F3

Faculty of Electrical Engineering
Department of Measurement

Master's Thesis

Design and Implementation of Systems for Railway Fail-Safe Platforms

Bc. Petr Kučera
Open Informatics
Computer Engineering

November 2024, May 2025
Supervisor: doc. Ing. Jiří Novák, Ph.D. and Ing. Bc. Martin Votava

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Kučera** Jméno: **Petr** Osobní číslo: **499325**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra měření**

Studijní program: **Otevřená informatika**

Specializace: **Počítačové inženýrství**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Návrh a implementace systémů pro platformy v železniční infrastruktuře odolné vůči selhání

Název diplomové práce anglicky:

Design and Implementation of Systems for Railway Fail-Safe Platforms

Jméno a pracoviště vedoucí(ho) diplomové práce:

doc. Ing. Jiří Novák, Ph.D. katedra měření FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **29.01.2025** Termín odevzdání diplomové práce: **23.05.2025**

Platnost zadání diplomové práce: **20.09.2026**

Jan Holub
Digitálně podepsal(a) Jan Holub
Datum: 02.04.2025 06:19:41

podpis vedoucí(ho) ústavu/katedry

Jiří Jakovenko
Digitálně podepsal(a) Jiří Jakovenko
Datum: 04.04.2025 08:25:31

podpis proděkana(ky) z pověření děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

29.04.2025
Datum převzetí zadání

Bc. Kučera Petr
Podpis studenta

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Kučera** Jméno: **Petr** Osobní číslo: **499325**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra měření**

Studijní program: **Otevřená informatika**

Specializace: **Počítačové inženýrství**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Návrh a implementace systémů pro platformy v železniční infrastruktuře odolné vůči selhání

Název diplomové práce anglicky:

Design and Implementation of Systems for Railway Fail-Safe Platforms

Pokyny pro vypracování:

1. Natudujte standardy implementace softwaru na platformách zabezpečených selhání pro železniční infrastrukturu. Identifikujte rozdíly mezi požadavky platformy SIL2 (SIL1) a SIL4 (SIL3). Diskutujte o konkrétních mechanismech, vedoucích k naplnění individuálních požadavků.
2. Vyberte příslušný hardware (pro SIL2 a možná vyšší) a navrhněte architekturu, která splňuje požadavky CENELEC pro SIL2. Návrh by měl zahrnovat sekvenci a mechanismy pro detekci a zmírnění náhodných chyb a dalších identifikovaných problémů se softwarem. Navrhované zařízení by se mělo skládat z bezpečnostní části, odpovědné za provádění bezpečných výpočtů, a neefektivní části, odpovědné za monitorování bezpečnostní části a přenosu dat přes Ethernet.
3. Implementujte navrženou architekturu softwaru na vybraném hardwaru.
4. Zdůvodněte implementaci SIL2, analyzujte a diskutujte o možných krocích a rozšířené argumentaci k dosažení vyšší hladiny SIL.

Seznam doporučené literatury:

• CENELEC. EN 50129, Railway applications - Communication, signalling and processing systems - Safety related electronic systems for signalling. November 2018.
• CENELEC. EN 50128, Railway applications - Communication, signalling and processing systems - Software for railway control and protection systems. June 2011.
• CENELEC. EN 50126-1, Railway applications - The Specification and Demonstration of Reliability, Availability, Maintainability and Safety (RAMS) - Part 1: Generic RAMS Process. October 2017.
• B. Evers, ”Hazard Modelling for Electronic Systems in Railway Operation and Control,” 2008 3rd IET International Conference on System Safety, Birmingham, 2008, pp. 1-5, doi: 10.1049/cp:20080741.
• K. Rástočný, J. Ždánsky and J. Hrbček, ”The Problems Related to Realization of Safety Function with SIL4 Using PLC,” 2020 Cybernetics & Informatics (K&I), Velke Karlovice, Czech Republic, 2020, pp. 1-5, doi: 10.1109/KI48306.2020.9039878.
• T. G. Markovits and G. Rácz, ”Safety principles for designing a generic product for railway signalling systems,” 2021 Fifth World Conference on Smart Trends in Systems Security and Sustainability (WorldS4), London, United Kingdom, 2021, pp. 134-139, doi: 10.1109/WorldS451998.2021.9514005.
• O. Morgan, ”Certified Testing of C Compilers for Embedded Systems,” 2007 3rd Institution of Engineering and Technology Conference on Automotive Electronics, Warwick, UK, 2007, pp. 1-5.

• H.Ahangari, F.Atik, Y.I.Özkök, A.Yildirim, S.O.Ata and O.Ozturk, ”Analysis of Design Parameters in Safety-Critical Computers,” in IEEE Transactions on Emerging Topics in Computing, vol. 8, no.3, pp. 712-723, 1 July-Sept. 2020, doi: 10.1109/TETC.2018.2801463.
• H. Ahangari, Y. ˙I. Özkök, A. Yıldırım, F. Say, F. Atik and O. Ozturk, ”Analysis of design parameters in SIL-4 safety-critical computer,” 2017 Annual Reliability and Maintainability Symposium (RAMS), Orlando, FL, USA, 2017, pp. 1-8, doi: 10.1109/RAM.2017.7889787.

FAKULTA ELEKTROTECHNICKÁ
FACULTY OF ELECTRICAL ENGINEERING
Technická 2
166 27 Praha 6

DECLARATION

I, the undersigned

Student's surname, given name(s): Kučera Petr
Personal number: 499325
Programme name: Open Informatics

declare that I have elaborated the master's thesis entitled

Design and Implementation of Systems for Railway Fail-Safe Platforms

independently, and have cited all information sources used in accordance with the Methodological Instruction on the Observance of Ethical Principles in the Preparation of University Theses and with the Framework Rules for the Use of Artificial Intelligence at CTU for Academic and Pedagogical Purposes in Bachelor's and Continuing Master's Programmes.

I declare that I used artificial intelligence tools during the preparation and writing of this thesis. I verified the generated content. I hereby confirm that I am aware of the fact that I am fully responsible for the contents of the thesis.

In Prague on 23.05.2025

Bc. Petr Kučera
.....
student's signature

Acknowledgement / Declaration

I express my gratitude to my thesis advisor, doc. Ing. Jiří Novák, Ph.D., for taking on this project. I also thank Ing. Bc. Martin Votava from Siemens Mobility for his guidance and consultations. Additionally, I am grateful to Ing. Tomáš Hering and Ing. Jan Volný from Siemens Mobility for their regular consultations. Finally, I extend my thanks to all those not specifically named who contributed to this work, whether through discussions that guided me in the right direction or by providing direct advice on how to proceed.

The declaration of independent work and the use of artificial intelligence tools, verified and generated in the KOS system, is included in this thesis on a separate sheet in accordance with regulations.

Abstrakt / Abstract

Diplomová práce se zabývá návrhem monitorovacího systému pro světelnou signalizaci železničního přejezdu s využitím vícejádrového embedded zařízení. První část práce se věnuje analýze požadavků na systémy s funkční bezpečností (fail-safe design) v souladu s platnými normami pro železniční aplikace. Následuje specifikace systémových požadavků, výběr vhodné hardwarové platformy a návrh softwarové architektury s důrazem na bezpečnostní mechanismy. Třetí část se zaměřuje na implementaci funkčního prototypu, včetně bootovacího procesu. Závěrečná část diskutuje strategie testování a návrhové přístupy, které přispívají ke zvýšení úrovně bezpečnostní integrity (SIL). Výstupem práce je návrh systémové architektury a implementace softwarového prototypu připraveného k testování na cílovém embedded zařízení.

Klíčová slova: funkční bezpečnost, návrh odolný vůči sehnáním, bezpečně kritický vývoj, železniční přejezd, úroveň integrity bezpečnosti, železniční standardy, CENELC, Sitara AM243x, vestavěné zařízení, více jádrová architektura, jádro reálného času, izolované jádro, nahrávání firmware, SPI komunikace, bezpečnostní manuál

Překlad titulu: Návrh a implementace systémů pro platformy v železniční infrastruktuře odolné vůči selhání

The diploma thesis focuses on the design of a monitoring system for level crossing signaling using a multi-core embedded device. The first part analyzes the requirements for systems with functional safety (fail-safe design) in compliance with applicable railway application standards. This is followed by the specification of system requirements, selection of an appropriate hardware platform, and design of a software architecture with an emphasis on safety mechanisms. The third part concentrates on the implementation of a functional prototype, including the boot process. The final part discusses testing strategies and design approaches that contribute to achieving higher Safety Integrity Levels (SIL). The outcomes of the thesis are a proposed system architecture and an implemented software prototype ready for testing on the target embedded device.

Keywords: Functional Safety, Fail-Safe Design, Safety-Critical Development, Railway Crossing, Safety Integrity Level (SIL), Railway Standards, CENELEC, Sitara AM243x, Embedded System, Multi-Core Architecture, Real-Time Core, Isolated Core, Firmware Deployment, SPI Communication, Safety Manual

Contents /

Tables / Figures

Chapter 1
Introduction

This chapter introduces the motivation for this thesis and outlines its objectives.

1.1 Motivation

Rail transport is one of the most energy-efficient and high-capacity modes of transportation, enabling the movement of large volumes of freight and passengers with relatively low energy consumption. It forms a critical component of global infrastructure, with approximately 8% of goods and passengers worldwide transported by rail [1].

Given these benefits, ensuring safety and minimizing risks are critical priorities. This section justifies the importance of railway safety before exploring specific safety mechanisms.

1.1.1 High Impact

The low rolling resistance of steel wheels on rails, combined with efficient electric propulsion, makes rail transport ideal for transporting large volumes of passengers and freight. As illustrated in Figure 1.1, rail systems offer superior passenger capacity relative to time and space in urban environments.

Moreover, rail transport is highly energy-efficient, consuming approximately seven times less energy per passenger than car travel in urban settings [2]. However, the large scale of rail operations means that accidents can have severe consequences, underscoring the need for robust safety measures.

1.1.2 Strategic Importance

Railways are vital for industrial supply chains, transporting critical materials such as coal and chemical components due to their high capacity and ability to handle heavy loads. They also play a key role in strategic logistics, enabling efficient transport of military equipment and supplies in various global and regional contexts [3]. Moreover, rail transport is essential for delivering humanitarian aid during natural disasters, provided the infrastructure is designed to withstand environmental challenges such as floods, earthquakes, or extreme weather.

Beyond these roles, railways are strategically critical for global and regional connectivity. They facilitate trade by linking urban centers, ports, and industrial hubs, as seen in initiatives like China's Belt and Road, which enhances cross-border commerce [4]. Railways also support the energy transition by enabling low-carbon transport and the movement of renewable energy components, aligning with sustainability goals like the EU's Green Deal [5]. Additionally, rail networks drive urban and regional development by connecting cities and rural areas, fostering economic equity and mobility [6]. Finally, transcontinental rail corridors strengthen international trade resilience, offering alternatives to maritime routes and mitigating geopolitical risks [7].

Consequently, railway infrastructure must be engineered for reliability across diverse conditions. Robust design and operational measures should address human error, environmental risks, and capacity demands, with requirements varying by geography. For example, northern railways must endure extreme cold and heavy snowfall, while southern systems need resilience against high temperatures and coastal rainfall.

1.1.3 Personal Motivation

My primary interest is in embedded systems, focusing on direct hardware control, peripheral interaction, and low-level software development. Designing safe software for fail-safe railway platforms aligns with these interests, as it requires ensuring reliability, addressing physical constraints, and minimizing abstraction layers.

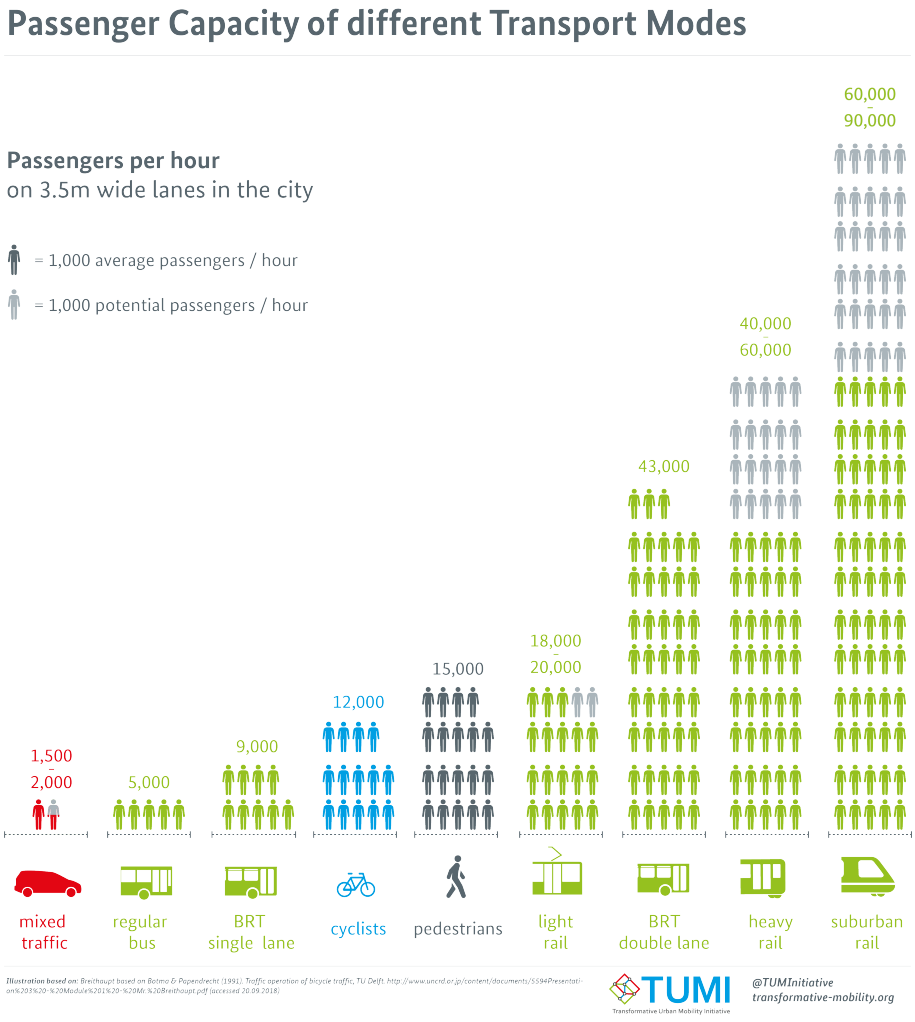

Figure 1.1. Passenger capacity of various transport modes, showing the number of passengers per hour on 3.5 m wide lanes in urban environments [8].

1.2 The Aim of This Thesis

This thesis aims to design and implement a fail-safe system for railway applications, specifically a component for a railroad crossing system that controls LED lights. The

system must comply with CENELEC[1] standards. Detailed system and functional requirements are provided in Chapter 3.

To achieve this objective, the following steps are proposed:

- Analyze fail-safe software standards for railway systems. Identify the differences between SIL2 and SIL4 platforms and examine specific mechanisms that ensure compliance with safety requirements.
- Choose suitable hardware and design a CENELEC-compliant SIL2+ architecture. The design should include a boot sequence and mechanisms for detecting and mitigating random errors and software faults. The proposed device will consist of a safety part responsible for critical computations and a non-safety part responsible for monitoring and data transmission over Ethernet.
- Implement the designed software prototype architecture on the selected hardware.
- Justify the implementation for SIL2 and evaluate potential steps required to achieve higher SIL levels.

[1] European Committee for Electrotechnical Standardization

Chapter 2
Background Research

In this chapter, I studied the standards for design and implementation of software for fail-safe platforms in railway infrastructure environments. I have tried to identify the differences between the different safety integration levels and discuss the specific mechanisms that lead to meeting the safety requirements given by the standards.

2.1 The Difference between Security and Safety

Before studying the issue, it is necessary to get the terminology right. In the English language, there are two words: safety and security.

The term **security** refers to the discipline that aims to protect against harmful attacks or actions such as vandalism, cyber-attack, or terrorism. Physical barriers, authentication, encryption, or monitoring systems are used to protect [9].

The term **safety** refers to the discipline aimed at preventing unintentional incidents, such as accidents, errors, or technical failures, which may result in injury, loss of life, or property damage. Typical threats include natural hazards, equipment malfunctions, or human error. Safety measures encompass robust design standards, regular maintenance, and human factor engineering to minimize risks [10].

The importance of safety in railway infrastructure and rolling stock cannot be overstated, as the potential consequences of technical failure or human error can be catastrophic. A stark reminder is the derailment of the ICE 884 near Eschede, Germany in 1998, which resulted in the deaths of 101 people and injuries to over 80. The accident was caused by a fatigue crack in a wheel tire, which had not been properly detected during maintenance inspections [11]. This tragic event exposed the vulnerabilities in safety assurance processes and highlighted the necessity for rigorous inspection protocols, fail-safe design principles, and continuous monitoring. Since then, significant advances have been made in railway safety, including the adoption of condition-based maintenance, advanced track monitoring systems, and automated train control technologies [12]. These improvements illustrate how safety in railways depends not only on robust engineering but also on a systemic approach integrating human factors, operations, and organizational learning from past incidents.

To illustrate the critical role of safety in a more familiar context, the importance of safety in railway systems can be compared to the automotive industry, which we encounter more frequently. The emphasis on safety over security in car transportation is driven by statistical evidence: road traffic accidents remain a leading cause of death globally, with approximately 1.19 million fatalities annually [13]. In contrast, security-related incidents, such as cyberattacks on vehicles, while emerging as a concern, are less frequent and typically have less immediate impact on human lives. For example, a malfunctioning braking system poses a direct and immediate risk to occupants, whereas a security breach, such as unauthorized access to a vehicle's infotainment system, may compromise privacy but is less likely to cause physical harm. Therefore, prioritizing

Aspect	Safety	Security
Focus	accident prevention and technical failures	prevention of deliberate attacks
Threats	technical faults, random failures, technical failures	human intentional actions, malicious actions
Methods	system reliability, standards	security, encryption, monitoring
Example of measures	fail-safe systems, resilience testing	physical security, cybersecurity systems

Table 2.1. Main differences between the terms safety and security

safety through robust engineering, driver training, and regulatory frameworks is critical to enhancing the reliability and trustworthiness of car transportation systems.

Table 2.1 compares the different aspects of safety and security in order to highlight the different meanings. In this thesis, I will deal primarily with the second term - safety.

2.2 Materials and Standards

The safety requirements for transport infrastructure in the European Union are defined in standards issued by CENELEC.[1] These standards are *EN 50126*, *EN 50128*, *EN 50129*, and *EN 50159*.

The standard **EN 50126** relates to railway equipment in general and specifies and defines how to demonstrate reliability, availability, maintainability, and safety (RAMS). For more information see chapter 2.3.

The norm **EN 50129** describes the standard for the railway communication and signaling system and the data processing system (software). The annexes provide detailed instructions on how to test hardware components including microcontroller components. The standard **EN 50159** complements the standard for safety communications.

The norm **EN 50128** defines a standard on how to properly develop safety software for railway infrastructure.

2.2.1 New Standard EN 50716

In November 2023, CENELEC issued the new standard *EN 50716:2023*, titled Railway Applications - Requirements for software development, which replaces *EN 50128:2011* and *EN 50657:2017* [14]. This new standard unifies software development requirements for both signaling systems and vehicles, eliminating the need to separate specifications. It now applies to software with basic integrity (formerly *SIL0*), removing the restriction that excluded non-safety-related software. Key changes include:

- **Scope and Normative References**: The standard no longer lists normative references, and the requirement for separate safety and non-safety specifications is clarified, with all system requirements now covered under a single *System Requirements Specification*.

[1] European Committee for Electrotechnical Standardization

- **Terms and Definitions**: Terms are aligned with ISO[2] and IEC[3] standards, and the *Integrator* role abbreviation is removed.
- **Software Integrity**: A quality assurance process is now mandatory for non-safety-related software, with *basic integrity* requirements introduced. No assessor is required for basic integrity systems (SIL 0).
- **Organization and Management**: Organizational independence is emphasized, addressing pressures from peers, supervisors, or profit-driven motives that could compromise safety. The integrator role is removed from the organizational structure.
- **Software Assurance**: Validators can now perform additional audits, and documentation requirements for basic integrity are relaxed (e.g., software architecture and design specifications are now recommended rather than highly recommended).
- **Software Development**: The integrator role is replaced by a tester, and programming language criteria are updated to focus on features like modular programming and strict typing, enabling the use of modern languages like Rust.
- **Application Data and Maintenance**: Chapters 8 and 9, covering application data/algorithms and software deployment/maintenance, are critical for system configuration and safety, but require separate detailed analysis due to their complexity.

This thesis primarily builds on the *EN 50128:2011* standard, as the core work was completed prior to the application of *EN 50716:2023* to the working process.

2.3 RAMS

The acronym RAMS is composed of the initial letters of the words **Reliability** (components do not fail too often), **Availability** (to be sure that components will work as required and that if they fail, they will not affect the functionality of the equipment), **Maintainability** (and easy replaceability in the event of damage), and **Safety** (described in the Chapter 2.1).

No approach can achieve 100% success in any RAMS aspect; it can only be approximated. The desired level of achievement must be clearly defined.

It is necessary to remember the economic side of the issue. The ratio of the investment in making the platform safer and how much it will cost to develop and produce is growing exponentially, in general. Thus, going from 20% to 60% safety will be much less economically challenging than going from 99.99% to 99.999999%. If we develop this theory into implications, we necessarily arrive at the question - how expensive is human life? Theoretically, this cost could be quantified according to standards, state-specific requirements, and crash statistics.

If we go back to all four aspects, we find that some of the essences are contradicting. For example, we're going to make the equipment safer and more robust; it's going to reduce availability and reliability.

I also find it important to answer the question - why is it necessary to pay attention to system availability or repairability? After all, it is only the safety itself that is important to us. If we think about it, as a result, all the aspects already mentioned are important because they complement each other. You cannot achieve safety without having a low failure rate system, or you cannot achieve safety with a possible system failure without affecting the requirement that the component is supposed to fulfill.

[2] International Organization for Standardization
[3] International Standards

2.4 Systematic and Random Failures

Random failures can be described by statistical distributions [15]. We cannot control failures; we can only predict and model them mathematically. In most cases, the failure is due to a physical cause such as material fatigue, wear, corrosion, or random failures of electronic components.

The *EN 50126* standard defines **systematic failures** as failures caused by errors in system life cycle activities that lead to deterministic failure of a product, system, or process under certain combinations of inputs or conditions [15]. Unlike random failures, systematic failures are usually caused by human errors at various stages of the system life cycle, namely incorrect specification, system design, development, manufacture, or operation and maintenance. These failures, on the other hand, are preventable from accidental failures by process, testing, and validation phases.

By the norm *EN 50129*, these errors are, for example: specification errors, design errors, manufacturing errors, installation errors, operation errors, maintenance errors, or modification errors.

Aspect	Random Failure	Systematic Failure
Cause	physical processes (fatigue, wear, environment)	human error during design, implementation or maintenance
Characteristic	stochastic, unpredictable at the individual level	deterministic, predictable under certain conditions
Solution	predictive maintenance, statistical analysis, redundancy	validation, verification, process measures
Repetition	not repeatable under the same conditions	repeated every time under the same conditions
Examples	rail breakage, bearing failure	software error, sensor miscalibration

Table 2.2. Main differences between random and systematic failures.

2.4.1 RAMS Development Cycle

The RAMS development cycle is described by the V-model, which uses *top-down* and *bottom-up* approaches [16]. It is a graphical representation of the complete development lifecycle [17]. The advantages of this model are clear structure, easy traceability between the design, respective requirements, and testing results.

The model also has many disadvantages such as the lack of flexibility; the later a bug is discovered, the more expensive it is to fix [18]. However, in the context of rail transport, these weaknesses are a necessity to ensure safety.

EN 50128 describes in more detail the different phases and roles that are responsible for each phase. The V-model is specific to two activities - validation and verification.

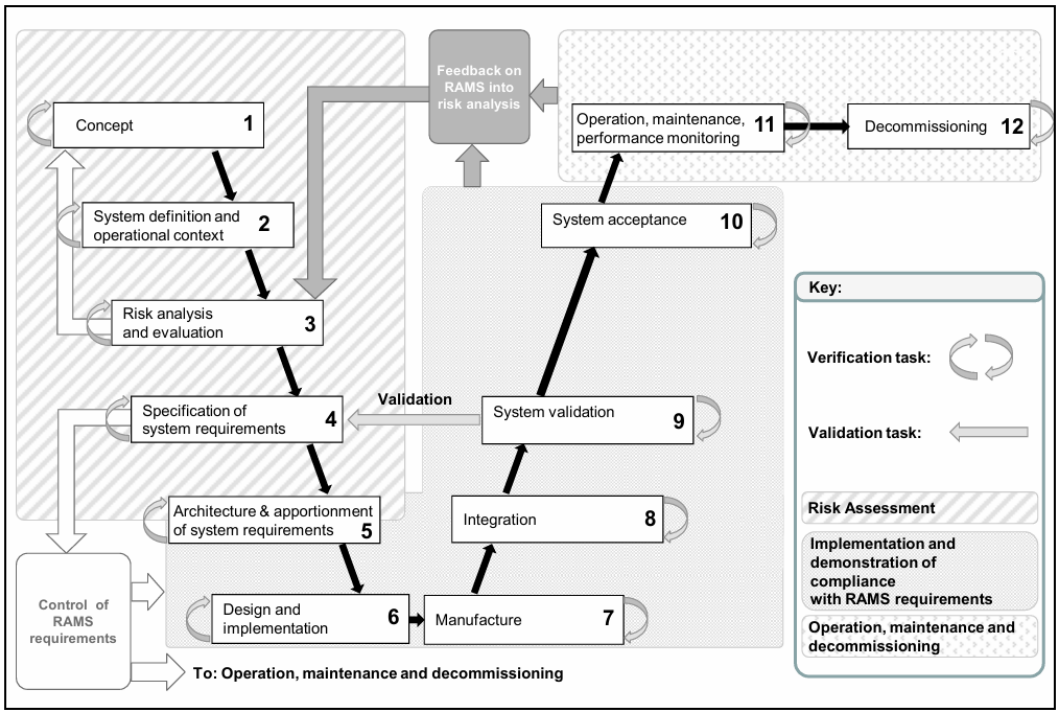

Figure 2.1. V-model illustration by *EN 50128* norm definition [19].

2.4.1.1 Verification and Validation

The standard defines **verification** as the confirmation, through provided objective evidence, that defined requirements have been met, and **validation** as the confirmation, through provided objective evidence, that requirements for a specific purpose or application have been met [15].

Simply, verification answers the question *are we building things correctly*, validation answers the question *are we building the right thing*?

2.4.1.2 System Description

Defining system requirements is the first and very important step for a successful and safe system or component. In this phase, it is important to define the limits, interfaces, functions, and the environment of the system.

It is important to note that SIL levels,[4] which we will discuss later, are always related to a component - e.g. a function. So we cannot say that the whole platform is a SIL, but only that the platform provides a function that has that SIL.

To illustrate, let's take a specific example: automatic train control, specifically emergency braking in case of a dangerous situation, e.g. when a train runs a red light. In this case the **function** is: automatic braking when passing a dangerous signal. The driver's cab, the track equipment and the signal transmission path are the **system**. The **Interfaces** for us are the wheels, the dashboard which serves as an API for the driver to communicate with the system, and the warning light which signals the entry prohibition.

2.4.1.3 Risk Analysis

In the context of safety-critical systems, we define the hazards relevant to this study. The **hazard** standard defines a hazard as any condition that could precipitate an accident [15].

[4] Safety integrity level

In the context of a railway monitoring system, a hazard can be exemplified by the failure to initiate emergency braking or the inability to receive a stop command. In safety-critical systems, the presence of a hazard inherently introduces risk. According to relevant standards, risk is defined as the combination of the anticipated frequency of a hazard's occurrence and the severity of its potential consequences.

To make a system safe, our goal is to minimize risk to the point that it is equal to or less than the acceptable risk. We achieve this by adding just safe features to our system such as the redundancy principle, the watchdog principle, or the error detection principle.

Risk analysis consists of some of the methods already mentioned: hazard identification, frequency minimization, and consequence classification.

2.4.1.4 Acceptable Risk

The European Union specifies in the CSM regulation [5] which methods can be used to calculate acceptable risk.

- **Application of codes of practice**: The use of established norms, standards, and best practice methodologies for risk assessment and management that have been validated in practice.
- **Comparison with similar system**: Risk assessment based on analysis and comparison with other systems that have similar functionality, architecture, or operating conditions. This approach is particularly useful when assessing a new context.
- **Explicit risk estimation**: Direct and detailed quantitative or qualitative risk assessment using specific methods. This approach requires data collection and detailed modeling [20].

2.4.1.5 Hazard Analysis (FTA, FMEA)

The standard *EN 50126* recommends the use of top-down analysis for hazard analysis. Specifically, **Fault Tree Analysis (FTA)** which allows us to work with dependencies between individual hazards. The second method used is **Failure Modes and Effects Analysis (FMEA)**. The FTA is suitable for analyzing multiple system failures; FMEA is suitable for analyzing a single failure, including all its consequences.

2.4.1.6 Tolerated Risk Ratios (THR, TFFR)

Tolerable Hazard Rate (THR), which represents the maximum frequency of occurrence of a particular hazard. It therefore determines how many times a potentially hazardous event can occur in a defined period without affecting the overall safety of the system [19].

Tolerable Functional Failure Rate (TFFR), which represents the maximum frequency of failure of a specific system function, focuses on the system's functional aspects and considers how often a function can fail without leading to unacceptable risk [19].

THR will be sufficient if our system consists of only one safe function. However, if there is more than one, we need to use TFFR to combine them.

2.4.1.7 Safety Integrity Level (SIL)

The term SIL is a method used to evaluate and classify the level of integrity of the safety functions of a system. Previous methods have described to us how to deal primarily with random error. SIL tells us what type of architecture, validation, testing, etc. to choose depending on the level of system error.

[5] Common Safety Method for Risk Evaluation and Assessment

TFFR [h^{-1}]	SIL level
$10^{-9} \leq TFFR < 10^{-8}$	4
$10^{-8} \leq TFFR < 10^{-7}$	3
$10^{-7} \leq TFFR < 10^{-6}$	2
$10^{-6} \leq TFFR < 10^{-5}$	1
$10^{-5} \leq TFFR$	basic integrity

Table 2.3. Relationship between TFFR and required Safety Integrity Level (SIL) [16].

The norm describes 4 Safety Integrity Levels from 1 to 4 and Basic Integrity. The SIL 4 is the highest level and the Basic Integrity is the lowest, as described in the table 2.3.[6]

2.4.2 The Five Base Questions

I would like to mention that an excellent mechanism to verify that we haven't forgotten anything elementary is to answer these five basic questions that are based on *EN 50126*.

- **What** - What features must be implemented?
- **How** - What steps must be followed during implementation?
- **With what** - What tools must be used for implementation?[7]
- **Assurance** - How can I ensure that the first three questions are answered correctly?
- **Traceability** - Have I recorded everything in a way that allows for audit and verification?

2.5 Mechanisms to Achieve SIL Level

I would like to discuss the mechanisms, architecture types, principles, and policies that ensure that a given platform contains SIL functionality. The thesis does not address the processes behind the development of safety software, which are an integral part of the overall development process.

In this chapter we will use terms defined by *EN 50128* and *EN 50129*, which are widely used in their annexes and can be found in this thesis in tables or figures. In order to understand them, let's clarify them.

- **'M' - Mandatory**: This symbol means that the use of a technique is mandatory [19].
- **'HR' - Highly Recommended**: This technique or measure is highly recommended for the SIL. It is considered a key measure, and implementation is mandatory. In case of absence, clear reasoning is required.
- **'R' - Recommended**: This technique or measure is recommended but not required or necessary to meet the safety requirement for a given SIL. Implementation of this technique can improve safety and reliability.
- **'-' No Recommendation**: There is no recommendation or non-recommendation for this technique or measure. Use is discretionary and should be decided based on the requirement specification.

[6] The norm *EN 50129* from year 2003 introduced term **SIL 0** to indicate non-safety-related functions. This terms in no longer used in the new norms version.

[7] The tool problematic is detailed discussed int the Chapter 2.5.4.

- **'NR' - Not Recommended**: This technique or measure is not recommended for a given SIL. Implementation may be ineffective, inadequate, or even counterproductive.

2.5.1 The Principles in Developing High-integrity Software

The *EN 50128* standard defines ten basic principles, the application of which in the development of high integrity software is not mandatory. Let's highlight the six most important ones.

- **Top-down design method**: This method divides the system into smaller parts and gradually works its way from generalities to specific issues.
- **Modularity**: The modularity aims to divide software into smaller independent blocks that can be maintained and tested independently. This division often makes it possible to work on as many components in parallel as possible at the same time. In addition, it allows for easier testing or making changes. The disadvantage can be a certain necessary level of abstraction, which in the case of embedded systems can be undesirable.
- **Verification of each phase of the development life-cycle**: This method aims to minimize the risk of errors during development by thoroughly verifying each stage. This leads to the early identification of problems and therefore saves time and money.
- **Verified components and component libraries**: The goal is to reduce the risk of bugs in systems by reusing verified components. In addition to preventing errors, this measure also saves money. Since the validation process of developing new software is very demanding, it is therefore preferable to use components that are already validated.
- **Clear documentation and traceability**: The aim is to provide a clearly interpretable, understandable, and quickly comprehensible description of the component.
- **Auditable documents**: The aim is to provide evidence that the system is developed and tested in accordance with the requirements.

2.5.2 Architecture

In designing SIL software, the standard defines 7 basic principles. However, in relevance to the use of a microprocessor, they can be generalized to 3 basic principles or approaches to ensure that the code is safe and meets all requirements.[8] Each of these approaches has its advantages and limitations depending on the SIL level required.

- **Inherent fail-safety**: This approach ensures the safety of the system due to the intrinsic properties of the design. Functions are designed to be fault-tolerant without the need for external intervention. It is therefore more a matter of hardware design. This method cannot be implemented in software. As an example, the signal remains in a safe state even after a power failure, without external intervention. This mechanism is recommended (R) for *SIL 1* and *SIL 2*, and highly recommended (HR) for *SIL 3* and *SIL 4*.
- **Reactive fail-safety**: The approach focuses on responding to faults typically through fault detection and subsequent activation of protection mechanisms. The system is therefore able to react on its own without external intervention to ensure a safe state. Examples include the activation of emergency braking or safety shutdown when a failure of the main system is detected. The mechanism is recommended (R) for *SIL 1* and *SIL 2* and highly recommended (HR) for *SIL 3* and *SIL 4*.

[8] The sub-parts of these mechanisms are described in *EN 50129*, Annex A, Table E.4.

- **Composite fail-safety**: The approach combines multiple channels with fail-safe mechanisms for peer-to-peer comparison. This provides greater resilience to failure through redundancy and independence. An example would be the need to obtain a positive signal from at least two of the three independent units to activate an action element. This mechanism, like the inherent and reactive mechanisms, is recommended (R) for *SIL 1* and *SIL 2* and highly recommended (HR) for *SIL 3* and *SIL 4*.

The standard also defines other architectural approaches that are less suitable for higher levels of safety integrity (*SIL 3* and *SIL 4*). These approaches can only be used in less critical applications such as *SIL 1* and *SIL 2*. These include, for example, a duplicated electronic structure, but where the channels may not be completely independent and the comparison of results may not be fail-safe. Another test is, for example, a simple electronic structure with self-tests, thus supervising its own functions. However, this approach again lacks independence between the function and its supervision. This limits this method, like the previous one, for higher SIL levels.

Technique/Measure	SIL 1	SIL 2	SIL 3	SIL 4
Separation of safety-related functions from non-safety-related functions to prevent unintended influences	R	R	R	R
single electronic structure with self-tests and supervision	R	R	NR	NR
single electronic structure based on inherent fail-safety	R	R	HR	HR
single electronic structure based on reactive fail-safety	R	R	HR	HR
Dual electronic structure	R	R	NR	NR
Dual electronic structure based on composite fail-safety with fail-safe comparison	R	R	HR	HR
Diverse electronic structure with fail-safe comparison	R	R	HR	HR

Table 2.4. Table showing techniques and measures for different SIL levels according to *EN 50129*, Annex A, Table E.4 [16].

Table 2.4 clearly shows the different mechanisms depending on the SIL level as defined by the standard. It is also not a bad idea to combine different mechanisms together. By definition, however, this cannot be implemented for all techniques.

Technique/Measure	SIL 1	SIL 2	SIL 3	SIL 4
Defensive Programming	HR	HR	HR	HR
Fault Detection & Diagnosis	R	R	HR	HR
Error Detecting Codes	R	R	HR	HR
Failure Assertion Programming	R	R	HR	HR
Safety Bag Techniques	R	R	R	R
Diverse Programming	R	R	HR	HR
Recovery Block	R	R	R	R
Backward Recovery	NR	NR	NR	NR
Forward Recovery	NR	NR	NR	NR
Retry Fault Recovery Mechanisms	R	R	R	R
Software Error Effect Analysis	R	R	HR	HR
Graceful Degradation	R	R	HR	HR
Information Hiding	-	-	-	-
Information Encapsulation	HR	HR	HR	HR
Fully Defined Interface	HR	HR	M	M
Formal Methods	R	R	HR	HR
Modeling	R	R	HR	HR
Structured Methodology	HR	HR	HR	HR
Modeling supported by CAD and specification tools	R	R	HR	HR

Table 2.5. Software Architecture mechanism for different SIL levels according to *EN 50128*, Annex A Table A.3 [19].

2.5.3 Software Architecture Technique

EN 50128 in Annex A defines specific mechanisms to be used in the design of the software architecture and its implementation for different SIL levels. We have listed some of them in Table 2.5.

2.5.3.1 Defensive Programming

The **Defensive Programming** is one of the most important techniques in safety programming. The aim is to produce programs that detect anomalous control flow, data flow, or data values during their execution and react to these in a predetermined and acceptable manner [19].

There are three basic techniques of defensive programming:

- All data[9] is important until proven otherwise.
- All input data is potentially hazardous until proven otherwise.
- All code is dangerous until proven otherwise.

There exist many techniques of defensive programming. For example, to ensure that the numbers manipulated by the program are reasonable, the norm *EN 50128* recommends that:

- Variables should be range-checked.
- Where possible, values should be checked for plausibility.[10]
- Parameters to procedures should be type, dimension, and range checked at procedure entry.

Safe software should be designed to expect failures in its own environment. The norm *EN 5018* also defines three techniques:

- Input variables and intermediate variables with physical significance should be checked for plausibility.
- The effect of output variables should be checked, preferably by direct observation of associated system state changes.
- The software should check its configuration. This could include both the existence and accessibility of expected hardware and also that the software itself is complete. This is particularly important for maintaining integrity after maintenance procedures.

There are more techniques like reusing quality code, handling I/O, testing, low tolerance, canonization, or control flow sequence checking, but the CENELEC norms do not explicitly mention them [19, 22–23].

2.5.3.2 Fault Detection and Diagnosis

The goal of **Fault Detection and Diagnosis** is to detect faults in a system, which might lead to a failure, thus providing the basis for countermeasures in order to minimize the consequences of failure [19].

Fault detection is based on the principles of **redundancy** and **diversity**. The redundancy can detect hardware faults, and diversity can detect software faults. For correct results interpretation, it is necessary to use a voting system.

Special applicable methods for software level are **assertion programming**, **N-version programming**,[11] or **safety bag technique**. For hardware level control loops, error checking codes, etc.

2.5.3.3 Error Detecting and Correcting Codes

The techniques **Error Detecting and Correcting Codes** aim to detect and correct errors in sensitive information [19].

[9] By data we mean all values stored in memory such as variables, objects, arrays, etc.
[10] Plausibility means quality of seeming likely to be true, or possible to believe [21]
[11] The **N-version programming**, also known as a multiple-version dissimilar software, is the method where multiple functionally equivalent programs are independently generated from the same initial specifications [24].

To safety n bits information is necessary to generate k bits of block code. The block code can be generated by different methods, like: Hamming codes, cyclic codes, polynomial codes, hash codes, or cryptographic codes.

2.5.3.4 Failure Assertion Programming

The goal of **Failure Assertion Programming** is to detect residual faults during the execution of a software program [19].

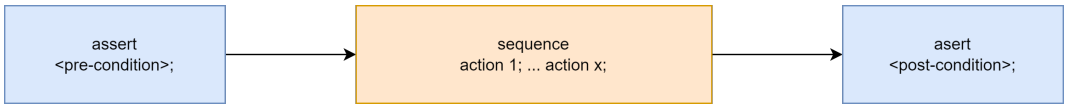

Figure 2.2. The diagram illustrates the failure assertion programming mechanism.

The assertion programming method in safety software development follows the idea of checking a **pre-condition**[12] and a **post-condition**[13] as illustrated in Figure 2.2. If either the pre-condition or the post-condition is not met, the processing terminates with an error.

2.5.3.5 Safety Bag Techniques

The **Safety Bag** technique is aimed at protecting against residual specification and implementation faults in software that adversely affect safety [19].

Practically, it is an external monitoring system implemented on an independent computer with a separate specification. Its primary role is to ensure that the main computer performs safe actions, though not necessarily correct ones. The safety bag continuously monitors the main computer and prevents the system from entering an unsafe state. Additionally, if it detects that the main computer is approaching a potentially hazardous state, it must restore the system to a safe condition, either on its own or in coordination with the main computer.

2.5.3.6 Diverse Programming

The goal of **Diverse Programming** technique is to detect and mask residual software design faults during the execution of a program, in order to prevent safety-critical failures of the system, and to continue operation for high reliability [19].

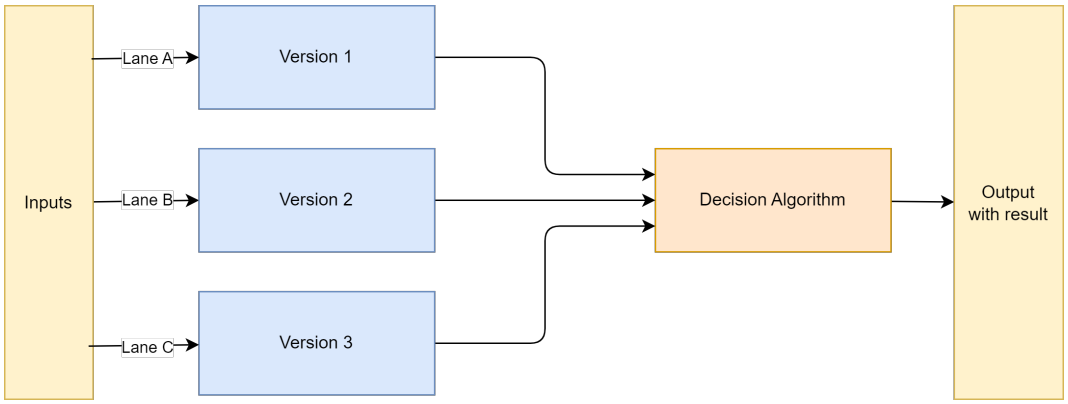

Figure 2.3. The diagram illustrates the diverse programming mechanism.

The technique is based on relying on multiple implementations of the same specification. Consequently, we expect that if the inputs are the same for all implementations,

[12] The initial conditions are checked for validity before a sequence of statements is executed.
[13] Results are checked after t he execution of a sequence of statements.

the outputs will be the same as well. If this is not the case, we need to have a voting strategy that tells us whether we accept a given outcome and possibly which one [25].

This technique does not eliminate residual software design faults but provides a mechanism to detect and mitigate them before they impact safety.

Studies and experiments[14] indicate that N-version programming does not always achieve the expected effectiveness. Despite using different algorithms, diverse software versions frequently fail on the same inputs.

2.5.3.7 Recovery Block

The **Recovery Block** is the technique to increase the likelihood of the program performing its intended function [19].

This technique may appear similar to Diverse Programming (N-version programming), but there are key differences between them. In the **N-version programming** technique, N independent groups or individual developers, who do not share the programming process, each develop a separate version of a software module. The underlying idea is that different developers will make different mistakes, thereby covering all possible faults. In the **Recovery Block technique**, different algorithms are assigned to distinct try blocks, which serve as redundant components. Unlike N-version programming, the redundant copies do not execute simultaneously. Instead, the result of each try block is evaluated using an acceptance test to determine its validity [26].

2.5.3.8 Retry Fault Recovery Mechanism

The goal of the **Retry Fault Recovery Mechanism** is to attempt functional recovery from a detected fault condition by re-try mechanism [19].

If an error is detected in a condition or procedure, re-execution of the same code can be started. In case of failure of a major part, a re-boot or re-start can be performed. In the case of using tasks and re-scheduling or re-starting the task.

2.5.3.9 Software Error Effect Analysis

The goal of the **SEEA (Software Error Effect Analysis)** is to identify software components and their criticality to propose means for detecting software errors and enhancing software robustness, and to evaluate the amount of validation needed on the various software components [19].

The norm *EN 50128* describes three phases of the SEEA, that is a powerful bug-finding method if it is carried out by an independent team:

- **Vital software components identification** - This phase aims to determine the depth of the analysis needed for each software component, from its specification.
- **Software error analysis** - The second phase aims to provide the following information:
  - component name,
  - error considered,
  - consequences of the error at the module level,
  - consequences at the system level,
  - violated safety criterion,
  - error criticality,
  - proposed error detection means,
  - violated criterion if the detection means is implemented,
  - and residual criticality if the detection means is implemented.
- **Synthesis** - The third phase aims to highlight the remaining unsafe scenarios and determine the validation effort required based on the criticality of each module.

[14] Source is the norm *EN 50128*, Annex D, Chap D.16.

2.5.3.10 Graceful Degradation

The technique **Graceful Degradation** aims to maintain the more critical system functions available despite failures by dropping the less critical functions [19].

The graceful degradation refers to the ability of a system to maintain a partial level of functionality when some components fail or are otherwise damaged. Instead of failing completely, a system designed with graceful degradation can reduce its quality of service while still providing essential functionality. This contrasts with `fail-stop` behavior, where the system completely ceases to function upon failure. The goal of graceful degradation is to ensure that users benefit from a reduced but functional level of service, thus minimizing the impact of failure. A system that is designed in this way is sometimes called fail-soft or fail-safe [27–28].

2.5.3.11 Information Encapsulation

The goal of the **Information Encapsulation** technique is to increase the robustness and maintainability of software [19].

Globally accessible data can be unintentionally or incorrectly modified by any software component, potentially requiring thorough code reviews and extensive changes. To mitigate these issues, information hiding is a widely used approach. It involves restricting direct access to key data structures, allowing them to be modified only through a predefined set of access procedures. This ensures that internal changes—such as altering data structures or adding new procedures—do not impact the overall functionality of the software. For instance, a directory system might include access procedures like *Insert*, *Delete*, and *Find*. These procedures, along with the underlying data structures, could be re-implemented (e.g., by adopting a different lookup method or storing data on a hard disk) without changing the logical behavior of the software that relies on them.

2.5.3.12 Fully Defined Interface

The technique **Fully Defined Interface** leads to modularization. A **Modular Approach** contains several rules for coding, maintenance phases, and design of the software project. The norm *EN 50128* defines a list of these rules to reach the modular methods for interfaces.

- A component or module shall have a single well-defined task or function to fulfill.
- Connections between components or modules shall be limited and strictly defined; coherence in one component or module shall be strong.
- Collections of subprograms shall be built providing several levels of components or modules.
- Subprograms shall have a single entry and a single exit only.
- Components or modules shall communicate with other components or modules via their interfaces. Where global or common variables are used, they shall be well structured, access shall be controlled, and their use shall be justified in each instance.
- All components or module's interfaces shall be fully documented.
- Any component or module's interface shall contain the minimum number of parameters necessary for the component or module's function.
- A suitable restriction of parameter number shall be specified.[15]

2.5.3.13 Formal Methods

The **Formal Methods** refer to mathematically rigorous techniques and tools for the specification, design, and verification of software and hardware systems. The application

[15] The norms *EN 50128* defines that suitable number is typically 5.

of formal methods in software and hardware design is driven by the idea that, similar to other engineering fields, conducting rigorous mathematical analysis enhances the reliability and robustness of a design [19, 29].

There is no universal formal method that is suitable for all scenarios. Rather, an appropriate mathematical model must be chosen for each situation. Therefore, we have several models. The standard *EN 50128* describes CSP, CCS, HOL, LOTOS, OBJ, Temporal Logic, VDM, Z Method, B Method, and Model Checking. Let's at least take a closer look at the most important ones.

The **Communicating Sequential Processes (CSP)** is a technique for the specification of concurrent software systems. CSP defines a language for specifying process systems and verifying their implementations via traces. A system is modeled as independent processes composed sequentially or in parallel. Processes communicate through channels, synchronizing only when both are ready, with optional event timing modeling. As an example of a suitable application, this method is systems of communicating processes operating concurrently.

The **Calculus of Communicating Systems (CCS)** is a means for describing and reasoning about the behavior of systems of concurrent, communicating processes. Like CSP, CCS is a mathematical calculus for modeling system behavior as independent processes running sequentially or in parallel. Processes communicate via ports, synchronizing only when both are ready. Non-determinism is supported, and systems can be refined top-down from traces or built bottom-up using composition rules.

The **Higher Order Logic (HOL)** is a formal language intended as a basis for hardware specification and verification. It was developed at the University of Cambridge Computer Laboratory. The notation is largely derived from Church's Simple Theory of Types, while the support system is based on the LCF (Logic of Computable Functions) framework.

The **Language for Temporal Ordering Specification (LOTOS)** is a means for describing and reasoning about the behavior of systems of concurrent, communicating processes.

The **OBJ** is an algebraic specification language. Users specify requirements in terms of algebraic equations. The goal of this technique is to provide a precise system specification with user feedback and system validation prior to implementation.

The **Temporal logic** aims to direct the expression of safety and operational requirements and formal demonstration that these properties are preserved in the subsequent development steps. Standard First-Order Predicate Logic does not account for time. Temporal logic extends it by introducing modal operators like `henceforth` (for ongoing conditions) and `eventually` (for future conditions). These help define system behaviors—e.g., safety properties must always hold, while certain states must be reached eventually. Temporal formulas describe sequences of system states, which can represent the entire system, a component, or a program. However, temporal logic does not handle precise time intervals directly; instead, additional time states must be defined within the system.

The **Vienna Development Method (VDM)**, **Z notation**, and **B method** are all formal methods used for specifying and verifying software systems, but they differ in their approaches and areas of application.

VDM is a mathematically based, model-driven specification technique developed at the IBM Laboratory Vienna in the 1970s [30]. It enables high-level abstraction in system modeling, facilitating early detection of design flaws and key system features. By providing a structured refinement process, validated models can be systematically transformed into detailed implementations while ensuring correctness through proof

obligations. With its formal semantics, VDM allows for rigorous verification of model properties, offering a high level of assurance. Additionally, an executable subset supports testing and validation, making it accessible even to non-experts through graphical user interfaces.[16]

The **Z method**, like VDM, employs a model-based approach with set-theoretic structures to define system states and their transitions. However, Z focuses on structuring specifications through schemas, making it well-suited for data-oriented, sequential systems. Unlike VDM, Z is more of a notation rather than a complete development method, relying on additional methodologies to transition from specification to implementation.

The **B method** extends the principles of Z by incorporating rigorous proof obligations and refinement steps that ensure the correctness of both system models and their implementations. It provides a structured approach to software development through abstract machines and step-by-step refinements, leading to deterministic implementations. Unlike VDM and Z, B emphasizes automatic proof generation and verification, making it particularly useful for safety-critical applications.

While all three methods share a foundation in formal logic and model-based specification, their primary distinctions lie in their scope and intended use cases. VDM is best suited for early-stage design validation and refinement into sequential programs. Z excels in formalizing complex data structures and relationships, whereas B extends these capabilities with rigorous proof obligations and structured refinements, ensuring the correctness of both design and implementation.

The last formal method mentioned in the norm *EN 50128* is the **Model Checking**. It aims to, given a model of a system, test automatically whether this model meets a given specification [19]. Model checking verifies whether a given structure satisfies a specified logical formula. It applies across various logic and structures, with a common example being the evaluation of propositional logic formulas. Primarily used in hardware design, model checking algorithmically ensures that a system model meets a formal specification, often expressed in temporal logic. For software, full automation is limited due to undecidability. The system is typically represented as a finite-state machine, where nodes correspond to system states, edges define transitions, and atomic propositions describe key properties. When the model is finite, verification reduces to a graph search.

2.5.3.14 Data Modeling

The technique **Data modelling** is the process of creating a data model by applying formal data model descriptions using data modelling techniques.

In software engineering, a data model is an abstract representation of how data is structured, stored, and accessed within a system. It defines data objects, their relationships, and the rules governing their interactions within a specific domain. Data modeling is essential for designing databases, ensuring data consistency, and facilitating seamless data exchange across systems.

2.5.3.15 Structured Methodology

The approach **Structured Methodology** aims to improve software quality by emphasizing the early stages of the development life cycle. This method is related to the V-model and one of its goals is to.

Structured methodologies provide systematic procedures and notations to define requirements and implementation features in a logical and structured manner. These

[16] Recommending for simple human understanding description on Wikipedia for detail understanding of this method: `https://en.wikipedia.org/wiki/Vienna_Development_Method`.

methodologies break down complex problems into manageable stages, ensuring a clear understanding of the system and its environment. They facilitate the decomposition of data and functions, use checklists for completeness, and maintain a balance between precision and intuitive understanding.

Different structured methods are suited for various domains: traditional data-processing methodologies such as SSADM[17] and LBMS[18] focus on transaction-based systems, whereas process-control and real-time applications benefit from methods like MASCOT,[19] JSD,[20] and real-time Yourdon. Many structured methodologies incorporate graphical or semi-formal notations, improving visibility and reducing the likelihood of misinterpretation. Some, like JSD and SDL, partially integrate mathematical formalism, enhancing their reliability and enabling automated processing.

By providing a clear, logical framework, structured methodologies help developers systematically analyze, specify, and verify system designs, making them particularly valuable in complex or safety-critical applications.

2.5.4 Tools Classification

The norm *EN 50128*[21] defines three basic categories of tools used for safety software development.

- **T1**: This class includes tools that do not produce any output that could directly or indirectly contribute to the resulting executable software code. An example of a tool is a text editor that has no ability to generate code.
- **T2**: This class includes tools that support testing or verification of the design or executable code. Errors in a tool may occur in certain situations, but cannot directly cause errors in the resulting executable software. Examples of such tools include static analysis tools or tools for measuring test coverage.
- **T3**: This class includes tools that generate outputs that directly or indirectly contribute to the final executable code of the resulting system. These include, for example, compilers.

It is important to note that depending on the class of a given tool, requirements are made on them, which are checked during validation and verification. Therefore, it is important to keep them in mind when designing a system. Often it may be more appropriate for a project to use an older but validated tool for which its behavior has already been proven and de-validated.

[17] **Structured systems analysis and design method (SSADM)** - waterfall method for the analysis and design of information systems.

[18] **Location-based method (LBMS)** - A scheduling approach that optimizes construction workflow by tracking work crews' movement across locations, ensuring continuity and efficient resource allocation. It extends traditional project management by structuring work packages into interconnected entities, improving scheduling flexibility and project efficiency [31].

[19] **Modular Approach to Software Construction Operation and Test (MASCOT)** is a methodology for designing, constructing, and testing software with a strong emphasis on modularity, concurrency, and data flow representation. It ensures component decoupling, facilitating reusability and simplifying testing, making it particularly suitable for complex real-time embedded systems [32].

The interesting fact is that the UK Ministry of Defense has been the primary user of the MASCOT method through its application in significant military systems, and at one stage mandated its use for new operational systems. Examples include the Rapier missile system, and various Royal Navy Command & Control Systems.

[20] **Jackson System Development (JSD)** is a linear software development methodology developed by Michael A. Jackson and John Cameron in the 1980s [33].

[21] Specifically, clauses 3.1.42-44 in the aforementioned EN 50128.

Technique/Measure	SIL 1	SIL 2	SIL 3	SIL 4
Coding Standard	HR	HR	M	M
Coding Style Guide	HR	HR	HR	HR
No Dynamic Objects	R	R	HR	HR
No Dynamic Variables	R	R	HR	HR
Limited Use of Pointers	R	R	R	R
Limited Use of Recursion	R	R	HR	HR
No Unconditional Jumps	HR	HR	HR	HR
Limited Size and Complexity of Functions, Subroutines, and Methods	HR	HR	HR	HR
Entry/Exit Point strategy for Functions Subroutines, and Methods	HR	HR	HR	HR
Limited Number of subroutine parameters	R	R	R	R
Limited Use of Global Variables	HR	HR	M	M

Table 2.6. Coding standards for different SIL levels according to *EN 50128*, Annex A, Table A.12 [19].

2.5.5 Programming Technique

The standards also define the programming standards for the individual SIL levels. The individual techniques are shown in table 2.6.

Compared to other types of software, the development of the safe one differs mainly in techniques such **not recommending the use of dynamic memory allocation**, both for objects and variables. This is due to the greater possibility of failure, as the software allocates memory at runtime, and this can lead to failures. The standard recommends **limited use of recursion and pointers**. These are methods that can potentially mishandle memory - either filling it up quickly or being ill-defined. Therefore, unless necessary, it is not recommended to use them.

Specific techniques for safe development are also **limited size and complexity of functions, subroutines, and methods** and **input or output strategies for functions, subroutines, and methods**.

The other techniques are not, in my view, specific to safety development, but are principles that are appropriate for any software development, such as following the **coding standard**, the **coding style guide** that leads to improved readability, or the **limited use of global variables**.

The norm *EN 50128* defines a set of essential rules that coding standards should include. The most important ones are:

- **Language specification** defines the programming language and its allowed features, ensuring **portability** across different platforms.
- **Scope and base standard when available** clarifies the coding standard's applicability and references any underlying standards, while **the procedure for changing the coding standard** describes how updates and modifications should be handled.
- **Analysis of the potential faults and recommended treatment** identifies possible issues and provides solutions, complemented by **restrictions to avoid the faults**, which list constraints to prevent common errors.

It may also be interesting to note that the standard defines **recommended programming languages** for different SIL levels.[22] It will come as no surprise that C, C++ or Assembler are among the recommended languages. However, the presence of languages such as C# or Java may also come as a surprise. The standard also mentions more historical languages such as Pascal.

2.6 Level Crossing System in CENELEC Countries

According to research,[23] the average number of level crossings per 100 km in the European Union in 2023 is over 50. The record holder is Norway, which has more than 90 level crossings, while the Czech Republic, along with Austria, Hungary, and the Netherlands, has more than 80.

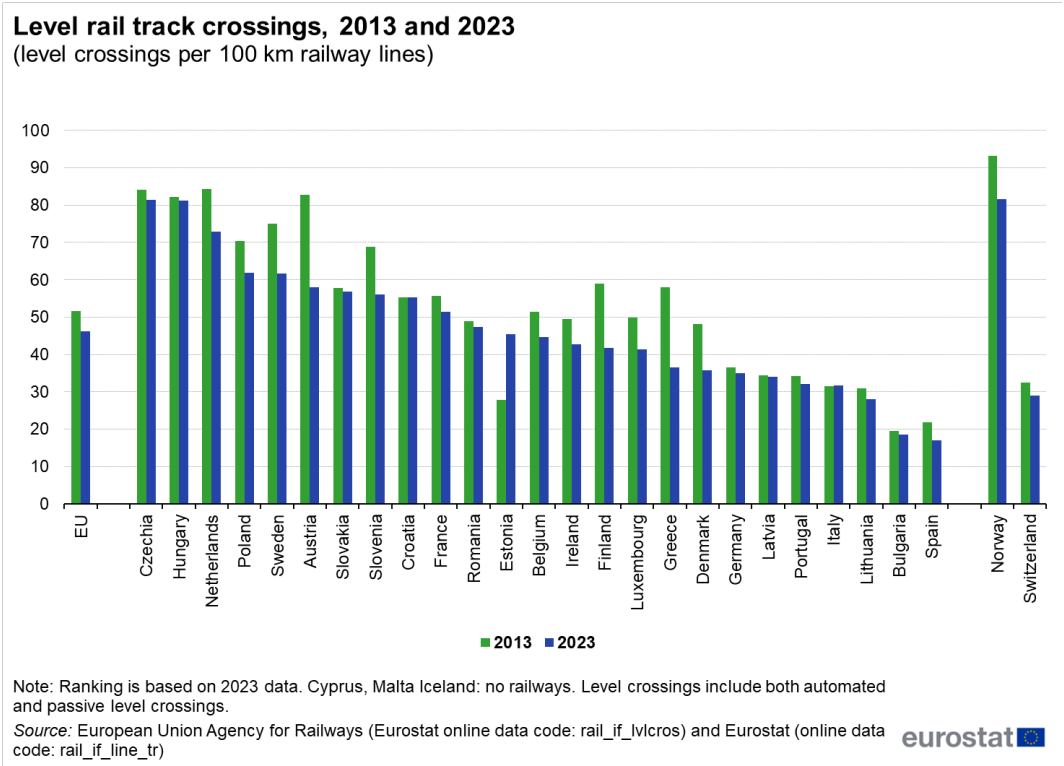

Figure 2.4. Level rail track crossings per 100 km by countries.

Each country uses a different signaling standard, mainly due to historical development. I am describing a few country systems with potential deployment of the developing device.

[22] The EN 50128 standard describes the different programming languages in Table A.15 in Annex A

[23] Eurostat, 2023 - `https://ec.europa.eu/eurostat/statistics-explained/images/6/65/F6_Level_rail_track_crossings%2C_2013_and_2023.png`

In Figures 2.5, 2.6, 2.7, and 2.8, the **passive status** indicates that no train is approaching or present on the crossing, meaning there are no immediate restrictions or hazards. The **active status** occurs when a train is in the immediate vicinity or on the crossing, signaling that entry is prohibited due to an ongoing or imminent train passage. Lastly, the **warning status**, used in Germany and some other countries, serves as a cautionary signal, indicating that the active status will soon follow.

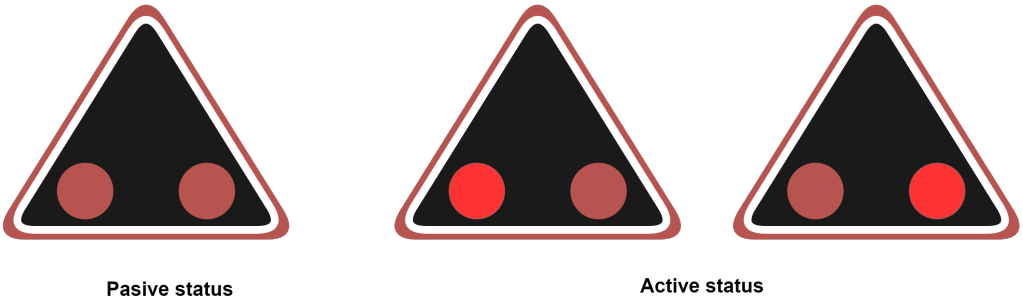

Figure 2.5. Level crossing signalization in Switzerland.

In **Switzerland**, the crossing lights remain completely off in the passive state. In the active state, the red lights flash alternately, as illustrated in Figure 2.5.

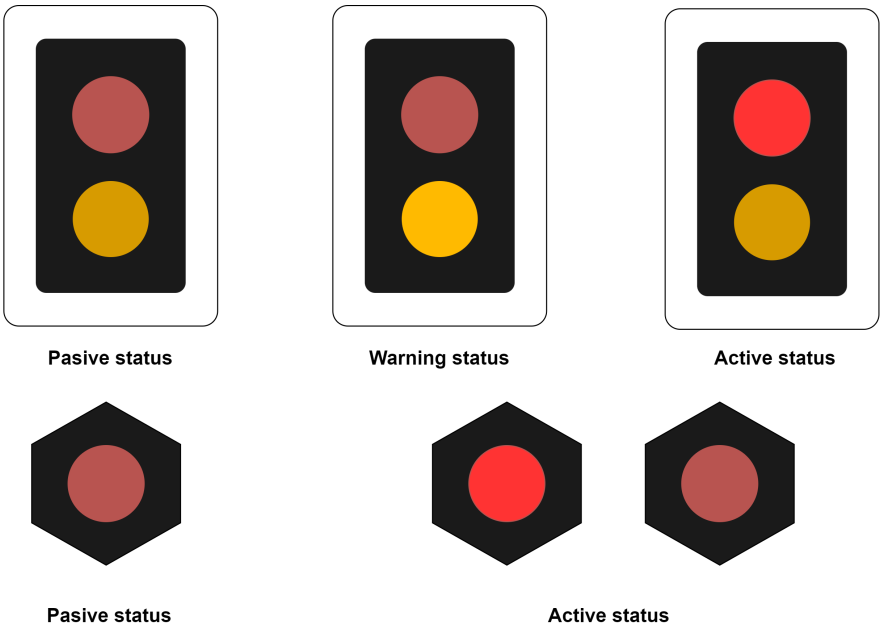

Figure 2.6. Level crossing signalization in Germany.

In **Germany**, multiple signaling types are used. The first type consists of a red and an orange light. In the passive state, both are off. When transitioning to the warning state, the orange light turns on while the red remains off. In the active state, the red light is on, and the orange light is off. The second type features only a single red light, which is off in the passive state and flashes at regular intervals in the active state. Both options are illustrated in Figure 2.6.

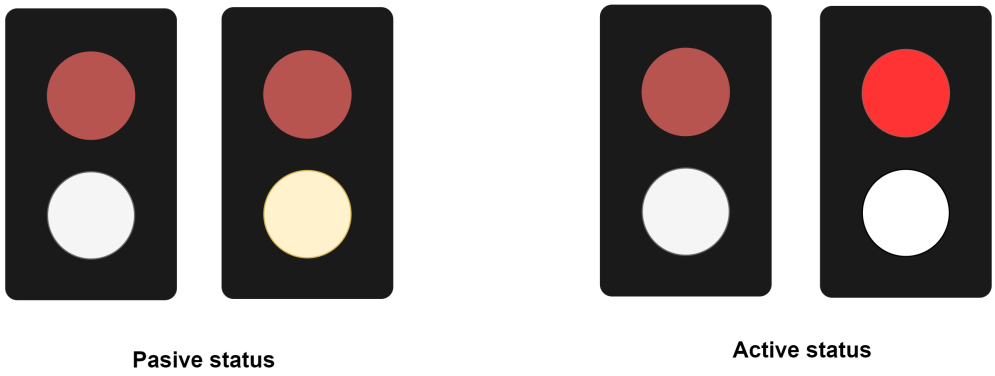

Figure 2.7. Level crossing signalization in Norway.

Norway employs a two-color signaling system similar to Germany's. The upper light is red, and the lower light is white. In the passive state, the red light is off, while the white light flashes at regular intervals. In the active state, the white light turns off, and the red light flashes at a slightly faster, consistent interval, as illustrated in Figure 2.7.

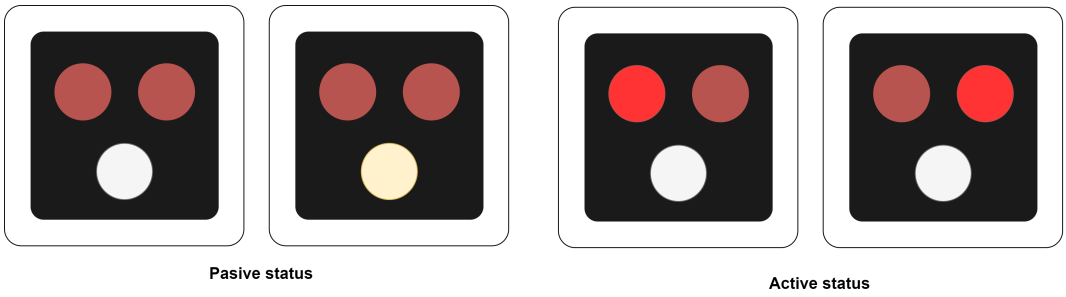

Figure 2.8. Level crossing signalization in Czech Republic.

In the **Czech Republic**, three lights are arranged in an inverted equilateral triangle, as shown in Figure 2.8. The upper two lights are red, and the lower one is white. In the passive state, the white light flashes at regular intervals while the red lights remain off. In the active state, the two red lights flash alternately, and the white light turns off.

Some countries not mentioned here use the same signaling systems. For example, Austria follows the same signaling as Germany, while Slovakia uses the same system as the Czech Republic.

Additionally, some level crossings are equipped with audible signals and barriers to prevent vehicles, pedestrians, and other road users from entering the railway. However, since these mechanisms are not addressed in the developed equipment, they are not covered further in this thesis.

Chapter 3
System Requirements

In this chapter, I would like to specify the system, functional requirements, and interfaces that I want to design and implement. Finally, I will discuss the hazards that need to be addressed or questions that need to be answered when designing the system architecture.

3.1 System Context

The proposed equipment is part of a level crossing safety system that controls traffic lights and signals. Previously, the system relied on a *Simatic PLC*[1] with digital inputs and outputs. However, it lacked the ability to analyze system functionality; issues could only be diagnosed manually using a multimeter. This becomes particularly problematic when the equipment is located in inaccessible areas or exposed to harsh conditions, such as freezing temperatures in Norway.

The goal of the new system version is to integrate a component that enables such analysis, which is precisely the purpose of this device.

The system is designed for use in a country that follows CENELEC standards. The device itself is a road signal monitoring module that controls LED modules on road and train signals based on commands from the *Simatic PLC*. In addition to managing the signals, it provides diagnostic data about both its own operation and the LED modules, helping to detect faulty components. The device can also transmit monitoring data to a higher-level server.

The term system and device will always mean a component of the overall safety system. If this is not the case, this will always be explicitly mentioned.

3.2 System Specification

My implemented and proposed project serves as a prototype for the final device. Unlike the final version, it does not need to communicate with the PLC.[2] It only needs to read information from LED modules and transmit the data further.

In the following diagrams and discussions, I will include all device components, not only relevant for the prototype device. This is necessary because their presence must be considered when designing the overall architecture, selecting hardware, and progressing through other development phases.

The device's safety functions must comply with SIL2, while the final system as a whole is designed to meet SIL4. However, since this prototype is purely a monitoring device and does not introduce any hazards, SIL2 is sufficient.

[1] `https://www.siemens.com/cz/cs/products/automation/systems/industrial/plc.html`

[2] The feature can be using a *daisy-chain* wiring scheme, where multiple devices are connected in sequence, allowing communication with the *Simatic PLC* through a single connection. This approach simplifies wiring, reduces cabling complexity, and enables efficient signal distribution across multiple devices.

3.2.1 Interface Specification

Three interfaces are provided by the device shell:

- **analog interfaces** to LED, that support relevant road signal modules,[3]
- interface for **communication with Simatic PLC**[4]
- and **two Ethernet ports** supporting 100Base-Tx, that shall act as a switch to support daisy-chain connection of multiple devices.

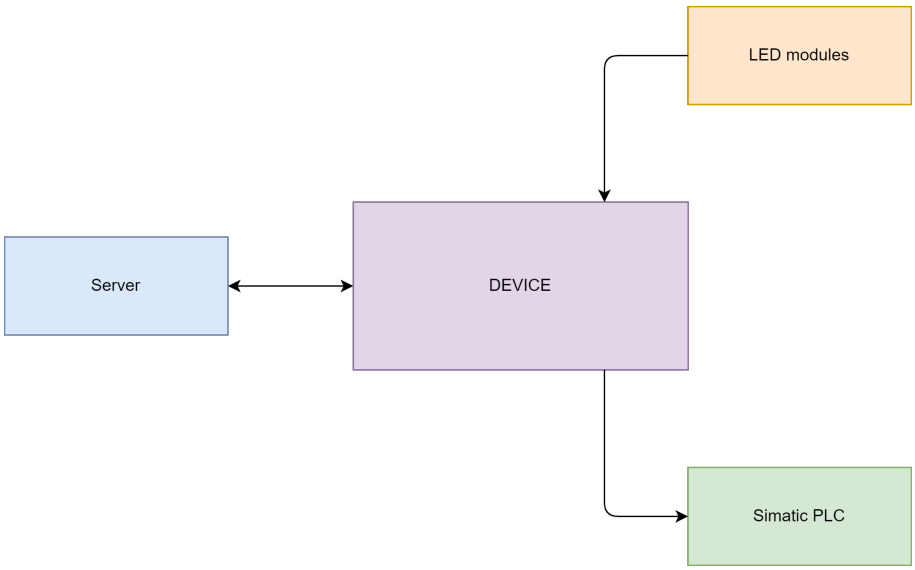

Figure 3.1. The schematic describes a system with all components.

3.2.2 Function Requirements

As previously mentioned, since this is solely a monitoring component, compliance with SIL2 is sufficient, even though the entire system is rated SIL4.[5]

The system shall be capable of controlling multiple LED modules installed on road signals, organized into multiple groups. The reason for multiple groups is that a railway crossing is typically illuminated by several railroad crossing signals. Each group should have common control and status signals.

To ensure proper signal operation, the system will evaluate LED functionality by measuring current. It must support three operational states: power on, static illumination, and flashing at frequencies specified by railroad regulations.[6]

Additionally, the system shall support *emergency signaling* in the event of a *Simatic PLC* shutdown.

The system shall be compatible with both *high-side* and *low-side switching* of road signals, accommodating different wiring requirements depending on national regulations. It should also support various hardware configurations for different signaling states, as described in Chapter 2.6. These configurations include:

- two alternating red lights,
- two alternating red lights with white light,
- flashing red light with flashing white light,
- static red light with static yellow light.

[3] Relevant for this project is modules *Silux/Yulux 2.40RS* and *Silux/Yulux 1.40*.
[4] `https://www.siemens.com/cz/cs/products/automation/systems/industrial/plc.html`
[5] Safety integrity level 4, detailed describes Chapter 2.4.1
[6] 60 flashes per minute and 90 flashes per minute, both with a 1:1 duty cycle.

Monitoring will be provided via a simple *REST API*.

Within the overall system, my device takes on the role of monitoring. In terms of SIL functions, which ensure the safety and reliability of critical components operating the level crossing actuators—the device shall:

- Read diagnostics from the current measurement components.
- Receive signals from the *Simatic PLC* indicating the intended signal state.
- Transmit data outside the SIL environment using the *REST API*.
- Support additional monitoring beyond SIL functions, enabling advanced analysis in an easily accessible format.

The Figure 3.1 illustrates a monitoring system with all components.

3.2.3 Human Interfaces

The system shall be designed to include an **LED interface** that allows maintenance personnel to quickly and easily identify potential issues or failures. The *LED indicators* must be clearly visible even in daylight and should accurately indicate the faulty component.

The LED interface should provide information on the following conditions: - Overall system status (OK / Failed) - Communication status with the diagnostics module (my device) - Status of connected LED modules (OK / Failed) - Status of individual inputs (Active / Inactive) – this interface is optional

Additionally, it is crucial for my device to indicate its own status using an *LED indicator*, which can be integrated into the system's human-reporting interface.

3.2.4 Safety

The system should meet SIL4 for isolation between two independent LED chain channels. According to CENELEC standards, this can be achieved with two independent LEDs connected to one main unit.

The system control and input channels shall be implemented as primarily independent of each other, in accordance with norm *EN 50129*. The system shall fulfill SIL2 for control LED modules.

The generic device shall fulfill SIL2 for the evaluation of its state. The Ethernet communication and diagnostics LEDs shall be developed as non-SIL functions.

3.2.5 Diagnostics

The generic device shall provide diagnostic data supporting failure detection, such as input voltage and current through each LED module. Optionally, the generic device shall also provide information on internal signals that could assist with failure detection.[7]

The generic device shall provide an interface for the measurement of the supply voltage of road signals (e.g., DC power supply rail).

The generic device shall provide its internal temperature (diagnostic feature only).

3.2.6 Project Limitation

As outlined earlier, I define the final target device within the device requirements. However, the scope of this thesis focuses solely on developing a prototype. This prototype does not implement all functions, such as communication with the Simatic PLC, daisy-chained wiring, or firmware updates via Ethernet.

[7] Note: For failure detection, it is sufficient to identify the faulty component without determining the precise cause. Indication of a faulty generic device itself, a broken LED module, or a missing power supply is adequate.

Nevertheless, understanding the full context is essential for a comprehensive overview. Therefore, the system requirements are defined for the entire device, not just the prototype.

3.3 Hazard Descriptions

The railway crossing signal unit provides indications to road traffic participants, informing them whether it is safe to cross the railway. In the following discussion, the term *signalization* refers primarily to the signals intended for road traffic participants, not for trains.

The system can operate in one of the following four states:

- The **system is operational**, and the signalization functions correctly.
- The system is faulty, and the **signalization is non-functional**.
- The system is faulty, and the signalization continuously indicates that the railway crossing **cannot be crossed**.
- The system is faulty, and the signalization continuously indicates that the railway crossing **can be crossed**.

3.3.1 State Analysis

In the **first state**, the system operates in its normal and desired condition. This is the ideal state, and the objective is to maximize the time the system remains in this state.

The **second state** presents a problem but does not constitute a hazard. When the signalization is non-functional, the responsibility shifts to the users of the crossing, i.e., those attempting to cross the railway. The crossing does not provide incorrect signals; it provides no signals at all. This situation impacts system availability, and thus, efforts should be made to minimize the occurrence of this state.

The **third state** is problematic and represents a partial hazard. If the signalization continuously indicates that the crossing cannot be traversed, users may choose to cross despite the signal, acting contrary to the indication. This behavior can lead to potential hazards, as the system appears unreliable, eroding trust among users. A solution, such as using an alternative crossing, may not always be feasible, particularly in mountainous regions like Switzerland,[8] where railway crossings are often widely spaced.

The **fourth state** is the most critical, as it constitutes a significant hazard. The signalization continuously indicates that the crossing is safe, even when it may lead to a collision with an approaching train. This state must be rigorously prevented and mitigated to ensure it does not occur.

[8] Based on Figure 2.4, a simple calculation indicates that railway crossings in Switzerland are approximately 3 km apart. However, it is necessary to consider that the concentration of crossings is higher in populated areas. Consequently, the average distance may increase to several kilometers, particularly in mountainous regions such as Switzerland, where this distance can be significant.

Chapter 4
Hardware

In this chapter, I would like to specify the hardware requirements, design, and describe the selected microcontroller development kit.

4.1 Hardware Requirement

The proposed device must support communication through the following interfaces:[1]

- An analog interface for reading LED module values,
- I/O communication with the *Simatic PLC* (though out of scope for this work, the necessary pins must be included),
- Two Ethernet ports supporting 100Base-TX, functioning as a switch to enable daisy-chain wiring.
- Additionally, the device should manage indication LEDs as a user interface and support the connection of non-SIL peripherals for analysis.

The device shall also comply with the SIL2 standard.

4.2 Microcontroller Options

Based on these options, I looked for different microcontrollers on which to build the device. I shortlisted 5 devices that could be used. I compare four of them in Table 4.1.

Two of them meet the requirement for an internal ADC converter, which in the end could not be used anyway, because ADC converters must be able to measure voltages up to 36 V. The second reason for using an external ADC is the necessity to meet safety requirements defined in 3.2.4, which mandate that the device must employ two independent channels for the LED modules connected to the main control unit. So you will need to use external converters that can be communicated with via SPI, I2C or other peripherals. Therefore, I discarded the *STM32MP1* and the *NXP i.MX RT1060* devices whose great advantage was this functionality.

Microchip SAM E70 microcontroller has only one Ethernet port; it would be necessary to connect an external switch, so I rejected it.

As a result, I was deciding between devices from Texas Instrument's Sitara series, specifically the *AM335x* and *AM243x*. However, the devices from the *AM335x* series are built on the Arm Cortex-A8 core, which is unnecessarily powerful for my purposes and is more suited for handling 3D graphics, graphics accelerators, running higher-end operating systems like Android or Linux, etc [38]. That's why in the end I opted for the AM243x processor from Texas Instruments. For development, I will use the **Evaluation Board LaunchPad AM-243**.

[1] By requirements from the Chap 3.2.

Feature	TI Sitara AM335x	STM32MP1
Analog Inputs	External ADC (SPI/I2C)	Integrated ADC
Serial Communication	UART (RS232/RS485)	UART (RS232/RS485)
Ethernet Ports	2x 100Base-Tx (Switch)	2x 100Base-Tx (Switch)
Indicator LEDs	GPIO Support	GPIO Support
Non-SIL Peripherals	SPI, I2C, USB	SPI, I2C, USB, SDIO
Feature	**NXP i.MX RT1060**	**Microchip SAM E70**
Analog Inputs	Integrated ADC	Integrated 12-bit ADC
Serial Communication	UART (RS232/RS485)	UART (RS232/RS485)
Ethernet Ports	2x 100Base-Tx (Switch)	1x 100Base-Tx (Ext.Switch)
Indicator LEDs	GPIO Support	GPIO Support
Non-SIL Peripherals	SPI, I2C, USB, CAN	SPI, I2C, USB, CAN

Table 4.1. Comparison of suitable Microcontrollers [34–37]

4.3 Microcontroller Description

The **Evaluation Board LaunchPad AM-243x** (LP-AM243x)[2] is equipped with the AM2434 ALX MCU, which is composed of 4x Arm Cortex-R5 and 1x Arm Cortex-M4 cores. The Evaluation Board has a large number of interfaces including Industrial Ethernet, Fast Serial Interface, CAN transceiver, and 512MB QSPI flash memory. For debugging, the Evaluation Board is equipped with the On-Board debugger XDS110 [39].

The device is organized into two domains: MAIN and MCU.

4.3.1 The MAIN Domain

The **MAIN domain** is made up of Arm Cortex-R5, which are 4 on the development kit. Actually, it is 2x two real-time cores because it is a Dual-Core Cortex-R5F subsystem. The architecture is based on the ARMv7-R instruction set and supports two configurations at boot time:

- **Dual core mode**: means two independent free-operating cores (Asymmetric Multi-Processing, no coherence).
- **Single core mode**: one free-operating core and one non-operating core.

The cores also allow operating in lockstep mode. **Lock-step mode** refers to a fault-tolerant operational configuration in which a redundant instance of the CPU logic (and

[2] This processor series shares a similar architecture, registers, and most features with the AM64x series. Consequently, the majority of documentation for these two processor series is identical. This can occasionally be confusing, primarily because the documentation and manuals focus on the AM64x version, which offers a broader range of resources, internal modules, and peripherals.

optionally the Accelerator Coherency Port - ACP) executes in parallel with the primary CPU. Both the primary and redundant logic are driven by identical inputs and share the same cache memory, eliminating the need for duplicate cache RAMs. The redundant logic operates synchronously with the functional CPU, replicating its behavior in real time. However, it does not influence the system's outputs or processor behavior directly. Its primary purpose is to enable the detection of faults by comparing the results of the redundant logic with those of the functional CPU [40]. However, I don't use it in my project, as I use the *isolated core* option for safe functions. By TI safety documentation, AM243x doesn't support full Lockstep mode[3] [41].

The MAIN domain also integrates several subsystems that serve to operate real-time systems, meet industry standards for communication, memory access or inter-core communication. Specifically, these are:

- **Programmable Real-Time Unit** and **Industrial Communication Subsystem** (PRU-ICSSG), which in addition to real-time support for cores, provides support for industrial standards such as EtherCAT, PROFINET, EtherNet/IP, PROFIBUS, Ethernet Powerlink and SERCOS.
- 16-bit DDR memory subsystem (**DDR16**, also referred to as DDRSS) for SDRAM support.
- **Region-based Address Translation Module** (RAT), which handles the translation of a 32-bit address into a 36-bit output address.
- **Data Movement Subsystem** (DMSS) for efficient data movement between software, firmware and hardware in all combinations.
- **Mailbox** (MAILBOX) system for inter-core communication.
- Other standard subsystems to support microcontroller and microcomputer functions such as: **Spinlock**, **ADC**, **GPIO**, **I2C**, **SPI**, **UART**, **CPSW3G** (3-port Gigabit Ethernet Switch), **PCIE**, **Serializer/Deserializer** (SERDES), **USBSS** (Universal Serial Bus 3.1 Subsystem), **GPMC** (General Purpose Memory Controller), **ELM** (Error Location Module), **FSS with OSPI** (Flash Subsystem with Octal Serial Peripheral Interface), **MMCSD** (Multi-Media Card/Secure Digital Interface), **ECAP** (Enhanced Capture Module), **EPWM** (Enhanced Pulse-Width Modulation Module), **EQEP** (Enhanced Quadrature Encoder Pulse Module), **MCAN** (Controller Area Network) to support CAN, **FSI-RX and FSI-TX** (Fast Serial Interface Receiver and Transmitter) and Timers.
- **Internal Diagnostics Modules** that provide monitoring and diagnostic functions required to achieve certain safety compliance levels.

4.3.2 The MCU Domain

The **MCU domain** is based on the Arm Cortex-M4F, which is seeded once on the development kit. In its documentation, TI refers to the MCU domain as *M4FSS Island*, *MCU Island*, *MCU Channel*, or *MCU Subsystem*. I will stick to these terms.

The processor core can be configured as an isolated safety MCU or general-purpose MCU. The processor uses the ARMv7-M instruction set architecture, supports the Nested Vectored Interrupt Controller (NVIC) with 64 inputs, and has the ability to execute code from internal or external memories, etc.[4]

Similarly to the MAIN domain, the MCU domain consists of multiple subdomains in addition to the core, which support standard microcontroller functions in hardware.

[3] Details are described in the *Functional Safety for AM2x and Hercules Microcontrollers* on the official TI webpage `https://www.ti.com/product/AM2434`.

[4] More details are described in the device documentation `https://www.ti.com/lit/ug/spruim2h/spruim2h.pdf?ts=1745386413600`.

These include: **MCU-GPIO**, **MCU-I2C**, **MCU-SPI**, **MCU-UART** and **MCU Timers**. Like the MAIN domain, MCU island supports **MCU Internal Diagnostics Modules** which provide monitoring and diagnostic functions required to achieve certain safety compliance levels. The mechanisms are:

- One instance of the Dual Clock Comparator (MCU-DCC) module, which is used to determine the accuracy of the clock signal while the application is running.
- One instance of the Error Signaling Module (MCU-ESM) for safety-related events and errors aggregation from throughout the device into one location.
- Multiple ECC aggregator modules supporting ECC mechanisms for providing increased system reliability via reduction of memory software errors by allowing single-bit errors to be detected and corrected (SEC) and double-bit errors to be detected (DED).
- Memory Cyclic Redundancy Check module used to perform CRC to verify the integrity of a memory system.

A limitation of the chip placement on the development board used in this work is that not all pins of the isolated core are accessible. The isolated core exposes only four peripheral interfaces, to which the UART is connected; no other interfaces are available. This presents a minor complication for future development. A more detailed discussion of this issue is provided in the following chapter, in the context of designing the specific system architecture.

The MCU domain is designed to ensure **isolation from the broader SoC architecture**, incorporating Freedom From Interference (FFI) mechanisms to enhance system reliability and security. The key isolation features include:

- Independent interconnect architecture to segregate MCU communication pathways.
- Implementation of firewalls and timeout gaskets to enforce access control and prevent unauthorized interactions.
- Controlled reset isolation to enable independent reset management for the MCU domain.
- Dedicated Phase-Locked Loop (PLL) and Memory-Mapped Register (MMR) control for autonomous clock and configuration management.
- Separate I/O voltage supply rail to ensure electrical isolation and minimize interference from other SoC components.

4.4 ADC Module Description

For reading data from the LED modules, I selected an Analog Devices ADC module, specifically the **CN0254** evaluation board with the **AD7682** ADC. This cost-effective, highly integrated 16-bit, 250 kSPS[5] 8-channel data acquisition system can digitize industrial-level signals ranging from ±10 V [42]. The module supports communication via various peripherals, including I2C, SPI, or USB, when paired with an external converter from Analog Devices. For this application, I chose to use the SPI interface.

[5] kilosamples per second

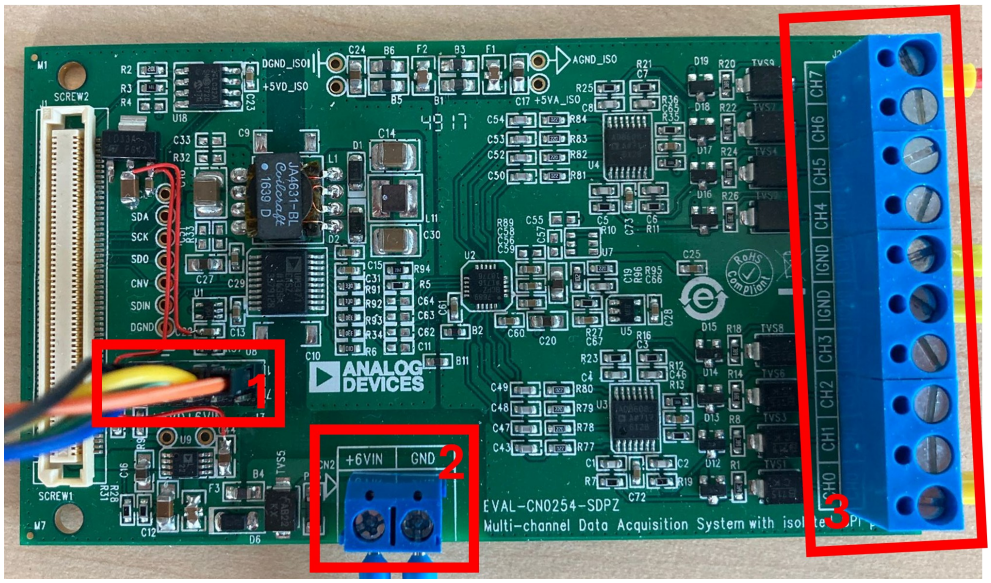

Figure 4.1. Photo illustrates the ADC Evaluation Board connected with a few modifications. 1) On-board SPI pins, 2) 6 V power supply, 3) ADC inputs.

The device requires a 6 V power supply and can measure input voltages in the range of 0 to 10 V.[6] To utilize the CN0254 evaluation board, two points on the printed circuit board need to be connected.[7] Further details on ADC communication via SPI are provided in Chapter 6.7.

4.5 Hardware design

The primary component of the entire system is the aforementioned Sitara AM243x development board. Connected via SPI, an external Analog-to-Digital Converter (ADC) device is responsible for reading values from the LED module. The processor communicates with a Simatic PLC through binary outputs. The device employs LED diodes to display its current status. Additionally, it communicates with a higher-level device, functioning as a gateway, via Ethernet. All system is illustrated in the figure 4.2.

[6] Operation of the final system requires hardware modifications, as the input signal must be in the range of 6 to 24 V.

[7] Specifically, two modifications were required to enable the use of the onboard SPI and power the ADC evaluation board from a 6 V supply. These involved adjusting the resistance and grounding. I consulted with experts on this matter and will not discuss these changes further, as I am not an expert in this field, and this thesis does not focus on the hardware aspects of the project.

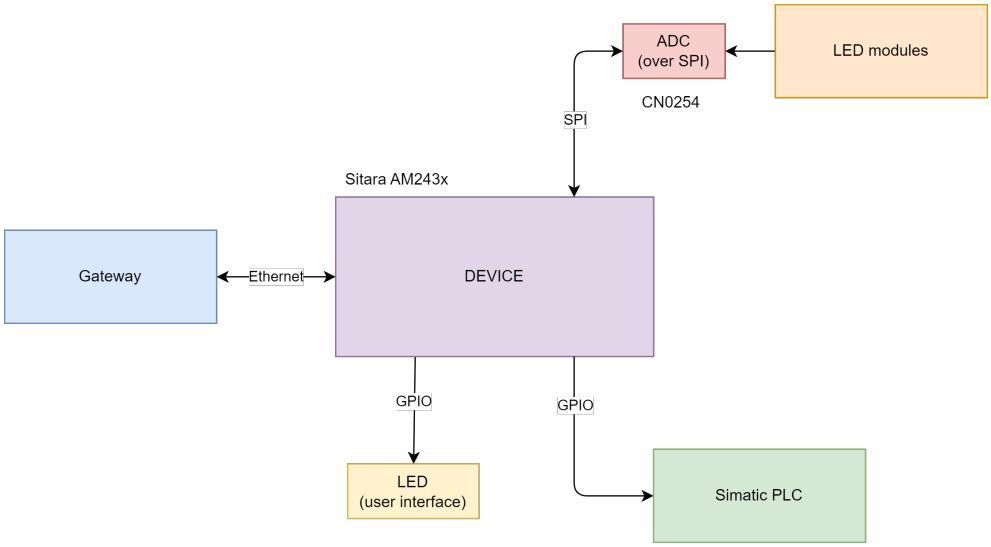

Figure 4.2. Hardware design with peripherals

Chapter 5
Device design and architecture

In this chapter, I propose the architecture, encompassing all mechanisms and processes intended for inclusion in the final device, as opposed to the prototype developed within the scope of this thesis.

5.1 System Requirements Analysis

The system is designed to meet both Safety Integrity Level (SIL) requirements and non-SIL requirements, ensuring robust functionality for safety-critical and supplementary tasks. For clarity in development and analysis, the device is divided into two primary components: the SIL component and the non-SIL component.

The SIL component, operating on the isolated M4 core, monitors the functionality of LED modules, communicates their status to the Simatic PLC, and shares this data with the non-SIL software. Due to the M4 core's isolation requirements, shared memory is not feasible; instead, communication is facilitated by hardware mechanisms provided by the AM243x system, such as inter-core messaging or dedicated registers.

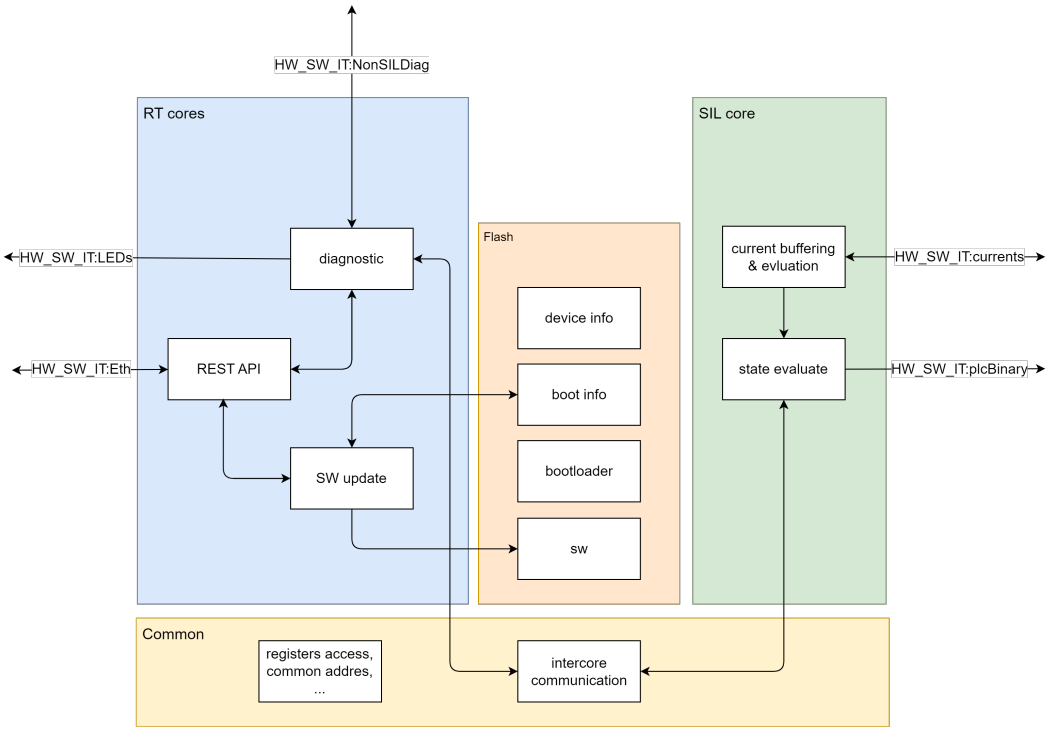

Figure 5.1. Diagram illustrating the software architecture overview.

The non-SIL component retrieves status data from the SIL component regarding the LED modules, ensuring reliable data access through AM243x inter-core communication mechanisms. It manages the bootloader and initialization of the SIL component and

hosts a REST API server to enable data monitoring and software updates. Additionally, it supports auxiliary non-SIL monitoring functions.

Figure 5.1 provides an overview of the software architecture, illustrating the relationships between the SIL and non-SIL components and specifying the interfaces used for communication with external systems. This diagram is relevant to the prototype's architecture.

To better understand, let's divide the software into four basic components:

- Booting
- Standard operation (Safety Software and Non-SIL software)
- Software update
- Safety shutdown

5.2 Booting

First, let's describe the boot process. The AM243x is a multi-core processor that utilizes a multistage booting sequence. By TI naming convention, the booting sequence is divided into *ROM boot* and *SBL boot*.

5.2.1 ROM Boot

The **ROM boot** (also RBL) is stored in read-only memory and is almost considered as part of the SoC. As soon as the board is powered ON, the ROM bootloader or RBL starts running. The RBL is the primary bootloader. Depending on which boot mode is selected, the RBL will load the secondary bootloader or SBL from a boot media. It is via UART in our case. The rest of the booting is done by the SBL [43].

The **ROM code** is code that is executed in this phase.

The ROM can operate in the three modes by the device type:

- **HS-Field Securable device** (HS-FS) - This is the HS device state before the customer keys are provisioned in the device (the state at which the HS device leaves the TI factory); secure features are not available and the device protects the ROM code, TI keys, and certain security peripherals; the device does not force auth for booting.
- **HS-Field Enforced device** (HS-SE) - This is the HS device state after the customer keys are successfully provisioned in the device; all security features are enabled, all secrets within the device are fully protected, all of the security goals are fully enforced, the debug override sequence is supported, and the device forces security booting.
- **General-purpose device** (GP) - This is the non-secure device state intended for general-purpose applications; no security features are enforced, secure booting is not required, and the device allows open access to all peripherals and debug features without authentication.

Final system is targeting the HS-SE device type. The thesis prototype device is targeting the GP device type only.

In a general-purpose device, DMSC ROM (Device Management System Controller ROM) runs on the Arm Cortex M3 and performs the following functions:

- Device management
- Configures the boot vectors and controls reset release of R5 core. That is, DMSC is the boot controller of the R5 core.
- IPC[1] configuration via Main DMSS rings and Secure Proxy

[1] Inter-Processor Communication

- PLL configuration (R5 and SA2UL[2])
- X509 certificate parsing
- SA2UL configuration to SHA512 for image integrity checks
- DMSC firmware loading [45]

5.2.2 SBL Boot

The **SBL boot** (Secondary Bootloader) typically does a bunch of SoC-specific initializations and proceeds to the application loading. Depending on the type of SBL loaded, the SBL looks for the multicore app image of the application binary at a specified location in a boot media.

5.2.3 Boot Modes

Device allows several boot modes divided into two classes:

- **host** boot modes (Ethernet, UART, USB, ...)
- and **memory** boot modes (MMCSD,[3] GPMC,[4] QSPI, ...).

Device provides **Primary Boot Mode** and **Secondary (Backup) Boot Mode**, that is started when Primary Boot Mode fails (it is illustrated in Figure 5.5). The mode is set up by BOOTMODE pins.[5] On the Devkit, a hardware mutex is placed to control BOOT pins (illustrated in Figure 5.2).

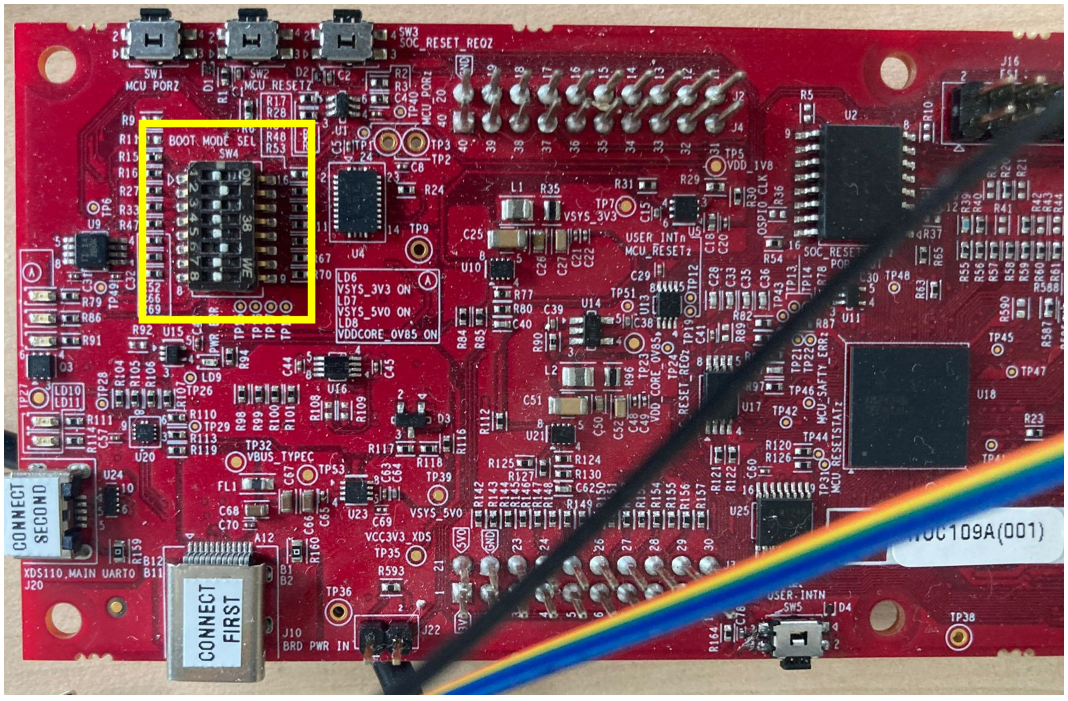

Figure 5.2. The Boot Pin Mutex on the Devkit board LP-AM243x.

[2] The SA2UL subsystem is designed to provide a generic cryptographic acceleration for different use cases such as secure boot, secure content, key exchange etc. [44].
[3] eMMC Flash or SD card
[4] NOR flash, NAND flash
[5] Detail description can be found in the *AM243x Technical Reference Manual* in the Chapter 4.3.1 [45].

5.2.4 Device Boot Phases

In my device, I divide the boot process into two phases. These phases are not the same as RBL and SBL. It is a different division. RBL and SBL are, to some extent, part of both phases.

The **first phase**, which I call the **bootloader**, is responsible for loading software from flash memory into individual CPUs, verifying the validity of certificates, performing basic initialization, and managing software versions.

During the **second phase**, the device is already running standard operations on each core of the device.

In addition to these two phases, I include a third phase called **flash writer**. This phase is tasked with downloading software during the initial loading into the device and is only executed when the device does not boot from the primary flash memory source but from a secondary (backup) Ethernet source. This part of the code is provided to the device before the actual loading from a server that uploads the software and is never stored in the device's flash memory.

Overall, the software comprises the following parts:

- **Flash writer**: a component that ensures the loading of software into flash memory.
- **Bootloader**: a component that is launched first after a successful system startup and manages other software parts, such as loading and validation.
- **Images for each core**: a component that contains running standard operations.

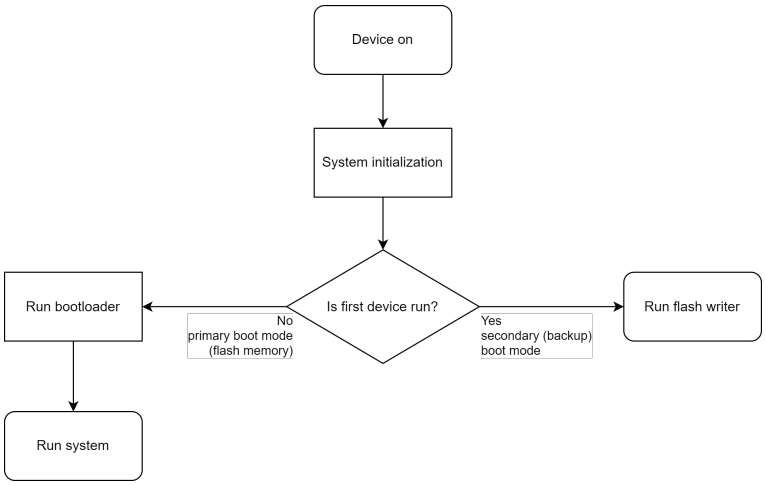

Figure 5.3. Overview of the boot process, illustrating the relationships and transitions between the bootloader, flash writer, and core operation phases.

Due to the multi-core architecture, it is necessary to use the DMSC (Device Management System Controller) in the device, which is managed by firmware referred to by TI as SYSFW (System Firmware). This layer functions as a *black box* and provides an API for Resource Management, Power Management, and Security. The DMSC is uploaded in the device as part of the SDK with device images.

5.2.4.1 Detailed SYSFW description

SYSFW comprises two primary components: TI Foundational Security (TIFS) and Device Manager (DM). TIFS provides security services, including authentication and decryption of binary data using root-of-trust keys, processor boot control, JTAG unlock, access to device-unique keys, and management of Customer One-Time Programmable

(OTP) eFuses. Additionally, TIFS configures device firewalls and Initiator-Side Security Control (ISC) to manage initiator credentials, controlling access to memories, peripherals, and other System-on-Chip (SoC) resources. These services support critical security use cases, such as authenticated processor boot, device configuration, trusted execution environments, and isolation [46].

The DM component delivers centralized Resource Management (RM) and Power Management (PM) services essential for device operation. PM services encompass configuration and control of module power states, clock states, frequency settings, multiplexer selections, and system resets. RM services manage the allocation and assignment of key SoC resources, including Direct Memory Access (DMA), Ring Accelerator, Interrupt Aggregator, Interrupt Router, and various Interrupt Router instances across the SoC.

SYSFW is distributed as part of TI's Processor SDKs. TIFS is provided as a binary-only image, which, on high-security devices, is signed and encrypted with TI's proprietary keys, rendering it closed to development and intended for integration as a black-box component. TIFS is loaded via the MAIN R5F bootloader during the default boot sequence in the SDK, with TI supplying standard board configuration entries to support SDK examples and use cases.

The DM is implemented through a set of libraries and a reference TI System Control Interface (TISCI) server running on the DMSC core, provided via the SCIClient component in the Processor SDK RTOS. TI SDKs include a prebuilt RTOS-based DM reference implementation, loaded through standard device boot procedures.

Neither TIFS nor DM (including RM and PM services) is safety-certified, as they are developed according to TI's baseline quality process rather than functional safety standards.

The DMSC-Lite, an ARM Cortex-M3-based subsystem, is the first subsystem activated after a power-on reset in the AM243x device. It serves as the central hub for both device management and security control, orchestrating the initial boot sequence, resource allocation, and security configurations. The DMSC-Lite executes the SYSFW, ensuring seamless integration of TIFS and DM services to support the multi-core architecture's operational and security requirements [46].

5.2.5 Boot Image Format

Each block consists of a certificate and the software itself and is loaded into flash memory at a specific address (illustrated in the Figure 5.4). The device uses X.509 certificates described in the norm RFC 5280[6].

In general, an X.509 certificate contains a public key which has been signed by a private key. The public ROM code does not directly use the keys. In non-secure devices (GP), the public key value is in general a don't care condition. The exception is certificates containing a degenerate RSA public key. GP devices with a degenerate RSA key allow for integrity checking of most (but not all) of the certificate. As I mentioned below, we are using the device in the GP mode. It means that our device integrity in the prototype is validated by the degenerate RSA public key [45].

The ROM code is getting two additional pieces of information from the X.509 certificate:

- The total size of the X.509 certificate
- The total size of the boot image

[6] `https://datatracker.ietf.org/doc/html/rfc5280`

The ROM defines several extensions that are used only by TI for boot. These are placed in the extensions field of the TBS[7] certificate.

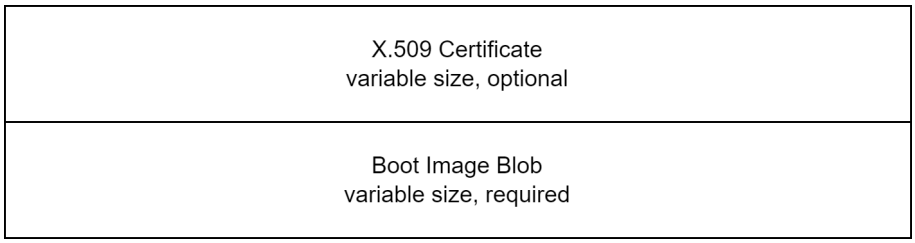

Figure 5.4. The Boot Image format.

5.2.6 Detailed System Initialization Description

During the DMSC initialization phase, as depicted in Figure 5.6, the R5 core reads the boot mode pins to configure the appropriate peripheral interface, enabling access to the boot image. The R5 performs a preliminary validation of the image before transferring it to the DMSC. The DMSC ROM then verifies the code and loads the validated boot image into on-chip RAM. If the image verification fails but the boot mode pins indicate a secondary (backup) boot mode, the DMSC ROM loads the backup image into on-chip RAM. This functionality is utilized by our system to initially boot software using the flash writer component. After receiving the image, the R5 enters a clean state and idles. The DMSC ROM asserts a reset on the MCU, redirects the boot vector to the newly loaded image, and releases the reset. This process restarts the R5 with the Public ROM code fully disconnected, as illustrated in Figure 5.5.

The boot sequence then proceeds based on the boot mode: for the primary mode, the system boots from flash memory and executes the bootloader; for the secondary mode, the system boots from a backup source, activating the flash writer component to download the initial software to flash memory.

[7] to be signed

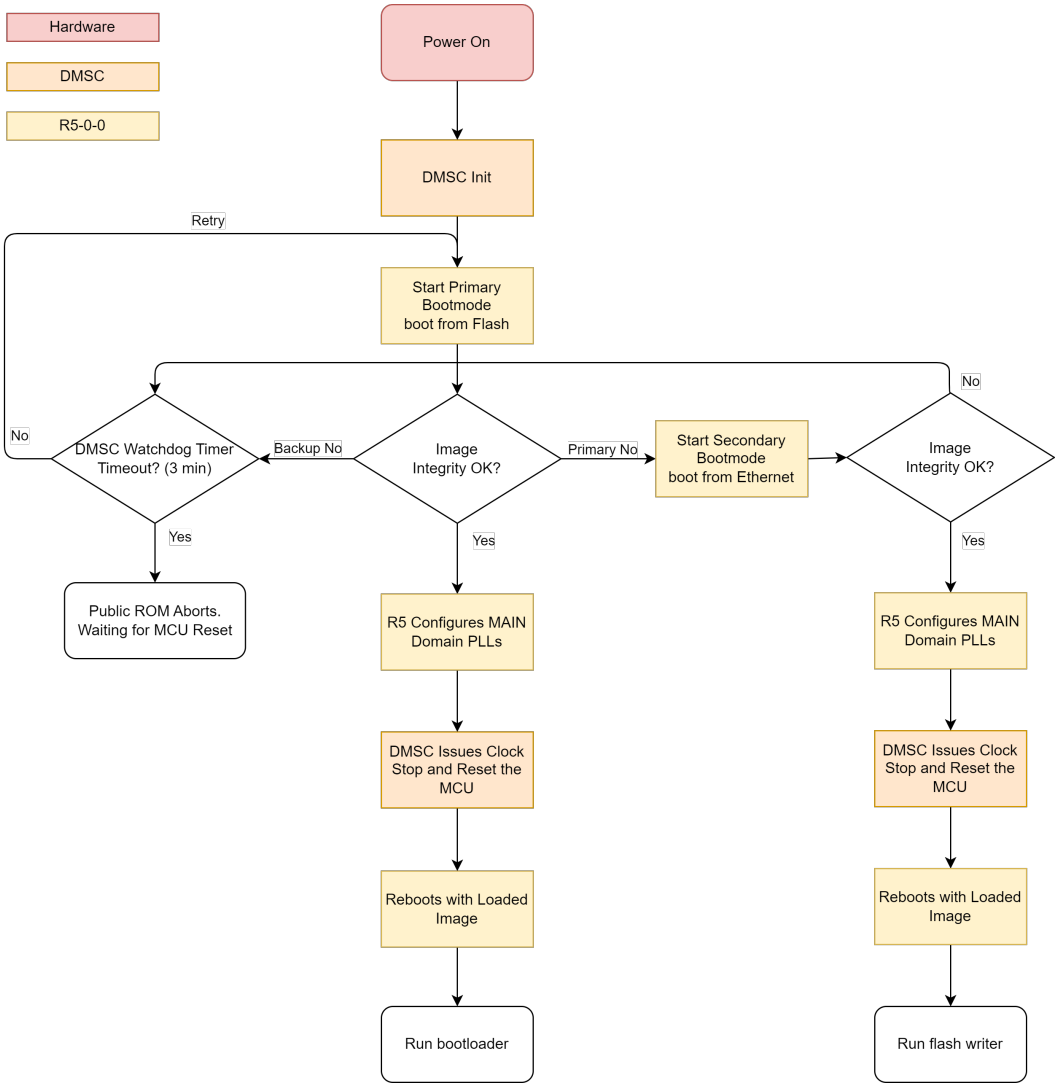

Figure 5.5. Diagram illustrating the complete system initialization process, detailing the sequence of boot image loading, verification, and core reset for R5 startup.

Note: The DMSC ROM configures a 3-minute watchdog timer (MCU_RTI0) timeout. The MCU boot must complete within this period; otherwise, a watchdog timer reset occurs. Once the R5 image (SBL) is loaded, the DMSC ROM restarts the watchdog timer for an additional 3 minutes upon entering the R5 SBL. The customer-provided MCU image must load and install the TI-provided SYSFW image into the DMSC, which manages the watchdog timer during runtime [45].

5.2.6.1 The DMSC Initialization

The DMSC serves as the boot controller for the Public ROM, managing essential configurations such as firewalls, clocks, PLLs, and inter-core communication modes.[8] As shown in Figure 5.6, the DMSC releases the reset for R5 CPU0.[9] Its configuration depends on the specified boot mode, with host boot mode imposing different requirements compared to memory boot mode.

[8] The device supports two primary inter-core topologies: Proxy and Ring.
[9] R5FSS_0-0

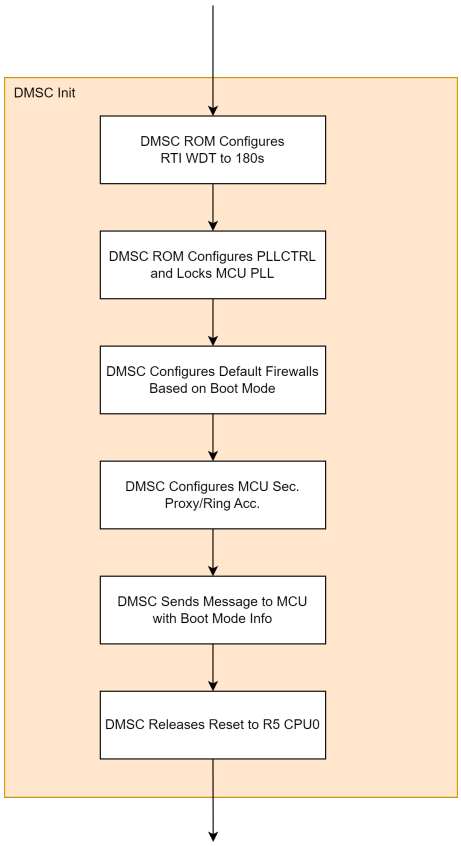

Figure 5.6. Diagram detailing the DMSC initialization process, a critical subset of system initialization, highlighting configurations for boot mode, peripherals, and R5 CPU0 reset, as part of the overall boot sequence in Figure 5.5.

5.2.7 Bootloader Function Description

The bootloader is a critical component of the system. Its primary tasks are:

- Loading software into the RAM of individual cores and verifying the authenticity of each core's image using an X.509 certificate.
- Performing basic system configuration required for the subsequent program execution, such as setting up PLLs, firewall rules, communication topology, and other parameters. These configurations are similar to those managed by the DMSC controller, as illustrated in the Figure 5.6. Additionally, the bootloader must establish an isolation layer to enable the M4 core to operate in isolated mode. The precise clock and PLL configuration is later finalized by the M4 core upon its startup.

The bootloader always runs on the real-time core *R5_0-0*. It is activated after the DMSC transfers control to it. The bootloader configures essential system resources and then reads information stored in flash memory. Based on this information, it loads either the original or new software into the RAM of all processors, verifying its authenticity using an X.509 certificate. If all images are valid, the system reboots and initiates standard operation. If the original software fails, the bootloader restarts and attempts to boot again. If authentication of new images fails, the system saves an error code to the failure info variable in the boot info structure in the flash, and the system transitions to a process defined within the software update sequence, as described in chapter 5.3.

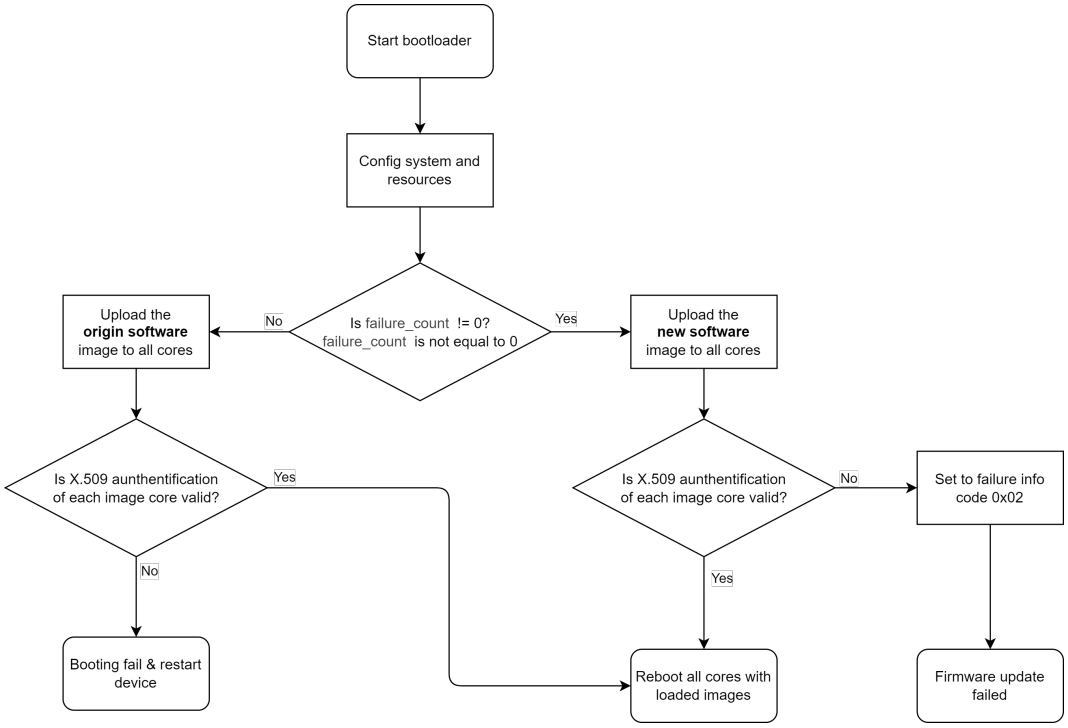

Figure 5.7. The diagram illustrates the bootloader process.

5.2.8 Flash Memory

Flash memory is used to store software for individual cores and information that must be retained in the device even after a power outage. It consists of four main components:

- **Bootloader Image**: Contains the software for running the bootloader. The image comprises an X.509 certificate and a compiled binary code file, as illustrated in Figure 5.4.
- **Device Info**: Stores static information such as the serial number, MAC address, certificates, and other device-specific data.
- **Boot Info**: A simple structure containing three values: *failure count*, *failure info*, and the addresses of the original and new software. Access to this memory section must always use pseudo-atomic operations to ensure error-free updates, particularly during software updates. (The pseudo-atomic operation mechanism is described in Chapter 5.2.9.) The *failure count* indicates whether the new software launched successfully and *failure info* variable contains the error code, with further details provided in the software update section 5.3.
- **Original Software**: A memory region for the functional, already in-use software for all cores.
- **New Software**: A memory region for new, unverified software for all cores.

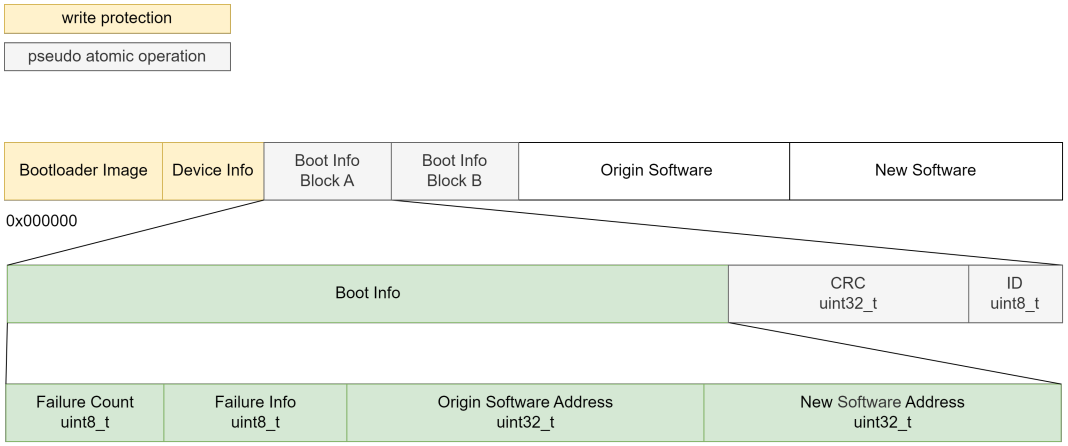

Figure 5.8. The illustration of flash memory map structure.

The Bootloader Image and Device Info sections are written only once during the initial software loading via the **flash writer** component. Afterward, these flash memory regions are locked, allowing read-only access and preventing overwrites.

5.2.9 Pseudo-atomic Flash Update Operations

Pseudo-atomic operations are necessary primarily to prevent a state where memory is only partially modified, which could occur due to events such as power failures or high-priority interrupts. To mitigate this, a modified **bitwise memcpy** operation using the Compare-and-Swap instruction, which verifies update consistency, would ideally be employed [47]. However, this approach is not feasible, as the system operates over external flash memory rather than stack memory. Flash memory updates rely on paging, where an entire page is rewritten for any change. While this might seem to satisfy the requirement for single-operation updates, the challenge lies in the fact that updates are managed by an external controller via Quad Serial Peripheral Interface (QSPI) flash. Consequently, an alternative solution is required.

The proposed solution employs **encapsulation** to ensure data integrity, guaranteeing that the most recent valid data is used during reads. It duplicates the boot information block, storing each in a distinct memory block, referred to here as Block A and Block B. Each block comprises a data section (as described previously) and a header containing a Cyclic Redundancy Check (CRC) and an identifier (ID). The CRC verifies data integrity, while the ID indicates which block is newer.

The **read operation** involves loading both blocks, with the block having a valid CRC and the higher ID designated as the valid data.

The **write operation** targets the memory block without a valid CRC or, if both blocks have valid CRCs, the block with the lower ID. The ID is cyclic, and to prevent overflow, the value 0xFF is treated as lower than 0x00. After writing, a read operation is performed to verify that the data was written correctly. If verification fails, the write operation is repeated.

5.2.10 Flash Writer Description

The **flash writer** is a specialized component integral to the software update process, yet designed to operate independently as it is also used during the initial software loading of the device. The term *flash writer* refers to a standalone module responsible for the first-time provisioning of the device. When the device attempts to load data from flash memory and finds it empty—typically during factory settings, as no data has been

written yet—it activates the secondary (backup) boot mode, where the flash writer plays a critical role. The primary objectives of the flash writer are:

- To receive software images and store them at designated locations in flash memory.
- To create the flash memory structure and configure write protection for static sections.

The flash writer is employed in host-based boot modes. The AM243x processor supports booting from various external sources, but for this work, two are relevant: UART and Ethernet. **UART** is used in the prototype development phase, while **Ethernet** is intended for the final device.

The Public ROM code provides the **BOOTP/TFTP protocol** for Ethernet-based booting. The device supports both Ethernet interfaces RMII and RGMII based on the hardware configuration. For UART-based booting, the **XMODEM protocol** is utilized.

5.2.10.1 The Ethernet Booting Process Description

After device configuration, the bootloader performs a standard BOOTP/TFTP boot. The device sends a BOOTP request with its MAC address to a host TFTP server to be assigned an IP from a pool of addresses. The timeout for each BOOTP packet is 4 seconds, and the ROM will attempt 10 BOOTP retries, after which the boot mode will fail. If the connection is established, the device initiates a TFTP download and is able to receive image data encapsulated in Ethernet packets.[10] There is a timeout of 1 second to receive a response for the READ request, and the ROM will retry the READ request 10 times, after which the boot mode will fail. If the TFTP download is successful, data received is stripped of its network headers and the boot data is stored in internal RAM. When the transfer completes and the image is found to have good integrity, the ROM Code will branch to the address defined in the Boot Info field of the boot header [45].

There are a few limitations as received packets cannot be IP fragmented or only DIX Ethernet headers are supported.[11] But they don't limit my project in any way.

5.2.10.2 The UART Booting Process Description

The ROM Code is always configured on the UART with a transmission speed of 115200 kbaud in 8-n-1 mode.[12]

After the ROM code configures the UART interface, the device transmits ASCII capital 'C' characters as pings for several seconds, detectable by the host. An example of the XMODEM protocol in half-duplex transfer is shown in Figure 5.10. The UART boot mode exclusively supports the CRC mode of the XMODEM protocol, with no support for CHECKSUM mode, and accommodates block sizes of 128 or 1024 bytes. The host must initiate the transfer of the boot image using the XMODEM protocol before the device's pings cease [45].

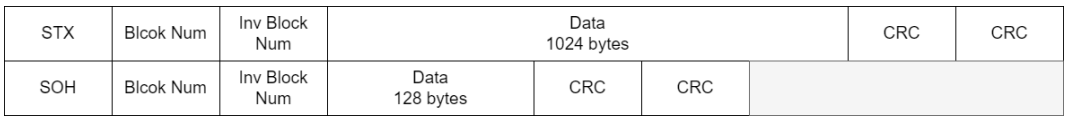

Figure 5.9. The format of XMODEM 1024-byte and 128-byte long frame.

[10] Device supporting only IPv4 packets.
[11] Device does not support 802.3 with SNAP/LLC, DIX Ethernet with VLAN, and 802.3 with VLAN and SNAP/LLC.
[12] This mode means 8 data bits, no parity, and 1 stop bit [48].

Field	Value	Description
STX	0x02	The start character for 1024-byte CRC data blocks.
SOH	0x01	The start character for 128-byte CRC data block.
Block Num	0x01-0xFF	The block number. The first block has value 1, and the block number wraps around 0xFF to 0.
Inv Block Num	0xFE-0x00	The inverse block number (bit inverse of the block number).
CRC	Based on data	The 16-bit CRC generated from the polynomial 0x1021.

Table 5.1. The table describes fields in the XMODEM frame format illustrated in the figure 5.9.

The Figure 5.9 illustrates the XMODEM frame format. The meaning of each field is as follows:

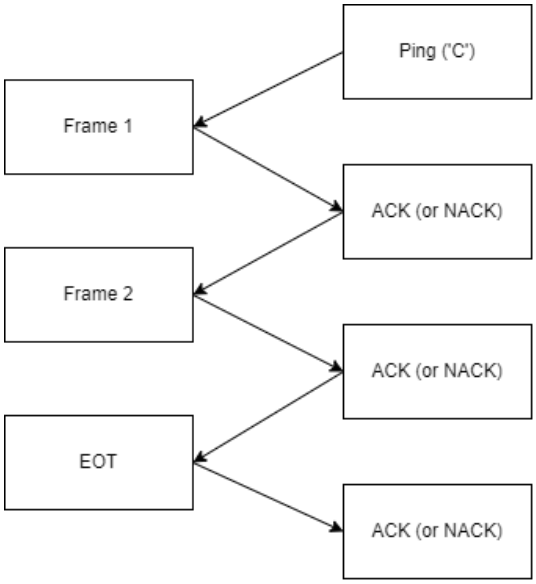

Figure 5.10. The example of XMODEM Transfer Protocol in half-duplex mode.

5.2.10.3 Flash Writer Process

The flash writer sequentially receives and stores data in flash memory according to the flash layout illustrated in Figure 5.8. The process begins by storing the bootloader image for the *R5_0-0* core at address *0x00000000*. Next, it creates the device info structure, populating it with received data, and generates an initial boot info structure, which may be empty.

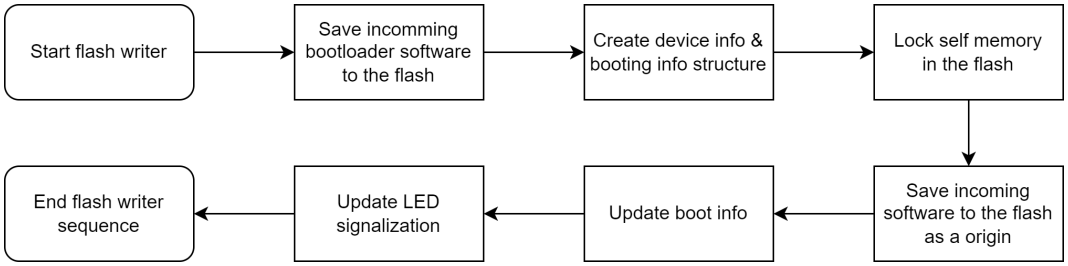

Figure 5.11. The diagram illustrates the flash write sequence.

The flash writer then locks the flash memory regions containing the bootloader image and device info to prevent future overwrites, ensuring these sections remain immutable. Subsequently, it stores the incoming images for each CPU core in the original software section of the flash memory. The boot info structure is updated to specify the address of the original software, with the new software address left undefined and the *failure count* and *failure info* set to 0. Finally, the flash writer illuminates an LED to indicate the successful completion of the software download process.

5.3 Software Update Process

The software update component operates during the standard runtime of the program on the primary real-time core, *R5_0-0*. Its primary objective, as implied by its name, is to facilitate the migration of the system to a new software version. Given the need to update secure software, this process is non-trivial and requires careful design to prevent undefined states or unrecoverable errors.

The software update process is structured as follows: The update is initiated when the web interface receives a POST request. First, the *failure count* variable in the flash memory's boot info structure is set to 1 using a pseudo atomic operation. The *failure count* serves as a counter for attempts to launch the new software, with its role in determining the software version to boot detailed in the bootloader section 5.2.7. A *failure count* of 0 indicates that the original software is valid, while a non-zero value signifies an ongoing attempt to boot the new software. The maximum number of attempts is configurable, set to 4 in our case, allowing three retries. Incoming data is then received and stored at the *new software* address in flash memory, as defined in the flash memory layout Figure 5.8.

After storing the data, the device restarts, and the bootloader detects the non-zero *failure count*, prompting it to attempt booting the new image. The system then enters standard operation to verify whether the device functions correctly. Two outcomes are possible:

- **Failure**: The *failure count* is incremented by 1, and the device restarts. If the *failure count* reaches the maximum limit (e.g., 4), the boot info structure is updated to reset *failure count* to 0, ensuring the original software is booted on the next attempt. An error code corresponding to the failure is recorded in the *failure info* field, with all states detailed in Table 5.2. The Figure 5.12 illustrates the *failure count* variable states.
- **Success**: The boot info structure is updated to set the *failure count* to 0, and the addresses of *original software* and the *new software* are swapped, marking the update as successful.

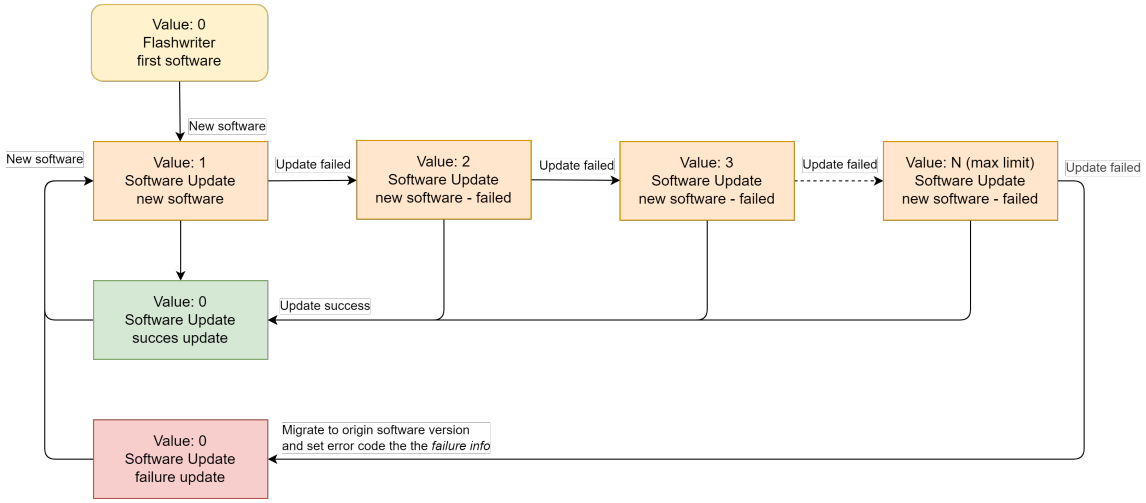

Figure 5.12. Diagram illustrates the *failure count* variable from boot info structure in the flash memory.

Value	Description
0x00	No error code.
0x01	Software update failed, the system uses the original version.
0x02	Software update failed, the new software has not valid certificate.
0x03	Software initialization check failed, the SPI communication doesn't operate correctly.
0x04	Software initialization check failed, the Inter-core communication doesn't operate correctly.
0x05	Software running check failed, the Inter-core communication doesn't operate correctly.
0x06	Software initialization check failed, CPU resources
0x07	Testing Initial Sequence Timeout
0x08	Error detected in the ESM
0xFF	Not defined error

Table 5.2. The table describes codes in the Failure Info Memory Block in the flash at the position defined by 5.8.

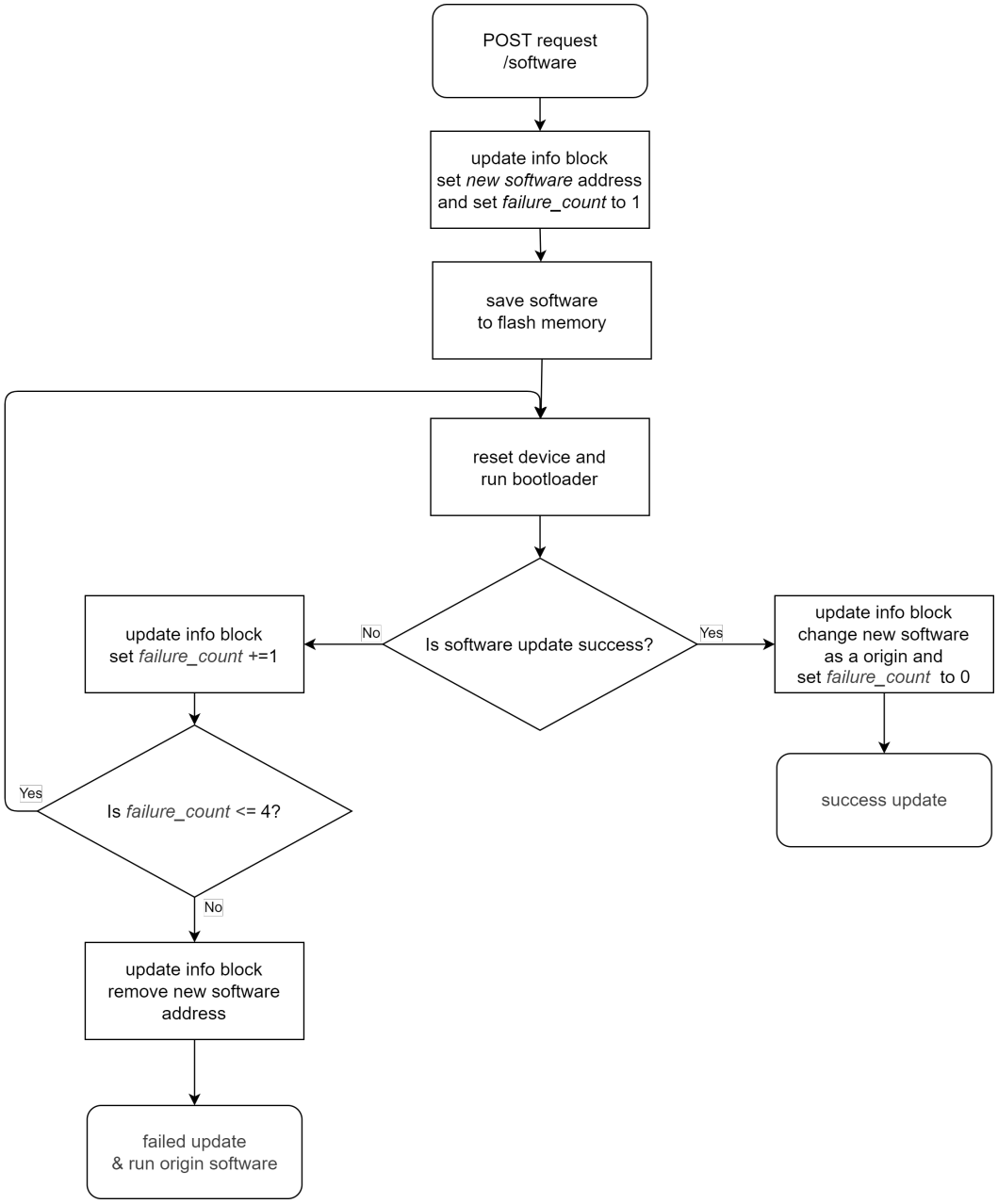

Figure 5.13. Diagram illustrating the software update process.

5.3.0.1 Ensuring Robustness in Software Updates

A key challenge is verifying that the software update completed successfully and preventing failures that could render the system inoperable. The potential issues and their mitigations are as follows:

- **Corrupted or Malicious Software**: If invalid software is uploaded (e.g., due to an attack or process error), the bootloader's X.509 certificate validation during RAM loading detects the issue, logging error code *0x02* in the *failure info* field of the boot info structure.
- **Damaged Data**: Corrupted data is similarly addressed through certificate-based integrity checks, ensuring the software image's validity [45].
- **Restart During Update**: If the device restarts (intentionally, unintentionally, or due to a power outage) during the update:

- **Incomplete Software Upload**: Certificate validation prevents partial uploads from being executed.
- **Interruption During Runtime Validation**: The system increments the *failure count* and retries, as the bootloader uses the *failure count* to boot the new software upon restart.
- **Restart Before Boot Info Update**: The device uploading the software receives a negative acknowledgment, indicating an incomplete download. Once the boot info is updated, the internal mechanism handles such errors autonomously.

- **Residual Old Code in a Core**: This scenario is impossible, as the device restart clears all CPU RAM contents, and the bootloader reloads the new software into RAM from flash memory.

The second question addressed Ascertaining **whether the software update was successful** was partially addressed in the software update section. To confirm that the updated software operates correctly, the system transitions to the standard runtime phase, which begins with an initialization check. This check verifies peripheral configurations, communication integrity, and other critical parameters. If an error occurs during this phase, the device enters a safety shutdown, as detailed in Chapter 5.4. Upon successful completion of the initialization check, further described in Chapter 5.5, the system is deemed operational, and the boot info structure is updated, as previously outlined.

5.4 Safety Shutdown

One of the device's operational states is the safety shutdown, which is triggered when an error occurs, allowing the device to respond to the fault. The safety shutdown state can lead to one of two actions: **device reboot** or **termination**.

5.4.1 External Monitor Device

The TI Functional Safety Manual[13] recommends using an external monitoring device for higher SIL levels to disconnect the power supply in case of a device failure.[14] Specifically, TI advocates for an external Power Management Integrated Circuit (PMIC) suitable for SIL-2 solutions. TI PMIC devices provide the following functionalities:

- Overvoltage and undervoltage monitoring of power resource voltage outputs.
- Overvoltage monitoring of the PMIC input.
- Watchdog monitoring of the safety processor.
- MCU error monitoring.
- MCU reset.
- I2C communication with Cyclic Redundancy Check (CRC).
- Error indicator to drive external circuitry.

A potential illustration of the integration between the AM243x and a Power Management Integrated Circuit (PMIC) is presented in Figure 5.14. The PMIC regulates the AM243x power supply based on safety functions, including a Q&A watchdog that expects periodic responses and employs Cyclic Redundancy Check (CRC) to protect against errors. Additionally, a safety unit integrated within the AM243x can notify the PMIC of any faults. The PMIC also supports sending interrupts to the MCU to alert

[13] The internal Functional Safety Manual is under NDA. [46].
[14] See points [SA_7], [SA_8], and [SA_19] in the Safety Function Manual. [46]

it of critical conditions. While the configuration can be significantly more advanced, this schematic aims solely to illustrate a possible implementation.

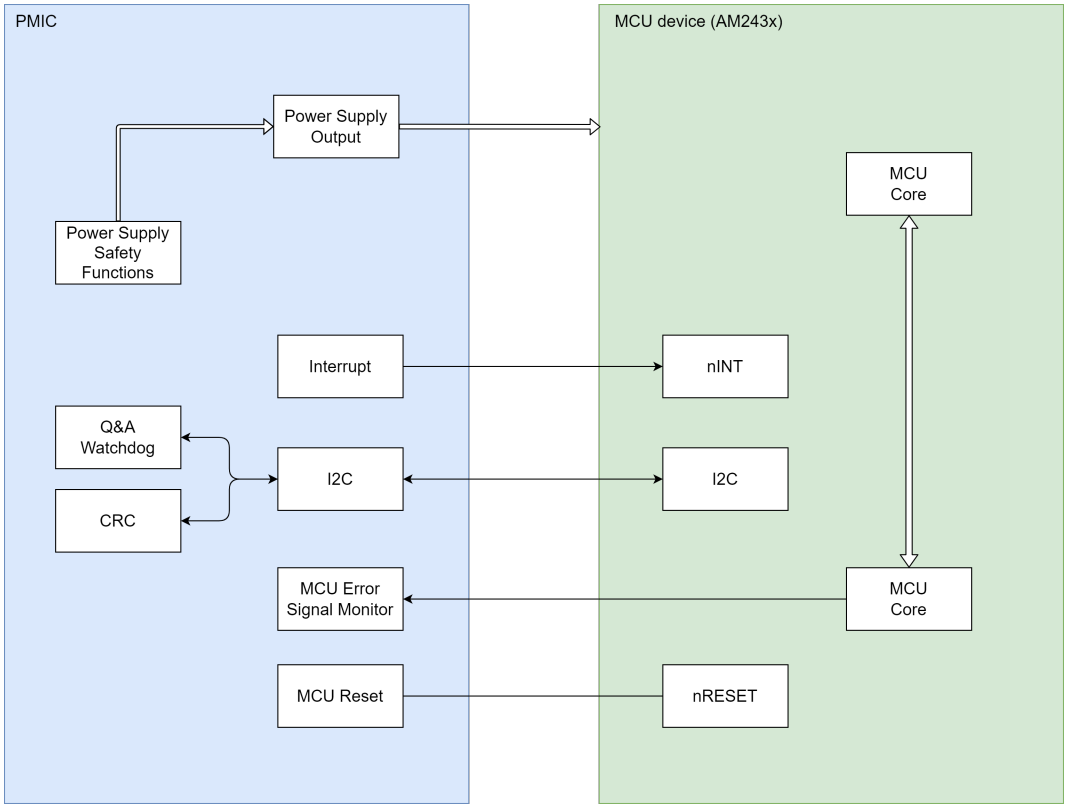

Figure 5.14. Diagram illustrates the possible connection between AM243x and the PMIC device.

5.4.2 Safety Shutdown Process Description

There is used an **external Q&A Watchdog** monitor with a similar function as the described PMIC. It monitors the device's operation, expecting periodic confirmation that the processor is functioning correctly. If this confirmation is not received, the watchdog hardware disconnects the power supply, shutting down the entire device. The device may also be powered off if an error occurs in another component communicating with it, with this action managed by the hardware.

The external watchdog's functionality is utilized for one type of termination: if the device stops sending the periodic confirmation message and the **error pin output**[15] signals a problem, the watchdog initiates a shutdown. The safety shutdown process is straightforward, as depicted in Figure 5.15, and consists of two primary steps: storing an error code in the flash memory's boot info structure, specifically in the *failure info* field, and executing termination based on the specific error state. When the device boots a new image, in addition to setting an error code, the boot failure count variable is incremented if an error occurs. The various scenarios are detailed in Table 5.3, with corresponding error codes and descriptions listed in Table 5.2.

The safety shutdown process executes on the R0 core, as the M4 core cannot directly modify flash memory. In the event of a failure in the M4 core, it ceases sending periodic messages to the external watchdog based on the error code and sends a request to the R5 core to update the flash memory.

[15] MCU_SAFETY_ERRORn

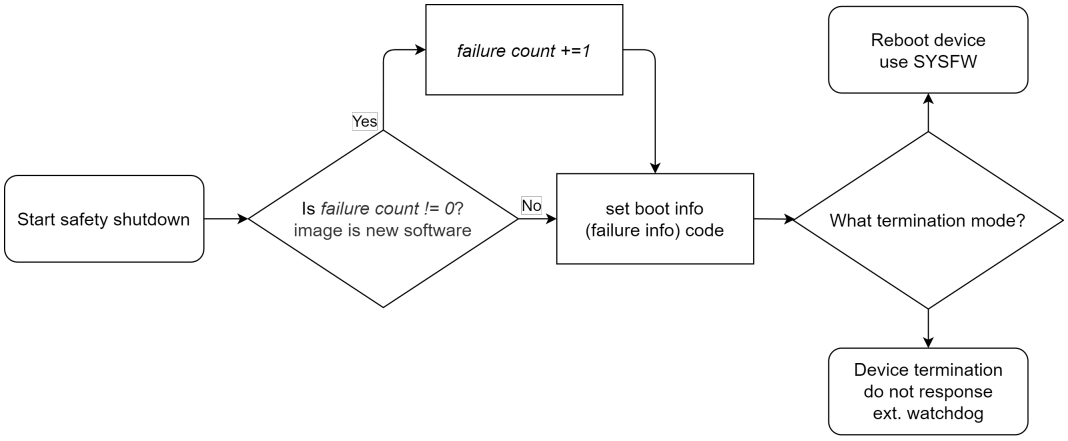

Figure 5.15. Diagram illustrating the device safety shutdown process.

The safety shutdown function can be triggered by errors detected during the initialization test sequence, runtime software failures, an invalid new software image, or faults identified by the Error Signaling Module (ESM). The ESM is an independent unit that monitors both the MAIN and MCU domains, with a dedicated ESM instance in each domain.[16] When an error is detected by the ESM, the safety shutdown process is initiated. The ESM aggregates safety-relevant events from across the System-on-Chip (SoC) into a single point and signals them to the R0 core via interrupts for processing. The R0 core handles the error with the corresponding error code, as illustrated in Figure 5.15. Errors reported by the ESM are categorized into two groups: corrected faults and non-corrected faults.[17]

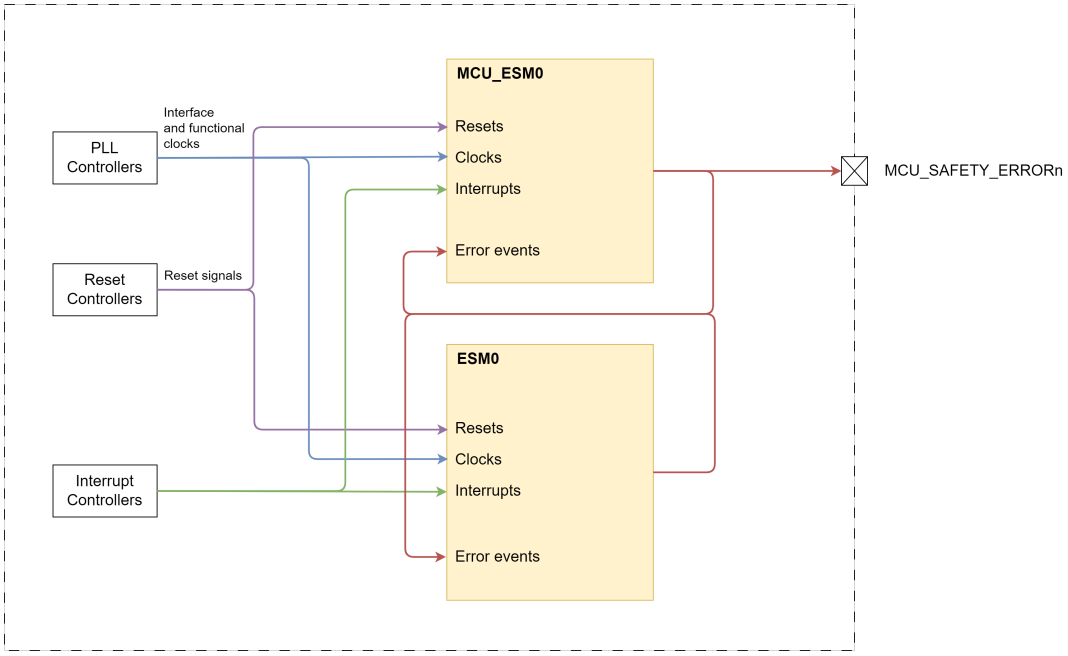

Figure 5.16. Diagram illustrating the ESM module overview connection. The *MCU_ESM0* is placed in the MCU domain, the *EMS0* is placed in the MAIN domain.

A failure to initialize the device correctly typically poses no safety hazard, as it prevents operation from commencing. The critical risk emerges when the device starts,

[16] The MCU domain includes *MCU_ESM0*, while the MAIN domain includes *ESM0*.
[17] A detailed description of how to work with this module is available in the Technical Reference Manual.

functions normally for a period, and subsequently fails. To address this, initialization errors trigger an automatic reboot, prompting the device to attempt a full restart. In contrast, runtime errors lead to immediate termination, as they represent a significant safety hazard.

Description	Error Code	Termination Mode
SPI initialization error	0x03	Reboot
Inter-Core communication initialization error	0x04	Reboot
Inter-Core communication runtime error	0x05	Termination
CPU resources initialization error	0x06	Reboot
Testing Initial Sequence Timeout	0x07	Reboot
ESM detected error	0x08	Termination
Undefined error	0xFF	Termination

Table 5.3. This table outlines the safety shutdown states, their corresponding error codes, and the associated termination modes.

5.5 Initialization Test Sequence

Before launching the application, an initialization test (check) sequence is executed to:

- Verify that all SIL software components function correctly.
- Validate the correct behavior of the software during an update.

The process consists of a few steps, each testing a single component. If an error occurs, the device transitions to a safety shutdown state with the corresponding error code, and the safety shutdown process handles the fault. The process is illustrated in the Figure 5.17.

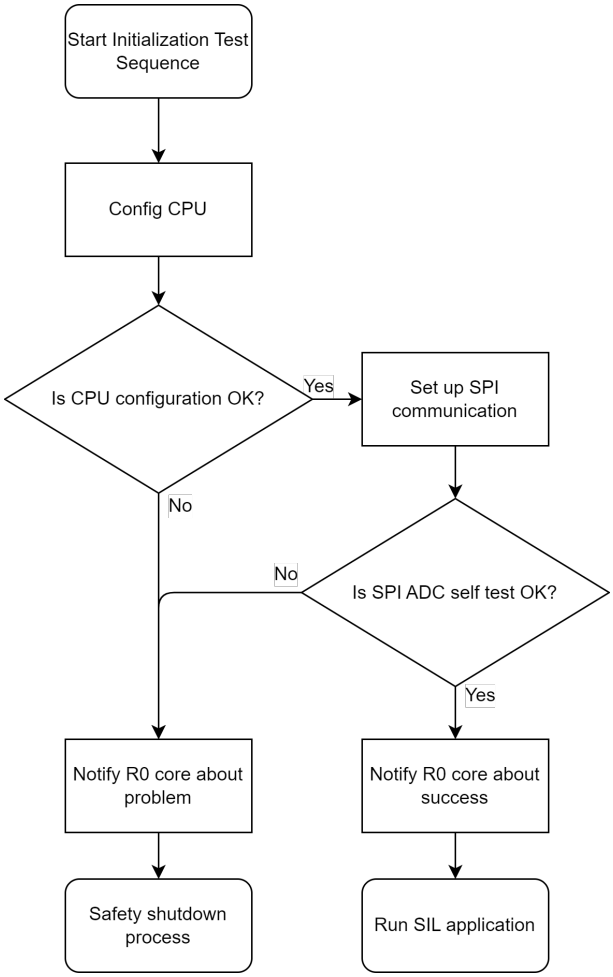

Figure 5.17. Diagram illustrating the software initialization check process.

- CPU verification involves configuring clocks, PLL, GPIO, IPC mechanisms, memory allocation, and other essentials. An initialization error occurs when internal checks fail, prompting the device to enter a safety shutdown state with the appropriate error code. This description is not entirely precise, as checks typically occur after each initialization step. Based on the implementation, it is advisable to expand and refine the error codes to correspond to specific issues.
- The ADC verification over SPI involves initializing the SPI interface, configuring the ADC converter, and performing a self-test. The ADC converter allows reading values from its registers using a callback configuration. Verification is conducted by configuring the ADC and expecting the configuration to be read back correctly. If this fails, the process is repeated twice more (a total of three attempts). If none of the attempts succeed, the SPI test via ADC is deemed unsuccessful.
- Inter-core communication does not require a dedicated test step, as successful completion of the initialization sequence depends on its proper functioning. The entire sequence executes on the M4 core in isolated mode. Upon completion, the M4 core must send a message to the R5 core indicating either successful initialization or an error with the corresponding error code. If the R5 core does not receive this message within a specified timeout period, it determines that the M4 core is malfunctioning, and an error code is logged in the flash memory. This process is illustrated in the Figure 5.18.

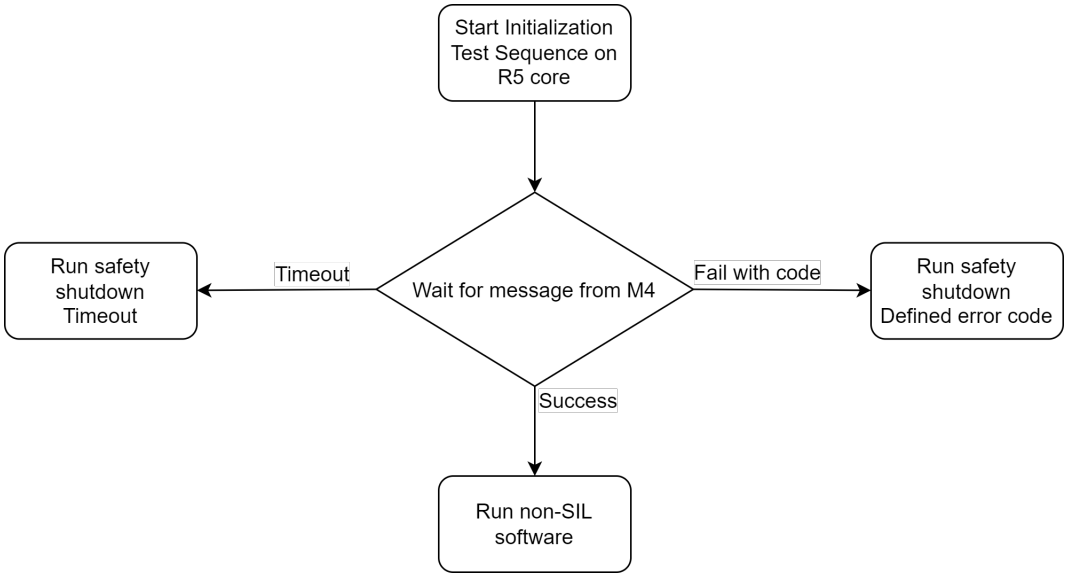

Figure 5.18. Diagram illustrating the software initialization check process in the R5 core.

The success of the **software update** process critically depends on this initialization sequence. These initialization checks serve as the primary mechanism for determining whether the software update was successful.

To summarize, SIL software is considered **successfully** launched if:

- The X.509 certificate is valid (verified by the bootloader).
- The Testing Initialization Sequence in the M4 core completes successfully.

5.6 Safe Software

The SIL component, operating on the isolated M4 core, is tasked with fulfilling safety functions that are relatively straightforward. Its primary role is to read data from ADC modules, evaluate the data, and send a corresponding binary signal to the Simatic PLC via a binary output. This constitutes its core safety function. Its secondary function is to provide the collected data to the R5 core, which uses the information for monitoring purposes.

5.6.1 Core Isolation Description

The safety processor is separated out in its own domain called the MCU domain (safety channel) and the rest of the SOC, which includes the application processor. MCU domain owns dedicated peripherals such as GPIOs, I2C, UART, SPI, interconnect, and configuration logic. This makes sure that the M4F in the MCU channel can execute independently using dedicated resources. After the AM243x device is booted up, the MCU domain can be configured to be isolated from the MAIN domain. This is done by enabling **clock gating isolation**. Dedicated Local Power Sleep Controller (LPSC) controllers present in the device can provide for all clock stop control for all the transactions between the MAIN domain and MCU domain. When this isolation is enabled, any transaction from the MAIN domain will be blocked from entering the safety channel. These transactions are terminated gracefully, and the MAIN domain processor will be notified of these violations. Note that IPC communication is not hindered by clock gating.

Independent reset pins for both the MAIN and MCU domain provide flexibility to reset the MAIN domain while the MCU domain is still active. Additionally, both the MAIN and the MCU domain can issue a reset to the MAIN domain independently through dedicated reset control registers, which can result in a warm reset or POR[18] of the MAIN domain. At this point, the MCU domain can function independently. The reset of the MCU domain, however, will result in the reset of the entire device [46].

5.6.2 Process Description

The operation of the SIL core is cyclic, as illustrated in Figure 5.19. The core is isolated, running independently of the R5 cores' resources (MAIN domain), including its own clock, RAM, and peripherals. The system supports separate restarts for the MAIN domain and the MCU domain (where the isolated core operates), though this functionality is not utilized from the perspective of the M4 core.

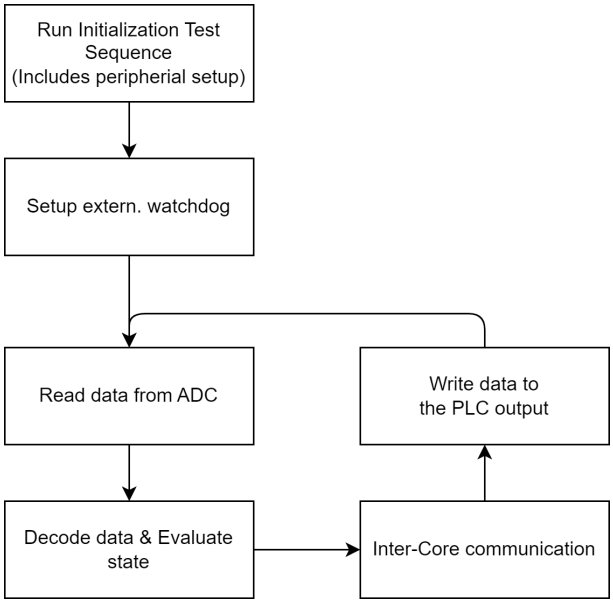

Figure 5.19. Diagram illustrating the process in the SIL component.

The process begins with the initialization of essential components, including the configuration of clocks, PLL, peripherals, and memory resources. This sequence is part of the Initialization Test Sequence described in Chapter 5.5. Subsequently, the Q&A watchdog is initialized to oversee the operation of the entire device. Initially, I intended to use a timer for its management, but this approach undermines the watchdog's functionality. It is expected that checks will be implemented in the main loop, ensuring that in the event of software failure (e.g., jumping to undefined memory), the watchdog responds and terminates the device. The watchdog function is described in greater detail in Chapter 5.4.

Once all components are initialized, a loop is executed that performs the following sequence: reading data from the SPI interface, processing the data, and evaluating its state. The evaluation varies based on the hardware configuration, specifically depending on the type of railway crossing and the country in which the device is deployed. Further details on these configurations are provided in Chapter 2.6.

After evaluating the state, data is exchanged with the MAIN domain (i.e., the R5 core), which serves as a monitoring device. Data is not sent in every cycle but rather

[18] Power-on Reset

once every n cycles. This is because ADC data reading requires near-real-time performance with low latency (in practice, high-speed data transfer from the ADC module via SPI is limited due to hardware protections, such as high-impedance resistors), whereas the monitoring device does not require real-time data. Inter-core communication is further described in Chapter 5.8.

In the final step of the loop, a signal is sent to the Simatic PLC as a binary output. This signal is not transmitted via serial communication but consists of simple binary states: *HIGH* or *LOW*.

5.7 Non-SIL Software

In contrast to the SIL software, which focuses on safety-critical functions, the non-SIL software encompasses a broader range of functionalities and operates primarily on a single R5 core within the MAIN domain. Its responsibilities include:

- Communicating with the SIL software (i.e., the MCU domain) to acquire measured and evaluated data.
- Running a web server to provide measured data and facilitate software updates.
- Indicating the device's status using informational LEDs.
- Performing additional measurements to better understand the device's state, though these do not fulfill SIL-relevant functions.

The core runs on a real-time operating system, specifically FreeRTOS. An alternative, Zephyr, is better suited for larger-scale projects that function as platforms,[19] but its use is not justified in the context of this project. The primary motivation for using a real-time operating system is to optimize the operation of the web server and enable task prioritization. Since the system runs on FreeRTOS, a decision must be made regarding the structure of the non-SIL software.

Two general approaches exist. The first involves parallelizing all subtasks, as illustrated in Figure 5.20. In this scenario, each task is handled by a separate FreeRTOS task. The advantage of this approach is the ability to prioritize certain tasks, such as those responsible for measurements, over the web server's operation. However, a disadvantage is the need to manage access to shared data.

[19] It is the result of discussion with Texas Instrument engineers on the workshop.

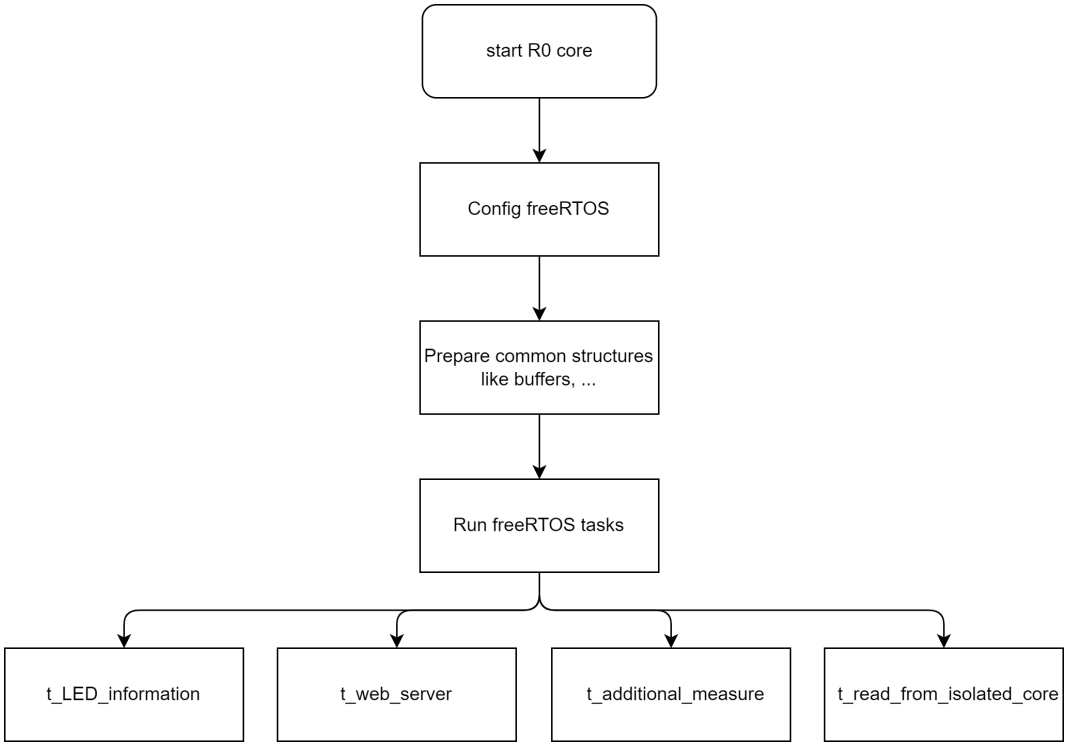

Figure 5.20. The diagram illustrates the parallel version of non-SIL software

The second approach adopts a more sequential structure, as depicted in Figure 5.21. Here, only two main tasks are used (though more may exist, as one way to implement the web server is to spawn a new task for each client request). One task handles the web server, while the other manages cyclic measurement tasks. The advantage of this approach is the minimization of shared data access, as the measurement task primarily writes to memory, and the web server task only reads from it. A potential drawback could be limited task prioritization. However, this is not an issue in this project, as measurement components are prioritized over the web server. Additionally, the measurement components are relatively simple and do not involve complex operations in terms of code scope. For these reasons, I opted for the second approach, the more sequential structure.

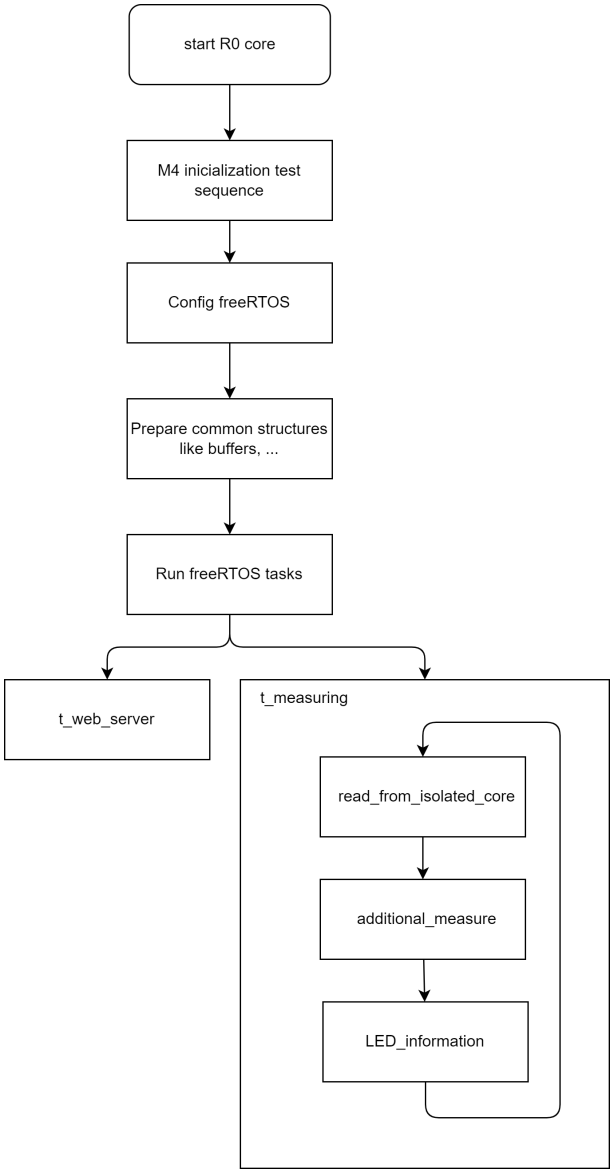

Figure 5.21. The diagram illustrates the sequential version of non-SIL software.

Let us examine the operation of the non-SIL software in greater detail. Upon startup, the system performs initial configuration of data structures and FreeRTOS settings. Additionally, it participates in the **Initialization Test Sequence**, specifically by echoing back a test message for inter-core communication (IPC) verification. Following this basic configuration, the FreeRTOS system is launched, initiating its individual tasks.

The cyclic operations are managed within the *t_measuring* task, with its configuration performed at the task's outset. For clarity and to illustrate the sequence of steps cohesively, I have chosen to present the process as a unified loop. The *t_measuring* task comprises three sequences: *read_from_isolated_core*, *additional_measure*, and *LED_information*.

The *read_from_isolated_core* sequence handles data acquisition from the isolated M4 core, as discussed in detail in Chapter 5.8, which covers inter-core communication.

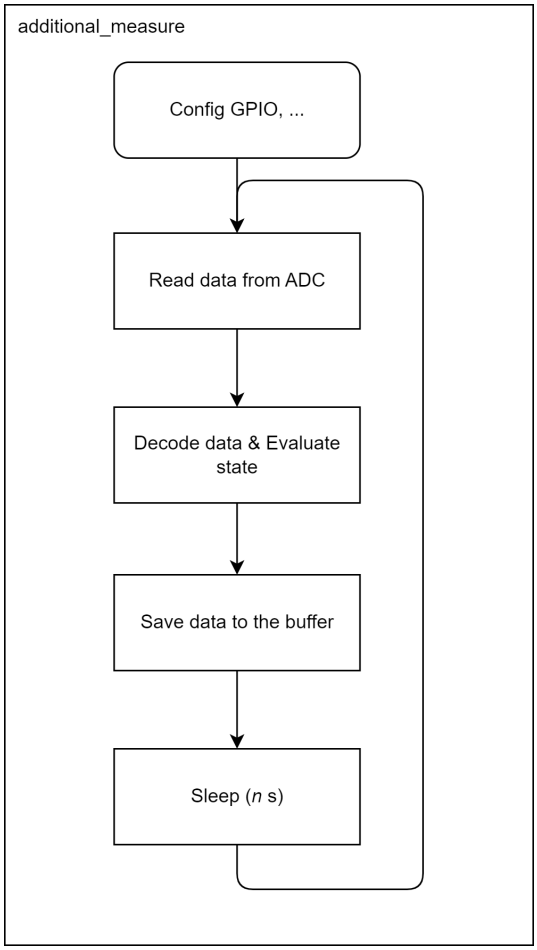

Figure 5.22. The diagram illustrates *additional_measure* sequence.

The *additional_measure* sequence aims to provide supplementary data to better identify error states and their causes. Currently, it includes one additional ADC module measuring the voltage of a non-SIL2-relevant component, thus not safety-critical. This sequence configures the peripheral, cyclically reads data from the ADC module, decodes and evaluates the data, and stores the results in a buffer. These results are made available via the web server upon request, alongside other data. As this sequence is non-critical, it can be executed intermittently. A sleep function pauses the sequence for n seconds, with an estimated interval of 1 second deemed sufficient, to be finalized during implementation.

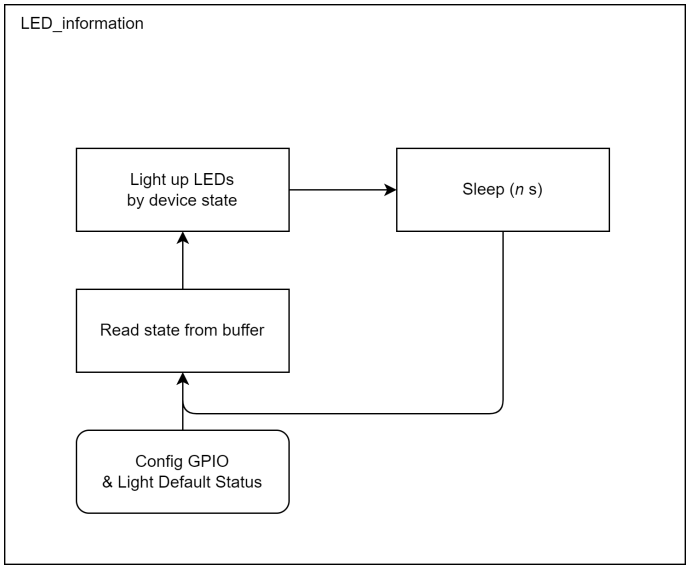

Figure 5.23. The diagram illustrates *LED_information* sequence.

The *LED_information* sequence displays the device's status using LEDs, with one LED assigned to each ADC module connected to the M4 core. Each LED can indicate three states across four conditions: unpowered (a), powered (b), off (c), or undefined (d). Two LEDs are used for signaling: one lights up when the device is powered, the other when it is off, and both illuminate for an undefined state. These states are illustrated in Figure 5.24.

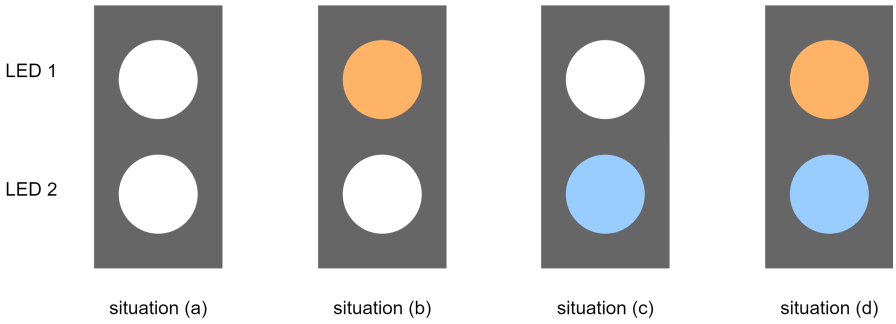

Figure 5.24. Information LED Status Illustration.

The *t_web_server* task delivers measured data via HTTP responses and supports software updates. This task runs at a lower priority than the *t_measuring* task, which is responsible for data acquisition. The process begins by launching the web server, which involves configuring the Ethernet interface available on the development kit. Once active, the web server awaits incoming requests, handling two types: a GET request to the */data* URL path, returning measured data in a JSON structure, and a POST request to the */software* URL path, initiating a software update. The update process is detailed in Chapter 5.3. If an unknown request is received, the web server returns an error code in the response.

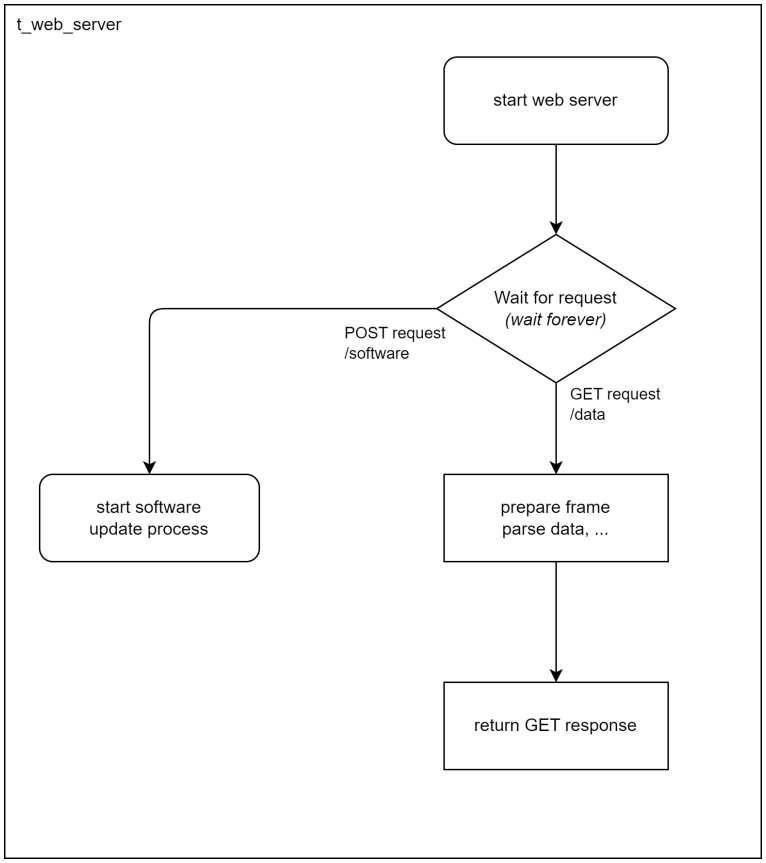

Figure 5.25. The diagram illustrates *t_web_server* task process.

As noted in the introductory discussion, the sequential approach minimizes conflicts in accessing shared resources. The web server reads data structures populated by the measurement sequences but does not modify them, avoiding write conflicts. The only requirement is to ensure the web server does not return inconsistent data, such as data being actively modified by the measurement task.

5.8 Inter-Core Communication

The device requires the capability to communicate between cores, specifically between the isolated MCU domain and the MAIN domain, where the non-SIL software operates. This necessitates the configuration of an inter-core communication mechanism. The objectives of this layer are:

- To enable the transmission of measured data from the isolated M4 core to the non-SIL R5 core for display purposes.
- To facilitate the reporting of error codes and their logging in flash memory, particularly during the Initialization Test Sequence (described in Chapter 5.5).

Two primary approaches are available. The AM243x device provides an IPC mechanism that utilizes a **mailbox system** to send notifications and messages. Alternatively, a **shared memory region** can be defined and used as a communication medium between cores.

Mailbox module serves to facilitate the communication between the various on-chip processors of the device by providing a queued mailbox-interrupt mechanism. The

queued mailbox-interrupt mechanism allows the software to establish a communication channel between two processors (users) through a set of registers and associated interrupt signals. The module does not support the hardware protection to prevent users from reading FIFO mailboxes that they don't own as receiver or writing to FIFO mailboxes that they don't own as sender [45].

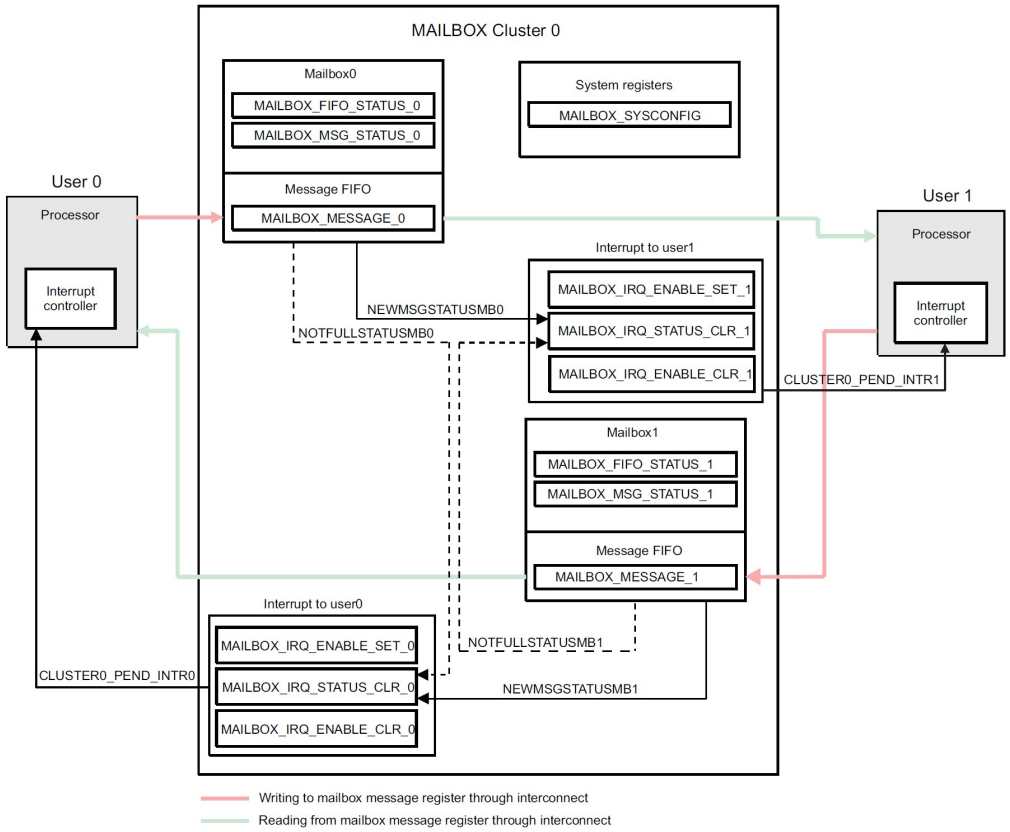

Figure 5.26. The figure illustrates inter-core communication between two processors using the AM243x mailbox system. A 32-bit message written to the *MAILBOX_MESSAGE_y* register is appended to a FIFO queue with a capacity of four messages; if the queue is full, the message is discarded. Overflow is prevented by checking the *MAILBOX_FIFO_STATUS_y* register to ensure the queue is not full before writing. Reading the *MAILBOX_MESSAGE_y* register retrieves and removes the first message from the queue, returning 0 if the queue is empty. A new message interrupt is triggered when at least one message is present, with the *MAILBOX_MSG_STATUS_y* register indicating the number of queued messages [45].

The alternative approach to inter-core communication is to utilize **shared memory**. However, to ensure safety, memory operations must be protected. The advantages of this approach include simpler implementation and the elimination of the need for a dedicated subsystem. Consequently, I opted for this method, though it requires enhancements to meet safety requirements.

Communication is designed to be unidirectional: the M4 core can access and write to the shared memory, while the real-time R5 core is restricted to read-only access. To prevent memory corruption due to erroneous jumps, **firewall mechanisms** are employed.

The AM243x device supports two primary types of firewalls: region-based and channelized. **Channelized firewalls** are implemented solely to protect registers controlling the power domain and Local Power Sleep Controller (LPSC) for the security enclave, with

their settings fixed and non-modifiable by users. **Region-based firewalls**, positioned at the target endpoint in the interconnect, are shared by groups of target interfaces. Each region-based firewall is configured with:

- **FWID**: A unique identifier for the firewall block.
- **Number of protected regions**: Specifies the number of user-definable protected regions.
- **Memory regions behind the firewall**: Transactions targeting these regions are subject to firewall protection.

Most memory regions in the System-on-Chip (SoC) are safeguarded by region-based firewalls, except for the public boot ROM, debug cell, and System Trace Module (STM).[20]

Memory protected by a firewall must be aligned to a 4KB boundary, with a minimum size of 4KB. Each firewall can protect only a single memory block, preventing the stacking of multiple firewalls [49].

The firewall is configured to permit write access exclusively to the M4 core, while other cores, such as the R5 core, are limited to read-only access. This mechanism ensures that the memory used by the M4 core for SIL-relevant operations, allocated in the MAIN domain, cannot be overwritten. It also guarantees that only the M4 core can perform writes, eliminating potential conflicts. The primary challenge is ensuring data consistency, preventing the R5 core from reading data during a write operation, which could result in incomplete or inconsistent values.

Multiple solutions exist to address this issue. An atomic operation, similar to those used for modifying flash memory structures, is unnecessary in this context. Alternatively, a spinlock or mutex mechanism could lock the structure during access, but this is overly complex for the given scenario, as absolute data correctness is not required. It suffices to discard and ignore inconsistent data, a situation expected to occur rarely due to the rapid nature of read operations. Nevertheless, this possibility must be accounted for.

To ensure consistency, the shared memory is organized into multiple structures. Each structure includes an identifier at its start and end, incremented by one with each write. Data is written sequentially, and consistency is verified if the start and end identifiers match. This mechanism is applied to every structure.

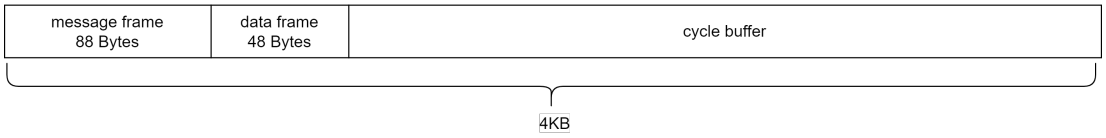

Figure 5.27. The overview of inter-core communication shared memory.

Two primary structures are defined in the shared memory, both protected by the described consistency mechanism. The first, illustrated in Figure 5.28, facilitates message transmission from the M4 core to the R5 core, primarily for reporting errors or confirming the successful completion of the Initialization Test Sequence.

start integrity id uint8_t	message code uint8_t	message data uint8_t [8]	end integrty id uint8_t

Figure 5.28. The message frame of inter-core communication.

[20] The device using Arm CoreSight STM with supporting logic that maps initiator IDs to specific STP Major Source IDs.

The second structure, depicted in Figure 5.29, transfers measured data from ADC modules to the R5 core for reporting via its web server. This structure comprises a header, protected by the consistency mechanism to ensure integrity, which includes metadata about the subsequent circular buffer, such as the latest element's ID and the block size, in addition to integrity IDs.

Figure 5.29. The data frame of inter-core communication.

The circular buffer is updated as follows: First, the *start integrity id* is incremented. Based on the latest element's ID and *block size*, the system locates the oldest element. The *start position* (the oldest element, typically one position behind the latest element, except at the buffer's end) is incremented, and the new value is written. Finally, the *end integrity id* is incremented.

5.9 Safety Mechanisms and Requirements

As discussed in previous chapters, the system is divided into SIL (Safety Integrity Level) and non-SIL components. For the SIL component, it is critical to ensure that no unhandled error states occur. This section analyzes potential failure scenarios, their mitigations, and the general mechanisms defined by the relevant standard that apply to our project.

As discussed in the chapter reviewing relevant manuals (Chapter 2.3), two primary types of faults must be addressed: systematic and random faults. **Systematic faults** are mitigated through development processes and adherence to established guidelines. In contrast, **random faults**, which cannot be prevented, must be managed during device operation. The following sections primarily focus on methods for minimizing systematic faults. Toward the end of the chapter, I will examine the techniques and features provided by the AM243x device for addressing both fault types.

The system is designed to address safety-critical issues primarily through hardware, leveraging inherent fail-safety mechanisms (Chapter 2.5.2). This approach enhances maintainability and simplifies the software update process. Nevertheless, the SIL component must operate in a safe mode. To achieve this, in addition to procedural guidelines, several techniques and architectural approaches are employed. As previously mentioned, most safety requirements are addressed through inherent hardware mechanisms. Additionally, an external watchdog is connected to the device to monitor its behavior, expecting a confirmation code at regular intervals. If the code is not received, the watchdog disconnects the device from its power supply. Further details on the watchdog's functionality are provided in Chapter 5.4.

The standard outlines several mechanisms, discussed in Chapter 2.5.3. For this project, the following methods relevant to the software architecture were selected:

- **Defensive Programming**: Defined by coding standards, further detailed in Chapter 2.5.3.
- **Fault Detection and Diagnosis**: Implemented within the railway crossing control system by this device.
- **Information Encapsulation**: Incorporated as part of object-oriented programming, described in Chapter 2.5.3.
- **Fully Defined Interfaces**: Also part of object-oriented programming, with principles outlined in Chapter 2.5.3.

From the programming techniques listed in Table 2.6, this project employs:

- Coding Standard and Coding Style Guide
- No Unconditional Jumps
- Limited Size and Complexity of Functions, Subroutines, and Methods
- Entry/Exit Point Strategy for Functions, Subroutines, and Methods
- Limited Use of Global Variables

As specified by the standard, the project is developed using a strongly typed programming language, specifically C and C++.

For object-oriented development, the SIL component adheres to the following standard-defined mechanisms:

- Inheritance is used only when the derived class is a refinement of its base class.
- Depth of inheritance is restricted by coding standards.[21]
- Overriding of operations (methods) is strictly controlled.
- Multiple inheritance is used exclusively for interface classes.

5.9.1 TI Functional Safety Constraints and Assumptions

The TI Functional Safety Manual outlines functional safety constraints and assumptions that, when adhered to, enable software to be classified as SIL-compliant. I aim to highlight and discuss those constraints relevant to system design and development, which must be followed.[22]

- Constraints **[SA_6]**, **[SA_7]**, and **[SA_19]** pertain to external monitoring devices, in my case an external watchdog rather than a PMIC. The watchdog's role is further detailed in **[SA_8]**, with additional information provided in Chapter 5.4.
- Constraint **[SA_13]** addresses power requirements and a safety-critical approach, which is fulfilled by the device's design.
- Paraphrased, **[SA_20]** stipulates that safety functions should only be activated after successful device initialization. This condition is met through the Initialization Testing Sequence, described in Chapter 5.5.
- Constraint **[SA_22]** requires the MCU domain to control the power device. This is achieved by employing an external debugger that expects periodic messages from the MCU domain, specifically the isolated M4 core.
- Constraint **[SA_25]** mandates that when multiple components access the same resource, the SIL level must correspond to the higher requirement. This condition is satisfied, as the device shares no resources except for shared memory used for inter-core communication, which is adequately protected.

Other statements primarily address procedural aspects or conditions irrelevant to this work.

5.9.2 Safety Perspective on Process Execution Analysis

To ensure the safety of processor operations, we analyze the system's execution in detail, dividing it into distinct phases and addressing key safety-related questions for each. For this discussion, the system's operation is categorized as follows:

[21] In this work, I'm using the Siemens internal Coding Standard.

[22] *NOTE*: For identification, I will use the notations from the TI manual, i.e., [SA_<number>], such as [SA_12]. The Functional Safety Manual is under a NDA, preventing its inclusion as an appendix. To access this document, request permission through the forms at `https://www.ti.com/drr/opn/AM64X-RESTRICTED-DOCS-SAFETY` and `https://www.ti.com/drr/opn/AM243X-RESTRICTED-SECURITY`.

- Booting and Initialization
- Device Operation
- Termination

Software Update: This is detailed in Chapter 5.3 and does not involve safety-critical code, so it will not be further discussed here.

5.9.2.1 Booting and Initialization

- **How do we ensure the software's integrity?** Each core's image is signed with an X.509 certificate, which verifies the software's integrity (ensuring it was not corrupted during download) and authenticity (confirming it was signed with our private key).
- **How do we prevent non-SIL code from running on the SIL core or SIL code on the non-SIL core?** The AM243x incorporates an Arm Cortex-R5F and an Arm Cortex-M4F, both utilizing the ARMv7 instruction set (specifically ARMv7-M and ARMv7-R). These cores share identical register address spaces and are based on the same ARMv7 versions. The primary difference is that ARMv7-M architectures always include divide instructions, whereas ARMv7-R includes them in the Thumb instruction set but optionally in its 32-bit instruction set [50–51].

  Theoretically, it might be possible to execute code intended for one core on the other. However, successful initialization requires a core-specific sequence, ensuring that non-SIL code (designed for the R5F core) cannot successfully execute on the SIL M4 core, nor can SIL code execute on the non-SIL R5F core. Running incorrect code on the safety-critical M4 core poses a significant hazard due to potential direct safety impacts, whereas the reverse scenario (SIL code on the non-SIL core) is less critical for our system.
- **How do we verify the correct configuration of the device's clocks and PLL?** Incorrect configuration of the clocks or PLL may cause failures in components synchronized by the clock signal, such as erroneous data reading from flash memory during the boot phase. While this issue is not critical for our system, the priority lies with processes running on the isolated M4 core, particularly data acquisition via SPI and inter-core communication. Correct configuration is verified using a predefined sequence, specifically the initial setup of the ADC via SPI on the M4 core. If the timing is incorrect, the device will fail, and the initialization sequence will not complete successfully. Additionally, an external debugger monitors the device during runtime to ensure proper operation.
- **How do we verify the correct configuration of firewalls and communication modes?** If the cores cannot communicate with each other, the R5 core will be unable to read data sent by the M4 core, compromising the device's primary functionality, though SIL functions would remain intact. The verification process is similar to the clock and PLL case: a predefined sequence of data is transmitted between the cores in both directions to confirm correct inter-core communication. If this communication fails, the condition for successful processor operation, as described in the software update process (Chapter 5.3), is not met, indicating that the code did not execute correctly, and an error is logged in the flash memory.
- **How do we ensure that the CPU's memory is properly initialized?** The bootloader loads the code into the internal RAM, and its integrity is verified using an X.509 certificate. If the certificate or the entire image is invalid, an error occurs during the bootloader process, and this information is stored in the flash memory.
- **How do we verify the correct initialization of peripherals?** The SIL component utilizes only two peripherals: GPIO for sending binary signals to the Simatic PLC and SPI

for communication with the ADC converter. The correct initialization of SPI is verified through the aforementioned initialization sequence for the ADC converter's initial configuration. If an error occurs, it is logged in the flash memory, and the device restarts. The functionality of GPIO pins is not verified during each device initialization; it is only tested during manufacturing as part of the firmware loading process.

- **How do we verify the correct operation of SYSFW and DMSC?** The System Firmware (SYSFW), which manages the Device Management Security Controller (DMSC), is detailed in Chapter 5.2.1. SYSFW facilitates access to system resources, including authentication mechanisms, inter-processor communication (IPC) for messaging and notifications, and mechanisms for loading software from flash memory into the RAM of individual CPUs during the boot phase. From the perspective of SIL functions, SYSFW is not a critical component. The M4 core utilizes this layer only during its startup and not during runtime. The bootloader, which primarily relies on SYSFW, does not handle safety-critical code. In the event of a failure, the DMSC implements internal mechanisms to address the error, ensuring system stability [45].

5.9.2.2 Device Operation

- **How do we verify that the software performs its intended functions correctly?** The correct transmission of binary signals to the Simatic PLC is ensured through signal duplication.

  Data reading from the ADC is verified by an initial sequence, as described in the previous section. During runtime, the ADC is not periodically rechecked. If a read error occurs, data cannot be retrieved, triggering a transition to a safety shutdown. In the case of a hardware failure, the hardware addresses the issue by disconnecting the device from its power supply.

  However, the proposed solution may significantly reduce EMC resilience. Therefore, it is advisable to consider implementing a mechanism in the future that ensures an error must occur multiple times before triggering a shutdown.

  Other functions employ standard-defined approaches outlined earlier, such as defensive programming, to prevent potential errors.
- **How do we ensure the memory operates correctly?** Memory verification is not required for SIL2 compliance. However, the memory is monitored by the Internal Diagnostic Module, as described in the hardware introduction (4.3.2). If code corruption occurs during runtime, the device addresses this by transitioning to a safe shutdown state, performing a reboot, and reloading the RAM for all cores from flash memory.
- **How do we confirm that the clocks and PLL function correctly?** Unlike initialization, long-term discrepancies during runtime on the R5 cores can be detected using the Internal Diagnostic Module, which operates in both the MAIN and MCU domains. This module includes the Dual Clock Comparator (MCU-DCC), which assesses the accuracy of the clock signal during application execution. Additionally, an external debugger detects persistent errors by expecting regular responses. If the expected message is not received, the debugger disconnects the device from its power source, effectively shutting it down.
- **How do we verify that GPIO functions correctly?** In the SIL component, GPIO and SPI are critical. SPI verification was addressed previously. For GPIO, I rely on hardware protection implemented through the required combination of multiple signals to enable positive transmission, illustrated in the Figure 5.30. Specifically, this involves

the output from the ESM and an output from the power management unit responsible for the device's power supply. These three signals are combined in a logical AND operation. The initial configuration is verified only during initialization, and the software does not modify the configuration during runtime.

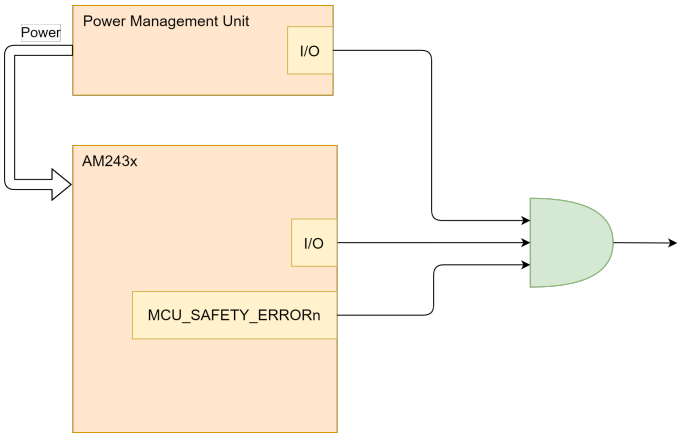

Figure 5.30. The schema illustrates the GPIO logical safety connection with ESM Status Output and Power Management Unit Status Output.

- **How do we ensure that firewalls and inter-CPU communication function correctly?** Firewall settings do not require runtime verification, as they are configured only during initialization, with testing described earlier. Inter-CPU communication is validated during initialization using predefined sequences. If the non-SIL component does not receive a message from the isolated core within the specified deadline, it assumes an error has occurred and transitions to a safety shutdown, which addresses the situation.

  It is necessary to consider whether the options for device termination and reboot are sufficient. The M4 core is isolated and executes the safety function. However, if the R5 core fails, only the monitoring and error logging to flash memory will be affected. The question is whether such an error warrants halting railway traffic by terminating the device. The proposed system is undoubtedly safe, but to enhance system trustworthiness and availability, it would be advisable in the future to design a mechanism that merely notifies of the error without shutting down the entire system.
- **How do we ensure secure Ethernet communication?** The Ethernet network is isolated from external access and operates within a physically secured environment to minimize the risk of unauthorized access. For communication beyond the local network, a dedicated gateway unit is used. This gateway enforces encryption standards (e.g., TLS) and implements secure communication protocols. To further enhance security, HTTPS with properly configured TLS certificates should be used for all web-based interfaces. Regular certificate rotation and the use of certificate pinning can help prevent man-in-the-middle attacks. Physical access to networking equipment should be restricted to authorized personnel only, with access controls such as locked cabinets, surveillance, and audit logs in place.

5.9.2.3 Device Termination

- **How is device termination ensured in the event of an error?** Termination is achieved through an external watchdog that expects periodic confirmation messages. If these messages are not received, the watchdog disconnects the power supply, resulting in

device termination. Thus, if the SIL core does not operate as expected, the external hardware automatically initiates a shutdown.

The termination process is discussed in greater detail in Chapter 5.4, which describes the safety shutdown mechanism.

5.9.3 Functional Safety Mechanism Examples

The AM243x device provides a range of mechanisms to mitigate random failures. Some are purely hardware-based but require software activation, while others are pre-implemented by TI, leaving the system integrator to decide whether to utilize them. These mechanisms, along with principles for designing peripheral hardware and development processes, are detailed in the Functional Safety Mechanism Manual.[23]

5.9.3.1 Access Management Using Firewalls

The interconnect subsystem incorporates firewalls, primarily at the slave termination points, to restrict resource access to authorized masters based on their privilege ID (privID), transaction privilege, and transaction type. Unauthorized transactions are blocked, with writes failing to complete and reads returning zeros. Such attempts are logged, and an error event is generated. These features can be tested through software by initiating transactions that violate firewall permissions. This mechanism is employed for inter-core communication and to secure shared memory, as detailed in Chapter 5.8.

5.9.3.2 Alignment Error Detection

If an Ethernet frame contains an uneven byte count (not a multiple of 8), the module flags an Alignment Error. Error response and software requirements are defined by the system integrator.

5.9.3.3 Auto Mode CRC Check

The Multi-Channel CRC (MCRC) Controller, when operated in Auto mode, combines Direct Memory Access (DMA) with Cyclic Redundancy Check (CRC) calculations. This hardware accelerator offloads the CPU, increasing its capacity for other tasks. Additionally, it supports high-speed operations, enhancing system efficiency.

5.9.3.4 Bit Multiplexing in Memory Array

SRAM modules implement a bit multiplexing scheme, ensuring that bits forming a logical word are not physically adjacent. This reduces the likelihood of physical multi-bit faults manifesting as logical multi-bit faults, instead presenting as multiple single-bit faults. In Single Error Correction and Double Error Detection (SECDED) SRAM, where Error Correcting Code (ECC) corrects single-bit faults, this scheme enhances ECC diagnostic effectiveness. Similarly, in parity SRAM, it improves parity diagnostic reliability. Bit multiplexing is a mandatory architectural feature and cannot be modified by software.

5.9.3.5 Dual Clock Comparator (DCC)

One or more Dual Clock Comparators (DCCs) are implemented as versatile safety diagnostics to detect incorrect clock frequencies or drift between clock sources. Each DCC consists of two counter blocks: one serves as a reference time base, and the

[23] Note that this manual covers both the AM64x and AM243x processor families. The AM243x series, used in this work, is significantly reduced in functionality. Thus, it is essential to verify that specific features are implemented. For example, Lockstep mode is supported only on AM64x, not AM243x, despite the presence of dual ARM R5 cores designed to enable this mode.

other tests the target clock. Both reference and test clocks, as well as the expected frequency ratio, are software-configurable. Deviations from the expected ratio trigger an error signal to the Error Signaling Module (ESM). DCC diagnostics are disabled by default and must be enabled via software. In our case, DCC is used to detect clock timing errors, preventing issues such as communication failures with ADC modules or malfunctions in internal components.

5.9.3.6 External Monitoring of MCU Safety Error Pin

The *MCU_SAFETY_ERRORz* pin, part of the MCU domain's ESM, significantly enhances safety, as discussed in Chapter 5.4.2. This signal indicates the reset error state of the MCU domain. An external monitor can detect expected or unexpected changes in the reset signal's state. Error response, diagnostic testability, and software requirements depend on the external monitor chosen by the system integrator. The AM243x also supports additional external monitoring via signals such as *SOC_PROC_OUT*, which indicates resets in the MAIN domain, and SYSCLKOUT and OBSCLK, which enable monitoring of internal clock signals using external peripherals. Further external monitoring devices, such as PMIC and watchdog, are detailed in Chapter 5.4.

5.9.3.7 Flash Authentication Protection and Flash ECC Protection

Encryption and authentication are applied to data blocks stored in attached flash memory (e.g., Hyperbus or Octal SPI). Error Correcting Code (ECC) enhances protection against soft errors by detecting and correcting single-bit errors and detecting double-bit errors. Flash ECC is calculated on 32-byte or 36-byte data blocks, with the block address included in the ECC calculation for 32-byte blocks. These functions are available only in Non-Bypass Mode.

5.9.3.8 Illegal Operation and Instruction Trapping

The Cortex-R5F processor includes diagnostics for illegal operations and instructions, serving as safety mechanisms. Many of these traps are disabled after reset and must be configured during R5 processor initialization. Examples include traps for illegal instructions, floating-point underflow/overflow, floating-point division errors, and privilege violations.

The AM243x supports numerous additional safety techniques, such as Over-temperature Warning, Information Redundancy through various approaches, and periodic checks of memory and peripheral functionality using register readback.

5.10 Modular Architecture and Component Design

The project is not built on an existing platform, necessitating the development of all components, including those responsible for launching the application, such as the Flash Writer (Chapter 5.2.10) and Bootloader (Chapter 5.2.7). For clarity, the project's components, or modules, are described below. Each module resides in its own repository, submodule, or subrepository within the project. The design philosophy enables independent versioning of each module, allowing new releases without impacting other components. For instance, although both the Bootloader and non-SIL software run on the R0 core, they are maintained in separate modules. The Bootloader is intended to remain fixed at a single version, while the non-SIL software may undergo regular updates.

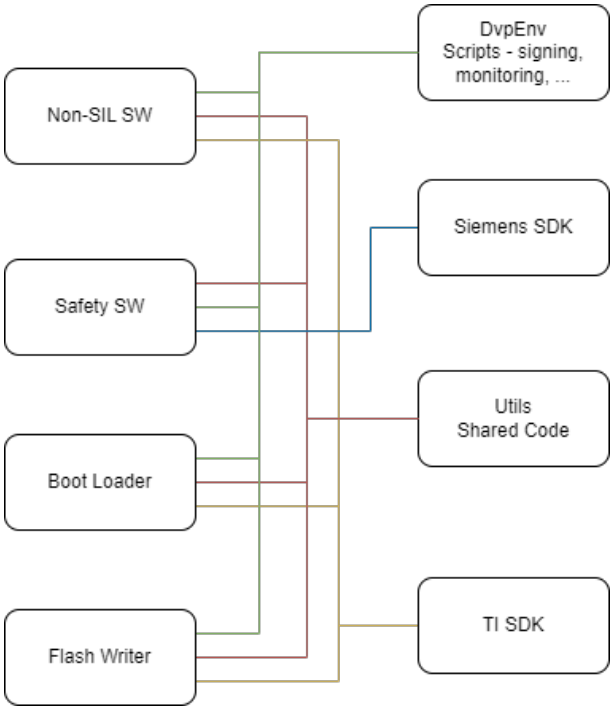

Figure 5.31. The diagram illustrates the module structure for the entire project.

The modules, along with their dependencies, are illustrated in Figure 5.31. The following outlines each component:

- **Non-SIL Software**: This module encompasses the non-SIL software, producing software images for the R5 cores. Its behavior is detailed in Chapter 5.7.
- **Safety Software**: This module contains the SIL software, generating an image for the M4 core to fulfill safety-critical functions. Its behavior is elaborated in Chapter 5.6.
- **Bootloader**: Operating on the R0 core, this component loads the SIL and non-SIL software images into the CPU's internal RAM. Its functionality is described in Chapter 5.2.7.
- **Flash Writer**: This module facilitates the initial software upload via Ethernet in the factory. It collaborates with the Factory Test Runner to verify hardware functionality post-upload, as detailed in Chapter 5.2.10.
- **DvpEnv**: This module includes project and automation scripts to streamline development, such as pre-commit hooks. It also contains scripts used across modules for tasks like compiling, signing images, monitoring devices, and decoding TI Trace Logs.
- **Siemens SDK**: This component provides pre-implemented functions from other projects using the same hardware.
- **Utils (Shared Code)**: This module houses libraries shared across multiple components, such as ADC module communication and communication frameworks.
- **TI SDK**: This refers to the Texas Instruments MCU PLUS SDK for the AM243x Sitara series. All components, except the M4 core, rely on this SDK. The M4 core uses a custom implementation, with register configuration performed via direct register writes, bypassing the TI SDK's HAL layer.

5.11 Architecture Notes

The selected AM243x device does not utilize the described architecture to its full potential. The software primarily runs on two cores: one isolated and one real-time.

To reduce costs, it could be advantageous to equip the final system with the AM2432 variant, which includes only a single pair of real-time R5 cores.

Chapter 6
Prototype Implementation

In this chapter, I describe the implementation and development of the prototype device, analyzing in detail the specific challenges encountered and the solutions devised to address them.

6.1 Prototype Specification

The objective of the implementation phase of my project was to design and develop software for a hardware prototype intended for testing hardware components. To accelerate development, software development commenced concurrently with hardware development. The development was conducted on the previously mentioned AM243x LaunchPad development board.

The prototype must be capable of communicating with an ADC module via SPI, managing GPIO, facilitating communication between the isolated and real-time cores, and reporting measured data via Ethernet. The prototype's goals include enabling device booting, establishing a development environment, and implementing device debugging. When developing software for the isolated core, it is ideal to adhere to the guidelines outlined in the relevant standard to simplify future safety-critical development.

6.2 Architecture Overview

The prototype's structure comprises the main Sitara AM243x MCU and the following components:

- **XDS110**: An integrated debugger used for code monitoring.
- **ADC Module**: Communicates via SPI.
- **UART1** Interface: Facilitates system-level log transmission.
- **GPIO**: In this case, directly connected to onboard LEDs, serving as a user interface.
- **Gateway**: Transmits measured data via an Ethernet interface.

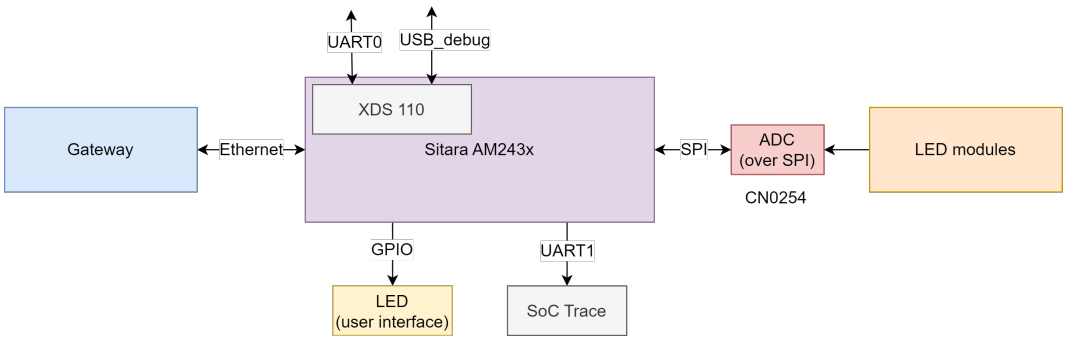

Figure 6.1. The diagram illustrates the prototype architecture overview.

The project's software is divided into the following three components:

- **Flash Writer**: Also referred to as Uniflash, this component handles the uploading of firmware to flash memory. In the prototype, it uses UART for this purpose. Further details are provided in Chapter 6.4.
- **Bootloader**: Referred to in the implementation as SBL OSPI Multi-Partition, it manages the boot process. Specifically, it ensures that firmware is loaded from flash memory into the RAM of individual CPUs and verifies the X.509 certificates used to sign software images. Further details are provided in Chapter 6.5.
- **System**: This component includes subcomponents containing software images for each core, responsible for managing the application's runtime.

6.3 Development Environment and Build System

The development was built upon the TI SDK[1] for AM243x and AM64x. TI recommends using its IDE, Code Composer Studio (CCS),[2] which is based on Eclipse, and also offers a newer version called Theia,[3] which is more similar to the VS Code design. Personally, I did not use either IDE for development due to two reasons: first, I encountered issues with missing components during installation, preventing a swift setup; second, I found it more efficient to use a build system like CMake or Make for better portability, such as in automated testing or pipeline builds, where a build system simplifies execution compared to an IDE. Instead, I used Visual Studio Code for development and Makefiles for building, which the TI SDK also employs.

However, I utilize Code Composer Studio for debugging, as it supports communication with TI's integrated XDS110 debugger. This is discussed further in Chapter 6.6. To ensure the project functions correctly, additional tools must be installed, on which the system depends. These include:

- **Code Composer Studio (CCS)**: As mentioned, it is used for debugging and is capable of running additional tools, such as the energy monitor for measuring device power consumption based on its activities, or installing dependencies, including drivers for XDS110 debugger communication.
- **TI SysConfig**: SysConfig is a configuration tool designed to simplify hardware and software configuration challenges to accelerate software development [52]. A detailed description is in Chapter 6.3.1.
- **TI ARM Clang**: For software compilation, specifically version 4.0.1.[4]
- **Node.js**: [5] This web server runs the SysConfig application, enabling code generation as an API between the TI SDK and my code.
- **Doxygen**: For generating SDK documentation.
- **OpenSSL**: For signing software images with X.509 certificates.
- **Python3**: For running scripts, such as flashing software via the flashwriter, decoding SoC Trace logs, or signing software images.

[1] Specifically, this refers to the mcu_plus_sdk, available at `https://www.ti.com/tool/MCU-PLUS-SDK-AM243X` or in the GitHub repository `https://github.com/TexasInstruments/mcupsdk-core`. However, I do not recommend using the GitHub version, as it is not regularly maintained, TI does not incorporate GitHub Issues into its development process, and the repository lacks properly resolved dependencies, requiring user intervention.

[2] `https://www.ti.com/tool/CCSTUDIO`

[3] `https://www.ti.com/tool/download/CCSTUDIO-THEIA/1.5.1`

[4] Available at `https://www.ti.com/tool/download/ARM-CGT-CLANG/4.0.1.LTS`

[5] TI specifies version 12.18.4 LTS, though I used a newer version.

- **Uniflash**: [6] This component is less critical. I used it occasionally to inspect flash memory contents.

6.3.1 System Configuration Tool (SysConfig)

An application serving as a configurator for AM243x, similar to STM32CubeMX, providing a graphical interface for configuring hardware resources like peripherals and clocks. However, its workflow differs significantly. While STM32CubeMX is primarily used for initial C file generation and occasional reconfiguration, TI SysConfig is employed in every build. Unlike STM32CubeMX, which generates source code with commented sections, SysConfig produces source files not intended for developer modification, acting as an API between the TI SDK and my code.

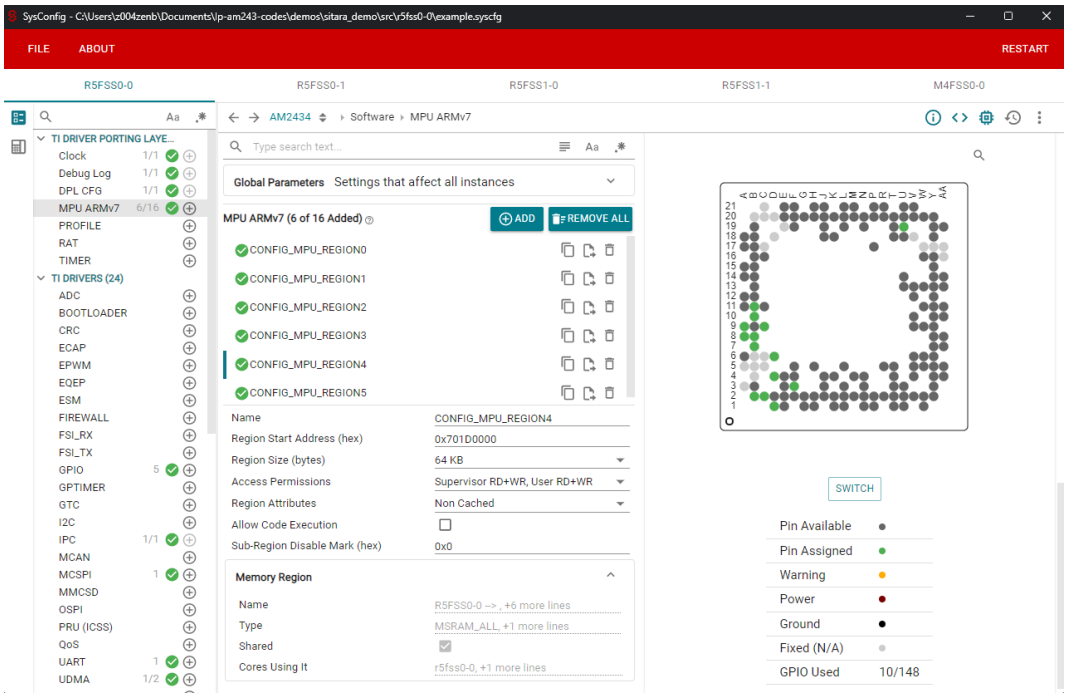

Figure 6.2. The screenshot illustrates multi-cores project configuration in the SysConfig tool.

The SysConfig environment significantly saves time by eliminating the need to study all registers in detail for testing, thereby greatly accelerating prototype development. However, I encountered two challenges that require attention.

The AM2434 is a multi-core system, and ideally, we would want to configure all cores within a single project. This is beneficial for tasks such as memory allocation, inter-core communication via the IPC mechanism, and firewall configuration. Achieving this in SysConfig is not entirely straightforward. When opening the application, you can load a configuration from a pre-existing configuration file, each specific to a core and bearing the *.syscfg* extension. However, this approach will not work for multi-core projects, as SysConfig's graphical interface does not support opening such projects. To address this, the application must be launched via the terminal with a special argument that links individual configuration files. In my project, this script is integrated into the Makefile located in the *system_nortos* directory.

The second limitation of the application is its inability to run two instances simultaneously. Practically, if you need to open the configurator for the system (all

[6] `https://www.ti.com/tool/UNIFLASH`

cores) and the bootloader concurrently to verify configurations, this is not natively supported. The issue stems from the application running on Node-Webkit, which does not allow multiple instances. A workaround exists: launch one instance, then modify the unique name for a new instance in the temporary data.[7] This enables running the application in multiple windows.

6.3.2 Build Process

For building, I utilize pre-existing and modified Makefiles from the TI SDK, all orchestrated by a single overarching Makefile that enables the execution of the following targets:

- **app**: Builds the system, generating images for each core.
- **app-syscfg**: Launches the SysConfig application for the system, configuring each image.
- **bootloader**: Builds the bootloader, specifically the *sbl_ospi_multi_partition* directory for the R5F0-0 core in my project.
- **bootloader-syscfg**: Launches the SysConfig application for bootloader configuration.
- **uniflash**: Builds the Uniflash component, i.e., the flash writer.
- **uniflash-syscfg**: Launches the graphical SysConfig interface for Uniflash configuration.
- **libs**: Builds the libraries included in the TI SDK.
- **flash**: Executes a Python script to upload firmware to the board.
- **sysfw**: Rebuilds the system firmware, required when enabling SoC Trace.
- **sysfw-config**: Opens the graphical configuration interface for system firmware.
- **sbl**: Rebuilds the Secondary Boot Loader (SBL), necessary when enabling SoC Trace.
- **sysfw-trace-log**: Parses log files obtained during SoC Trace.

The main Makefile also includes composite targets that combine individual targets, such as **all**, which natively builds the entire system (excluding sysfw and sbl) and initiates the flashing process, or **build**, which performs the same tasks as **all** but omits the flashing step. The Makefile is designed to support parallel building for improved efficiency.

The build process for each core's image is defined by a Makefile specific to each subproject. It generally consists of three steps:

- **Generation of Files**: Creating files defined in the *.syscfg* file using TI SysConfig.
- **Compilation and Linking**: Producing a binary file for the respective core. This process also generates a *.map* file, which describes the device's memory layout and is useful for debugging memory-related errors.
- **Signing and Image Creation**: Signing the generated binary file and creating an image that includes an X.509 certificate and the binary code.

6.3.3 Build Workflow

For development purposes, the following workflow is recommended for building, uploading, and monitoring the device. On the AM243x LaunchPad board, it is necessary to configure the *UART BOOT MODE* via hardware switches and power on the device. Next, I recommend opening the UART console, accessible through the integrated XDS110 debugger. In this mode, the ROM Code sends a "C" symbol

[7] For example, change the *name* variable in the *package.json* file, located in the SysConfig installation folder on Windows.

every 2–3 seconds and, upon restart, a hash code indicating the ROM Code's current version. The transmission of "C" symbols confirms the correct mode selection and functional power supply.

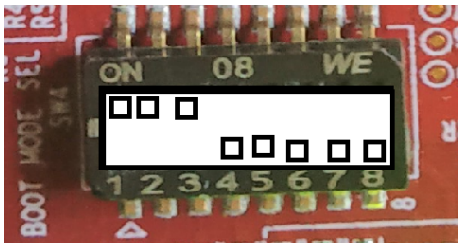

Figure 6.3. The image illustrates the *UART BOOTMODE* configuration.[8]

Subsequently, use the Makefile to compile all software and initiate the upload to the device. To upload software to flash memory, the UART console monitoring must be disabled. Then, switch the device to *QSPI BOOT MODE* and restart it. I recommend performing a full system reset using the *SOC_RESET_REOZ* button or resetting the M4 core, which also triggers a system-wide reset via *MCU_PORZ*.

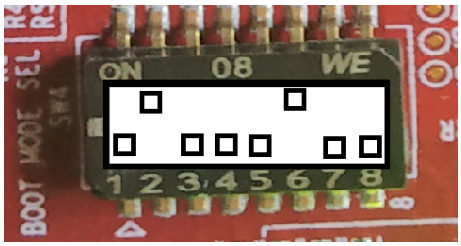

Figure 6.4. The image illustrates the *QSPI FLASH BOOTMODE* configuration.[9]

During the reset, I advise reopening the terminal to display information sent by the software to the debug console, provided debugging via UART is enabled.

6.4 Flash Writer

The Uniflash component is tasked with storing software images for individual cores in flash memory. Data is transferred via UART using the Xmodem protocol, which is detailed in Chapter 5.2.10. A Python script, included in the TI SDK, manages the upload process on the PC side. This script accepts arguments that define its behavior and influence the data layout in flash memory. For ease of configuration modification, these arguments are specified in a file named *sbl_ospi_partition.cfg*, which is passed as an argument within the Makefile. Let us examine its key components.

The argument *–flash-writer=<image location>* must always be defined first, specifying the location of the generated software image for the Uniflash component. Next, the bootloader's software image location is defined, which must be placed at an offset of *0x000000*. Subsequent arguments specify the locations and offsets for each core's generated software image, for example, *–file=<image location> –operation=flash –flash-offset=0x280000*.

[8] Image source: `https://software-dl.ti.com/mcu-plus-sdk/esd/AM243X/latest/exports/docs/api_guide_am243x/GETTING_STARTED_FLASH.html`.
[9] Image source: `https://software-dl.ti.com/mcu-plus-sdk/esd/AM243X/latest/exports/docs/api_guide_am243x/GETTING_STARTED_FLASH.html`.

In this configuration, the flash memory is not erased; it is only overwritten. A complete erase is advisable when changing the offsets of individual software images. However, an advantage is that unchanged images do not need to be rewritten, thereby accelerating the flashing process.

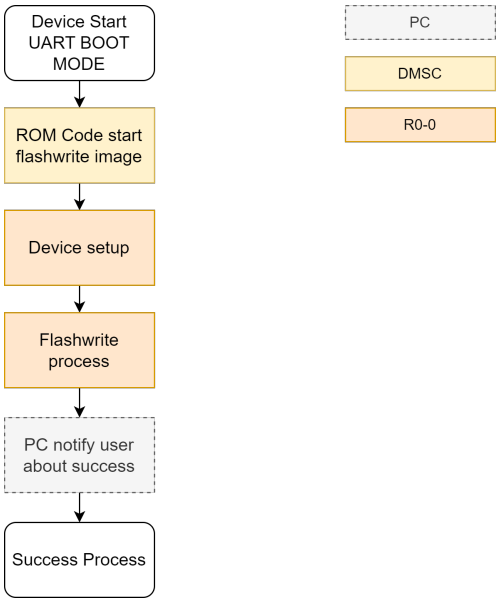

Figure 6.5. The diagram illustrates the implemented flash write process.

The entire process is illustrated in detail in Figure 6.5 and proceeds as follows: upon starting the device in UART BOOT MODE, the ROM Code loads the flash writer (including certificate authentication, as described in Chapter 5.2.6) into the *R5F0-0* core and executes it. The software initializes the device, specifically configuring clocks, OSPI for flash communication, UART for data transfer, GPIO for onboard LED status indication, and memory regions.

The OSPI interface uses the 4S-4D-4D protocol,[10] which is supported by the S25HL512T flash memory installed on the board, with a page size of 256 bytes. After initializing all necessary components, the Xmodem protocol is activated, utilizing functions defined in the TI SDK. If data is received successfully, a confirmation command is sent. If data reception fails, such as due to an overflow, an error code is transmitted to the PC.

If an error occurs during uploading, it is displayed on the terminal via the Python script. The most common solution is to restart the device and reattempt the upload.

6.5 Device Booting

The bootloader, referred to in the implementation as *sbl_ospi_multi_partition*, is always executed first. In the prototype, its role is to initialize the system, authenticate the X.509 certificate for each core's software image, and load these images into the CPU RAM for each core.

[10] Four DQ signals are used during command transfer at Single Data Rate (SDR), and four DQ signals are used during address and data transfer at Double Data Rate (DDR). Further details are available at `https://www.infineon.com/cms/en/product/memories/nor-flash/semper-nor-flash-family/semper-nor-flash/s25hl512tfabhm010/`.

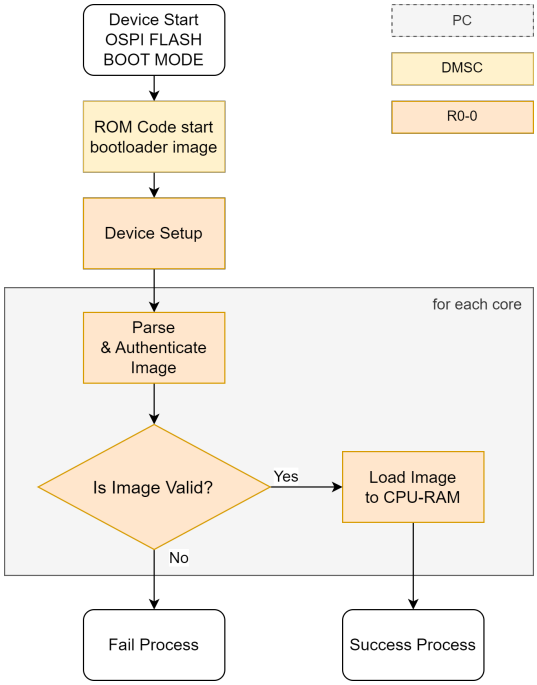

Figure 6.6. The diagram illustrates the implemented bootloader process.

The bootloader software is stored in flash memory at address *0x000000*. When configured in OSPI FLASH BOOT MODE, the ROM Code loads this memory into the R5F0-0 CPU from this address, verifies the code's integrity, and executes it. Further details on this phase are provided in Chapter 5.2.6.

In the second step, the program configures the device, specifically setting clocks and PLLs, boot addresses for individual CPUs, and OSPI communication with the flash memory, consistent with the Flash Writer implementation described in Chapter 6.4. It also configures onboard LEDs to indicate device status and sets up UART for both debugging via SDK logs and outputting SoC Trace.

The device then enters a loop to load software for each CPU into its RAM. Initially, each image is parsed, and its X.509 certificate is authenticated. If the software image passes this verification, it is loaded. If loading fails for any CPU, the process is immediately halted, and the program does not proceed further.

Authentication and loading of software utilize functions implemented within the TI SDK.

6.6 Debugging Device

Initially, I intended to use JTAG for debugging, which the device supports in combination with a Segger J-Link. However, I abandoned this approach because I could not find a J-Link configuration for the AM243x, and hardware modifications to the development board would have been necessary. While the related AM64x-LP development board supports two JTAG connections (one via the built-in XDS110 and one unconnected), the AM243x supports only one, which is occupied by the internal TI XDS110 debugger. In addition to standard JTAG pins (TDO, TDI, TMS, TCK, TRSTn), the device uses EMU0 and EMU1 pins, which are supported only by the XDS110, not the J-Link. Consequently, I opted to use the XDS110 instead of the J-Link.

The XDS110 is a cost-effective, entry-level debug probe designed by Texas Instruments for debugging microcontrollers, processors, and SimpleLink devices. It supports a wide range of TI devices through JTAG, cJTAG, and SWD/SWO interfaces. The probe includes features like Core Processor and System Trace via ETB, EnergyTrace for power measurement, and support for UART and GPIO control. It connects via a standard TI 20-pin JTAG connector and is compatible with Code Composer Studio [53].

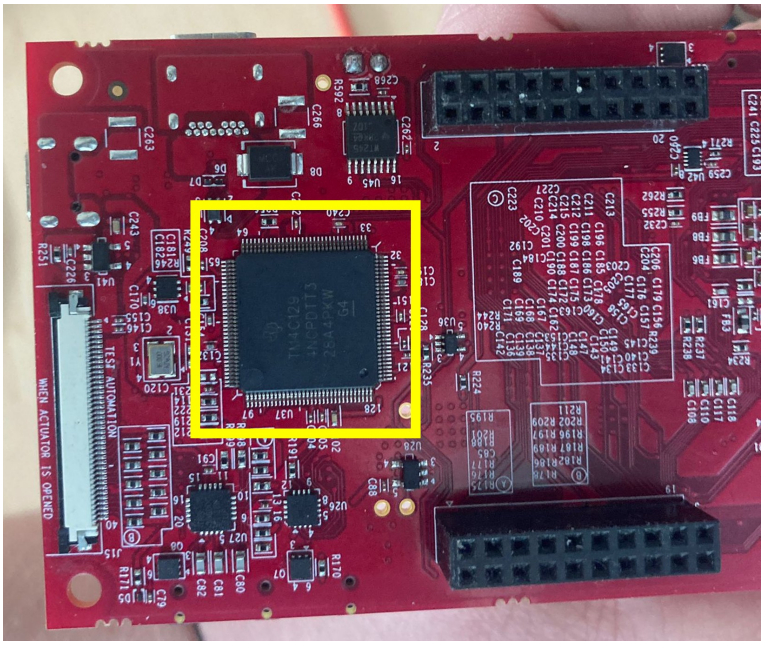

Figure 6.7. The image illustrates the XDS110 Built-in Debug Probe.

For operating this debugger, I utilize Code Composer Studio (CCS). Two configuration files with the *.ccxml* extension are defined in the project directory, specifying the hardware resources monitored by the debugger. One configuration file enables debugging, including the DMSC on the M3 core, which executes the ROM Code. The other supports debugging without the DMSC. To initiate debugging, the specific software image must be loaded into the target core. The device's BOOT MODE is irrelevant for this process, though I tested only OSPI FLASH BOOT MODE and UART BOOT MODE in this project. The debugger supports stepping through code, accessing all device registers, and displaying debug logs within the IDE. Additionally, the CCS debugger offers advanced multi-CPU debugging techniques, such as CPU grouping, which allows simultaneous operations across multiple CPUs, for example, during testing of multi-core communication.

However, the debugger operates on CMSIS-DAP, which is compatible with OpenOCD. In the future, it would be valuable to develop a configuration file to enable debugging directly in Visual Studio Code, the primary development environment. A challenge is that the AM243x is a multi-core device, which most debugging environments are not designed to handle.

6.6.1 SoC Trace

The AM243x is a multi-core processor, and thus I often refer to it as a system rather than an MCU. To debug errors at the system level, a mechanism known as SoC Trace, also referred to as SYSFW Trace or TI Trace, is available. This method

enables tracing through messages that can be converted into human-readable text. The Trace Layer provides logs from system firmware (SYSFW) components, such as firewalls and clocks.

SoC Trace data can be output via the UART interface. Each Texas Instruments (TI) device uses a specific UART port. Although the AM243x is not listed in the relevant documentation,[11] the AM64x, which uses the same UART interface (specifically UART1), is referenced.

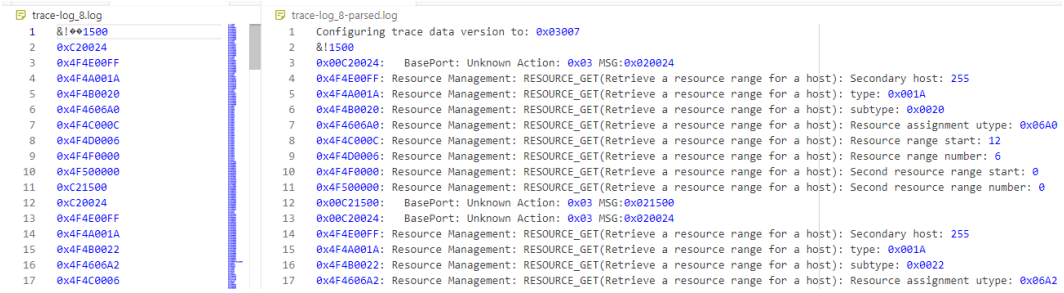

Figure 6.8. The image illustrates the received log and the parsed log from SoC Trace.

To utilize this trace layer, the UART1 interface must be enabled, along with DMSC logs in the SDK, followed by recompilation of all SYSFW files. Predefined commands in the main Makefile can be used for this purpose.

Captured log files should be saved in the format *trace-log_<number>.log* in the project's root directory. The main Makefile includes a sysfw-trace-log target that executes a script to translate machine-generated data into human-readable code. The data is parsed based on specifications in the TISCI documentation,[12] which describes the entire SoC Trace Layer.

6.7 ADC

The CN0254 evaluation board with the AD7682 ADC, manufactured by Analog Devices, was selected as the ADC module. In the final application, this module enables the measurement of voltages from LED modules. The ADC offers 16-bit resolution and 8 channels. For communication, the SPI interface is used, specifically the MCSPI on the AM243x.

6.7.1 Configuration Register

The ADC supports multiple configuration and operation modes, configured via a 14-bit register. Below, we describe the significance of each bit and the corresponding operating modes.

- **[13] CFG**: Determines configuration updates. A value of 0 indicates no change to the configuration, while 1 triggers an update of the configuration register.
- **[12:10] Input Channel**: Select the input channel mode, with five available options:
  - 0b00X[13]: Bipolar differential pairs, inputs referenced to $V_{REF}/2 \pm 0.1V$.
  - 0x2: Bipolar, inputs referenced to $COM = V_{REF}/2 \pm 0.1V$.
  - 0x3: Temperature sensor.
  - 0b10X: Unipolar differential pairs, inputs referenced to $GND \pm 0.1V$.
  - 0x6: Unipolar, inputs referenced to $COM = GND \pm 0.1V$.
  - 0x7: Unipolar, inputs referenced to GND.

[11] https://software-dl.ti.com/tisci/esd/latest/4_trace/trace.html#trace-uart-allocation
[12] https://software-dl.ti.com/tisci/esd/latest/4_trace/trace.html#trace-debug-data-format
[13] *Note*: X denotes "don't care."

- **[9:7] Channel Count**: Specify the number of channels.
- **[6] Bandwidth**: Configures the low-pass filter bandwidth. A value of 0 sets the bandwidth to one-quarter, while 1 enables full bandwidth.
- **[5:3] Reference**: Define the reference voltage source, with the following options:
  - 0x0: Internal reference and temperature sensor enabled, using $REF = 2.5V$ buffered output.
  - 0x1: Internal reference and temperature sensor enabled, using $REF = 4.096V$ buffered output.
  - 0x2: External reference used, temperature sensor enabled, internal buffer disabled.
  - 0x3: External reference used, internal buffer and temperature sensor enabled.
  - 0x4, 0x5: Unused and undefined.
  - 0x6: External reference used, internal reference, internal buffer, and temperature sensor disabled.
  - 0x7: External reference used, internal buffer enabled, internal reference and temperature sensor disabled.
- **[2:1] Channel Sequencer**: Configure the channel sequencer, which enables cyclic transmission of data from enabled channels. Four states are available:
  - 0x0: Sequencer disabled.
  - 0x1: Configuration updates enabled during sequencing.
  - 0x2: Sequencer scans values from channels IN_0 to IN_7, followed by the temperature sensor.
  - 0x3: Sequencer scans values from channels IN_0 to IN_7, excluding the temperature sensor.
- **[0] Read Back**: Determines whether read-back is enabled. A value of 1 sends the register value after data transmission, while 0 disables read-back.

6.7.2 Operation Modes

The ADC module supports SPI communication in two configurations: with and without a busy indicator. In my project, I utilize the configuration without a busy indicator to achieve cyclic communication at a constant rate, which is illustrated in Figure 6.9. The alternative configuration, which includes a busy indicator, is illustrated in Figure 6.10.

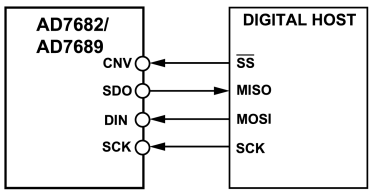

Figure 6.9. The image illustrates the SPI connection without a busy indicator.

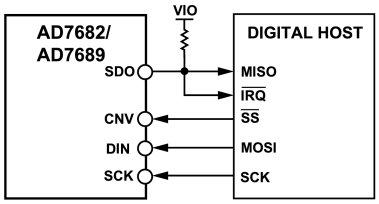

Figure 6.10. The image illustrates the SPI connection with a busy indicator.

The first configuration operates in SPI Mode 0, where both CPHA and CPOL are set to 0. This means data is sampled on the rising edge, shifted out on the falling edge, and the clock polarity in the idle state is logic low [54]. The second configuration uses SPI Mode 3, where CPHA and CPOL are both set to 1, indicating that data is sampled on the rising edge, shifted out on the falling edge, and the clock polarity in the idle state is logic high [54].

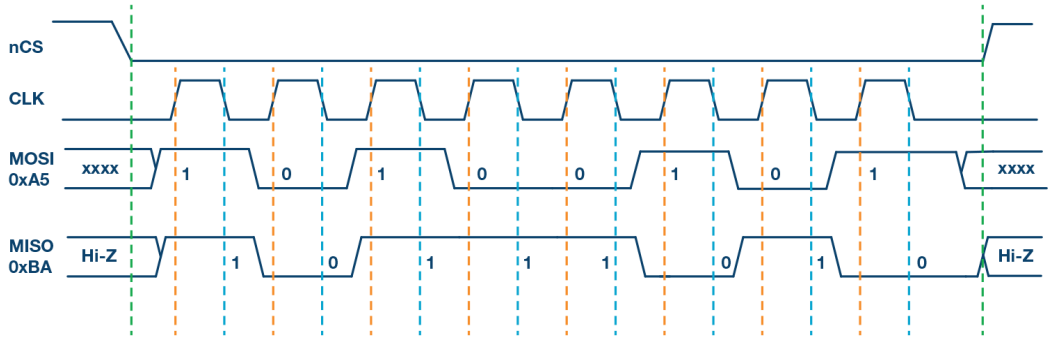

Figure 6.11. The image illustrates the SPI Mode 0, CPOL = 0, CPHA = 0, CLK idle state = low, data sampled on the rising edge and shifted on the falling edge.[14]

The operation of the ADC module can be divided into two continuously alternating periodic phases: acquisition and conversion. During the acquisition phase, the input signal is sampled and stored in internal memory. In the conversion phase, the input voltage is disconnected, and the converter transforms the measured analog value into a digital value.

The ADC supports three operating modes: read/write during conversion (RDC), read/write after conversion (RAC), and read/write spanning conversion (RSC). In my implementation, I utilize the RAC mode, which is illustrated in the timing diagram in Figure 6.12.

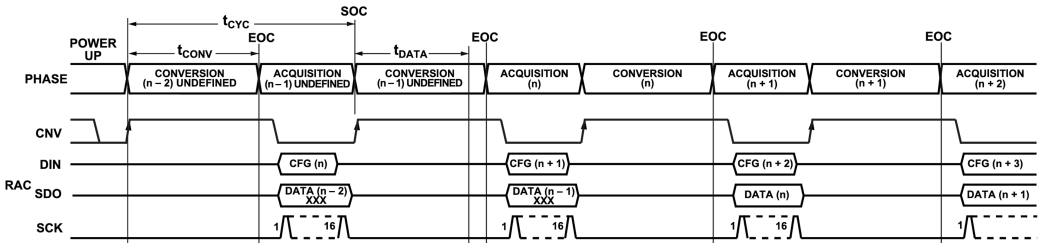

Figure 6.12. The timeline illustrates ADC SPI communication with timing in RAC operation mode without a busy indicator.[15] Consonants t_{CYC}, t_{CONV}, t_{DATA} are defined in the Data Sheet.

In RAC mode, the device operates with the CNV (conversion) signal in a logic 1 state when no data is being written. To initiate a write operation, the CNV signal is set to logic 0, followed by the exchange of 16 or 32 bits, depending on whether the write-back feature is enabled in the configuration register. Afterward, the CNV signal returns to logic 1. A two-cycle delay is required before the written configuration takes effect. This is accounted for in the implementation, where two *dummy cycles* are observed at startup to ensure proper device configuration.

6.7.3 AD7682 Driver Implementation

A dedicated driver was developed for communication with the ADC module, enabling operation in RAC mode. The driver supports two modes. The first mode utilizes the AM243x's MCSPI hardware peripheral, controlled via the TI SDK. The second mode, designed for the isolated core lacking an SPI peripheral, emulates SPI functionality through bit-banging.

[14] Image source: `https://www.analog.com/en/resources/analog-dialogue/articles/introduction-to-spi-interface.html`
[15] Illustration is edited Figure from AD7682 Data Sheet.

For reading values, the ADC's sequencer is always employed. The driver can return either raw values or values processed with a mathematical operation, providing flexibility for future prototype testing. The ADC also supports temperature readings, which are not interpreted by the driver. Interpreting temperature data requires testing on a physical prototype and analyzing the measured values.

Data is stored in a structure that associates each value with a timestamp, represented in my implementation as microseconds since the device started. Overflow is not a concern, as the timestamp uses a *uint64_t* type, which would require over 584,000 years of continuous operation to overflow.

During initialization, the driver performs a test to calculate the average data transfer duration, which is subsequently used for precise timing. If the transfer speed changes during operation, the implementation must be updated accordingly.

The driver also implements a self-test method to verify correct communication with the ADC module. This test leverages the write-back feature by performing a test configuration write, waiting for two dummy cycles, and reading the data back. If the retrieved configuration matches the written one, the test is successful. If it fails, I recommend repeating the test multiple times, as the test may be sent before the ADC module is ready to respond.

6.7.4 ADC Device Usage

The objective of the implementation phase is to develop software for a hardware prototype to be used during testing. For software development, only the AM243x, equipped with the AM2434-ALX package, is available. Unlike the AM2434-ALV package, which will be used in the hardware prototype, the ALX package lacks SPI hardware peripherals in the MCU domain. Consequently, this work required addressing this limitation. The challenge was to enable SPI functionality in the MCU domain while minimizing the effort required for future adaptations.

After studying the issue and evaluating options, I identified several potential approaches. The first was to utilize registers available in the AM2434 to create an internal interrupt that would set a specified value on a peripheral output upon writing to the register. This approach would allow future implementations to eliminate interrupts and directly use the peripheral. However, this solution was infeasible, as the AM243x manual revealed that the ALX package lacks SPI signals on the board. I consulted Texas Instruments (TI) support,[16] which confirmed that this approach was not viable.

[16] https://e2e.ti.com/support/microcontrollers/arm-based-microcontrollers-group/arm-based-microcontrollers/f/arm-based-microcontrollers-forum/1510685/mcu-plus-sdk-am243x-am243x-m4-core-spi-configuration-with-limited-uart-pins-and-internal-interrupt-routing/5825341

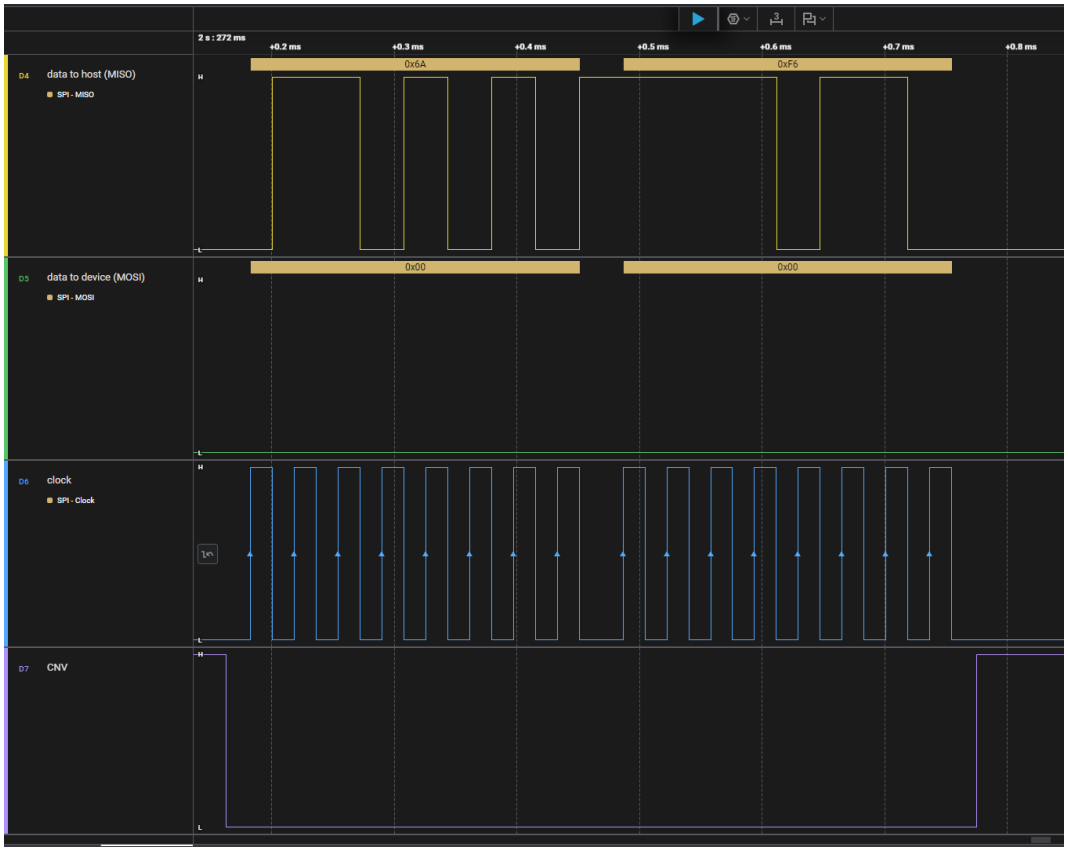

Figure 6.13. The image illustrates SPI communication analyzed with the logic analyzer.

In the MCU domain of the AM243x development board, only five ports are accessible. Four can be used for UART, and the fifth is reserved for an error code that can connect to an external watchdog device. The fifth pin is unavailable, leaving only three usable pins. Another option was to modify the UART protocol to emulate SPI behavior. However, SPI is a synchronous bus, whereas UART is asynchronous unless implemented as USART, which is not available on this board.

A third option was to simulate SPI behavior in software. This approach requires significantly reduced speed, which is not a concern in this case. However, maintaining synchronicity posed a challenge. This could potentially be addressed by configuring PWM, but none of the four available peripherals support this functionality. I implemented this software-based SPI simulation, which can be enabled via a macro acting as a switch. However, I do not recommend its use, as it is unreliable and frequently causes data corruption.

For the prototype with the ADC module, communication is handled through the R5 core.

6.8 Memory Layout

Proper operation of the prototype requires careful allocation of memory space. Although the application primarily uses the M4 and R5 cores, initializing the other cores is necessary to ensure future scalability. Let us examine the memory space allocation for the entire application in detail.

The memory for the R5FSS0-0 core is divided into nine regions. Region 0 is Tightly-Coupled Memory (TCM),[17] allocated to the vector table. Regions 1 and 2, also TCM, serve as fast-access memory, allowing the CPU to access them in every cycle. Region 4 defines the MSRAM, used as RAM for CPU operation, with a size of *0xD0000*. Region 5 corresponds to flash memory. Region 6 defines shared memory accessible by all cores. Region 7, another shared memory region, is used to distribute log messages from other CPUs to the R5FSS0-0 core. Region 8 defines shared memory for inter-core communication among all cores.

Other R5 cores differ from R5FSS0-0 in that they lack defined structures for shared memory and have a smaller MSRAM, specifically *0x10000*.

The M4FSS0-0 core has only three defined regions. As it is not a real-time core, it does not support TCM. Region 0 defines the vector table, Region 2 handles interrupts, and Region 3 uses DRAM for memory allocation.

A detailed memory map for each core can be viewed using the compiler, which generates it in a readable text format after linking.

6.9 Monitoring Output

6.9.1 Ethernet Configuration

To transmit data measured by the ADC module, I utilize the Ethernet port. The AM243x board is equipped with an Ethernet switch designated as CPSW (Common Platform Switch). One method to communicate with the Ethernet peripheral is to use the CPSW, accessible from the MAIN domain. This subsystem supports IEEE 802.3 standard Gigabit Ethernet packet communication and can be configured as an Ethernet switch. The CPSW interface supports both RGMII and RMII interfaces.

The second interface, which I employed in my implementation, is the Programmable Real-Time Unit and Industrial Communication Subsystem – Gigabit (PRU-ICSSG). This programmable firmware can emulate various peripherals, including industrial protocols such as EtherCAT, Profinet, and EtherNet/IP, as well as Ethernet Switch or Ethernet MAC. PRU-ICSSG supports both RGMII and MII modes.

The PRU-ICSSG runs the Industrial Communications Subsystem Ethernet Media Access Controller (ICSS-EMAC), which acts as a driver and provides an API for managing transmitted and received packets. The driver implements two Ethernet ports as a switch, supporting the 802.1D standard at 100 Mbps [45].

I chose this approach to reduce the load on the main core, as the process is fully dedicated to the PRU-ICSSG core. This choice was also driven by my personal interest in exploring the real-time subsystem's capabilities.

[17] Tightly-Coupled Memory (TCM) provides low-latency, deterministic access for critical operations. [55].

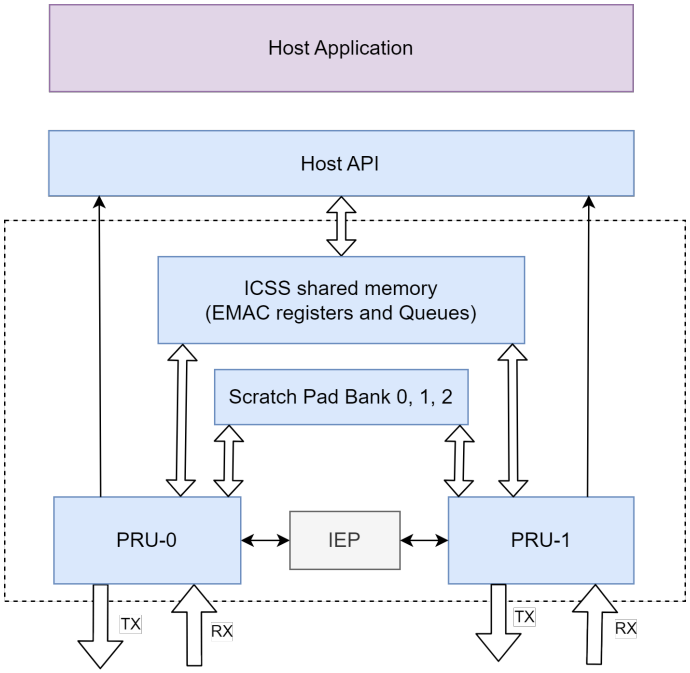

Figure 6.14. The diagram illustrates PRU-ICSSG System Decomposition [56].

The structure of the *ICSS_DUAL_EMAC* is illustrated in Figure 6.14. The implemented application communicates with the subsystem via an API that writes data to shared memory, specifically EMAC registers and queues. The PRU-0 and PRU-1 units read from these data structures to perform tasks such as receiving data on a port, transmitting on a port, collecting statistics, handling errors, and updating error counters.

For proper configuration, I recommend thoroughly studying the manual, which provides a detailed description of the subsystem's behavior. In my implementation, the device is configured in DUAL MAC mode rather than as a switch. This mode allows each port to have its own IP address, MAC address, and operate on a separate network, achieved through software-defined network masks. I used the MII interface, which is sufficient for my needs. For efficient data transmission and reception, I employed DMA, supporting up to 16 messages for transmission and 32 for reception. The MDIO module operates at a frequency of 2.2 MHz with a poll interval of 100. The transmission speed and duplex capability are set to auto-negotiate, and the same configuration is applied to the second interface.

To communicate with the subsystem, I2C instance 0 must also be initialized to enable access to the EEPROM used by the subsystem.

6.9.2 TCP Server

For the server implementation, I utilized the widely adopted open-source library lwIP, which provides a TCP/IP stack optimized for embedded systems. The device communicates using TCP packets. The implementation is based on an open-source example and supports both static IP address configuration and dynamic configuration via a DHCP server. The server operates on two interfaces: NetifIdx 0 with IP address 192.168.1.200, subnet mask 255.255.255.0, and gateway 192.168.1.1; and NetifIdx 1 with IP address 10.64.1.200, subnet mask 255.255.252.0, and gateway 10.64.1.1.

The server functions straightforwardly, transmitting data stored in a shared structure via TCP packets. I chose the TCP protocol for its reliability. Leveraging lwIP ensures that the implementation complexity remains manageable.

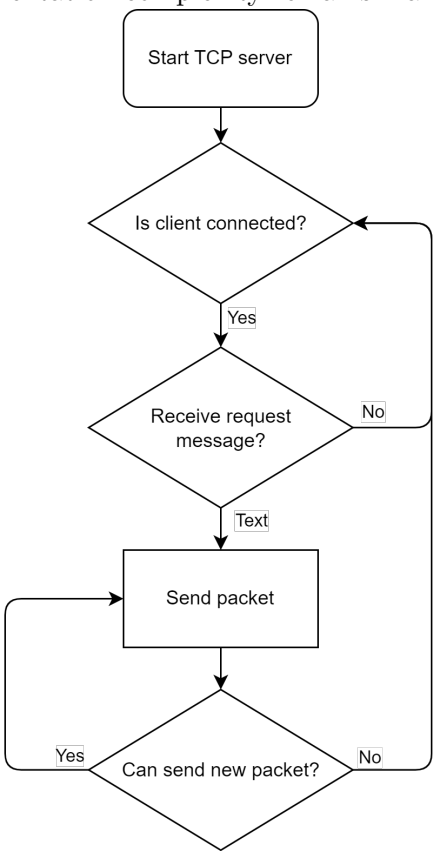

Figure 6.15. The diagram illustrates TCP Server Communication Flow.

Data transmission proceeds as follows: the server waits for a client to establish a connection. Once the client requests data, the server initiates a transmission loop, which continues until the client stops sending acknowledgments for the received data. As the Ethernet interface does not currently support switch functionality, I do not expect multiple devices to be connected to the network. Consequently, the server is not optimized for multiple concurrent connections. Due to performance requirements, the current implementation operates at approximately 70% of its capacity.[18]

6.10 Inter-Core Communication

In this prototype implementation, inter-core communication serves a testing purpose to verify that all cores are correctly initialized. The structure is, however, designed to support future implementations on a prototype where the M4 core can utilize an SPI peripheral, as detailed in Chapter 6.7.4.

Inter-core communication occurs via shared memory, structured according to the final implementation described in Chapter 5.8. Additionally, I specify the precise implementation of a cyclic buffer, illustrated in Figure 6.16. The buffer begins with a temperature value, followed by entries for the channel and timestamp indicating when the value was measured. The circular buffer consists of this structure.

[18] This value is measured using FreeRTOS.

Figure 6.16. The frame illustrates one element of the cycle buffer described in Chapter 5.8.

Communication is unidirectional, eliminating the need to manage memory access conflicts. Furthermore, no other core can write to this memory, as it is locked to all accesses except from the R5 core, as outlined in the architecture design.

6.11 Application

The implemented application differs from the introductory illustration in Chapter 6.1 in that data is read through the R5 core instead of the M4 core. However, all necessary provisions are in place for future implementations. The application is built on FreeRTOS, which enables the execution of multiple tasks with prioritized scheduling. Unlike the architecture design, its primary goal is not to respond to HTTP server queries but to transmit collected data as quickly as possible for testing and analysis. During experiments, I also implemented a web server capable of operating without FreeRTOS, returning current data with a short history (10 samples per channel) in JSON format. Due to the differing purpose, I utilized different data structures.

In the web server approach, communication involves a client sending an HTTP GET request and receiving an HTTP GET response containing JSON-formatted data. This approach benefits from shared memory, which the device can access at any time. For my implemented purpose, however, the communication follows a "open the floodgate and stream data" model—formally, the client sends a request and then waits for incoming data. Accordingly, I employed a suitable structure, specifically an *xQueue* in FreeRTOS, which facilitates data transfer between multiple tasks.

The application operates by launching a main task that configures the peripherals. Execution then splits into two tasks. The first task reads data from the ADC converters and is designed for future use to read from cyclic shared memory and forward data to a web server for client transmission. I deliberately separate the web server and shared data reading, creating an API to enhance modularity and simplify changes if multiple CPUs need to access the data. If the ADC fails to initialize correctly, the second task, which manages the web server, is terminated. The web server's operation is described in detail and best illustrated in Figure 6.15.

The application provides monitoring logs output to a virtual UART0 console, which indicate whether the ADC initialized successfully or if server-side errors occurred. The server also logs its CPU load, currently at 70% during operation.[19] The goal is to transmit data as quickly as possible to monitor subtle changes. In the final system, however, communication will likely be significantly slower due to the physical isolation of two independent channels, as defined by the safety architecture.

The application supports communication with the M4 core via shared memory or IPC notifications. However, this feature is disabled in the current implementation, as it is not relevant to the present objectives.

6.12 Monitoring Utility

For monitoring purposes, I developed a Python script that operates in multiple modes. The first mode enables storing data on the device in JSON format. In

[19] This value is measured using FreeRTOS.

this mode, the script acts as a client to the device's TCP server. Initially, it sends a packet to the IP address 192.168.1.200, requesting the transmission of additional data. The request can be in any format, as the device only verifies the client's connection without checking the request's content. The server then transmits data in the same format as stored in the frame illustrated in Figure 6.16. Individual data entries are separated by commas, and the end of the data is marked by a semicolon. Upon receipt, the script parses the data and saves it in JSON format.

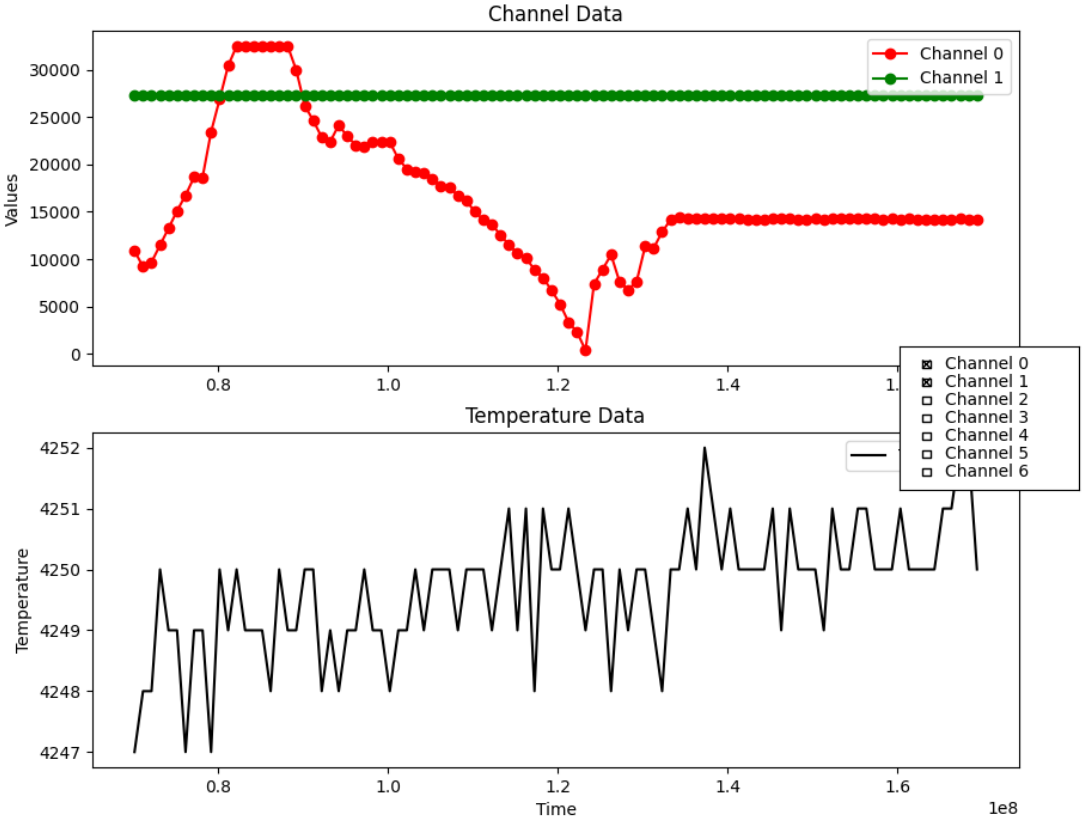

Figure 6.17. The figure displays a screenshot of the monitoring script's output, which enables real-time tracking of values on individual channels and the ability to enable or disable channels. It also shows temperature data in the lower graph. Time is represented by the number of processor ticks.

The second mode visualizes incoming data as a real-time animation while also storing it in the same JSON format. To enable this, the communication speed must be significantly reduced, or the script must be modified to ignore some received values and render only every nth frame.

The third mode loads previously stored data and generates a graph. Both the animation and graph are rendered using the Python library Matplotlib,[20] which must be installed as a dependency. The generated graph operates in an interactive mode, allowing users to select specific channels to display.

[20] `https://matplotlib.org/`

6.13 Implementation Challenges

During the implementation, beyond the challenge of the missing SPI module on the M4 core, I encountered several more complex issues. I aim to demonstrate and briefly describe them to provide inspiration for solutions in case similar problems arise.

The first issue occurred initially during X.509 certificate authentication and subsequently at every device startup. Specifically, if the device ran for more than three minutes, I was unable to successfully restart it for approximately 10–15 minutes after a reset, even after disconnecting the power supply. After considering possible causes, residual induced energy in the system seemed the likely culprit. While debugging, I also discovered a flaw in the SYSFW authentication process, which I reported to Texas Instruments. This was fixed in SDK version 11, but it did not resolve my issue. After extensive investigation, I identified the root cause: incorrect timing configuration for communication with the flash memory, likely exacerbated by component overheating. This would explain the need to wait before the system could be used again.

The second issue I frequently faced was combining C and C++ code. This was partly necessitated by the SDK and lwIP being implemented in C, while the specified requirements mandated C++ usage. Although mechanisms exist to address compatibility issues, this approach caused significant complications. Ultimately, I had to rewrite most of the code in C++, as C++ is not fully compatible with C.

During implementation, I often struggled with the compiler's optimization behavior, which sometimes reordered instructions in ways that were nonsensical or, to save space, transformed them into loops that introduced delays. This issue was most pronounced during software-based SPI emulation and when controlling the CNV pin via GPIO registers in normal operation. Several solutions were considered. The first was to disable compiler optimizations for specific code sections using pragmas defined by the ARM-based TI Clang compiler.[21] The second was to write the sequence directly in assembly, but this was not used, as it violates *coding standards* and would be impractical for the final system. The third and preferred solution was to leverage hardware peripherals, such as PWM or timers combined with interrupts, to avoid reliance on compiler optimizations. Compiler instruction reordering can also complicate debugging, so it is important to account for this issue.

[21] By official TI Compiler description with `"pragma FUNCTION_OPTIONS ( func, additional options )"` [57].

Chapter 7
Testing and Higher SIL Level Discussion

7.1 Device Testing

The project is currently at a stage where the hardware prototype is awaited, and my practical task was to develop a software prototype for testing purposes. This thesis minimally focuses on procedural aspects of development. However, validation and testing of the developed system are significant components of the development cycle. Therefore, I will briefly describe the tests most suitable for the system.

Numerous tests exist, but I will highlight those most relevant to my device, categorizing them into two groups: software testing and system testing.

7.1.1 Software Testing

Software testing aims to identify as many errors as possible during development to prevent potential issues. According to the EN 50128 standard for my SIL 2 level, testing is not mandatory. However, I strongly recommend it for SIL component development. From my perspective, unit tests are particularly valuable, as they verify the correctness of the implementation.

A unit test verifies that an implementation meets specified requirements [58]. For example, it involves executing a function or method with known inputs and comparing the output to the expected result. If the output matches, the risk of implementation errors is reduced. Unit tests are especially useful for testing edge cases, extreme scenarios, and error conditions.

When using unit tests, pair programming and task division can be beneficial. One person writes the unit test, specifying the expected behavior, while another implements the function or method. This approach increases the likelihood of detecting errors if one person misunderstands the function's purpose, as discrepancies between the test and implementation will reveal the issue.

Other software testing methods include tools for validating memory usage, such as Valgrind[1] or Sanitizer,[2] which can detect data race conditions and deadlocks, though Sanitizer is no longer actively developed.

Software testing should also involve monitoring vulnerabilities in dependencies. This includes tracking whether security threats emerge in third-party libraries, such as lwIP or FreeRTOS in my case, which could pose a security risk to the software.

7.1.2 System Testing

The second type of testing is system testing, which aims to verify the device's behavior within the complete system. This involves deploying the release version of the software on the device, simulating its operation, and observing whether it performs

[1] `https://valgrind.org`
[2] `https://github.com/google/sanitizers`

according to the expected specifications. For our device, this practically means controlling LED modules by turning them on and off, with the device correctly evaluating their state and transmitting the data to a Simatic PLC.

System testing also includes stress tests, where external conditions are applied to assess the software's behavior. These tests are conducted not on the released software but on a specially modified version designed to trace the logical paths taken during execution, such as which conditional statements (e.g., if statements) determine the program's flow.

It is advisable for these tests to be conducted by an independent tester, as defined by the standard, rather than the developer. This approach increases the likelihood of detecting errors that might result from human oversight.

7.1.3 Type Testing

Additional system tests include type tests, which evaluate the device's performance in extreme environments that could compromise its reliability, such as high temperatures, vibrations, or electromagnetic compatibility (EMC) issues that may disrupt communication or cause memory errors. The device should be resilient to these conditions to a certain extent.

7.2 Higher SIL Level Discussion

The proposed architecture complies with SIL 2 requirements. It leverages an architecture based on an isolated control device (our AM243x) and two independent channels, enabling the system to achieve SIL 4 compliance. To operate at SIL 3 or higher, a fundamentally different architectural approach would be required. According to Texas Instruments' Safety Manual, the AM243x can achieve SIL 3 under specific conditions but cannot reach SIL 4. These conditions are detailed in the manual's statements. The proposed approach employs a reactive safety method, where the device's operation is monitored, and in case of a fault, it is transitioned to a safe state. An alternative for achieving higher SIL levels would be to develop a platform with multiple independent units that communicate with each other, utilizing a composite safety architecture.

Generally, achieving SIL 3 or SIL 4 with the AM243x would require either two independent units or an external watchdog device, such as an advanced PMIC discussed in Chapter 5.4. However, higher SIL levels cannot be achieved with the AM243x alone.

If requirements were to change to demand a higher SIL level, a comprehensive redesign of the architecture and process allocation would be necessary, including an evaluation of whether the device would benefit from an isolated core. An isolated core could serve as an advanced internal watchdog, leveraging its ability to restart independently of the MAIN domain and operate with separate power sources.

The discussed mechanisms would likely lead to the development of a platform providing safety-critical services applicable across multiple projects, rather than a single device within a system. Developing such a platform would necessitate a different operating system. While FreeRTOS can operate at SIL levels, a more suitable choice for larger projects would be a real-time operating system like Zephyr.[3]

[3] `https://www.zephyrproject.org/`

Chapter 8
Conclusion

The objective of this diploma thesis was to study fail-safe mechanisms defined in the standard for achieving SIL 2 and SIL 4 levels, select suitable hardware, design a device architecture that meets the specified requirements, implement a prototype for testing purposes, and discuss the steps needed to achieve higher SIL levels. I described the individual mechanisms in Chapter 2, addressed the device requirements, hardware selection, and software architecture design in Chapters 3, 4, 5, detailed the implementation in Chapter 6, and discussed mechanisms for verifying the implementation's correctness and strategies for achieving higher SIL levels in Chapter 7.

The practical outcomes of my work include not only a deeper understanding of the subject matter and comprehensive familiarity with the AM2434 processor but also a prototype implementation that can be used to control a prototype device, verify its properties, and establish build, booting, and flash-write processes for firmware deployment to the device.

A limitation of my work was the inability to reliably establish SPI communication on the isolated core of the development kit. Personally, I found little value in pursuing a more robust implementation using timers and interrupts, which could have been more reliable than pure emulation, as the prototype hardware intended for the software is expected to include this peripheral.

Throughout the project, I collaborated with Siemens Mobility, for whom I developed the prototype. I greatly value this collaboration, which was both educational and rewarding. A significant but less visible aspect of my work was studying the architecture, behavior, and operation of the AM243x, specifically the AM2434, whose multi-core structure is non-trivial. The collaboration provided substantial benefits, including excellent facilities, technical support, and consultation opportunities. However, it also introduced process-related challenges, primarily due to the project's early stage, where colleagues were simultaneously designing the hardware. My work prompted several insights that necessitated adjustments, requiring me to adapt throughout the thesis.

This project marked my first practical encounter with safety-critical development, which involves studying numerous manuals and standards, frequent discussions on the validity of design choices, and rigorous formal verification of details. While this approach can slow progress, it offers the benefit of enabling a deep and comprehensive understanding of the subject matter. It was also my first experience with the Texas Instruments ecosystem and a processor with such an extensive set of peripherals that it requires dedicated firmware to manage them.

I consider the project successful. I met all specifications and delivered software that is not merely theoretical but has practical applications in a project that may one day control railway crossings across Europe.

References

and

[1] *The Future of Rail*. In: Paris: IEA, https://www.iea.org/reports/the-future-of-rail.

[2] *ENERGY EFFICIENCY*. In: UITP Europe, 2. https://web.archive.org/web/20160812185814/http://www.uitp.org/sites/default/files/cck-focus-papers-files/Energy%20Efficiency%20-%20Contribution%20of%20Urban%20Rail%20Systems.pdf.

[3] Jaroslav Gavenda. *Česko poslalo Ukrajincům tanky z 80. let, určitě jim pomohou, říká expert*. In: Seznam Zprávy, 2022. https://www.seznamzpravy.cz/clanek/domaci-politika-cesko-poslalo-na-ukrajinu-tanky-z-80-let-chybet-nebudou-rika-analytik-197464.

[4] H. Haralambides. *The Belt and Road Initiative*. In: *International Transport Forum Discussion Papers*. Paris: OECD Publishing, 2020. https://doi.org/10.1787/2281f06f-en.

[5] *Rail*. 2023. https://www.iea.org/energy-system/transport/rail.

[6] Alice Lunardon, Doroteya Vladimirova, and Benedikt Boucsein. How railway stations can transform urban mobility and the public realm: The stakeholders' perspective. *Journal of Urban Mobility*. 2023, 3 100047. DOI https://doi.org/10.1016/j.urbmob.2023.100047.

[7] The Middle Trade and Transport Corridor: Policies and Investments to Triple Freight Volumes and Halve Travel Time by 2030. 2023.

[8] *Passenger Capacity of different Transport Modes*. https://www.transformative-mobility.org/wp-content/uploads/2023/03/Passenger-Capacity-of-different-Transport-Modes_2021-09-08-071924_mmuh-AQ55yh.pdf.

[9] *SECURITY | English meaning - Cambridge Dictionary*. https://dictionary.cambridge.org/dictionary/english/security.

[10] *SAFETY | English meaning - Cambridge Dictionary*. https://dictionary.cambridge.org/dictionary/english/safety.

[11] Xiangwan, Pooja Purohit, and M. Mueed Jamal. Case Study Eschede Train Disaster-1998. *Department of Materials Science and Engineering, The University Of Sheffield*. March 19, 2011, 2-11.

[12] *Tagging for Track Components and Artificial Vision Inventory Systems*. 6 December 2023. https://uic.org/projects-99/article/tagging-for-track-components-and-artificial-vision-inventory-systems.

[13] *Global status report on road safety 2023*. In: Geneva: World Health Organization, 2023. 4. ISBN 978-92-4-008645-6.

https://iris.who.int/bitstream/handle/10665/374868/9789240086456-eng.pdf.

[14] *New standard EN 50716:2023 «Requirements for software development» Part 1.* https://www.csa.ch/en/blog/new-standard-en-507162023-requirements-for-software-development-part-1.

[15] *Railway Applications - The Specification and Demonstration of Reliability, Availability, Maintainability and Safety (RAMS) - Part 1: Generic RAMS Process.* October 2017.

[16] *Railway applications - Communication, signalling and processing systems - Safety related electronic systems for signalling.* November 2018.

[17] *V-model.* 2001-. https://en.wikipedia.org/wiki/V-model.

[18] *Mastering the V-Model: An In-Depth Guide to the V-Model SDLC.* https://teachingagile.com/sdlc/models/v-model.

[19] *Railway applications Gommunication, signalling and processing systems Software for railway control and protection systems.* 2011.

[20] *Common Safety Method for Risk Evaluation and Assessment.* https://www.era.europa.eu/domains/common-safety-methods/risk-evaluation-assessment-csm_en.

[21] *PLAUSIBILITY | English meaning - Cambridge Dictionary.* https://dictionary.cambridge.org/dictionary/english/plausibility.

[22] *Defenzivní programování.* 2001-. https://cs.wikipedia.org/wiki/Defenzivn%C3%AD_programov%C3%A1n%C3%AD.

[23] Joshua Bloch. *Effective Java.* 2nd ed ed. Upper Saddle River: Addison-Wesley, c2008. ISBN 978-0-321-35668-0.

[24] Liming Chen, and A. Avizienis. *N-VERSION PROGRAMMINC: A FAULT-TOLERANCE APPROACH TO RELIABlLITY OF SOFTWARE OPERATlON.* In: *Twenty-Fifth International Symposium on Fault-Tolerant Computing, 1995, ' Highlights from Twenty-Five Years'.* 1995. 113-.

[25] Algirdas Avizienis. *The Methodology of N-Version Programming.* In: 1995. 23-.

[26] *Difference between N-version programming and Recovery blocks Techniques.* https://www.geeksforgeeks.org/difference-between-n-version-programming-and-recovery-blocks-techniques/.

[27] William Stallings. *Operating Systems Internals and Design Principles.* Pearson Deutschland, 2014. ISBN 9781292061351. https://elibrary.pearson.de/book/99.150005/9781292061948.

[28] *Graceful Degradation in Distributed Systems.* https://www.geeksforgeeks.org/graceful-degradation-in-distributed-systems/.

[29] C. Michael Holloway. WHY ENGINEERS SHOULD CONSIDER FORMAL METHODS. *To appear in the Proceedings of the 16th Digital Avionics Systems Conference.* October 1997

[30] H. Bekic. *The meaning of names in PL/I.* In: C. B. Jones, eds. *Programming Languages and Their Definition: H. Bekic (1936–1982).* Berlin, Heidelberg:

Springer Berlin Heidelberg, 1984. 4–16. ISBN 978-3-540-38933-0. https://doi.org/10.1007/BFb0048936.

[31] Russel Kenley, and Toby Harfield. Location Breakdown Structure (LBS): a solution for construction project management data redundancy. *Swinburne University of Technology*.

[32] *The official handbook of MASCOT*. VERSION 3.1 ed. Worcestershire: Joint IECCA and MUF Committee on Mascot (JIMCOM), JUNE 1987. http://async.org.uk/Hugo.Simpson/MASCOT-3.1-Manual-June-1987.pdf.

[33] *Tutorial on JSP & JSD*. https://web.archive.org/web/20061121182445/http://cisx2.uma.maine.edu/NickTemp/JSP%26JSDLec/jsd.html.

[34] Texas Instruments. *AM335x Sitara™ Processors Datasheet*. 2023. https://www.ti.com/lit/ds/symlink/am3358.pdf.

[35] STMicroelectronics. *STM32MP157 Datasheet*. 2023. https://www.st.com/resource/en/datasheet/stm32mp157.pdf.

[36] NXP Semiconductors. *i.MX RT1060 Crossover Processor Datasheet*. 2022. https://www.nxp.com/docs/en/data-sheet/IMXRT1060IEC.pdf.

[37] Microchip Technology. *SAM E70 Series Datasheet*. 2021. https://ww1.microchip.com/downloads/en/DeviceDoc/SAM-E70-Datasheet-DS60001527E.pdf.

[38] *Texas Instruments AM335x Arm® Cortex™-A8 MPUs*. 2025. https://eu.mouser.com/new/texas-instruments/ti-am335x/.

[39] *LP-AM243 | AM243x general purpose LaunchPad™ development kit for Arm®-based MCU*. 1995-2025. https://www.ti.com/tool/LP-AM243.

[40] *Technical Reference Manual Cortex -R5*. 2010. https://documentation-service.arm.com/static/5f04288cdbdee951c1cd8969.

[41] *Functional Safety for AM2x and Hercules™ Microcontrollers - Product Overview*. 2023. https://www.ti.com/lit/po/sprt767/sprt767.pdf?ts=1745564392534.

[42] *CN0254 Evaluation Board Guide*. 2021. https://wiki.analog.com/resources/eval/user-guides/circuits-from-the-lab/CN0254.

[43] *Understanding the bootflow and bootloaders*. 2025. https://software-dl.ti.com/mcu-plus-sdk/esd/AM243X/latest/exports/docs/api_guide_am243x/BOOTFLOW_GUIDE.html.

[44] *SA2UL*. https://software-dl.ti.com/mcu-plus-sdk/esd/AM243X/latest/exports/docs/api_guide_am243x/SECURITY_SA2UL_MODULE_PAGE.html.

[45] *AM64x /AM243x Processors Silicon Revision 2.0 Texas Instruments Families of Products*. OCTOBER 2023.

[46] *AM64x, AM243x Functional Safety Manual*. 2024.

[47] *P1478R8: Byte-wise atomic memcpy*. https://www.open-std.org/JTC1/SC22/WG21/docs/papers/2022/p1478r8.html.

[48] *What does 8-N-1 mean?* 1999.
https://www.modemhelp.net/faqs/8n1.shtml.

[49] *FIREWALL | AM243x MCU+ SDK 11.00.00*. 2024.
https://software-dl.ti.com/mcu-plus-sdk/esd/AM243X/latest/exports/docs/api_guide_am243x/DRIVERS_FIREWALL_PAGE.html.

[50] *Arm® v7-M Architecture*. 2021.
https://developer.arm.com/documentation/ddi0403/latest/.

[51] *ARM® Architecture ARMv7-A and ARMv7-R edition*. 2018.
https://developer.arm.com/documentation/ddi0406/latest.

[52] *SYSCONFIG*. 2025.
https://www.ti.com/tool/SYSCONFIG.

[53] *XDS110 Debug Probe*. September 19 2024.
https://software-dl.ti.com/ccs/esd/documents/xdsdebugprobes/emu_xds110.html.

[54] Piyu Dhaker. *Introduction to SPI Interface*. 2018.
https://www.analog.com/en/resources/analog-dialogue/articles/introduction-to-spi-interface.html.

[55] *Tightly Coupled Memory*.
https://developer.arm.com/documentation/den0042/a/Tightly-Coupled-Memory.

[56] *ICSS DUAL EMAC FIRMWARE DESIGN GUIDE*. January 14, 2020.
https://git.ti.com/cgit/processor-sdk/pdk/plain/packages/ti/drv/icss_emac/firmware/icss_dualemac/docs/ICSS_DUAL_EMAC_Firmware_Design_Guide.pdf.

[57] *ARMOptimizing C/C++ Compiler v18.1.0.LTS*. January 2018.
https://www.ti.com/lit/ug/spnu151r/spnu151r.pdf.

[58] *Unit testing*. 2001-.
https://cs.wikipedia.org/wiki/Unit_testing.

Appendix A
List of Abbreviation

ADC	Analog-to-Digital Converter
API	Application Programming Interface
CCS	Calculus of Communicating Systems
CENELEC	European Committee for Electrotechnical Standardization
CSP	Communicating Sequential Processes
DMSC	Device Management System Controller
ESM	Error Signaling Module
FMEA	Failure Modes and Effects Analysis
FTE	Faul Tree Analysis
GP	General-purpose device
HOL	Higher Order Logic
HR	Highly Recommended
HS-FS	HS-Field Securable device
HS-SE	HS-Field Enforced device
I/O	Input / Output
IPC	Inter-Processor Communication
JSD	Jackson System Development
LBMS	Location-based method
LOTOS	Language for Temporal Ordering Specification
M	Mandatory
MASCOT	Modular Approach to Software Construction Operation and Test
MCU	microcontroller
NR	Not Recommended
PLL	Phase-Locked Loop
PMIC	Power Management Integrated Circuit
R	Recommended
RAMS	Reliability, Availability, Maintainability, Safety
RBL	ROM boot
SBL	Secondary Bootloader
SIL	Safety Integrity Level
SoC	System on Chip
SSADM	Structured systems analysis and design method
SYSFW	System Firmware
TFFR	Tolerable Functional Failure Rate
THR	Tolerable Hazard Rate
TIFS	TI Foundational Security
VDM	Vienna Development Method