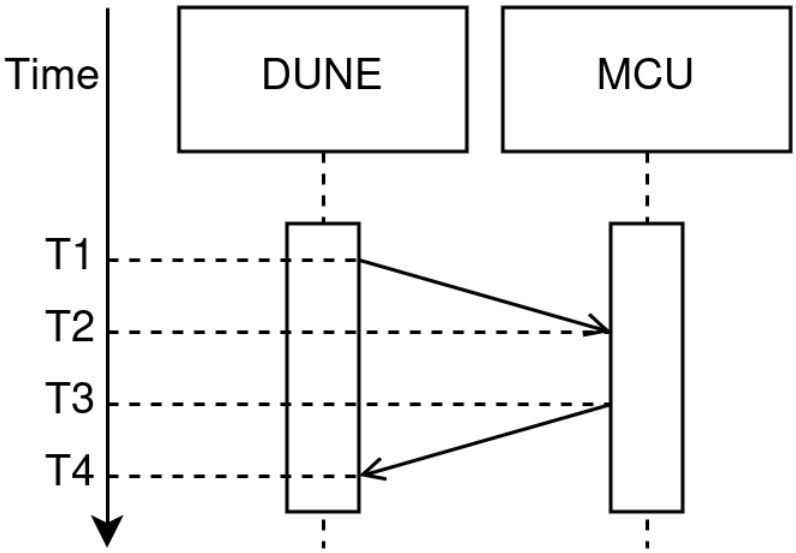

Figure 5.6: Synchronization Timestamps

5.3.6 Fault Recovery

In the CORE layer, fault recovery mechanisms are critical to ensure reliable operation. Each task within this layer is equipped with an internal software watchdog to monitor and manage its execution. The watchdog is a supervisory mechanism designed to detect and recover from malfunctions. It operates by periodically checking if a task is operating correctly within a predefined timeframe, referred to as the watchdog timeout period. If the task fails to reset the watchdog within this period, it is assumed to be malfunctioning or blocked. This watchdog timeout period is a task parameter allowing it to be adjusted at runtime. The watchdog timer is automatically reset during the normal execution of the task, provided the task completes its loop within the specified timeframe.

To detect the state of each task, a periodic routine checks the watchdog state of each running Task. Upon detecting a blocked task, this layer will restart it to its initial state execution but allow the task to operate with its current parameters. This is achieved by restarting the thread associated with the blocked task. In critical situations, where multiple tasks are restarting in a short period, the microcontroller will execute a software reboot. By implementing an internal software watchdog for each task, the system ensures that any task malfunction is promptly detected and addressed, enhancing the overall reliability and robustness of the microcontroller's operation.

5.4 Tasks

When a task is created, it subscribes to the IMC message ID it wants to receive. Each task has two default parameters: the Entity Label, which represents the task name, and a timeout parameter for checking if the task is blocked (Watchdog timeout), representing the timeframe that the task must conclude its loop cycle. Additional parameters can be created by associating a label with the parameter name and a variable.

Each task must implement its own setup and loop function. The setup function runs once when the task starts running and configures the required resources. The loop function runs continuously and is responsible for executing the task's main functionality.

A should be designed to execute one simple functionality and be abstract to other tasks, ensuring separation so that if one task fails, it can recuperate without impacting the entire microcontroller system.

When a task starts running, it announces its *IMC::EntityState* as "Boot", and once the setup function ends it announces its *IMC::EntityState* as "Normal" for normal execution.

Each task is pre-configured to interact with some messages:

- *IMC::QueryEntityParameters* - Represent a request of the Task Parameters. It will respond with an *IMC::EntityParameters* message containing all its Parameters.
- *IMC::SetEntityParameters* - Represents an update to the Task Parameters.
- *IMC::QueryEntityState* - Represents a request of the Task current State. It will respond with an *IMC::EntityState* message with the state of "Boot" if the setup function was not completed or "Normal" if it was completed.

Finally, when a task sends an IMC message to the Core Layer, it sets its source entity ID with its own IMC entity ID, allowing each message to be traced.

Chapter 6

Verification and Validation

The implementation of the IMC protocol in microcontrollers necessitates a rigorous approach to ensure that the system meets its design functionalities, as described in tables 4.3, 4.4 and 4.5, and project requirements, as exhibited in tables 4.1 and 4.2. This chapter describes the processes of verification and validation, which are essential for guaranteeing the correctness and usefulness of the software framework developed.

Verification is the process of evaluating the software to determine whether it complies with the specified functionalities, as described in the chapter 4. This section explains the verification activities carried out and outlines the methodology used to conduct unit tests and presents the tests created.

Validation, on the other hand, is the process of evaluating the software to ensure that it fulfills its intended purpose and meets the needs of the users, reflected in the project requirements. This section addresses the validation of the IMC protocol implementation against both functional and non-functional requirements. By validating the software, the chapter ensures that the system performs as expected in real-world scenarios and satisfies the criteria set.

Through a comprehensive examination of both verification and validation, this chapter aims to provide a clear understanding of the measures taken to achieve a reliable and robust implementation of the IMC protocol in microcontrollers.

6.1 Verification

Verification was conducted using unit tests. Unit tests are automated tests written and executed by developers to verify that individual units of the source code meet its design and behave as intended. They help identify bugs early in the development cycle, which allows developers to catch and fix issues before they escalate. By assessing if each unit of the code performs as expected, unit tests contribute to the overall reliability and maintainability of the software [1]. Lastly, with a comprehensive suite of unit tests, developers can refactor or enhance the code confidently, knowing that any deviations from expected behavior will be quickly identified.

During the development process, the unit testing methodology obeyed the following steps:

1. Initially, test cases are designed for each component based on specifications and anticipated behavior. These test cases encompass typical scenarios and some edge cases to ensure thorough testing.

2. Subsequently, the tests are implemented using a unit testing framework, Unity, provided by PlatformIO. Each test is autonomous and repeatable, allowing for an accurate evaluation of component functionality. The tests are then executed automatically in an isolated environment to prevent interference from other components.

3. Afterward, the results are meticulously analyzed to identify failures or deviations from expected outcomes. Any detected issues are rigorously investigated and rectified.

4. Finally, post-resolution, regression testing is conducted to ensure that the modifications do not introduce new problems or adversely affect existing functionality.

The unit tests were organized around the system's layered architecture, the RTOS, Core, and user-created tasks layers, using PlatformIO. PlatformIO is an open-source ecosystem designed for embedded development, offering a cross-platform build system, library management, and a unit testing framework. This tool enables the segregation of the software developed into parts and tests if they are working correctly. PlatformIO also enabled the automatic setup, execution, and management of unit tests. With this tool, independent unit tests were created for each layer functionality, ensuring test isolation between. This isolation enabled the separate examination of each component's behavior without interference from other parts of the system or external components.

6.1.1 RTOS layer

Test Case	**Description**
Creation	RTOS allows for the creation of multiple threads.
Execution	RTOS provides priority-based scheduling.
Yielding	RTOS allows threads to yield execution.
Termination	RTOS allows threads to be deleted without affecting other components.

Table 6.1: Unit Tests for Thread

Test Case	**Description**
Creation	RTOS allows for multiple mutexes to be created.
Locking	RTOS allows only one thread to acquire an unowed mutex lock at a time.
Unlocking	RTOS allows only the mutex owner to release its lock.
Destruction	RTOS allows mutexes to be deleted without affecting other components.

Table 6.2: Unit Tests for Mutex

Test Case	Description
Creation	RTOS allows for multiple semaphores to be created.
Signaling	RTOS allows a thread to increment the semaphore count.
Waiting	RTOS allows a thread to decrement the semaphore count.
Destruction	RTOS allows semaphores to be deleted without affecting other components.

Table 6.3: Unit Tests for Semaphore

Test Case	Description
Creation	RTOS allows for multiple message queues to be created.
Push	RTOS allows threads to push data to the message queue.
Pop	RTOS allows threads to pop data from the message queue.
Destruction	RTOS allows message queues to be deleted without affecting other components.

Table 6.4: Unit Tests for Message Queue

6.1.2 Core layer

Test Name	Description
Periodic Announce	Ensure Announce messages are sent periodically.
Entity Reservation	Ensure the configuration sequence can reserve IMC entities.

Table 6.5: Unit Tests for Entity Configuration

Test Name	Description
Storage	Task parameters can be serialized and stored in Flash memory.
Retrieval	Task parameters can be retrieved from Flash memory and deserialized.
Modification	Task parameters can be updated, and Tasks are notified of the changes.

Table 6.6: Unit Tests for Parameter Management

Test Name	Description
Detection	Ensure the detection of unresponsive tasks.
Recovery	Ensure the attempted recovery of unresponsive tasks.

Table 6.7: Unit Tests for Task Recovery

Test Name	Description
Distribution	Ensure messages are distributed in a publisher-subscriber pattern.
Consumption	Ensure message deletion once all subscribers have processed it.

Table 6.8: Unit Tests for Message Distribution

Test Name	Description
Creation	Ensure the creation of the background Task to handle external communication.
Setup	Ensure the configuration of the communication interface.
Operation	Ensure the runtime capability to change the communication interface.

Table 6.9: Unit Tests for Background Task

6.1.3 User Tasks

Test Name	Description
Creation	Ensure Tasks can create new parameters.
Update	Ensure Task parameters are updated.
Report	Ensure Task parameter changes are reported.

Table 6.10: Unit Tests for Task Parameters

Test Name	Description
Association	Ensure Tasks are correctly associated with an entity state.
Update	Ensure that each task updates its entity state accurately during execution.
Report	Ensure that each task correctly reports its current entity state.

Table 6.11: Unit Tests for Task Entity State

Test Name	Description
Subscription	Ensure tasks can subscribe to IMC messages.
Dispatch	Ensure that messages are dispatched with the correct source information.
Consumption	Ensure that tasks consume received messages.

Table 6.12: Unit Tests for Task Message Handling

6.1.4 Conclusion

All unit test cases have resulted in a 'passed' status. During development, unit tests were crucial for verifying each functionality of the software architecture's layers. These tests were executed whenever changes occurred in the respective component's implementation. For a component to be

considered complete, it had to pass its respective unit test, ensuring alignment with the software system design. Since all components have successfully passed their associated unit tests, this confirms the correctness of the development process.

6.2 Validation

Validation is a crucial phase in the software development lifecycle where the software system is evaluated to ensure it meets the intended purpose. It involves the assessment of the software under real-world conditions to confirm that it behaves as expected. Validation ensures that the right product is built, aligning the final software with both functional and non-functional requirements. It focuses on answering the question: "Are we building the right product?"

This involves confirming that the system's functionalities perform correctly in the operational environment and that the system is robust, reliable, and usable. Validation aims to identify any discrepancies between the developed software and user expectations, ensuring the final product is fit for its intended purpose. The validation scope encompasses assessing the software against both functional and non-functional requirements.

The methodology for creating validation tests involved these steps:

1. Test Case Design - Development of specific test cases that directly validate each requirement.

2. Implementation of Tests - Coding and setting up the tests alongside DUNE to simulate real-world conditions.

3. Execution and Evaluation - Running the tests and evaluating the results to ensure the system meets the requirements.

The tests implemented use the DUNE framework to monitor and assess the results. Each test runs manually and separately to ensure the necessary attention to detail. The software developed by this work was connected to DUNE using the supported communication protocols, RS-232 and ethernet, ensuring complete testing of the system's communication capabilities. DUNE simulated the board's required operations for each test case, creating a realistic environment to test the system's functionality and performance. The combination of manual test execution and DUNE's monitoring and simulation capabilities ensured a thorough validation process, confirming if the system met the specified requirements under realistic operating conditions. This structured approach ensures that each requirement is validated through well-defined test cases.

6.2.1 Functional Requirements

This section demonstrates each test case created and discusses the validation of the functional requirements for this project. The validation process for these requirements ensures that the system behaves as expected and meets the predefined criteria.

Requirement ID		R1
Requirement		The system shall receive and send IMC packets.
Test Case		Verify that the system can correctly send and receive IMC packets.
Steps	1	Send a series of IMC packets from DUNE to the system.
	2	The system will then send the same IMC packets to DUNE.
	3	Verify that all packets are received and sent correctly.
Expected Result		All packets are received as sent.

Table 6.13: Validation Table for Functional Requirement R1

Requirement ID		R2
Requirement		The system shall reserve IMC entity IDs for communication.
Test Case		Check that the system can reserve unique entity IDs.
Steps	1	DUNE will start the Entity configuration sequence.
	2	Wait for the Entity configuration sequence to complete.
	3	DUNE will query all entities inside the MCU.
	4	DUNE will query all entities inside the MCU.
Expected Result		Entity IDs are reserved successfully without conflicts.

Table 6.14: Validation Table for Functional Requirement R2

Requirement ID		R3
Requirement		Each task shall be associated with a unique name and ID.
Test Case		Validate that each task has a unique name and ID.
Steps	1	Create multiple tasks and assign names.
	2	Wait for the Entity configuration sequence to complete.
	3	Check that each task associated a unique name to a unique ID.
Expected Result		Each task has a unique name and ID.

Table 6.15: Validation Table for Functional Requirement R3

Requirement ID		R4
Requirement		Tasks shall communicate with each other.
Test Case		Check tasks can send and receive messages to/from each other.
Steps	1	Create multiple tasks that subscribe to messages.
	2	Send those message types to the system.
	3	Verify message consumption.
Expected Result		Messages are successfully sent and received.

Table 6.16: Validation Table for Functional Requirement R4

Requirement ID		R5
Requirement		The system shall detect if tasks are stalled.
Test Case		Check if a task stalled and was detected during execution.
Steps	1	Create multiple tasks.
	2	Simulate a stall in a task (such as an infinite loop).
	3	Verify that the system detects the stalled task.
Expected Result		The system detects the task stall and raises an alert.

Table 6.17: Validation Table for Functional Requirement R5

Requirement ID		R6
Requirement		The system shall restart stalled tasks.
Test Case		Ensure the system can restart stalled tasks.
Steps	1	Create multiple tasks.
	2	Force the system to restart a task.
	3	Verify if the restarted successfully.
Expected Result		The restarted task resumes normal operation.

Table 6.18: Validation Table for Functional Requirement R6

Requirement ID		R7
Requirement		The system shall continue to operate if a task fails.
Test Case		Validate system operation after task failure.
Steps	1	Initialize the system and create multiple tasks.
	2	Simulate a failure in one of the tasks.
	3	Verify that the other tasks continue to operate.
Expected Result		The other tasks remain operational despite one task failure.

Table 6.19: Validation Table for Functional Requirement R7

Requirement ID		R8
Requirement		The system shall be able to switch communication interface at runtime.
Test Case		Validate runtime communication interface switching.
Steps	1	Initialize the system with a default communication interface.
	2	Switch to an alternative communication interface at runtime.
	3	Verify that communication continues seamlessly after the switch.
Expected Result		Communication is uninterrupted after switching interfaces.

Table 6.20: Validation Table for Functional Requirement R8

Requirement ID		R9
Requirement		The system shall be able to synchronize its clock.
Test Case		Validate clock synchronization.
Steps	1	Initialize the system and set the initial clock time.
	2	Perform a clock synchronization operation.
	3	Verify that the system clock matches the reference time.
Expected Result		The system clock is accurately synchronized with the reference time.

Table 6.21: Validation Table for Functional Requirement R9

Requirement ID		R10
Requirement		The system shall be able to save task parameters to Flash memory.
Test Case		Validate saving task parameters to Flash Memory.
Steps	1	Initialize the system and create tasks with specific parameters.
	2	Save the task parameters to Flash memory.
	3	Verify that the parameters are correctly saved.
Expected Result		Task parameters are successfully saved to Flash memory.

Table 6.22: Validation Table for Functional Requirement R10

Requirement ID		R11
Requirement		The system shall be able to retrieve Task parameters from Flash.
Test Case		Validate Retrieving Task Parameters from Flash Memory.
Steps	1	Initialize the system and with Task Parameters in Flash memory.
	2	Retrieve the task parameters from Flash memory.
	3	Verify that the retrieved parameters match the original parameters.
Expected Result		Task parameters are successfully retrieved from Flash memory.

Table 6.23: Validation Table for Functional Requirement R11

ID	Requirement Description	Priority	Met
R1	The system shall send and receive IMC packets.	High	✓
R2	The system shall reserve IMC entity IDs for communication.	High	✓
R3	Each task shall be associated with a unique name and ID.	High	✓
R4	Tasks shall be able to communicate with each other.	High	✓
R5	The system shall detect if tasks are stalled.	Medium	✓
R6	The system shall restart stalled tasks.	Medium	✓
R7	The system shall continue to operate if a task fails.	Medium	✓
R8	The system shall be able to switch communication interfaces at runtime.	Medium	✓
R9	The system shall be able to synchronize its clock.	Medium	✓
R10	The system shall be able to save Task parameters to Flash memory.	Low	✓
R11	The system shall be able to retrieve Task parameters from Flash memory.	Low	✓

Table 6.24: Functional Requirements Results

Overall, the validation of functional requirements yielded positive results since the outcome of each test case matched their expected result. As such, this process confirmed that this implementation meets its functional specifications. Testing each requirement through dedicated test cases ensures the system behaves according to user expectations, as indicated in the functional requirements.

6.2.2 Non-Functional Requirements

In this section, we focus on the validation of the non-functional requirements. These requirements ensure the system's performance, usability, and other quality attributes.

Requirement ID		R1
Requirement		The system shall handle up to 200 messages per second.
Test Case		Validate Message Throughput.
Steps	1	Initialize the system and create a test task for sending and receiving messages.
	2	Generate and send 200 messages per second from DUNE to the system.
	3	Measure the system's ability to process these messages within the time frame.
	4	Monitor for any message loss or delay.
Expected Result		The system successfully handles 200 messages per second without message loss or significant delay.
Result Obtained		The system successfully handled up to 250 messages per second without message loss or significant delay. In both protocols tested, RS-232 and Ethernet.

Table 6.25: Validation Table for Non-Functional Requirement R1

Requirement ID		R2
Requirement		The system shall use less than 5KB of RAM.
Test Case		Validate RAM usage.
Steps	1	Initialize the system with its basic components.
	2	Monitor and measure the RAM usage during basic operation. Without any additional user-tasks
	3	Verify that the RAM usage does not exceed 5KB.
Expected Result		The system's RAM usage remains below 5KB during basic operation.
Result Obtained		The system's RAM max value was 4.0 KB while using Ethernet and 2.8 KB while using RS-232.

Table 6.26: Validation Table for Non-Functional Requirement R2

Requirement ID		R3
Requirement		The system shall use less than 300KB of flash memory.
Test Case		Validate Flash memory usage.
Steps	1	Compile the application.
	2	Measure the Flash memory usage.
	3	Verify that the total flash memory used does not exceed 300KB.
Expected Result		The system's flash memory usage remains below 300KB.
Result Obtained		The system's flash memory usage was 200KB while using Ethernet and 183 KB while using RS-232.

Table 6.27: Validation Table for Non-Functional Requirement R3

Requirement ID		R4
Requirement		The system shall correct any clock drift every 60 seconds.
Test Case		Validate clock drift correction
Steps	1	Initialize the system and set an initial reference time.
	2	Introduce a known clock drift.
	3	Wait for 60 seconds and observe the clock correction mechanism.
	4	Verify that the system corrects the drift and synchronizes with the reference time.
Expected Result		The system corrects the clock drift every 60 seconds and remains synchronized with the reference time.
Result Obtained		The system corrected the clock drift every 60 seconds and remained synchronized with the reference time for the entire test duration, 10 minutes.

Table 6.28: Validation Table for Non-Functional Requirement R4

Requirement ID		R5
Requirement		The system shall enter low-power standby mode while no task is running.
Test Case		Validate low-power standby mode operation
Steps	1	Initialize the system and perform typical operations.
	2	Yield all tasks.
	3	Verify that the system successfully enters low-power standby mode.
Expected Result		The system successfully enters low-power standby mode.
Result Obtained		The system successfully entered low-power standby mode.

Table 6.29: Validation Table for Non-Functional Requirement R5

Requirement ID		R6
Requirement		The system shall operate using RS-232 in a point-to-point external network.
Test Case		Validate RS-232 pperation in point-to-point network.
Steps	1	Initialize the system with an RS-232 communication interface.
	2	Establish a point-to-point connection with another device.
	3	Send and receive data through the RS-232 interface.
	4	Verify data integrity and communication reliability.
Expected Result		The system successfully communicates using RS-232 in a point-to-point network.
Result Obtained		The system successfully communicated using RS-232 in a point-to-point network.

Table 6.30: Validation Table for Non-Functional Requirement R6

Requirement ID		R7
Requirement		The system shall operate using Ethernet in a multipoint external network.
Test Case		Validate Ethernet operation in multipoint Network.
Steps	1	Initialize the system with an Ethernet communication interface.
	2	Connect the system to a multipoint network with multiple devices.
	3	Send and receive data through the Ethernet interface.
	4	Verify data integrity and communication reliability.
Expected Result		The system successfully communicates using Ethernet in a multipoint network.
Result Obtained		The system successfully communicated using Ethernet in a multipoint network.

Table 6.31: Validation Table for Non-Functional Requirement R7

ID	Requirement Description	Priority	Met
R1	The system shall handle up to 200 messages per second.	Medium	✓
R2	The system shall use less than 5KB of RAM.	Medium	✓
R3	The system shall use less than 300KB of flash memory.	Medium	✓
R4	The system shall correct any clock drift every 60 seconds.	Medium	✓
R5	The system shall enter low-power standby mode while no task is running.	Low	✓
R6	The system shall operate using RS-232 in a point-to-point external network.	Low	✓
R7	The system shall operate using Ethernet in a multipoint external network.	Low	✓

Table 6.32: Non-Functional Requirements Results

The validation of non-functional requirements for the IMC protocol implementation in microcontrollers has confirmed compliance across diverse operational aspects. As demonstrated by the results obtained. This includes efficient operation within specified memory constraints, accurate time synchronization, and enhanced system versatility in various communication environments.

Chapter 7

Conclusion

7.1 Results and Discussion

The primary objective of this research is porting the IMC protocol to microcontrollers, particularly to an STM32F767ZI, enabling microprocessors running DUNE and microcontrollers to communicate with the same protocol. Additionally, this research aimed to enhance the computational system of autonomous vehicles by distributing the workload across processing units, such as microprocessors and microcontrollers. To accomplish this goal, a framework for an STM32F767ZI was created allowing it to communicate using IMC and perform some of DUNE tasks.

This framework was created as a communication interface that abstracts the protocols, such as RS-232 and Ethernet. This was facilitated by IMC's independence from the transport methods used. Thus, by adapting the IMC protocol to microcontrollers, data transmission can occur regardless of the underlying transport mechanisms. This separation of the communication interface from the transport layer allows for changes to the underlying transport protocols while maintaining consistent interaction between components, leading to a more flexible computational system design.

Another significant achievement is the standardization of the protocol communications across all computational devices within a vehicle. This standardization simplifies the integration process, as each device follows the same communication rules and formats. Consequently, data exchange between different devices becomes more straightforward, reducing complexity and minimizing potential errors that can arise from using multiple proprietary protocols. Making IMC a common language of microprocessors and microcontrollers enhances interoperability, facilitating the design, implementation, and maintenance of the computational systems where these devices need to interact seamlessly, even across different communication protocols.

Furthermore, by abstracting the communication protocol and standardizing the communications, each processing unit within a vehicle can be encapsulated as an isolated module. Each isolated module will have set clear boundaries and connections to other devices. This creates a modular approach, where each module is designed to operate within its boundaries while interacting with other modules through well-defined connections. These well-defined boundaries and

connections between modules ensure that internal changes in one module do not affect others, thereby enhancing vehicle computational system stability, as issues in one module are less likely to propagate throughout the system.

The modular nature of the computational system also significantly enhances its scalability. New components, functionalities, and hardware can be incorporated without requiring a complete redesign. This scalability allows the system to grow and adapt over time, accommodating new technological advancements and expanding its capabilities without compromising the integrity of the existing setup.

Moreover, this framework allows for a more distributed design for autonomous vehicles computational systems, where the workload can be allocated more efficiently. To achieve this, some tasks, such as data processing, are offloaded from the main CPU, which runs DUNE, to microcontrollers. Previously, these devices with lower power consumption only acquired data, but now they process data at their source and send it to the computational system as IMC messages. This relieves the main CPU of excessive effort, making the overall system more efficient. Additionally, the tasks previously performed by DUNE are now carried out by microcontrollers, without interfering with DUNE's remaining responsibilities. This is possible because IMC messages from microcontrollers emulate those previously created by DUNE when processing data.

Lastly, the implementation of an RTOS in a microcontroller environment brings substantial benefits, particularly in ensuring deterministic behavior and enhancing real-time capabilities, as analyzed in section 3.3 with the *PX4* framework. An RTOS optimizes resource usage by scheduling and managing multiple concurrent tasks, ensuring that critical tasks meet their deadlines. This is crucial in real-time applications where timely and predictable responses are essential. Additionally, an RTOS provides mechanisms for protecting resources from concurrent access, facilitating inter-task synchronization, and enabling efficient communication between tasks. This deterministic behavior ensures that high-priority tasks are executed within their designated time frames, which is vital for applications requiring precise control and responsiveness, such as those in autonomous systems.

One of the primary limitations of this implementation is the necessity for low-level programming, specifically in Assembly language, due to the use of am RTOS. While this approach allows for precise control over hardware and resource management, it also means that the implementation is closely tied to the specific CPU architecture employed. For instance, in this project, the adaptation was performed for ARM Cortex microcontrollers, but this same implementation will not directly apply to other CPU platforms. Each new architecture, such as RISC-V or MIPS, will require its tailored implementation.

Another significant limitation is the variability in microcontroller platforms. Each platform, such as the ESP32 or Raspberry Pi Pico, comes with its own unique set of peripherals and implements its communication configuration differently. As a result, additional codebases must be developed for each target platform to accommodate the distinct characteristics of each microcontroller.

7.2 Future Work

Several paths for future work have been identified to enhance the current implementation and widen its applicability.

One significant improvement involves expanding support for additional communication protocols such as CAN, SPI, and I2C. Incorporating these protocols will provide a more comprehensive set of communication options where the IMC can be employed, thereby enhancing flexibility and broadening the potential use cases.

Another area for enhancement is the optimization of flash memory usage. This can be achieved by filtering IMC messages to include only those used by the microcontroller application. Creating a script to analyze the messages utilized by the software and generate only the necessary IMC message code can help achieve this optimization. The extent of reduction varies significantly depending on the number of messages the software employs. In applications with a small number of messages, reductions of up to 50% are possible, potentially decreasing flash memory usage from about 200 KB to around 100 KB. This particular aspect was not addressed in the current work, as the specific microcontroller being used already has 2 MB of flash memory, resulting in flash usage of only 10%. Additionally, the current memory footprint was already below 33% of the project requirements, so other requirements were prioritized to meet.

Furthermore, implementing this framework in a real-world scenario could demonstrate significant improvements in power efficiency achieved by maximizing microcontroller functions. For instance, deploying a low-power distributed control system for autonomous systems capable of real-time operations, integrated with DUNE and utilizing the IMC protocol, would showcase the practical benefits and energy savings. The average power consumption for a single Cortex-A8 core, Armv7-A, at 0.6 GHz is 800mW [3] while the Cortex-M7 core, Armv7-M, at 200 Mhz typically consumes 283.8 mW (with peripherals disabled) [35].

References

[1] Amazon Web Services. What is unit testing? - unit testing explained - aws, 2024. Available at https://aws.amazon.com/what-is/unit-testing/, last accessed in 10 Jun 2024.

[2] Arm. Arm cortex-m7 processor technical reference manual r1p2, June 2022. Available at https://developer.arm.com/documentation/ddi0489/f/?lang=en, last accessed in 03 Jun 2024.

[3] Emily Blem, Jaikrishnan Menon, and Karthikeyan Sankaralingam. Power struggles: Revisiting the risc vs. cisc debate on contemporary arm and x86 architectures. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–12, 2013.

[4] Maurantonio Caprolu, Roberto Di Pietro, Flavio Lombardi, and Simone Raponi. Edge computing perspectives: Architectures, technologies, and open security issues. In *2019 IEEE International Conference on Edge Computing (EDGE)*, pages 116–123, 2019.

[5] Cloudflare. What is the osi model?, n.d. Available at www.cloudflare.com/learning/ddos/glossary/open-systems-interconnection-model-osi/, last accessed in 15 May 2024.

[6] Steve Corrigan. Introduction to the controller area network (can). *Application Report, Texas Instruments*, 2002.

[7] P.S. Dias, S.L. Fraga, R.M.F. Gomes, G.M. Goncalves, F.L. Pereira, J. Pinto, and J.B. Sousa. Neptus - a framework to support multiple vehicle operation. In *Europe Oceans 2005*, volume 2, pages 963–968 Vol. 2, 2005.

[8] Universidade do Porto Faculdade de Engenharia LSTS. Plan supervision messages — imc v5.4.30 specification. Available at https://www.lsts.pt/docs/imc/master/Plan%20Supervision.html#plan-specification, last accessed 02 Jan. 2024).

[9] Erik Einhorn, Tim Langner, Ronny Stricker, Christian Martin, and Horst-Michael Gross. Mira - middleware for robotic applications. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2591–2598, 2012.

[10] CSS Electronics. Can bus explained - a simple intro [2024] – css electronics. Available at https://www.csselectronics.com/pages/can-bus-simple-intro-tutorial, last accessed 02 Jan. 2024.

[11] António Sérgio Ferreira, José Pinto, Paulo Dias, and João Borges de Sousa. The lsts software toolchain for persistent maritime operations applied through vehicular ad-hoc networks. In

2017 International Conference on Unmanned Aircraft Systems (ICUAS), pages 609–616, 2017.

[12] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, jul 1987.

[13] Ray Kurzweil. *The Law of Accelerating Returns*, pages 381–416. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[14] Frederic Leens. An introduction to i2c and spi protocols. *IEEE Instrumentation & Measurement Magazine*, 12(1):8–13, 2009.

[15] Qing Li and Caroline Yao. *Real-time concepts for embedded systems*. CRC press, 2003.

[16] LSTS. About | laboratório de sistemas e tecnologia subaquática. Available at https://lsts.fe.up.pt/about, last accessed 02 Jan. 2024).

[17] LSTS. Dune | laboratório de sistemas e tecnologia subaquática. Available at https://lsts.fe.up.pt/software/64, last accessed 02 Jan. 2024.

[18] LSTS. Imc | laboratório de sistemas e tecnologia subaquática. Available at https://lsts.fe.up.pt/software/63, last accessed 02 Jan. 2024).

[19] Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. Robot operating system 2: Design, architecture, and uses in the wild. *Science Robotics*, 7(66):eabm6074, 2022.

[20] Luís Madureira, Alexandre Sousa, José Braga, Pedro Calado, Paulo Dias, Ricardo Martins, José Pinto, and João Sousa. The light autonomous underwater vehicle: Evolutions and networking. In *2013 MTS/IEEE OCEANS-Bergen*, pages 1–6. IEEE, 2013.

[21] Ricardo Martins, Paulo Sousa Dias, Eduardo R. B. Marques, Jose Pinto, Joao B. Sousa, and Fernando L. Pereira. Imc: A communication protocol for networked vehicles and sensors. In *OCEANS 2009-EUROPE*, pages 1–6, 2009.

[22] P. Marwedel. Chapter 1. In *Embedded System Design*, page 1. Kluwer Academic, 2003.

[23] Lorenz Meier, Dominik Honegger, and Marc Pollefeys. Px4: A node-based multithreaded open source robotics framework for deeply embedded platforms. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 6235–6240, 2015.

[24] micro ROS. micro-ROS/micro-ROS-Agent: ROS 2 package using Micro XRCE-DDS Agent. https://github.com/micro-ROS/micro-ROS-Agent, last accessed 02 Jan. 2024.

[25] micro-ROS. Features and Architecture | micro-ROS, 2024. Available at https://micro.ros.org/docs/overview/features/, last accessed 17 Jul 2024.

[26] Malcolm S. Mollison, Jeremy P. Erickson, James H. Anderson, Sanjoy K. Baruah, and John A. Scoredos. Mixed-criticality real-time scheduling for multicore systems. In *2010 10th IEEE International Conference on Computer and Information Technology*, pages 1864–1871, 2010.

[27] G. Pardo-Castellote. Omg data-distribution service: architectural overview. In *23rd International Conference on Distributed Computing Systems Workshops, 2003. Proceedings.*, pages 200–206, 2003.

[28] José Pinto, Paulo S. Dias, Ricardo Martins, João Fortuna, Eduardo Marques, and João Sousa. The lsts toolchain for networked vehicle systems. In *2013 MTS/IEEE OCEANS - Bergen*, pages 1–9, 2013.

[29] José Pinto, Paulo Sousa Dias, and João Borges de Sousa. Coordinated operation of multiple auvs using the lsts toolchain. In *2018 IEEE/OES Autonomous Underwater Vehicle Workshop (AUV)*, pages 1–6, 2018.

[30] Pixhawk. Pixhawk autopilot v5x standard, 2022. Available at https://github.com/pixhawk/Pixhawk-Standards/blob/master/DS-011PixhawkAutopilotv5XStandard.pdf, last accessed in 26 Jul. 2024.

[31] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016.

[32] IEEE Computer Society. Ieee standard for ethernet 802.3-2018. *IEEE Standard for Ethernet*, 2018, 2018.

[33] Alexandre Sousa, Luis Madureira, Jorge Coelho, José Pinto, João Pereira, João Borges Sousa, and Paulo Dias. Lauv: The man-portable autonomous underwater vehicle. *IFAC Proceedings Volumes*, 45(5):268–274, 2012.

[34] STMicroelectronics. NUCLEO-F767ZI, 2024. Available at www.st.com/en/evaluation-tools/nucleo-f767zi.html, last accessed in 17 May 2024.

[35] STMicroelectronics. STM32F767ZIT6 Microcontroller Datasheet, 2024. Available at www.st.com/resource/en/datasheet/stm32f767zi.pdf, last accessed in 17 May 2024.

[36] Marilyn Wolf. Chapter 1 - introduction. In Marilyn Wolf, editor, *Embedded System Interfacing*, pages 1–22. Morgan Kaufmann, 2019.

[37] Lifang Zhai, Chunyuan Li, and Liping Sun. Research on the message-oriented middleware for wireless sensor networks. *J. Comput.*, 6(5):1040–1046, 2011.

[38] Yousaf Zikria, Sung Won Kim, Oliver Hahm, Muhammad Afzal, and Mohammed Aalsalem. Internet of things (iot) operating systems management: Opportunities, challenges, and solution. *Sensors*, 8:1–10, 04 2019.