Interner Bericht

DLR-IB-FT-BS-2024-116

Developing an FPGA based Redundancy Framework for Integrated Modular Avionics

Hochschulschrift

Phillip Nöldeke

Deutsches Zentrum für Luft- und Raumfahrt

Institut für Flugsystemtechnik
Braunschweig

Deutsches Zentrum
DLR für Luft- und Raumfahrt

Institutsbericht
DLR-IB-FT-BS-2024-116

Developing an FPGA based Redundancy Framework for Integrated Modular Avionics

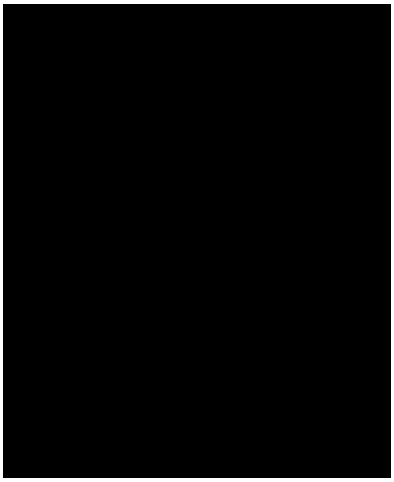
Phillip Nöldeke

Institut für Flugsystemtechnik
Braunschweig

138 Seiten
24 Abbildungen
29 Tabellen
13 Referenzen

Deutsches Zentrum für Luft- und Raumfahrt e.V.
Institut für Flugsystemtechnik
Abteilung Sichere Systeme & Systems Engineering

Braunschweig, den 22.07.2024



Institutsleitung: Prof. Dr.-Ing. S. Levedag

Abteilungsleitung: A. Bierig

Betreuer:in: Prof. Dr. U. Durak

Verfasser:in: P. Nöldeke

Technische Universität Braunschweig

Deutsches Zentrum für Luft- und Raumfahrt
German Aerospace Center

Master's Thesis

Developing an FPGA based Redundancy Framework for Integrated Modular Avionics

Phillip Nöldeke
[redacted]

May 31, 2024

DLR Institute of Flight Systems

Supervisors:
Prof. Dr.-Ing. Stefan Levedag
Institute of Flight Systems, DLR e.V.

Prof. Dr.-Ing. Peter Hecker
Institute of Flight Guidance, TU Braunschweig

Supervisor at DLR:
Prof. Dr. Umut Durak

In cooperation with the German Aerospace Center (DLR e.V.) Braunschweig

Statement of Originality

This thesis has been performed independently with the support of my supervisors. To the best of the author´s knowledge, this thesis contains no material previously published or written by another person except where due reference is made in the text. I also confirm that the electronic version submitted is identical to the printed version.

Braunschweig, May 31, 2024

Danksagung

Ich möchte mich an dieser Stelle bei Herrn Prof. Dr.-Ing. S. Levedag, Institutsleiter des Institutes für Flugsystemtechnik am DLR Braunschweig, für die Betreuung dieser Masterarbeit bedanken.

Weiterhin gilt mein Dank Umut Durak für die fachliche Betreuung und persönliche Unterstützung wärend der Bearbeitung.

Darüber hinaus möchte ich mich bei meinen Kolleginnen und Kollegen der Abteilung SSY und insbesondere der Arbeitsgruppe Avionics Systems für die Unterstützung während meiner Abwesenheit in den letzten Monaten bedanken.

Schließlich danke ich meiner Familie und meinen Freunden für die mentale Unterstützung während der Anfertigung dieser Arbeit.

Abstract

Fault tolerance is a key element in the design of safety-critical avionics systems. According to ARP4754A, avionics computers must not fail due to a single fault. That is why they must be capable of enduring a specified number of random component failures, to meet the stringent safety and reliability requirements. Since the occurrence of a fault leading to a failure event cannot be ruled out completely, avionics systems are designed and developed with a combination of fault avoidance and tolerance. The main objective is to preserve the avionics system functionality even when faults occur in the system. The stages a fault-tolerant system must provide are fault detection, fault containment and isolation, as well as reconfiguration or recovery.

Today's avionics systems are complex and use a combination of fault tolerance techniques to meet safety and reliability requirements. The complexity of a fault-tolerant design is further increased by the introduction of the Integrated Modular Avionics (IMA) concept. Redundancy is primarily used to ensure the integrity and reliability of an avionics system by replicating avionics computers (channels) and communication paths. This involves comparing and/or voting of replicas in order to identify faulty channels. The passivation of a failed channel is a consequence of the avionics system fault containment strategy. These redundancy management mechanisms result not only in processing overhead, but also increase the development effort and makes validation of the avionics system challenging. Because of their specific design for particular applications, the majority of redundancy management systems additionally pose challenges with reusability.

This thesis presents an approach for a framework that supports the development of an avionics redundancy management system from system design to integration on the target hardware. The aim is to provide a generic FPGA-based redundancy management system that is configurable according to the safety and reliability requirements of the use case. The avionics systems hardware and software architecture provide additional configuration inputs to the framework. The presented framework involves multiple steps to implement the intended functionality. First, it provides an environment where the redundancy architecture and the required fault-tolerance mechanisms are configured. This includes determining the required number of replicas and defining monitoring, voting, and consensus metrics. If required, dissimilarity patterns may also be considered in this

step. The second step employs the assembly of the redundancy management system implementation by utilizing essential building blocks containing generic FPGA logic. The process is completed by generating the intellectual property component in preparation for integration into the FPGA part of a system-on-chip processing module.
The thesis outcome shows that the redundancy framework facilitates the development of redundancy management systems by providing generic patterns for various fault tolerance techniques. During the configuration process, safety, reliability, and integrity requirements, as well as architectural system requirements are considered. In addition, the use of an FPGA allows separation from application development and significantly reduces the computational overhead on the target hardware's processing module.

Kurzfassung

Fehlertoleranz ist ein Schlüsselelement bei der Entwicklung von sicherheitskritischen Avioniksystemen. Gemäß ARP4754A dürfen Avionikcomputer nicht durch einen einzigen Fehler ausfallen. Deshalb müssen sie in der Lage sein, eine bestimmte Anzahl von zufälligen Ausfällen zu überstehen, um die strengen Sicherheits- und Zuverlässigkeitsanforderungen zu erfüllen. Da das Auftreten eines Fehlers und der damit verbundene potentielle Ausfall eines Avionikcomputers nicht vollständig ausgeschlossen werden kann, werden Avioniksysteme mit einer Kombination aus Fehlervermeidung und -toleranz entworfen. Das wesentliche Ziel besteht darin, die Funktionalität des Avioniksystems auch dann zu erhalten, wenn Fehler im System auftreten. Dazu stellt ein fehlertolerantes System Funktionen zur Fehlererkennung, Fehlereindämmung und -isolierung sowie Rekonfiguration oder Systemwiederherstellung zur Verfügung.
Die heutigen Avioniksysteme in Flugzeugen sind komplex und verwenden eine Kombination von Fehlertoleranztechniken, um die Anforderungen an Sicherheit und Zuverlässigkeit zu erfüllen. Die Komplexität eines fehlertoleranten Systementwurfs wird hierbei durch die Einführung des Integrated Modular Avionics (IMA) Konzepts zusätzlich erhöht. Redundanz dient dabei in erster Linie dazu, die Integrität und Zuverlässigkeit eines Avioniksystems zu gewährleisten, indem Avionikcomputer und Kommunikationselemente repliziert werden. Dies beinhaltet den Vergleich und/oder die Abstimmung von Computern, um fehlerhafte Kanäle zu identifizieren. Die Passivierung eines fehlerhaften Kanals ist dabei die Folge der Fehlereindämmungsstrategie des Avioniksystems. Diese Redundanz Management Mechanismen führen nicht nur zu einem Mehraufwand bei der Programmausführung, sondern erhöhen auch den Entwicklungsaufwand und stellen eine Herausforderung bei der Validierung des Avioniksystems dar. Aufgrund ihres spezifischen Designs für bestimmte Anwendungsfälle, erschweren die meisten Redundanz Management Systeme zudem die Wiederverwendbarkeit in anderen Avionik Systemen.
In dieser Arbeit wird ein Ansatz für ein Framework vorgestellt, das die Entwicklung eines Avionik-Redundanz Management Systems vom Systementwurf bis zur Integration auf der Zielhardware unterstützt. Das Ziel ist es, ein generisches FPGA-basiertes Redundanz Management System bereitzustellen, das gemäß den Sicherheits- und Zuverlässigkeitsanforderungen des entsprechenden Anwendungsfalls konfigurierbar ist. Die Hard-

und Software-Architektur des Avioniksystems liefert dabei zusätzliche Informationen für die Konfiguration. Das vorgestellte Framework umfasst mehrere Schritte, um die geforderten Funktionalitäten umzusetzen. Zunächst stellt es eine Konfigurationsumgebung bereit, in der die Redundanzarchitektur und die erforderlichen Fehlertoleranzmechanismen definiert werden. Dazu gehören die Bestimmung der erforderlichen Anzahl an redundanten Komponenten sowie die Festlegung von Monitoring-, Voting- und Konsensmetriken. Falls erforderlich, kann in diesem Schritt auch ein dissimilares Systemdesign berücksichtigt werden. Im zweiten Schritt wird die Implementierung des Redundanz Management Systems unter Verwendung von generischen FPGA-Bausteinen vorgenommen. Der Prozess wird durch die Generierung des Redundancy Management Intellectual Property Moduls zur Vorbereitung der Integration in den FPGA-Teil eines System-on-Chip Prozessor Moduls abgeschlossen.
Das Ergebnis dieser Arbeit zeigt, dass das Redundanz-Framework die Entwicklung von Redundanz Management Systemen erleichtert, indem es generische Muster für verschiedene Fehlertoleranztechniken bereitstellt. Während des Konfigurationsprozesses werden Sicherheits-, Zuverlässigkeits- und Integritätsanforderungen sowie Anforderungen an die Systemarchitektur berücksichtigt. Darüber hinaus ermöglicht die Verwendung eines FPGA eine Trennung von der Entwicklung der eigentlichen Applikation und reduziert den Rechenaufwand im Prozessor Modul der Zielhardware erheblich.

Task Definition

Integrated Modular Avionics (IMA) is a new approach for architecting avionics systems. An IMA system consists of multiple computing modules, each supporting multiple applications at different levels of criticality. This leads to design of novel avionics architectures. However, it also makes it challenging to meet the already stringent reliability and safety requirements. Avionics system designers use redundancy practices and health monitoring concepts to build integrity and fault tolerance on top the system at design time to meet these safety and reliability requirements.
The increasing use of system-on-chip (SoC) modules in IMA platforms provides additional implementation options for avionics and redundancy system designers. Using the Field Programmable Gate Array (FPGA) part of such SoCs to host the redundancy management logic significantly reduces the computational overhead on the processor and enables a generic, platform-independent redundancy management implementation.
The demonstrator platform of the redundancy framework is a Flight Control Computer (FCC) developed by the DLR.

The following aspects will be dealt with in detail:

- Literature review on avionics related development standards and guidelines.
- Identify the possible failures and errors that could occur in an avionics system and analyze how to mitigate them.
- Development of feasible redundancy architectures and techniques applicable to the DLR-FCC avionics hardware.
- Conceptual design of an FPGA-based redundancy framework for the DLR-FCC considering the developed architectures and techniques.
- Implementation of the redundancy framework on an FPGA.
- Perform simulation and hardware-in-the-loop testing of the implementation.
- Discuss the results obtained.

List of Abbreviations

Notation	Description
AMD	Advanced Micro Devices
APU	Application Processing Unit
ASIC	Application-Specific Integrated Circuit
AXI	Advanced eXtensible Interface
B2B	Board to Board
BF	Byzantine Fault
BR	Byzantine Resilience
CC	Cross-Channel
CCI	Cross-Channel Interface
CL	Cross-Lane
CLB	Configurable Logic Block
CLI	Cross-Lane Interface
CMF	Common-Mode Failure
CRC	Cyclic Redundancy Check
DDDI	Duplex Data Duplex Interface
DDR	Double Data Rate
DDSI	Duplex Data Simplex Interface
DLR	German Aerospace Center
DUT	Device Under Test
EEPROM	Electrically Erasable Programmable Read-Only Memory

Notation	Description
EPROM	Erasable Programmable Read-Only Memory
FCC	Flight Control Computer
FCR	Fault-Containment Region
FiFo	First-in-First-out
FPGA	Field Programmable Gate Array
GNSS	Global Navigation Satellite System
HDL	Hardware Description Language
I/O	Input/Output
IC	Integrated Circuit
IMA	Integrated Modular Avionics
INS	Inertial Navigation System
IP	Intellectual Property
JSON	JavaScript Object Notation
LUT	Lookup Table
MPSoC	Multi Processor System on Chip
MPU	Message Processing Unit
MU	Management Unit
PCB	Printed Circuit Board

Notation	Description
PL	Programmable Logic
PLD	Programmable Logic Device
PS	Processing System
RAM	Random Access Memory
RM	Redundancy Management
RMC	Redundancy Management Channel
ROM	Read Only Memory
RPU	Real-Time Processing Unit
RTL	Register Transfer Level
RX	Receive
SDDI	Simplex Data Duplex Interface
SDRAM	Synchronous Dynamic Random Access Memory
SDSI	Simplex Data Simplex Interface
SiL	Software-in-the-Loop
SoC	System on Chip
SoM	System on Module
TMR	Triple Modular Redundancy
TX	Transmit
VHDL	Very High Speed Integrated Circuit HDL
WCET	Worst Case Execution Time

Notation	Description
YAML	YAML Ain't Markup Language

List of Figures

List of Tables

Listings

Contents

1 Introduction

In modern aircraft, avionics systems are playing an increasingly important role due to growing digitalization. At the same time, the demands on avionics to perform more tasks in less time are increasing, adding to the complexity of these systems. In addition, these systems are required to meet the highest aerospace standards for safety and reliability during their operation. The key is to detect and contain faults at runtime and maintain system functionality. This approach is referred to as fault tolerance.
To meet these requirements, avionics computers are replicated so that a standby computer can maintain system functionality if another computer fails. The occurrence of a fault is detected by extensive comparison and voting mechanisms between redundant units. It can be assumed that the more complex the avionics system itself is, the more complex these redundancy management mechanisms become. The development of such fault tolerance measures is based on the system's safety and reliability requirements identified through the safety assessment process (see ARP4761A [1]) and is specifically designed for the system, the hardware, and the software architecture. This means that an individual redundancy management system must be developed for each avionics system and component to implement fault tolerance. Because of the direct interface between the actual application and the redundancy management at the hardware and software level, the two development processes are closely coupled. With the increasing complexity of avionics systems and the introduction of more modern system concepts such as Integrated Modular Avionics (IMA), the development of fault-tolerant avionics systems is facing major challenges.

1.1 State of the Art

Conventional avionics systems typically use distributed computing architectures. These computers are designed and developed for individual functions. To achieve the required level of fault tolerance, redundancy architectures such as dual-duplex, triplex or quadruplex are implemented, depending on the criticality of the system. This applies to both

computers and communication elements. However, because these systems are individually designed for each avionics component, they tend to be very inflexible when it comes to subsequent system adaptations.
In contrast, the Integrated Modular Avionics (IMA) concept, introduced by [2], proposes a modular avionics architecture with generic computer platforms and standardized communication networks. The resources of IMA platforms are shared by multiple mixed-criticality applications. Access to shared resources is managed through strict partitioning and active resource management. The goal is to ensure that hosted applications do not interfere with each other through unintended behavior [2].
The concept of IMA is only rarely used in current avionics systems and is only used in low criticality systems [3]. Airbus for example uses Core Processing and Input/Output Modules (CPIOMs) in the A380 and the A400M whereas Boeing implemented the B777´s Airplane Information Management System (AIMS) according to the IMA concept [3].

1.2 Scope of the Thesis

This thesis addresses the aforementioned challenges in the development of fault-tolerant avionics systems and presents a possible solution with the development of a modular redundancy framework. An avionics computer developed by the German Aerospace Center (DLR) is used to demonstrate the system. Based on the identified failure modes and safety requirements, a redundancy framework is developed to support the implementation of FPGA-based redundancy management systems for modular avionics platforms. This thesis documents the structure and approach of the framework and explains the development of a generic, FPGA-based redundancy management system. The goal of the work is to present a first version of the redundancy framework and a redundancy management system for the DLR dual-lane avionics computer.

2 Fault-Tolerant Avionics Systems

Avionics systems are safety-critical systems, that are defined by the fact that a fault or failure of such a system can result in a hazardous or even catastrophic event.
In the context of avionics systems, this includes various components such as flight control systems, autoland systems, avionics computers, and data communications. In accordance with ARP4754A, avionics systems are required to have a certain degree of fault tolerance to withstand a specified number of random component failures [4, 5]. Therefore, it is essential to ensure the continuous and safe operation of these systems throughout their operational duration. This chapter provides an introduction to the concept of fault tolerance in avionics systems, serving as the foundation for this thesis.
An early approach to the design of safety for avionics computers is the avoidance of faults [6]. Lala et al. [6] further discovered that this is achieved through strict quality control and component engineering. However, engineering high reliability into devices and components results in significant costs.
Furthermore, the occurrence of design errors is unpredictable and their effectiveness in preventing them is only partial [7]. Moreover, it is not possible to address Byzantine Faults (BFs) and Common-Mode Failures (CMFs) through fault avoidance measures during the development process [6].
As noted by Hitt et al. [7], avionics systems must be designed and developed to incorporate both fault avoidance and fault tolerance. The objective is to ensure the continued functionality of the system in the event of faults. However, developing a fault-tolerant avionics system can lead to increased system complexity and validation challenges. The necessity for additional hardware and software modules, as well as the introduction of greater connectivity between system elements, is largely responsible for this [7].
Nevertheless, the possibility of a fault leading to a failure event cannot be entirely excluded, even if the avionics system is developed in accordance with the highest safety standards and all conceivable error cases are considered during the development process. The principal functionalities of fault-tolerant systems are the detection, containment, and isolation of faults, as well as reconfiguration or recovery [8]. One of the most important

techniques used to implement these features is redundancy. The replication of functionally identical compute modules (called channels) and communication interfaces increases system reliability and integrity.

2.1 Redundancy

With the introduction of microprocessors, it became more cost effective to design redundant components [6]. As mentioned in [6], the concept of redundancy is employed as a means of ensuring safety, and this involves a trade-off between the avoidance of faults and fault tolerance. In addition to the advantage of fault tolerance, redundant systems present a challenge in terms of validation. Moreover, redundancy does not guarantee fault tolerance, but it does significantly reduce the rate of fault occurrences. However, this approach comes at a cost, as fault-tolerant systems can lose up to 50 % of their performance in order to manage redundancy. While initial redundant systems are more susceptible to failure than simplex systems, redundancy is meanwhile a well-established technique in the aerospace domain. It is used to meet stringent safety standards with respect to system availability [6].
The study by Lüttig [3] states that the functionality of a component is classified as either *correct* or *failed*. A component is considered correct when it performs its intended function as expected. A design error (introduced during the component development process) or a random fault can cause a component no longer being able to perform its intended function. Subsequently, the component is then classified as failed. However, it is possible for a system or component to have both errors and faults without resulting in a failure. This can happen if, for example, there is a software bug in a disabled feature that would only manifest itself as a failure when enabled. Similarly, there may be a hardware failure in an unused component that only causes a failure when activated. This implies that a component can actually exist in one of three states: *correct*, *passive*, or *out-of-control*. A component is classified as passive when a failure has been identified and successfully treated. If a component has a potential failure that has not been identified or treated, it is classified as out of control. The identification of a failure within a component can be achieved by comparing the results of two functionally identical components (redundant

replicas). According to Lüttig [3], this approach is implemented, for example, in the commanding-monitoring principle.
The first step in implementing fault tolerance in an avionics system is to partition redundant items into Fault-Containment Regions (FCRs) [6]. Lala et al. [6] also describe a FCR as a collection of components designed to operate correctly even in the presence of arbitrary or electrical faults outside the region. It is essential that any faults that may occur within the FCR are prevented from propagating beyond the FCR boundaries and affecting or compromising other components. Additionally, measures of fault containment at system level are crucial. Voting mechanisms are therefore used to mask faults at different stages of the fault-tolerant system. The process of using input voting involves the masking of failed sensor values. Internal computer voting plays a fundamental role in preventing the propagation of faults from a failed channel to other channels within the system. The combination of output voting and interlock mechanisms serves to prevent the outputs of failed channels from propagating beyond the FCR boundaries [6]. An interlock mechanism can enable or disable the outputs of a channel. The majority of channels within an avionics system can collectively change the lock state to disable a failed channel [6]. The presence of multiple redundant channels allows for fault masking, eliminating the need for immediate fault diagnosis, isolation and recovery. This ensures that the majority of channels continue to operate, thereby creating a redundant system that is capable of meeting the stringent real-time response requirements. The implementation of actuator voting is used to effectively mask errors that may occur within the data transmission medium. [6]

2.2 Architectural Redundancy Categories

As outlined in [7], redundancy can be classified into three architectural categories: static, dynamic, and hybrid. A statically redundant system is able to detect and mask faults. A simple example is the Triple Modular Redundancy (TMR) architecture. The system consists of three redundant replicas, each of which generates an output that is then voted on to determine which signal to select. This passive approach is sufficient to prevent fault propagation through the FCR boundaries. If the replicas are able to detect internal errors, the system's overall reliability would be improved, and the number of replicas required

for the voter would be reduced. It is important to note that the reliability of the system is strongly dependent on the individual replicas and the reliability of the voters. The reliability of a voter is defined as the probability that the voter provides correct output and does not assert an unsafe signal. The voter safety is defined by its reliability and the probability that the voter will assert an unsafe signal. A redundant system must be designed with a balance of safety and reliability, as both are interrelated. An increase in one can lead to a decrease in the other [7].
Figure 2.1 depicts a dynamic redundant system that is capable of reconfiguring in the event of a failure. The figure shows a triplex redundant system that does not select the correct output, but uses the majority of the channel outputs to disable a failed channel if the failure is determined to be persistent.

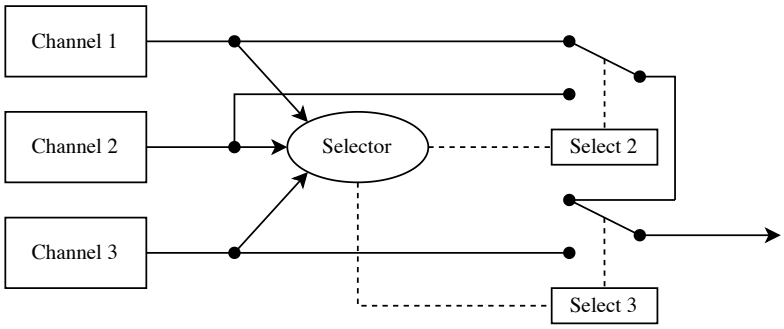


Figure 2.1: Dynamic triplex redundancy example [7]

A hybrid redundant system is a combination of static and dynamic fault detection, masking, and recovery (which may involve reconfiguration) [7]. As introduced by [7], a hybrid fault-tolerant system includes active and redundant channels, along with a spare channel. The outcome of each channel output is determined by a central voter. In addition, each channel is equipped with a comparator unit, which is responsible for verifying that the channel output is different from the voted output. In the event of a mismatch between the selected output and the channel output, a switch disables the failed channel output and replaces it with the backup channel. This functionality ensures that the system remains operational in the event of a channel failure. A failed channel can be restored to operational status if it is reestablished with the voter for a specified period of time [7].

2.3 Reliability and Integrity in Redundant Systems

According to Lüttig [3], the configuration of an avionics system with multiple redundant channels can be approached in two ways: *active-standby* and *all-active*. In active-standby systems, a single channel assumes the operational role. If the active channel fails, one of the standby channels takes over. In all-active systems, all channels are operational and interact with external systems (e.g. actuators). Compared to active-standby systems, all-active redundant systems provide higher reliability. However, with regard to integrity, double failures and inconsistent failures (Byzantine failures) tend to be the most common. In this context, active-standby systems have better integrity. While all-active systems can improve both reliability and integrity, the increased number of redundant modules requires the implementation of sophisticated mechanisms for inter-channel communication mechanisms and modifications to the actuators. In this configuration, the actuators are independently responsible for selecting a channel's data. In active-standby systems, it is essential that integrity is maintained independently in each channel. To achieve this goal, it is necessary to design the channels as duplex modules to avoid common mode errors. The introduction of inter-module redundancy serves to increase the overall reliability of the system, which is achieved through the implementation of n-duplex systems. A fundamental requirement for the effective operation of an active-standby system is the establishment of a consensus regarding the active status. Each duplex channel must be in agreement with the others as to which channel is active while all others remain in standby mode. The implementation of this concept requires the establishment of a highly reliable inter-channel communication network [3].

2.4 Consensus

In fault-tolerant systems, comparing data from redundant channels is essential to detect and mask faults [6]. This should allow redundant replicas to reach consensus on their input and output data [3]. This can reduce the likelihood of erroneous values being transmitted beyond the boundaries of the FCR [3].
Consensus between redundant replicas is achieved when they agree on each others computed output or received input data. A distinction is made between exact and approximate

consensus, as explained by Lala et al. [6]. For exact consensus, the compared data must be bit-wise identical. This method is useful for comparing channel states, for example. To reach an approximate consensus, the compared data must not deviate from each other by more than a defined threshold. The approximate consensus principle is useful, for example, for comparing measured physical quantities. Determining the appropriate threshold, however, is a challenging task. The values are often obtained empirically or by experience. A balance must be achieved so that faults are correctly detected without generating false alarms [6].
Still, there is always the possibility that the output data is invalid despite consensus if the replicas agree on identical but incorrect values. Nevertheless, in order to achieve output consensus, it is at first necessary for the replicas to agree on each other's input data (input consensus) [3]. This ensures that all redundant computers receive the same input data for computation.
Ensuring input validity also involves filtering out grossly misbehaving sensors, especially since bit-for-bit voting is impractical due to the analog nature of sensor values [6]. As a result, separate sensor redundancy management algorithms are required to generate a valid input value for replicated sensors through mid or mean value selection, or averaging [6].
The comparison of the data is done by a voter or a comparator, depending on the number of replicas. In addition to the configuration for exact or approximate comparison, the possible error types must be considered when designing the voter. Fault modes such as random faults, drifts, constant offsets, or transient pulses must be considered in this matter [6]. As pointed out in [6], certain conditions must be met in order to perform a bit-identical comparison between the data of two digital computers. This includes both replicas being initialized with the same state, having identical input data, and performing the same sequence of operations. Temporal synchronization of the two components is also critical to ensure that the two replicas do not drift apart [6].

2.5 Byzantine Faults

One of the primary requirements for the development of fault-tolerant avionics systems is resilience to BFs, as stated by Lala [6]. A BF manifests itself in the arbitrary behavior of

a failed component. This can result in a lack of continuity in the execution of a program, or the transmission of conflicting information to different destinations. Consequently, a BF causes any behavior within a failed component to corrupt the system [6]. To ensure the resilience of an avionics architecture prone to BFs, it is necessary to impose a lower bound on the number of FCRs, as well as to guarantee the connectivity and synchronicity between them, by implementing specific information exchange protocols [6].
As noted in [6], proving that an appropriate architecture satisfies these criteria is less expensive and time-consuming than proving that certain failure modes cannot occur with a probability of, for example, 10^{-5}. Typical triplex and quadruplex system architectures require a minor redesign of channels and inter-channel communication protocols to achieve the required Byzantine Resilience (BR). The aforementioned techniques are primarily concerned with the hardware architecture. As a result, Byzantine resilient systems are more transparent to software programmers. It is possible to develop and validate operating systems and applications in a simplex environment without worrying about redundant copies. In addition, hardware redundancy management is transparent from the programmer's perspective. This means that the application layer is separated from the hardware and software that manages channel and module redundancy. This separation enables independent validation of redundancy management and application-related hardware and software [6].

2.6 Common-Mode Failure Tolerance

CMF are the most significant source of failures in safety-critical systems [7].
Unlike random failures, CMFs have the potential to affect multiple FCRs simultaneously, often due to common causes such as design errors (imperfections in requirements, design or implementation) or internal hardware and software errors [6].
A single CMF can cause all similar redundant copies to fail. For example, achieving design diversity through dissimilar replicas is essential to avoid fault propagation through FCR boundaries [7].
Traditional redundancy approaches have proven to be less effective in avoiding CMFs [6]. Therefore, Lala's research presented alternative approaches combining fault avoidance and fault tolerance must be considered. CMF avoidance can be implemented using tools

that support the development process from the design and requirements phase through to implementation. The use of standards and formal methods is also appropriate to achieve consistency and correctness and to reduce errors in specification, design, and implementation. CMF removal approaches include design reviews, simulations, tests, and fault injections to identify potential errors. Detection mechanisms such as watchdog timers and hardware exceptions help identify unwanted hardware and software states to implement CMF tolerance at runtime. Corrective measures, such as resetting the system to its last known correct state, may help to handle an identified CMF [6]. This may require rebooting modules and re-synchronization of redundant channels.

2.7 Dissimilarity

As mentioned above, a dissimilar system design can be used to avoid CMFs in redundant replicas [7]. For example, in commanding-monitoring architectures and active-standby systems, dissimilarity is typically used to avoid CMFs or reduce the probability of its occurrence [3].
Dissimilarity can be implemented at both the hardware and software level to maintain independence between redundant replicas [3].
At the hardware level, dissimilarity could include the use of electronic components from different manufacturers. Software dissimilarity could be achieved by using different programming languages for redundant replicas. A more general approach is to develop replicas with different development teams, each performing its own development process. Since these steps can significantly increase the development effort, a trade-off must be made to determine the appropriate level of dissimilarity and CMF tolerance given the safety and reliability requirements of the system.
In principle, a dissimilar system architecture can prevent fault propagation across FCR boundaries [6].

3 DLR Flight Control Computer

The German Aerospace Center (DLR) developed an avionics Flight Control Computer (FCC) for an unmanned high-altitude solar-powered aircraft. In addition to many digital interfaces, the FCC contains a dual-redundant power supply, several air data sensors and Inertial Navigation Systems (INSs), and two processor cards. In order to meet weight, space and cost requirements, the FCC is a self-contained avionics unit, making it suitable for general use in small aircraft.
Within this thesis, the FCC is used as a demonstrator and validation platform for the developed redundancy management framework.
The following sections describe the FCC architecture, the available interfaces, and the structure of the processor modules.

3.1 FCC Hardware Architecture

As shown in figure 3.1 the hardware architecture of the FCC consists of a total of five Printed Circuit Boards (PCBs). The basis is the interface board with connectors for the external digital interfaces and the power supply. It also includes transceivers and protection circuitry. Both the power board and the processor boards are connected to the interface board.

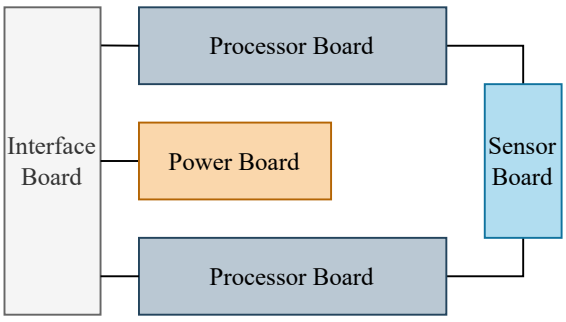


Figure 3.1: Flight Control Computer architecture

The power board contains the power stages that convert the external supplied voltage into 5 VDC and 3,3 VDC for the internal power supply. Both processor lanes are developed according to a generic design which facilitates interchangeability of lanes. Further, the lanes have fully physical access to all external data interfaces through the interface board connectors. The signal routing as well as access permissions are managed in the respective processor of each lane. Redundancy concepts such as Commanding-Monitoring could be implemented with these hardware capabilities. The designated configuration is developed and built during design phase and stored in each processor's non-volatile memory. The behavior and permissions of the processor lanes can thus be adapted by reprogramming the configurations.

3.2 FCC Interfaces

The FCC has electrical and data interfaces, both of which are routed to their designated connectors on the interface board. The power supply is provided by two independent, galvanically isolated channels, one of which is sufficient if the other fails. In addition to the physical air pressure and the two Global Navigation Satellite System (GNSS) ports, the interface panel also includes several digital communication interfaces (see Table 3.1).

Interface	Quantity	Protocol	Supported Data rate
CAN	2	CAN/CAN-FD	1 Mbit/s
RS-232	4	UART	500 kbit/s
RS-485	2	UART	500 kbit/s
Ethernet	2	UDP	1 Gbit/s

Table 3.1: Flight Control Computer external data interfaces

3.3 FCC Processing Modules

The processor boards are hosting the processing power of the avionics computer. Each card is equipped with special Board to Board (B2B) connectors where a 4 cm x 5 cm System on Module (SoM) can be plugged onto. This also supports the interchangeability of

processing modules, since different SoMs can be installed with the appropriate form factor. These SoMs include a System on Chip (SoC), oscillators, volatile and non-volatile memory, interface transceivers, and high-speed B2B connectors to install the SoM on a carrier board. The SoM used in this project is the TE0821-01-3BE21ML from Trenz Electronic. It is equipped with a 128 MB flash memory for configuration and data storage, 2 GB Double Data Rate (DDR) Synchronous Dynamic Random Access Memory (SDRAM) and an internal power switch for the required voltages [9]. The main part of the SoM is the AMD/Xilinx Zynq Ultrascale+ Multi Processor System on Chip (MPSoC).
It includes a 64-bit quad-core ARM Cortex-A53 processor, a dual-core ARM Cortex-R5 real-time processor, power and memory management, a configuration security unit, DDR4 memory support, multiple general-purpose (CAN, UART, SPI) and high-speed (Ethernet, PCIe) interfaces, and a programmable FPGA part. The processing related components such as the Application Processing Unit (APU) and the Real-Time Processing Unit (RPU) as well as the platform management, memory and communication interfaces make up the Processing System (PS). The FPGA part with its connectivity interfaces and signal processing capabilities comprises the Programmable Logic (PL) as shown in Figure 3.2. The communication interface between both parts is realized by the on-chip Advanced eXtensible Interface (AXI) data bus [10].
The separation of PS and PL is clearly the advantage of using this chip architecture for avionics computers. While applications and Operating Systems (OS) are usually executed in the PS part, a redundancy management system implemented in the PL can be developed and executed independently. Given the massive computing power of FPGAs, the computational overhead required to ensure data integrity and consensus is significantly reduced. An additional advantage is the realization of memory mapped communication interfaces (internal and external), which allows the assignment of strict access permissions and thus spatial partitioning between concurrently executed functions.
For this reason, it was decided to implement the redundancy management system in the FPGA part of the FCCs processing modules.

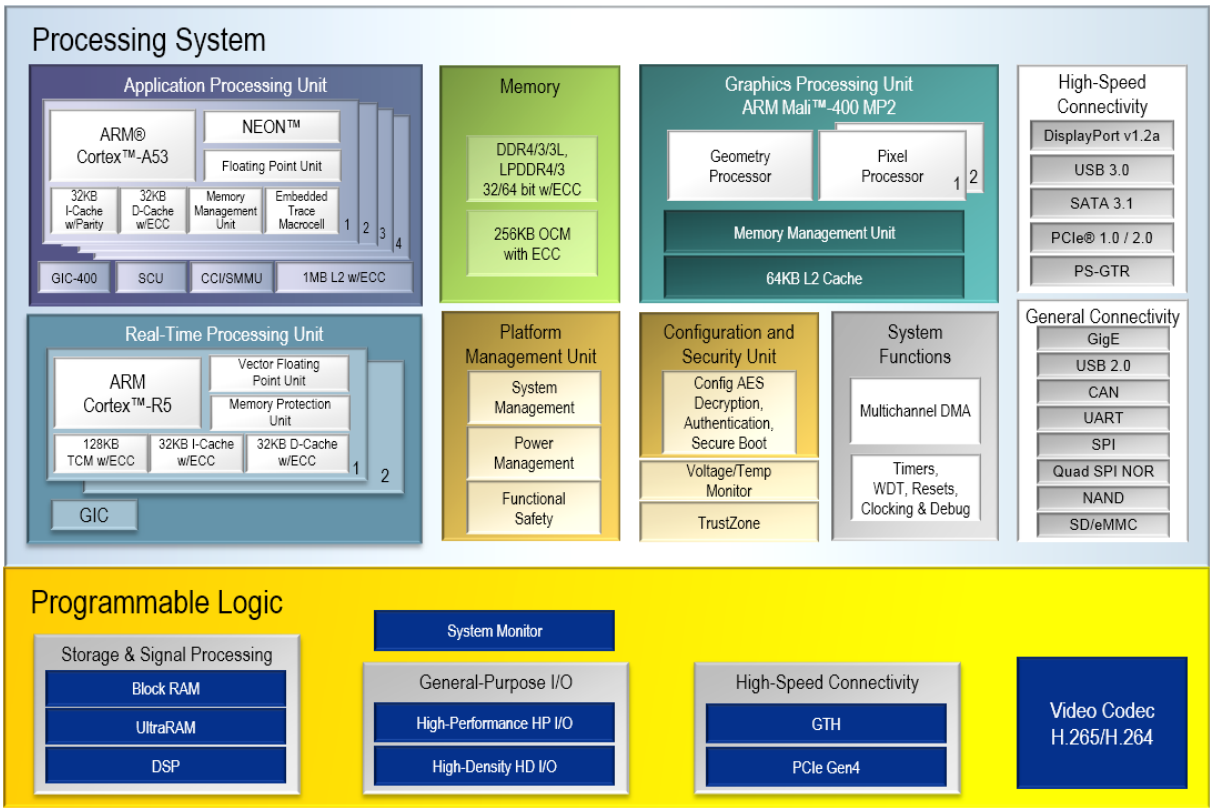


Figure 3.2: Zynq Ultrascale+ MPSoC overview [11]

4 Field Programmable Gate Arrays

Several embedded devices require the use of Integrated Circuit (IC) logic devices to implement static functions. Programmable Logic Devices (PLDs) or Application-Specific Integrated Circuits (ASICs) are commonly used therefore. However, these ICs require a lot of development effort and prototyping is quite challenging. Hence, the time-to-market leads to be long and thus increases the costs.
According to Brown [12], FPGAs are a suitable alternative to the above mentioned IC technologies to reduce development and production time. The first commercially available FPGA was developed by Xilinx in 1985. Since then, FPGAs have proven to be particularly useful for rapid prototyping of hardware logic [12].
Similar to PLDs and ASICs, FPGAs are integrated circuits that can be configured with digital logic by developers and end users. They consist of a two-dimensional array of Configurable Logic Blocks (CLBs) (see Figure 4.1). This CLB array is surrounded by a set of configurable Input/Output (I/O) blocks. A programmable interconnect network implements the communication between CLBs and I/O blocks [13].
The communication network connects the logic blocks with static wiring segments and programmable switches. The latter are realized by transistor controlled Random Access Memory (RAM) cells or Erasable Programmable Read-Only Memory (EPROM)/Electrically Erasable Programmable Read-Only Memory (EEPROM) transistors [12].
The CLBs consist of two or more logic cells. A logic cell is a basic grain of FPGAs. It consists of a four-bit lookup table (configured as Read Only Memory (ROM), RAM, or combinatorial logic). In addition, a carry data path is added for efficient arithmetic operators, and a D-type flip-flop is added to register the output of the logic cell [13].
Figure 4.2 shows the FPGA logic cell structure.

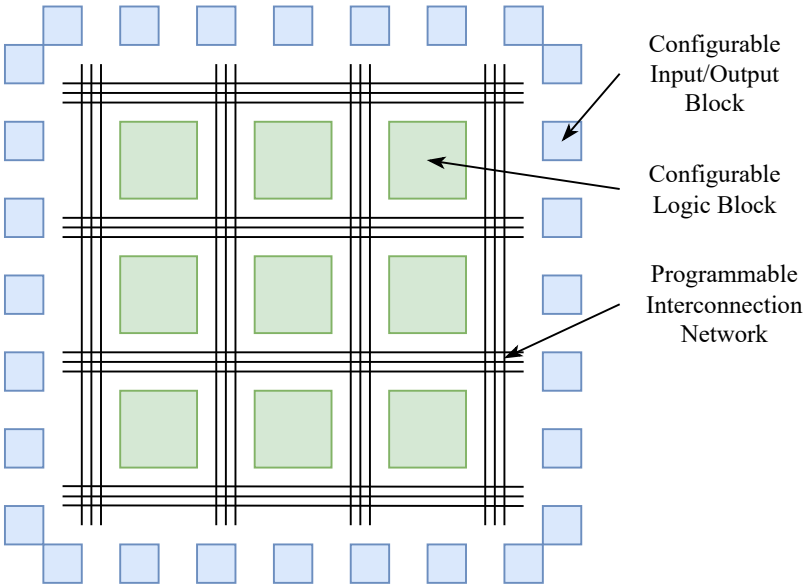


Figure 4.1: Generic FPGA architecture [13]

In recent years, modern FPGA design tools and methods have been developed to enable the implementation of complex logic and functions (e.g. complete digital systems or hard/soft processor cores such as ARM and MicroBlaze) [13]. These tools are based on Hardware Description Languages (HDLs) like Very High Speed Integrated Circuit HDL (VHDL) or Verilog.

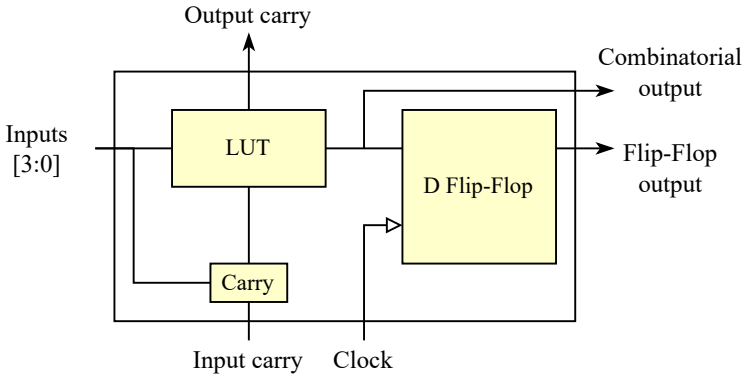


Figure 4.2: FPGA logic cell structure [13]

The FPGA design flow is divided into the following stages:

- System Level
- Behavioral Level
- Register Transfer Level (RTL)
- Physical Level

Following the work of Monomasson et al. [13], the FPGA designer describes the high-level circuit specification at the system level. At the behavioral level, the actual functionality of the circuit/logic algorithm is described by a HDL. Functional correctness can be verified by simulating the behavioral description in a dedicated test bench. In Register Transfer Level the behavior is synthesized into a circuit of logical hardware elements. Since the advent of analog HDLs, a simulation of the circuit is also possible at this level of abstraction. Performing the signal routing, logic mapping, and optimization results in the physical level bitstream that configures the FPGA. This last step takes into account the actual target hardware characteristics [13].
The configured logic is represented by a run-time static hardware circuit inside the FPGA. Hence, the Worst Case Execution Time (WCET) of an FPGA function is deterministically verifiable, which is a key requirement for safety-critical avionics. In addition, the FPGA facilitates the integration of multiple logic modules/functions in parallel, allowing simultaneous execution of independent functions. This approach also incorporates spatial partitioning of concurrently executed modules. This qualifies FPGAs as accelerators for computationally intensive applications such as high-speed communications, digital signal processing, etc.

5 Redundancy Management Framework

In the development of avionics systems, the requirements for reliability and safety play an important role. The primary metric in this context is fault tolerance, which is the ability of the system to tolerate certain failures. In the broadest sense, a fault-tolerant system is expected to continue to perform its required function to some extent even if a fault occurs in the system. In the context of avionics, this means that an avionics system must not fail just because a failure occurs in one of its computers. A common practice to achieve fault tolerance in a system is to use redundancy. This involves instantiating multiple functionally identical components (replicas) within the system so that one backup component can take over the tasks of another in the event of a failure. In addition, redundant instances can be used to compare against each other to detect random faults in the system. Therefore, the main tasks to ensure fault tolerance are fault detection and identification, fault containment, and fault recovery or system reconfiguration. These actions are performed by an active Redundancy Management (RM) system. The RM system operation is in addition to the actual avionics applications and must be performed at all times during system runtime. Implementing these functions in an avionics system is often very complex, requiring significant effort and overhead for application developers. Furthermore, such systems are rarely reusable because they are typically developed for a specific use-case or function.
Moreover, the development of redundancy management systems presents further challenges, particularly for IMA systems. The system flexibility and dynamics required by the IMA concept are not easily supported by legacy redundancy management systems. The primary challenges in achieving and guaranteeing fault tolerance are the dynamic scheduling of partitions on different computers and dynamic reconfiguration.
The objective of this thesis is to present a framework to facilitate the development of fault-tolerant avionics systems, encompassing both legacy avionics and IMA systems. The intent behind this framework is to minimize the effort required for developers to implement

fault tolerance mechanisms. For this purpose, the framework provides generic patterns (see section 5.3) and redundancy management building blocks (see section 5.4) for implementing the necessary fault tolerance measures. Additionally, the framework aims to provide a flexible and generic foundation so that adapting an avionics computer or even the entire avionics system does not necessitate a complete redesign of the redundancy management system. In contrast to conventional redundancy management systems, the proposed framework provides a range of tools that have been developed for use in complex and flexible avionics systems. The overall goal is to decouple the application and safety engineering with providing a redundancy management system transparent to the application developers point of view and vice-versa. The system shall be generic and able to operate independently so that avionics computers or even software developers do not need to get involved to closely in the redundancy management system development. However, preventing failures due to design, implementation, and human error is not an objective of this framework.
The primary objective of the first framework version is to provide a RM system that is configured and prepared for integration into the DLR FCC presented in section 3. Furthermore, the system shall be capable of adapting to different avionics architectures. The integration into the DLR FCC is intended to take place in the FPGA part of an Advanced Micro Devices (AMD) Ultrascale+ SoC. This allows for the implementation of the generic design and the degree of flexibility required for IMA systems. In addition, implementing the RM system in an FPGA also strengthens the decoupling from application software development, since FPGA logic must be developed according to DO-254.

5.1 Requirements

The previous chapter identified several goals and requirements that avionics systems must meet to achieve fault tolerance. In addition, many of them are mandatory for compliance with aerospace standards and guidelines. Derived from the chapter 2, this section presents the requirements derived from the system requirements that the redundancy management framework must satisfy in order to support the development of a redundancy management system that enables fault-tolerant behavior. These requirements are further divided into

general framework related requirements and those related to the implementation of the redundancy management system Intellectual Property (IP) itself.

5.1.1 Redundancy Framework Requirements

The requirements associated with the redundancy management framework mostly define the general functionality and workflow of the framework. The requirements primarily concern configurability and flexibility with respect to different avionics architectures, as well as the generic design of the framework. Moreover, the implementation of fault tolerance patterns that incorporate specific fault tolerance mechanisms is required. In addition, the specifications explicitly identify the particular mechanisms and functions that the framework must implement to achieve a given level of fault tolerance. The Table 5.1 contains some of the most important redundancy framework requirements. The full list can be found in the appendix section 7.1.

ID	Name	Text
0-1	Generate Redundancy Management System IP	The Framework must be able to generate an avionics redundancy management system implemented in an FPGA IP.
0-2	Avionics Architectures Support	The framework must be designed generic to support several avionics architectures.
0-3	Configurability	The framework must evolve configurability to adapt the redundancy management according to safety requirements.
0-4	Fault tolerance pattern	The framework shall provide pattern implementing several fault tolerance mechanisms.
0-11	Fail-Operational behavior	The framework must provide support to achieve fail-operational behavior.
0-23	Redundant channels	The framework must provide support for redundant channels within an avionics system.
0-25	Fault containment	The framework must provide support to achieve fault containment within the avionics systems channels.

Table 5.1: Selection of redundancy framework requirements

5.1.2 Redundancy Management IP Requirements

Based on the requirements for the redundancy framework, further requirements for the redundancy management IP can be derived. The above requirements describe the functions and properties that the IP must possess in order to guarantee the overarching functionality of the framework while implementing the required fault tolerance mechanisms. In the subsequent stages of this work, these requirements will be used in the implementation of the modules that comprise the redundancy management IP. A selection of redundancy management IP requirements is presented in Table 5.2. The complete list of redundancy management IP requirements can be found in the appendix section 7.2 as well.

ID	Name	Text
1-1	Generic design	The RM IP must have a generic and configurable design.
1-2	Hardware description language	The RM IP shall be developed in VHDL.
1-37	Target hardware	The RM IP must be developed to be embedded into the FPGA part of an AMD System-on-Chip.
1-7	Cross-lane interface	The RM IP must have two redundant bidirectional interfaces to communicate between the processing lanes.
1-9	Simultaneous processing	The RM must process all peripheral interfaces of a processing lane simultaneously.
1-14	Fault handling	The RM must perform channel internal fault identification and treatment.
1-16	Fault containment	The RM must prevent fault propagation through the channel boundaries.
1-17	Channel passivation	The RM must passivate a channel if a failure is detected.
1-26	Channel activation	The RM must switch the channel mode from standby to active if required.

Table 5.2: Selection of redundancy management IP requirements

5.2 Framework Concept

The framework is intended to provide a generic tool for developing redundancy management implementations for different avionics systems, architectures and concepts. The hardware and software architecture of the avionics system serves as the informational basis for using the framework. This includes the functional structure of the system, including avionics computers, communication interfaces/paths, and required software partitions. In addition, there are safety and reliability requirements which have an impact on the scope of the redundancy management system. With this information, the framework is intended to configure and provide an FPGA based redundancy management system for an avionics system. The implementation on an FPGA allows the redundancy management system the requisite degree of flexibility and independence from the actual avionics applications. The proposed framework process is depicted in Figure 5.1.

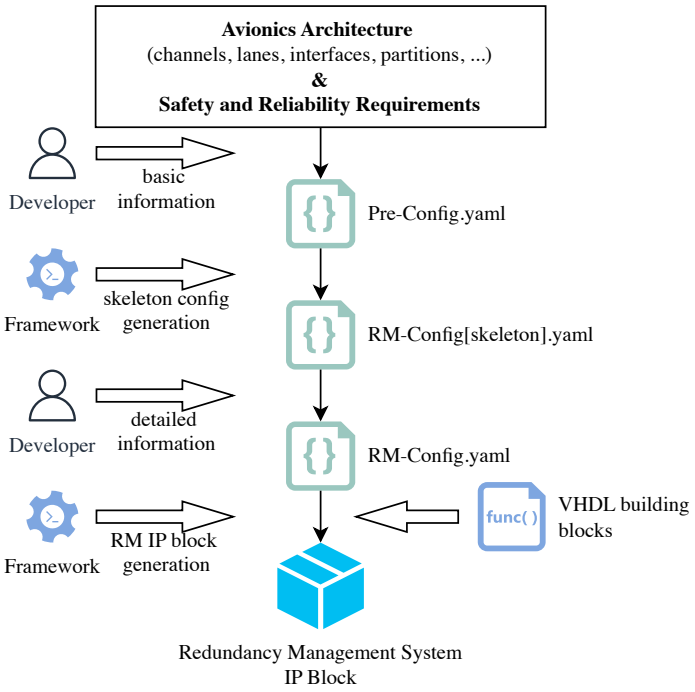


Figure 5.1: Redundancy management framework process

5.2.1 Redundancy Management System Configuration

The redundancy management system is created in two steps. First, the framework provides a configuration environment. The architecture of the avionics system is designed with the requisite fault tolerance mechanisms incorporated into its structure.
A format is established for the description of the configuration. The configuration description is presented in a top-down approach. The channels of the avionics system are defined at the highest level of abstraction. Each channel is assigned an unique identifier, an initial operational state (active or standby), and a list of external data interfaces. The external interfaces are also described with an ID, the name of the physical interface, and its fault tolerance configuration. Furthermore, redundant channels are linked to one another via their respective identifiers. The descriptions of the lanes are defined within a channel too. Additionally, each instance is assigned an unique identifier and a status (either commanding or monitoring) within the configuration file. A link parameter specifies which lanes form a commanding-monitoring pair. In addition, each lane specifies which external channel interfaces it can access and with what permission (read-write or read-only). Communication interfaces between the lanes are also described. Finally, the software partitions within each lane are defined. Each of these partitions is assigned an identification number and connected to one of the external interfaces. The parameters for the description of the communication protocol are also configured at this point. The specifications for each partition's input and output consensus are outlined in the configuration, in accordance with the avionics systems requirements.
A markup/data serialization language, such as JavaScript Object Notation (JSON) or YAML Ain't Markup Language (YAML), may be employed as the foundation for the configuration file. These facilitate the generation of a human- and machine-readable configuration, rendering them highly suitable for the exchange of information between the developer and the framework. An example YAML configuration of one avionics channel with one lane is shown in the following code example.

```yaml
---
channels:
  - channel 1:
        id: 1
        state: active
        replica_channels: none
        external_interfaces:
        - interface_1:
            id: 1
            protocol: uart
            physical interface: rs485
            interface_config: simplex
        lanes:
            - lane_1:
                id: 1
                config: commanding
                linked_lane [ID]: 2
                interfaces [ID]:
                    - 1
                        access: read-write
                cross_lane_interfaces:
                    - cross_lane_1:
                        id: 1
                        configuration: full-duplex
                partitions:
                    - partition_1:
                        id: 1
                        interface [ID]: 1
                        data_type: float32
                        message_header_width [bits]: 16
                        message_payload_width [bits]: 32
                        message_trailer_width [bits]: 16
```

```
33                    message_checksum: CRC16
34                    consensus: input/output
35                    consensus_type: exact
36 ---
```

Listing 5.1: Example YAML configuration of an avionics channel

The configuration is partially completed by the developer through manual input. In certain instances, however, patterns provided by the framework can also be employed by developers, which implement specific fault tolerance techniques (see section 5.3). To facilitate the configuration process, it is necessary to automate it as much as possible. Therefore, the framework provides a pre-configuration file into which the developer can enter the required base-level information:

- Number of available channels
- Number of external interfaces per channel
- Number of lanes per channel
- Number of software partitions per lane
- ID of computing fault tolerance pattern
- ID of general fault tolerance pattern
- ID of interface fault tolerance pattern

The framework generates a skeleton configuration file of the redundancy management system based on the aforementioned information (see Figure 5.1). This file is already pre-filled with the basic parameters, as illustrated in the YAML example above. The file contains the parameter structure for all subsequent descriptions. It is then the responsibility of the developer to enter the missing values. Upon completion of the configuration file, the framework proceeds with the generation of the redundancy management system FPGA IP in the second step.

5.2.2 Redundancy Management System Generation

In the second step, the framework employs the previously constructed configuration to implement the redundancy management system IP. The generation process is based on the concept of building blocks. These building blocks are generic VHDL modules that implement specific functions and fault tolerance mechanisms within a redundancy management system. Such fundamental building blocks include VHDL code implementing a redundant high-speed data interface or the comparison of data pairs, for example. The framework retrieves the configuration parameters from the previously created configuration file, instantiates the required VHDL building blocks within an FPGA project, and configures them in accordance with the gathered parameters. This process is supposed to be executed automatically. An unique RM system implementation is generated for each processing lane.
As illustrated in Figure 5.1 the outcome of this second step is an FPGA IP block in conjunction with the corresponding source code files. This IP block is designed to handle the software partitions hosted in the processing lane and implements the defined measures to achieve fault tolerance. Furthermore, the IP block facilitates both internal and external communication pathways, encompassing communication between partitions, lanes, and even channels. The generated redundancy management IP block can ultimately be directly integrated into the FPGA configuration of the SoC on the respective computing lane.
The development of script-based automation of VHDL IP for the framework is not addressed in this thesis. All process steps of the framework are initially carried out manually. The incorporation of automation through the use of shell scripts, for example, is anticipated as part of the framework's ongoing development.

5.3 Fault Tolerance Pattern

The framework provides techniques and mechanisms for implementing fault tolerance in avionics systems. These mechanisms are provided as patterns. These patterns describe a variety of fault tolerance mechanisms that can be combined in different ways. Patterns are defined for various application areas and purposes, including general fault tolerance,

computational fault tolerance, and interface fault tolerance. The patterns describe the function or mechanism in a general sense, allowing configuration according to the specific use case.

5.3.1 Pattern Structure

This section outlines the general structure of a pattern and describes the elements that must be included to adequately describe a desired function. A pattern must be readable by both humans and machines to be suitable for the redundancy framework. The basic requirement for a pattern within the RM framework is its generic structure. This implies that all patterns are structured in an identical manner and provide the same information. A pattern is defined as a specific configuration of elements that contains the following information:

- Name
- ID
- Type (general, computational, interface)
- Required parent pattern (if a high-level pattern is required for this one to work)
- Required child pattern (if a low-level pattern is required for this one to work)
- Description
- Required configuration input
- Output (what does this pattern create, which module is configured by this pattern)

5.3.2 Computational Fault Tolerance Pattern

The primary focus of computational fault tolerance patterns is the way computations are performed within an avionics system or channel. These patterns primarily involve different configurations of channels and lanes.

Commanding-Monitoring Pattern

The commanding-monitoring pattern defines a metric for configuring an avionics channel lane to either commanding or monitoring mode. For example, the command-monitoring configuration within a channel is essential for achieving consensus. When the pattern is applied to a channel configuration, the lane configuration is applied accordingly. In addition, a VHDL module is generated in which the configuration (commanding/monitoring) is programmed to facilitate its availability to other modules within the RM system. The entry in the channel configuration also results in corresponding adjustments to the logic of other RM modules. The specific modules affected by this are discussed in detail in the implementation chapter. A summary of the pattern is shown in Table 5.3.

Name	Commanding-Monitoring
ID	1
Type	computing
Parent pattern	none
Child pattern	none
Affected component	lane
Description	This pattern implements a commanding-monitoring configuration for an avionics channel.
Configuration input	• Lane ID and desired configuration (commanding/monitoring)
Output	• VHDL module for each lane programmed with the configuration • Entry in the channel and lane configuration • Adaptations in corresponding RM IP modules

Table 5.3: Commanding-Monitoring pattern

Active-Standby Pattern

The active-standby pattern applies an activity configuration to multiple channels in an avionics system. Their initial state is set to either active or standby. The state of a channel may change during runtime. An active channel fully participates in the communication with peripheral components (compute, send and receive). While a standby channel only

partially participates in a multi channel system (compute and receive). Applying this pattern to an avionics system would allow for a hot-redundant configuration where the standby channels operate simultaneously with the active channel. In the event of a failure that causes the active channel to passivate, one of the standby channels would change its activity state and take over the active part. A summary of the pattern is shown in Table 5.4.

Name	Active-Standby
ID	2
Type	computing
Parent pattern	none
Child pattern	none
Affected component	channel
Description	This pattern implements an active-standby configuration for a channel within an avionics system. The configuration applies the specific activity state (active or standby) to the channel. The state of the channel may change during runtime.
Configuration input	• Channel ID and desired initial configuration (active/standby)
Output	• Entry in the channel configuration • Adaptations in corresponding RM IP modules

Table 5.4: Active-Standby pattern

Reconfiguration Pattern

The reconfiguration pattern provides logic to reconfigure failed (passivated) components back to a correct state. This can be applied to channels, lanes, or even individual software partitions. The reconfiguration logic would trigger a reset routine to restore the default state after the failed components were identified and passivated. This could include power cycling a channel or rebooting the processor or individual software partitions. After restarting the dedicated component(s), they would receive a failed-standby state. The component works in parallel with the others. Meanwhile, the reconfiguration logic is responsible for checking the calculated outputs. The trust value of a recovered compo-

nent is increased if the generated outputs are correct, and decreased if the calculations are incorrect. Once the component has reached a sufficiently high trust level, it can be assumed that it is working properly again. Its state is then reset to correct-standby, and the reconfiguration of the component is complete. The pattern requires the provision of information about the configuration, specifically, which component is to be reconfigurable (channel, lane, partition) and how the trust threshold is to be set (the percentage of correctness required for the component). Accordingly, the pattern generates an appropriate entry in the configuration and VHDL logic for the component to be reconfigured. This pattern enables fail-operational behavior in a system with a limited number of channels or lanes. A summary of the pattern is shown in Table 5.5.

Name	Reconfiguration
ID	3
Type	computing
Parent pattern	none
Child pattern	none
Affected component	channel, lane, partition
Description	This pattern implements a configuration and logic to enable reconfiguration or failed components (channel, lane, partition). The logic would reset and restart the failed component in a standby state decoupled from the rest of the system. If the component works correctly again with a specified success rate, it would achieve a correct-standby state. Otherwise, it remains failed and passivated.
Configuration input	• Channel/lane/partition ID • Required trust value to regain correct state
Output	• Entry in the channel/lane/partition configuration • VHDL reconfiguration logic for dedicated component • Adaptations in corresponding RM IP modules

Table 5.5: Reconfiguration pattern

5.3.3 Interface Fault Tolerance Pattern

Interface fault tolerance patterns are designed to configure interfaces on a channel or lane. These tools are especially helpful when implementing interface redundancy (e.g., replicated interfaces). In addition, the system includes logic to ensure the security of data transmission and the integrity of the data itself.

Simplex Data Simplex Interface (SDSI) Pattern

The Simplex Data Simplex Interface (SDSI) pattern implements a configuration for a simplex interface that transfers simplex data. This is the simplest configuration and does not use any form of redundancy or integrity checking. An example would be a single sensor interface that connects one type of sensor to an avionics computer. A summary of the pattern is shown in Table 5.6.

Name	Simplex Data Simplex Interface
ID	4
Type	interface
Parent pattern	Cross-Lane Interface, Cross-Channel Interface
Child pattern	none
Affected component	interfaces
Description	This pattern implements a configuration for a simplex interface of a channel and/or lane. The SDSI configuration assigns a single interface to a corresponding partition. This pattern does not introduce redundancy or increase data integrity.
Configuration input	• Channel, lane ID • Interface ID • ID of corresponding partition
Output	• Entry in the channel/lane/partition configuration • Data port in VHDL module to connect the interface

Table 5.6: Simplex Data Simplex Interface pattern

Simplex Data Duplex Interface (SDDI) Pattern

The Simplex Data Duplex Interface (SDDI) pattern configures a port to drive two replicated hardware interfaces, each representing a simplex interface. Both interfaces are configured to operate simultaneously in a hot-redundant mode while using the data from only one interface. Implementing this pattern in an avionics system helps increase the reliability of the physical transmission and gives the interface fail-operational behavior in the event of a failed/incorrect transmission in one of the interfaces. Because the data is still simplex (e.g., one sensor with two interfaces), sensor integrity is not considered by this pattern. This pattern could be applied to both external and internal (cross-lane) interfaces. A summary of the pattern is shown in Table 5.7.

Name	Simplex Data Duplex Interface
ID	5
Type	interface
Parent pattern	Cross-Lane Interface, Cross-Channel Interface
Child pattern	none
Affected component	interfaces
Description	This pattern implements a configuration for a duplex interface of a channel and/or lane. The SDDI configuration assigns a duplex interface to an appropriate partition. Since the transmitted data is still simplex, a logic checks the integrity of both data streams and forwards the data from the interface whose transmission is correct.
Configuration input	• Channel, lane ID • Interface ID • ID of corresponding partition
Output	• Entry in the channel/lane/partition configuration • Data port in VHDL module to connect the interfaces • VHDL logic to determine an incorrect transmission and data forwarding

Table 5.7: Simplex Data Duplex Interface pattern

Duplex Data Simplex Interface (DDSI) Pattern

The Duplex Data Simplex Interface (DDSI) pattern implements the configuration of replicated data streams over a simplex interface. An example would be a redundant set of sensors sharing one data bus. This would increase the data integrity but not the transmission reliability. Since a replicated set of values is transmitted over one medium, the pattern also implements logic to sample and select the individual values from each source (e.g., sensor). Cross-checking these two sets would then allow for the detection of incorrect values. In the case of an avionics channel transmitting data to a duplex set of actors, the logic would merge both sets of data to prepare for transmission over a simplex interface. A summary of the pattern is shown in Table 5.8.

Name	Duplex Data Simplex Interface
ID	6
Type	interface
Parent pattern	none
Child pattern	none
Affected component	interfaces
Description	This pattern implements a configuration for a simplex interface of a channel and/or lane that transmits a duplex set of values. The DDSI configuration assigns a simplex interface to an appropriate partition. It then samples and selects the values from each source and provides a replicated set of values for further review. To transmit duplex data, the pattern provides logic to merge replicated records for transmission over a simplex interface.
Configuration input	• Channel, lane ID • Interface ID • ID of corresponding partition
Output	• Entry in the channel/lane/partition configuration • Data port in VHDL module to connect the interface • VHDL logic to sample, select and merge replicated data streams from one interface

Table 5.8: Duplex Data Simplex Interface pattern

Duplex Data Duplex Interface (DDDI) Pattern

The Duplex Data Duplex Interface (DDDI) pattern supports duplex interfaces that transfer replicated (duplex) data sets to and from multiple peripherals (e.g., sensors, actors). The pattern configures hardware interfaces to operate hot-redundantly to avoid switching delays and re-transmission of messages (similar to the SDDI pattern). Each interface then works on its own with a replicated data set (identical to the DDSI pattern). The sampling, selection, and merge logic is also provided by the pattern to enable monitoring of replicated data sources. This approach would increase both data integrity and communication reliability. A summary of the pattern is shown in Table 5.9.

Name	Duplex Data Duplex Interface
ID	7
Type	interface
Parent pattern	none
Child pattern	none
Affected component	interfaces
Description	This pattern implements a configuration for a duplex interface of a channel and/or lane that transmits a duplex set of values. The DDDI configuration assigns a duplex interface to an appropriate partition. Both interfaces are configured to operate hot-redundant. It also samples and selects the values from each interface source and provides a replicated set of values for further review. To transmit duplex data, the pattern provides logic to merge replicated data sets for transmission over each interface.
Configuration input	• Channel, lane ID • Interface ID • ID of corresponding partition
Output	• Entry in the channel/lane/partition configuration • Data port in VHDL module to connect the interface • VHDL logic to sample, select and merge replicated data streams from one interface • VHDL logic to determine an incorrect transmission and data forwarding

Table 5.9: Duplex Data Duplex Interface pattern

Cross-Lane Interface (CLI) Pattern

The framework is required to support communication and data exchange between lanes within a channel. This is realized by the CLI pattern. The pattern provides a VHDL building block for such an interface and configures it according to the information provided by the developer (message width, data rate, etc.). The basic building block represents a single bi-directional interface capable of full-duplex communication (sending and receiving simultaneously). This pattern can be combined with some of the above to create, for example, a dual-replicated communication interface (combined with the SDDI pattern). A summary of the pattern is shown in Table 5.10.

Name	Cross-Lane Interface
ID	8
Type	interface
Parent pattern	none
Child pattern	SDSI, SDDI, CRC
Affected component	lane, interfaces
Description	This pattern implements a CLI to enable communication and data exchange between lanes within a channel. By applying the pattern, it provides configured VHDL logic that implements the CLI. Optional applied child pattern may extend the CLI configuration. The basic configuration is a bi-directional, full-duplex interface (transmit and receive simultaneously). It can be configured up to a dual replicated bi-directional interface to increase module reliability. To ensure physical transmission, the pattern could be extended to include the Cyclic Redundancy Check (CRC) pattern.
Configuration input	• Lane ID • Message width • Data rate
Output	• Entry in the channel/lane configuration • Data port in VHDL module to connect the interface • VHDL logic implementing the CLI

Table 5.10: Cross-Lane Interface pattern

Cross-Channel Interface (CCI) Pattern

To increase the availability of an avionics system, multiple channels could be integrated into the system to provide computing resources even if one channel fails. However, this concept requires a reliable communication and data exchange interface between the channels to transmit failure and passivation events as well as reconfiguration commands. The Cross-Channel Interface (CCI) pattern provides a VHDL building block for such an interface. Similar to the CLI, the CCI has a bi-directional, full-duplex capable communication interface that can be configured as a dual-replicated interface if required. A summary of the pattern is shown in Table 5.11.

Name	Cross-Channel Interface
ID	9
Type	interface
Parent pattern	none
Child pattern	SDSI, SDDI, CRC
Affected component	channel, interfaces
Description	This pattern implements a CCI to enable communication and data exchange between channels within an avionics system. By applying the pattern, it provides configured VHDL logic that implements the CCI. Optionally applied child patterns can extend the CCI configuration. The basic configuration is a single, full-duplex, bidirectional interface (transmit and receive simultaneously). It can be configured up to a dual replicated bi-directional interface to increase module reliability. To ensure physical transmission, the pattern could be extended to include the CRC pattern.
Configuration input	• Channel ID • Message width • Data rate
Output	• Entry in the channel/lane configuration • Data port in VHDL module to connect the interface • VHDL logic implementing the CLI

Table 5.11: Cross-Channel Interface pattern

Cyclic Redundancy Check (CRC) Pattern

Applying a CRC is a common way to check whether a physical data transmission is correct or not. This check takes the data being transmitted and calculates a checksum, which is then appended to the message. The message receiver also calculates the checksum of the received data and compares it with the received checksum. The data exchange is considered correct if both calculated checksums are identical. A mismatch would indicate a transmission error. The pattern supports a 16 bit checksum (CRC16) with reflected inputs and outputs and requires the developer to configure the CRC polynomial used. The pattern provides the developer with a configured VHDL logic that implements the specified CRC algorithm. A summary of the pattern is shown in Table 5.12.

Name	Cyclic Redundancy Check
ID	10
Type	interface
Parent pattern	CLI, CCI
Child pattern	none
Affected component	interfaces
Description	This pattern implements a CRC with reflected inputs and outputs for a given data communication interface. The CRC can be used to ensure a correct physical transfer. The pattern outputs VHDL logic that implements the CRC algorithm.
Configuration input	• Channel ID • Interface ID • 16 bit CRC polynomial
Output	• Entry in the channel/lane configuration • VHDL logic implementing the CRC algorithm

Table 5.12: Cyclic Redundancy Check pattern

5.3.4 General Fault Tolerance Pattern

General fault tolerance patterns are independent of application partition computing or interface configuration. They focus primarily on consensus, component/module replication, and dissimilarity.

Output Consensus Pattern

Channels with multiple lanes can be configured to cross-check each other to detect errors in internal computations, such as in the commanding-monitoring principle. Before transmitting the calculated data to external systems, it is necessary for the lanes to agree on their data in order to reach a consensus. Although the result may still be erroneous, it is unlikely that all lanes will compute the same erroneous result identically. The agreement on the computed results is called output consensus and is implemented by this pattern. The developer specifies which components (lane, partition) should be subject to consensus analysis. The pattern then provides configured VHDL logic to implement the comparison of the data. A summary of the pattern is shown in Table 5.13.

Name	Output Consensus
ID	11
Type	general
Parent pattern	none
Child pattern	Exact Consensus, Approximate Consensus
Affected component	lane, partition, interface
Description	This pattern implements the necessary procedures to ensure output consensus. The developer defines which components (redundant replicas) to check, and the pattern creates the appropriate VHDL logic to compare the values. The consensus algorithms in the redundant replicas then decide whether or not the output consensus is reached.
Configuration input	• Channel ID • Replica IDs (lanes, interfaces, partitions)
Output	• Entry in the channel/lane configuration • VHDL logic implementing the output consensus algorithm

Table 5.13: Output Consensus pattern

Input Consensus Pattern

To ensure the reliability of the calculated data, it is also necessary to achieve consensus on the input data. This is a necessary prerequisite for achieving output consensus. To reach

a consensus on the input data, the information from the replica interfaces is compared. The configuration of the external interfaces plays a significant role in determining the suitability of replica interface pairing. Configuring replicated interfaces can result in a variety of different pairings, depending on the interface pattern used.

- In the context of SDSI interfaces, data is compared with the corresponding interface of the other lane.
- In the case of duplex interfaces (SDDI), both interfaces are compared with those of the other lane. This comparison is conducted in a manner that A is compared with A and B is compared with B.
- In the context of DDSI-configured interfaces, the duplex data pairs of both lanes are subjected to a comparison process.
- In the context of DDDI interfaces, the duplex interfaces of both lanes are cross-checked with each other (interface A with B and B with A).

Each comparison is performed independently in each lane, and then the results of both lanes are compared. If the results of both lanes are identical, it can be concluded that the received input data has been processed accurately. A summary of the pattern is shown in Table 5.14.

Name	Input Consensus
ID	12
Type	general
Parent pattern	none
Child pattern	Exact Consensus, Approximate Consensus
Affected component	lane, partition, interface
Description	This pattern implements the necessary procedures to ensure input consensus. The developer defines which interfaces and components (redundant replicas) to check, and the pattern creates the appropriate VHDL logic to compare the values. The consensus algorithms in the redundant replicas then decide whether or not the input consensus is reached. The actual implementation depends on the configuration of the input interfaces and which interface pattern is used (SDSI, SDDI, DDSI, DDDI).
Configuration input	• Channel ID • Replica IDs (lanes, interfaces, partitions)
Output	• Entry in the channel/lane configuration • VHDL logic implementing the input consensus algorithm

Table 5.14: Input Consensus pattern

Exact Consensus Pattern

When comparing data to demonstrate consensus, the comparison can be either exact or approximate. This pattern employs an exact (bit-wise) comparison of data. To achieve consensus, the data must be identical in every bit. This pattern serves as a child pattern for an I/O consensus pattern, defining the level of detail at which input or output data should be compared. The proposed methodology extends the comparison logic of an I/O consensus pattern by incorporating bit-wise comparison. A summary of the pattern is shown in Table 5.15.

Name	Exact Consensus
ID	13
Type	general
Parent pattern	Input Consensus, Output Consensus
Child pattern	none
Affected component	lane, partition, interface
Description	This pattern implements an exact (bit-wise) comparison of data sets. It acts as a child pattern for an I/O consensus pattern and extends the comparison logic by the bit-wise comparison.
Configuration input	• Channel ID • Replica IDs (lanes, interfaces, partitions)
Output	• Entry in the channel/lane configuration • VHDL logic implementing the bit-wise data comparison

Table 5.15: Exact Consensus pattern

Approximate Consensus Pattern

Unlike the exact consensus pattern, the approximate consensus pattern allows for some discrepancy in the data as long as it is within a defined tolerance range or maximum deviation. This allows for a more flexible approach to achieving consensus. This approach is only useful for sensor values that represent physical quantities, which are often not identical due to the inherent variability of measurement principles. The size of the tolerance range depends on the data format, the range of values, and the required accuracy or precision. However, determining an appropriate tolerance range is a complex process that is typically based on empirical results and experience (as previously discussed in chapter 2). Like the exact consensus pattern, the approximate consensus pattern functions as a child pattern for an I/O consensus pattern. It refines the required consensus check for an interface or partition by allowing a tolerance range or maximum deviation when comparing data. A summary of the pattern is shown in Table 5.16.

Name	Approximate Consensus
ID	14
Type	general
Parent pattern	Input Consensus, Output Consensus
Child pattern	none
Affected component	lane, partition, interface
Description	This pattern implements an approximate comparison of data sets. It acts as a child pattern for an I/O consensus pattern and extends the comparison logic to include approximate comparison. The maximum tolerated deviation between data sets must also be defined.
Configuration input	• Channel ID • Replica IDs (lanes, interfaces, partitions) • Maximal tolerated value deviation
Output	• Entry in the channel/lane configuration • VHDL logic implementing the approximate data comparison

Table 5.16: Approximate Consensus pattern

Replication Pattern

One of the most common redundancy practices is the replication of redundant (similar) modules. Instantiating multiple copies of the (functional) same module allows for cross-checking between modules, or would increase the reliability of a system by using redundant modules as standby modules to take over the active part when others fail. Replica modules are functionally identical copies. The replication pattern implements the instantiation of redundant copies of a given module/component. For example, the developer could assign the replication pattern to a compute lane, which is then instantiated n-times in the avionics channel. A summary of the pattern is shown in Table 5.17.

Name	Replication
ID	15
Type	general
Parent pattern	none
Child pattern	none
Affected component	channel, lane, partition, interface
Description	This pattern implements replication of a specified module/component to obtain redundant copies of it. The pattern can be assigned to either channels, lanes, partitions, or interfaces. It is one of the most important patterns and defines the overall configuration of the avionics as well as the redundancy management system.
Configuration input	• Component ID (channel, lane, partition, interface) • Amount of required replications
Output	• Entry in the component configuration • Configuration of the overall redundancy management system

Table 5.17: Replication pattern

Dissimilarity Pattern

As already discussed in chapter 2, dissimilarity is a key concept in avoiding common-mode failures in redundant copies (replicas). Therefore, the redundancy management framework provides a specific pattern to configure dissimilarity in the redundancy management system. The pattern does not intend to design or implement dissimilarity at the software partition or avionics hardware level, but for the redundancy management system instantiated on the avionics system lanes. This pattern basically defines if the RM modules for the FPGA are supposed to be implemented in different HDLs. A summary of the pattern is shown in Table 5.18.

Name	Dissimilarity
ID	16
Type	general
Parent pattern	none
Child pattern	none
Affected component	RM system on a lane
Description	This pattern allows for dissimilar implementations of redundancy management systems in the avionics lanes. When assigned, the pattern configures the RM system on each lane in which HDL it must be synthesized.
Configuration input	• Lane ID of RM system to be configured • Hardware Description Language (HDL)
Output	• Entry in the channel and lane configuration • Configuration and synthesis of the RM system in the specified HDL

Table 5.18: Dissimilarity pattern

5.4 Building Block Design

In addition to the configuration environment and patterns, the redundancy management building blocks represent a crucial component of the redundancy management framework. These generic VHDL modules serve as the foundation for the framework and are responsible for constructing the desired redundancy management system implementation. In this work, a generic FPGA-based redundancy management system is developed as a preliminary step in establishing the portfolio of these building blocks. The following sections address the design, implementation, and verification of building blocks forming a generic redundancy management system.

Considering the framework requirements (0-10 and 0-24 see section 7.1.1), the initial iteration of the redundancy management framework supports a redundancy management system developed for a commanding-monitoring avionics channel. Consequently, within such channels, there are two lanes that mutually verify each other in order to identify a calculation fault or a faulty sensor value. Both lanes operate as a self-checking pair,

meaning that they are functionally identical in principle. In order to verify the accuracy of the data, each lane transmits its calculated or received data to the other lane. This allows each lane to compare its own data with that of the other lane, thus ensuring the integrity of the data as demanded by the requirements 1-30 and 1-31, see section 7.2.1. Subsequently, both lanes provide feedback on whether the data is in agreement or in disagreement with one another. In order for a lane to be considered to have interpreted its data correctly, the results of the checks carried out in both lanes must match exactly. This ensures that data verification occurs in an equally rigorous manner in both lanes, thereby preventing the occurrence of a single point of failure. The objective of this process is to achieve consensus on the input and output data of a channel (requirements 1-21 and 1-22, see section 7.2.1). Should the data verification process be successful, the commanding lane will transmit the data via the peripheral interfaces. Conversely, the monitoring lane will be configured to only receive data.
In this configuration, a channel serves as a fault containment region. The objective of the RM system is to prevent faults from propagating beyond the boundaries of the channel's FCR and affecting or compromising other channels or peripheral systems (requirements 1-14 and 1-16, see section 7.2.1).
The data to be examined is encoded in a message protocol format that is specified by the use-case avionics system addressed in this thesis. Consequently, a message is 8 bytes (64 bits) in length and comprises a 16-bit header, a 32-bit payload, and a 16-bit checksum. The precise format of the message is illustrated in Table 5.19.

Byte	**Bits**	**Field Name**	**Description**
8	63 - 56	Label	The label indicates the message type
7	55 - 48	Destination	ID of message receiver
6	47 - 40	Payload-4	Payload bits 31 - 24
5	39 - 32	Payload-3	Payload bits 23 - 16
4	31 - 24	Payload-2	Payload bits 15 - 8
3	23 - 16	Payload-1	Payload bits 7 - 0
2	15 - 8	CRC-2	CRC bits 15 - 8
1	7 - 0	CRC-1	CRC bits 7 - 0

Table 5.19: Data message protocol format

Figure 5.2 provides a simplified overview of the SoC configuration of a computing lane. The upper part of the figure depicts the processing system of the SoC, which includes a hypervisor runtime environment that hosts the avionics software partitions and a health monitoring module. The specifics of these software components are not included in this thesis and are presented here for the sake of completeness.

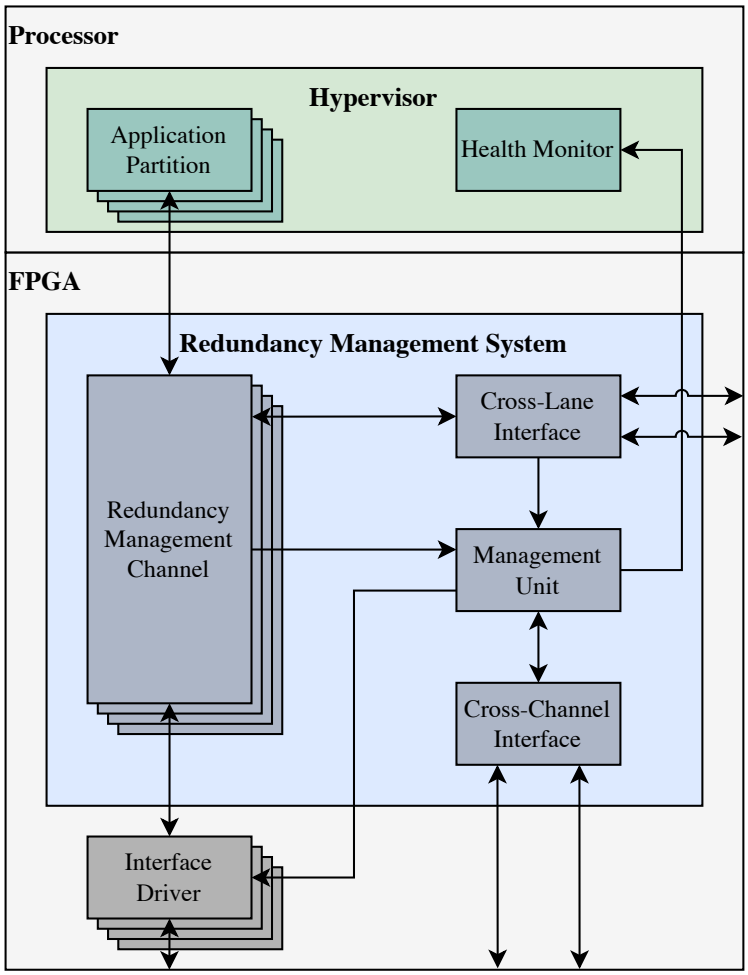


Figure 5.2: SoC configuration of an avionics computing lane

The lower part of the figure represents the FPGA component of the SoC. Alongside the interface driver modules, this section contains the RM system. The embedding of the RM system within the FPGA section serves to achieve the desired decoupling between the RM system and the application partitions. This decoupling permits the execution of the RM system and the application to proceed independently, thus improving performance. As illustrated in Figure 5.2, an RM system is comprised of four modules: the Redundancy Management Channel (RMC), CLI, CCI, and Management Unit (MU). Collectively, these modules implement the required fault tolerance measures for a computing lane.

5.4.1 Redundancy Management Channel

An RMC within the RM system is responsible for processing the payload data, which is either produced by an application partition or received from a sensor or other peripheral system. The processing activities of the system include data buffering, initiating data exchange with the other lane, comparing replica data, and evaluating this comparison. Consequently, the role of the RMC is to facilitate consensus on both input and output data. For each instance of an application partition, the RM system instantiates a separate RMC module. Consequently, the number of RMCs integrated in parallel within an RM system is contingent upon the number of application partitions. In this manner, the redundancy management system facilitates the dynamic reconfiguration of individual partitions across different lanes or channels. The FPGA's capability for dynamic reconfiguration of individual modules through partial bitstreams enables the framework to facilitate a joint exchange of application partitions and the associated RMC modules.

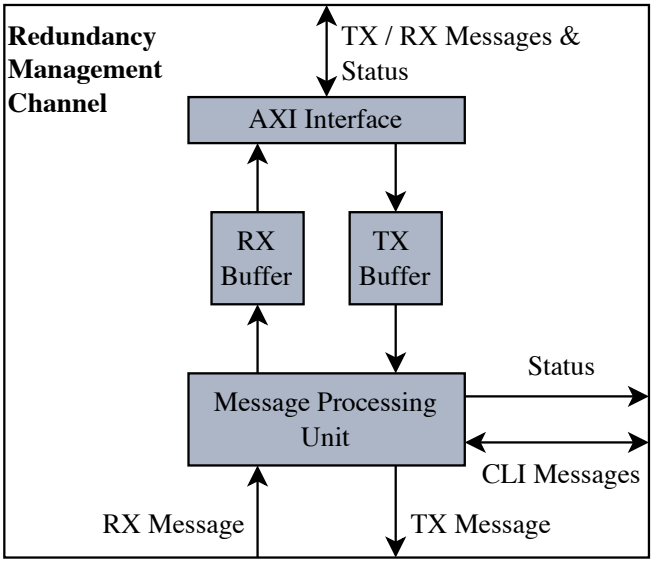


Figure 5.3: Redundancy Management Channel architecture

Figure 5.3 depicts the architecture of an RMC. An RMC is comprised of multiple elements, each of which implements a distinct function. The AXI interface (depicted at the top of the figure) is designed for interconnecting the application partition with the RMC. This interface enables bidirectional data communication between the partition within the processor and the RMC located within the FPGA. This interface facilitates the exchange of Transmit (TX) and Receive (RX) messages, as well as status information pertaining to the RMC. The AXI bus employs separate data paths, enabling the interface to operate in full-duplex mode. All TX messages transmitted by the software are initially stored in a buffer within the RMC. This prevents message loss due to the asynchronous nature of the communication between the processor and FPGA (partition and RMC).
The paths for TX and RX messages are also segregated within the RMC, thus enabling full-duplex operation within the module. Consequently, TX and RX messages can be processed simultaneously by the RMC. In light of the aforementioned considerations, it is evident that the RMC also possesses separate buffer memories. Subsequently, the buffered TX messages are individually subjected to processing by the Message Processing Unit (MPU). The MPU initiates the required steps for data exchange and comparison with

the other lane. To this end, the MPU reads a message from the memory and integrates it into a Cross-Lane (CL) message in accordance with the CLI message protocol.
The CLI module represents a shared resource within the RM system, as it is instantiated only once and multiple RMCs within the system can access it. For this reason, access to the CLI is secured by a lock managed by the CLI itself. In order for an RMC to transmit a CL message, it must first submit a request to the CLI module. Once the CLI has granted access, the RMC is then allowed to send its message. Upon completion of the transmission process, the next RMC in the queue is then able to transmit its message to the CLI.
Simultaneously, the MPU awaits receipt of a corresponding CL message containing the TX data from the other lane. Once the message is received, the comparison of the TX data begins. The verification process entails a comparison of the message header, payload, and checksum. It should be noted that the headers and checksum are always subjected to an exact consensus principle. This implies that the headers and checksum of both messages must be identical in every bit. The developer may configure the principle of verifying the payload data by applying a pattern to each RMC (see section 5.3.4). This may be done in accordance with either the exact (bit-wise) or approximate consensus principle. Once both messages have been verified, the MPU transmits the results to the other lane via the CLI, while simultaneously awaiting the results from the other lane. If both lanes reach the same conclusion regarding the data, the RMC of the commanding lane transmits the TX message to the respective interface driver. However, in the event that both lanes determine that the data does not match or if the results of both lanes differ, it is not possible to guarantee output consensus, and the transmission of the TX message is not initiated. In either case, a corresponding status message is transmitted to the management unit.
In parallel, the RX path of the MPU awaits receipt of an RX message. Once it arrives, the message is subjected to the same check as in the TX path. Consequently, the RMCs of both lanes exchange their received data, compare them, and report the results of their evaluation to each other. If both lanes determine that the received data is identical, the message is stored in the receive buffer. The application partition is then informed about the new message and can read it via the AXI interface. In the event that the verification process is invalid, the data will not be transmitted to the application partition.

Additionally, a corresponding status message will be sent to the MU. This methodology enables the implementation of the input consensus principle.
In general, an RMC must be capable of recognizing and accepting specific messages at any time, without exception. This encompasses TX, RX, and CL messages. The individual modules in the RM system are not synchronized with each other and are executed simultaneously in the FPGA. Consequently, an RMC is unable to predict the arrival of specific data. Accordingly, when implementing the RMC logic, it is of importance to ensure that input data can be received and buffered by the module at any time.

5.4.2 Cross-Lane Interface

The CLI is responsible for facilitating communication and data exchange between the lanes within a channel (requirement 1-7 , see section 7.2.3). In a commanding-monitoring channel, the data from individual partitions/RMCs must be exchanged between the commanding and monitoring lanes. The CLI provides a bidirectional, full-duplex capable data interface and has two redundant communication paths (CLI channels). The two CLI channels are operated in a hot-redundant mode, whereby the same data is transmitted over both channels. In the event of a transmission fault in one CLI channel, the receiver is able to access the data from the second channel directly, without the need to re-transmit the CL message. This not only enhances the dependability of the interface but also prevents a performance degradation in the event of a CLI channel failure.
Requirement 1-36 (see section 7.2.4) demands that the redundancy management system shall not produce a temporal offset resulting from message processing exceeding 10 ms. The serial data communication between the lanes represents a significant bottleneck in terms of timing, which is why the required performance of the CLI module has a significant influence on the module design. This becomes of particular significance when a large number of RMCs are instantiated within the system, which in turn increases the number of CL messages that must be transmitted per execution cycle.

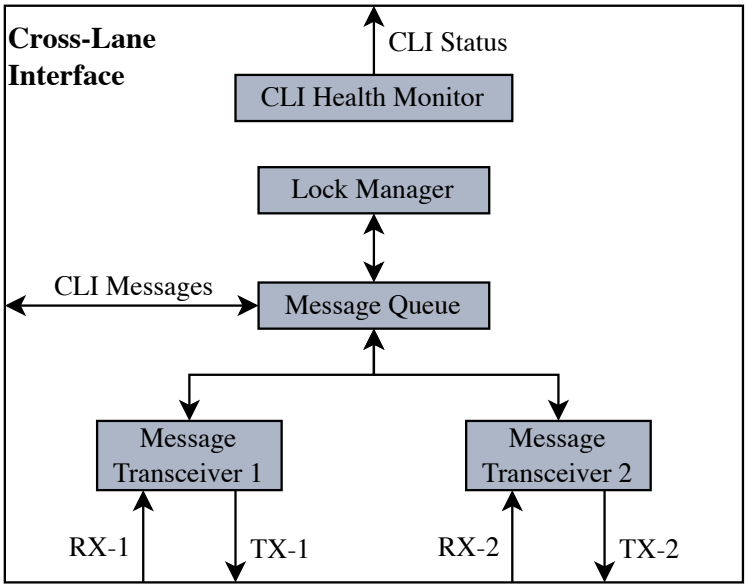


Figure 5.4: Cross-Lane Interface architecture

Figure 5.4 depicts the architecture of the CLI module. The CLI implements a bidirectional interface for each RMC in the system, enabling the exchange of CL messages in both directions. As explained in the RMC design section (5.4.1), the CLI is a shared resource utilized by multiple RMCs. While the FPGA implementation would permit the CLI to receive messages from multiple RMCs in parallel, the physical transmission of CL messages between the lanes occurs sequentially. Consequently, the serialization of CL messages is a necessity. In order to reduce the complexity of the CLI, the serialization of CL messages is implemented at the access to the CLI. In order to prevent the simultaneous access of multiple RMCs to the CLI, a locking mechanism has been designed. In order for an RMC to transmit CL messages to the CLI, it must first request access through a lock request. The CLI module then evaluates the requests and decides, based on a decision matrix, which RMC is granted access. Once the RMC has received the lock grant, it will transfer its message, which will then be stored by the CLI in the message queue. It is of importance that, when developing the decision matrix, only one RMC is permitted to receive a grant at any given time. Furthermore, it is essential that this grant cannot be revoked by another RMC while the transmission of a CL message is still ongoing. Furthermore, the distribution of lock grants should be fair, ensuring that no RMC is

permanently disadvantaged. However, prioritizing RMCs or partitions may be a viable option, particularly if they are scheduled with a high frequency and cannot wait long for a grant due to their short execution time window.
The management of lock requests and the granting of lock grants is the responsibility of the lock manager. In collaboration with the message queue, the lock manager ensures the orderly access to the CLI and the correct serialization and buffering of CL messages. Once the CL messages have been stored in the queue, they are read by the two message transceivers and transmitted via the physical interface. Simultaneously, the transceivers receive incoming CL messages from the opposing lane. Upon receipt of a CL message, it is transmitted to the corresponding RMC. The address of the target RMC can be identified from the header of the CL message. It is possible for the transmission of the CL message to the RMC to occur without a request and grant, given that an RMC must be capable of receiving incoming CL messages at any time.
The physical transmission between the lanes is secured by a CRC checksum, which allows the detection of multiple transmission errors in individual bits per message. A checksum is calculated for each CL message within the CLI module and appended to the message. Similarly, the checksum of received CL messages is verified within the CLI prior to forwarding the message to an RMC. This functionality enables the CLI to detect transmission errors and, if necessary, utilize the data from the second CLI channel.
In conjunction with the processing of CL messages, the CLI management logic oversees the execution and records all status information. This encompasses the recording of any transmission errors that may have occurred within a CLI channel, including instances of a faulty CRC, as well as the documentation of any messages that did not arrive or were not sent at all. The resulting status message is then transmitted to the Management Unit (MU) for assessment.
As with data messages, a distinct communication protocol is established for CL messages. Furthermore, CL messages distinguish between CL-request and CL-response messages. CL-request messages comprise a complete data bus message (see Table 5.19) and are transmitted to the other lane with the request to verify the data. Upon completion of the verification process in the other lane, a corresponding CL-response message is generated, which includes the result of the verification along with an unique identifier to map the result to the requested data bus message. Both CL message types are comprised of a

16-bit header and a 16-bit checksum. The header is composed of a message identifier as well as an RMC identifier. The CL message header serves to indicate the type of CL message (request or response), whether the message contains TX or RX data, and from which RMC the CL message originates. In the other lane, the message must then be forwarded to the corresponding RMC. However, the length of the payload differs between the two CL message types. Thus, an 8-byte payload is transmitted with a CL-request message, while a CL-response message carries a 3-byte payload. The structure of both CL message types is depicted in Table 5.20 and 5.21.

Byte	**Bits**	**Field Name**	**Description**
12	95 - 88	CL message ID	Type of CL message: 0x11 = CL-request (TX data) 0x21 = CL-request (RX data)
11	87 - 80	RMC ID	ID of originating and destination RMC
10	79 - 72	Payload-8	Data bus message byte 8 (label)
9	71 - 64	Payload-7	Data bus message byte 7 (destination)
8	63 - 56	Payload-6	Data bus message byte 6 (payload-4)
7	55 - 48	Payload-5	Data bus message byte 5 (payload-3)
6	47 - 40	Payload-4	Data bus message byte 4 (payload-2)
5	39 - 32	Payload-3	Data bus message byte 3 (payload-1)
4	31 - 24	Payload-2	Data bus message byte 2 (crc-2)
3	23 - 16	Payload-1	Data bus message byte 1 (crc-1)
2	15 - 8	CRC-2	CRC bits 15 - 8
1	7 - 0	CRC-1	CRC bits 7 - 0

Table 5.20: CL-request message protocol format

Byte	Bits	Field Name	Description
7	55 - 48	CL message ID	Type of CL message: 0x12 = CL-response (TX data) 0x22 = CL-response (RX data)
6	47 - 40	RMC ID	ID of originating and destination RMC
5	39 - 32	Payload-3	Data bus message byte 8 (label)
4	31 - 24	Payload-2	Data bus message byte 7 (destination)
3	23 - 16	Payload-1	Verification result
2	15 - 8	CRC-2	CRC bits 15 - 8
1	7 - 0	CRC-1	CRC bits 7 - 0

Table 5.21: CL-response message protocol format

The verification result byte comprises the outcome of the payload data check (bits 7 to 4) and the CRC check (bits 3 to 0). A data acknowledgment (indicating the presence of correct data) is coded with $0xA$, while a data rejection (indicating the presence of incorrect data) is coded with $0xB$.

5.4.3 Management Unit

The Management Unit is responsible for the evaluation of status messages, as well as for the determination of the state of each RMC, the CLI, and the CCI. The module states that have been determined are employed by the MU to ascertain the state of the lane and channel in the subsequent step. The MU classifies each element according to two distinct state types: *functional* and *activity*. The functional state of a module can be either correct or failed. The activity state indicates whether a module is active, standby, or passive. The overall state of an element is defined by its functional and activity state. Consequently, an element may be classified as either correct-active, correct-standby or failed-passive. This classification is applied to the RM system modules (RMCs, CLI, CCI), as well as its own lane and the entire channel. The classification is conducted in the status manager of the MU (see Figure 5.5).

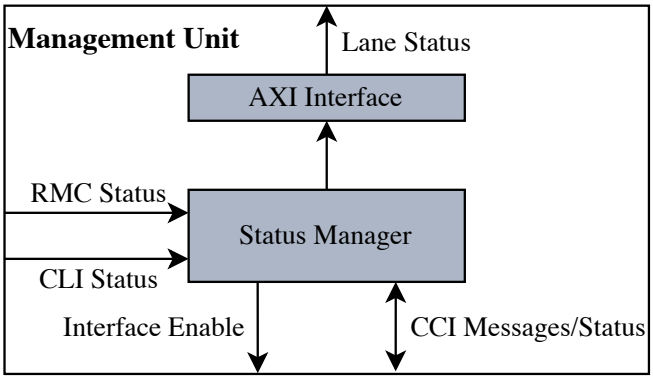


Figure 5.5: Management Unit architecture

The operational state of an RMC is determined by evaluating its status message. The status message contains a number of status flags that may indicate a variety of faults. It is employed by the MU to ascertain the functional state of the RMC. Based on this assessment, the MU then determines the RMC's activity state. With regard to the RMC, the activity state may be either active or passive. It should be noted that a standby mode is not yet supported, as this would require redundant RMC replicas, which is not an objective for the first version of the redundancy management framework. Given that multiple RMCs can be instantiated within the system, it is necessary for the MU to be capable of processing the status information of all RMCs at any given time.
As with the RMC, the state of the CLI is determined based on its status message. In this process, the MU examines both CLI channels as well as the entire module. A CLI channel may be classified as either functional or failed. Its activity state may be determined to be active, standby, or passive. It is necessary that only a correct CLI channel be active or standby. A failed one is always considered to be passive. The CLI module is deemed functional when both CLI channels are operational, or when one channel is operational and the other is failed. The CLI module is considered to be failed when both channels are failed. The activity state of CLI modules may be either active or passive. A standby state is not yet supported, as it would necessitate the existence of a redundant CLI replica.
The MU is responsible for determining the activity state of its own lane, based on the states of the CLI and RMC. A lane can be designated as either active or passive. Currently, a standby state is not supported due to the necessity of a third lane in the channel. In

the event that both the CLI and RMCs are operational, the lane is also considered active. The potential for an alternate state of the CLI to impact this decision is negated by the fact that at least one operational and functional CLI channel remains. In the event that either the RMC or the CLI module transitions to a failed-passive state, the MU must consider the entire lane to be failed. This also leads to the conclusion that the channel is, in fact, failed and requires passivation. The transition of a channel from an active to a passive state would then initiate the fault containment procedure. This implies that a fault containment region (channel) is isolated from the rest of the avionics system to prevent the propagation of faults. The fault containment procedure entails the disabling of all external interfaces, thus preventing the transmission or reception of messages. This action effectively separates the channel from the avionics system's communication network. Provided that the channel is in an active state, the MU enables the external interfaces to facilitate the sending and receiving of messages.
In order to prevent the MU from inadvertently or prematurely classifying a module as failed, the developer has the option of configuring the threshold at which an RMC is considered faulty. The default configuration presumes that an RMC is identified as failed if it produces three erroneous messages in a row or 10 erroneous messages out of a total of 1000 messages (requirement 1-18, see section 7.2.2).
The interface of the MU to the CCI serves to receive status information from other channels within the avionics system and to share its own status information with other channels. A periodic status message is transmitted by each MU to all channels. This serves both as a heartbeat signal and as a means of notifying other channels of any required channel passivation. In the event of channel passivation, a redundant standby channel would then be required to transition its operational state to active.
The AXI interface depicted in Figure 5.5 is also employed to notify the software layer within the processing lane regarding the status of the lane and channel.

5.4.4 Cross-Channel Interface

The architectural design of the CCI is illustrated in Figure 5.6. Its internal structure is in part similar to that of the CLI, with the CCI being less complex. As the management unit is the only module in the RM system that communicates with the CCI, it follows that

no further access restrictions need to be considered. Primarily, the CCI module serves as a driver for communication between multiple channels within an avionics system as demanded by requirement 1-24 (see section 7.2.3). In this way, status information as well as the occurrence of failure and passivation events are exchanged between the channels (requirement 1-19, see section 7.2.1). Furthermore, the interface is employed to facilitate consensus on the status of the channels (requirement 1-27, see section 7.2.1).

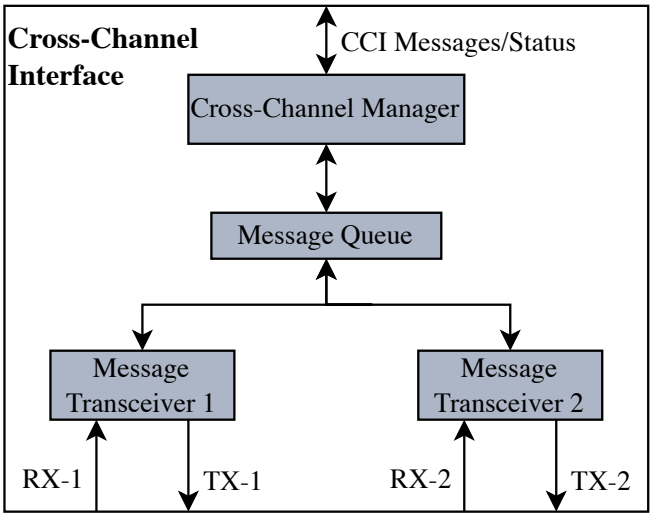


Figure 5.6: Cross-Channel Interface architecture

The CCI receives Cross-Channel (CC) messages from the management unit. The message queue is a necessary component of the system because the MU and CCI are not synchronized with each other, which could result in the loss of CC messages. The two interface transceivers in the CCI are responsible for the physical transmission of data. In a similar fashion to the CLI, the two redundant CCI channels ensure communication reliability in the event of transmission faults in one channel. Furthermore, the transmission is safeguarded by a CRC to identify any potential transmission faults. The CC Manager in the CCI is responsible for the management and processing of CC messages originating from the MU. Its functions include the coding and decoding of CRC checksums and the subsequent transmission of corresponding status updates back to the MU.

5.5 Building Block Implementation

This section focuses on the implementation of the generic FPGA modules which compose the frameworks building block portfolio. The implementation of the FPGA modules takes place in the AMD Vivado Design Suite. This enables the development and simulation of individual VHDL modules as well as the integration of the entire FPGA configuration. Vivado also comes with a synthesis toolchain that implements the design flow for generating an FPGA bitstream following the process explained in chapter 4.
The final redundancy management system is ultimately composed of several generic VHDL modules. These in turn can consist of several VHDL submodules integrated as instances into the high level RM module. The assembly of the RM system is performed according to a hierarchical structure of VHDL modules, which ultimately results in a VHDL IP block. This IP then contains the redundancy management logic as well as the required interfaces for the respective computing lane, taking into account the specific configuration of the developer/framework user. To apply detailed configuration to each VHDL module, the building blocks contain generic constants that are set by the framework according to the previously specified configuration file.
As part of this thesis, the FPGA-based redundancy management system is initially being developed for just one channel. The CCI is therefore not considered in the implementation.

5.5.1 Redundancy Management Channel

The RMC module is the direct interface between an application partition and an external interface. It is therefore responsible for processing all incoming and outgoing messages of this partition and is thus directly involved in the implementation of the configured fault tolerance mechanisms.
As already explained in the RMC design chapter, an RMC consists of an AXI interface, two separate message buffers, and the message processing unit. Each of these elements is implemented as a separate generic VHDL submodule and is instantiated into the RMC module. During instantiation, the configuration parameters and I/O ports of the submodule are then linked to the corresponding variables of the main module.
The modules within an RMC are executed simultaneously on the FPGA. Therefore, a synchronization mechanism is required when there are data dependencies between multiple

modules. In addition, all modules instantiated in the RMC are supplied by the same clock signal. This allows synchronization on the rising edge of the clock.

RMC AXI Interface

The RMC AXI interface enables bidirectional data communication between a partition in the processor and an RMC in the FPGA. This module implements the AXI communication protocol based on the ARM AMBA on-chip communication interface. The protocol defines a 3-way handshake for data transmission, which prevents the loss of data packets. The sender first transmits a send request. When the recipient acknowledges the request, the data packet is transmitted. Both the data and its destination address are exchanged according to this principle. AXI represents a multi-channel bus so that communication can take place in full-duplex operation. The exchange medium for data, addresses, control and status signals are corresponding registers in shared memory that can be accessed by the processor and the FPGA. Each AXI module is assigned to an individual base address inside the shared memory. The individual registers can then be accessed via offsets to the base address.
On the FPGA side, the operation of the interface is implemented by several VHDL processes. These processes include the exchange of addresses, data as well as request and acknowledgement signals. All processes within this module are executed simultaneously and are synchronized via the common clock source. The simultaneous design enables full-duplex operation of the interface. It should be noted that the processes for implementing the communication protocol were generated automatically by the Vivado design suite.
To connect the AXI interface to the custom user-logic, the RMC AXI module provides a register set with 10 registers of 32 bits each. These registers can be read in the FPGA to obtain data from the processor. When data is written to these registers, the data is in turn sent to the processor. The access permissions (read/write) of the individual registers are modified in the VHDL module according to the actual purpose.
The status information of an RMC (buffer capacity etc.) is transferred to the software partition in slv_reg1. This register can only be written by the FPGA. The processor has read access only. Registers 2, 3 and 4 are available to the partition to transmit a TX message. The RMC transmits the received RX messages via registers 5, 6 and 7. The

remaining registers are not currently in use and reserved for future use. Table 5.22 shows an overview of the registers of the RMC AXI module.

Register	Name	Offset Address	Description
slv_reg0	Reserved	0x00	Reserved
slv_reg1	Status	0x04	RMC status signals
slv_reg2	TX-Data-1	0x08	TX message byte 8 to 7: TX header
slv_reg3	TX-Data-2	0x0C	TX message byte 6 to 3: TX payload
slv_reg4	TX-Data-3	0x10	TX message byte 2 to 1: TX CRC
slv_reg5	RX-Data-1	0x14	RX message byte 8 to 7: RX header
slv_reg6	RX-Data-2	0x18	RX message byte 6 to 3: RX payload
slv_reg7	RX-Data-3	0x1C	RX message byte 2 to 1: RX CRC
slv_reg8	Reserved	0x20	Reserved
slv_reg9	Reserved	0x24	Reserved

Table 5.22: RMC AXI interface register overview

Additional I/O ports are defined for the connection of RMC signals to the AXI module. These implement the connection to the message buffer.

Message Buffer

In the RMC, the message buffer modules have the task of buffering incoming TX and RX messages. This is necessary because the FPGA and the processor operate asynchronously and no messages must be lost. Separate instances of the message buffer are integrated into an RMC for the TX and RX paths.
A message buffer module consists of an internal First-in-First-out (FiFo) memory that can be configured in terms of its word width and depth. The module also has separate ports for writing and reading data. In addition, the module is equipped with two status ports that indicate the used buffer capacity (full/empty). The message buffer must be accessed by setting the corresponding enable signal. In this way, the internal logic recognizes a read or write access and adjusts the read/write index accordingly. In particular, the logic enables the memory to be read and written simultaneously. This eliminates the need for special access restrictions. The logic of the buffer module is implemented in a process

that is executed on each rising clock edge. The output of data from the current memory location and the output of status information are implemented concurrently to the process and can therefore be read by external modules at any time. A control signal can be used to reset the entire module, including flushing the memory.

Message Processing Unit

The MPU is the central unit in an RMC and is responsible for the processing of incoming and outgoing messages. It initiates the data exchange with the other lane and compares the data from both lanes.
As can be seen from the block diagram in section 5.4.1, the MPU exchanges data with several modules. This is made possible in the VHDL module by appropriate I/O ports.
In order to make the processing of the various tasks as clear and efficient as possible, several parallel processes are integrated into the MPU module for different purposes. The processes are synchronized with each other by the common RMC clock signal.
The two main processes for processing TX and RX data are the MP-TX and the MP-RX processes. Both are internally equipped with a state machine that enables messages to be processed sequentially. This is required since the FPGA would usually execute VHDL statements concurrently. Initially, both processes remain in the idle state until a TX message is available in the buffer or an RX message is received from an external interface. The MP-TX process then reads the message from the buffer and prepares to send the CL request message in the next state, simultaneously to the MP-RX process. As both processes are running in parallel, it can happen that both want to access the only implemented CLI port at the same time. To prevent this, a separate lock manager process is implemented within the MPU for managing the access to the CLI. As in the CLI module, the MPU lock manager process checks whether a lock request has been set by a participant, and assigns a corresponding lock grant as soon as the shared resource is available. If both processes request a lock at exactly the same time, the MP-TX process is given priority. Once one of the processes has obtained the internal lock to access the port, it must then request the global lock from the CLI before sending its data. Only if the CLI has also granted access, the corresponding process in the MPU is allowed to transmit its CL message. From the perspective of an MPU process, a two-step authorization process

is implemented here in order to gain access to the CLI. Although this can lead to delays in MPU processes, it is necessary due to the parallel execution of asynchronous modules. Parallel to the MP-TX and MP-RX processes, another process is implemented in the MPU which is responsible for receiving CL messages. As explained in previous sections, the execution of MPUs (RMCs) and the CLI is not synchronized. The CL-RX process is used to avoid a loss of data when exchanging CL messages. This process listens on the CLI port in parallel with the other processes to detect incoming CL messages and store them internally. The process sorts the CL messages directly according to their type (TX/RX request/response) and informs the corresponding process about the available data.

As soon as the data exchange between the lanes is completed, the respective process (MP-TX/MP-RX) switches to the integrity check state where the comparison of the data is initiated. Data validation is implemented by an integrity process. It implements the logic for comparing the message header, CRC, and payload data. As already described in section 5.4.1, the CL header and the CRC of a message are compared bit-wise. To check the payload data of both messages, the integrity check process implements both exact (bit-wise) and approximate verification for various data types (signed/unsigned integer, float, signed/unsigned fixed). Depending on the RMC configuration, one of the two variants will be used for the comparison.

In the case of an exact comparison, the bit vectors of both replicas are compared bit by bit with an if-else branch. Approximate consensus comparison checks whether the absolute value of the difference between the two values is less than, equal to, or greater than a specified threshold. If the difference is less than or equal to the threshold, a consensus on the replica's data is reached. If the difference exceeds the threshold, there is no consensus on the replica values. The appropriate thresholds must be specified by the developer at design time. The VHDL file is then configured automatically by the framework.

The comparison of the data can be performed with the less than or equal to *(<=)* or greater than *(>)* VHDL operator. For this purpose, an additional IEEE open source VHDL library is integrated, which supports fixed and floating point variables as well as operators in addition to the classic VHDL bit vectors and integer variables.

Since the integrity process is implemented only once in the MPU and is used by multiple processes (MP-TX and MP-RX), it is also a shared resource. For this reason, access

to the integrity process is also managed by the MPU lock manager process. This can also cause a process to be delayed if the lock is already being held by another process. However, this delay is acceptable compared to the increased FPGA resource utilization due to instantiating a second integrity process.
Once the integrity check is complete, the MP-TX and MP-RX processes send the results (CL response) of the check to the other lane. Again, the transfer must first be granted by the internal MPU lock manager and then by the global CLI lock manager. When the verification results of both lanes have been exchanged, the MP-TX/MP-RX process checks whether both lanes have reached an identical result. A status message is compiled from the individual results and sent to the Management Unit (MU). The structure of the RMC/MPU status message is shown in Table 7.1 in the appendix. If both lanes determine that the data is correct, the TX message is forwarded to the appropriate interface drivers for transmission or the RX message is stored in the RX buffer. The MP-TX and MP-RX processes then start the execution of their state machine from the beginning, provided that further TX or RX data is available.
The MPU protects all states or logic sections with counters where a communication response is expected from a peripheral module. While a process is waiting, it increments the counter variable on every clock cycle. If the peripheral module does not responds or responds too late, the counter reaches a defined threshold and the process aborts the wait and continues processing. This is to ensure that a process does not get stuck in a deadlock. The threshold at which the wait is aborted depends on the expected transmission time (depending on data rate and data width). Furthermore, an additional buffer value is added to the threshold, which takes any waiting times and delays into account that may occur in the peripheral module. It is very difficult to specify an exact value here. The MPU is initially implemented to add a buffer of 1 % to each threshold variable. Later tests will show if this buffer is suitable or if the values need to be adjusted.

5.5.2 Cross-Lane Interface

As a communication interface between two lanes, the CLI plays a crucial role in implementing fault tolerance measures. As already explained in the design section, this module is subject to corresponding reliability and performance requirements. Unlike the RMC

module, the CLI only consists of one VHDL module. However, comparable to the MPU, it implements several simultaneous processes that fulfill different functions. The CLI also uses the simultaneous execution capabilities of the FPGA to increase the performance of the module. Some of these processes work independently or are triggered by external events. Others are managed by an internal management logic. If there are dependencies between processes (e.g. during data exchange), internal flags are used to synchronize the corresponding processes. In addition, all processes in this module are also driven by the central FPGA clock signal and therefore run synchronously with the rising edge of the clock. During the implementation, the CLI has been internally divided into three domains.

Lock Manager and Message Queue

The processes in the first domain are responsible for managing permissions on the RMCs and for caching the CL messages. A lock manager process manages the allocation of lock grants based on which RMCs have requested a lock. The decision as to which RMC gets the grant is resolved by an if-else branch with a check of the lock requests. The complexity in this logic lies in implementing which RMC gets the lock grant when multiple RMCs make a lock request in parallel. In addition, the logic must prevent one RMC from taking away the lock from another before its transaction is complete. This is avoided by including the current lock grant in the query in addition to the lock request. With 4 RMCs, this results in 44 possible combinations of lock requests. For each of these combinations, a corresponding lock grant assignment is implemented. It has also been taken into account that the same RMC is not always preferred. RMC prioritization is not yet implemented in this version of the lock manager. However, this could be integrated in an extension of the module with an additional factor in the if-else query. Once a lock is granted, an RMC sends its CL message to the CLI. The lock is granted to an RMC as long as it maintains its lock request. A deadlock by permanently blocking a lock is prevented on the RMC side. The message queuing process automatically detects a transfer and first stores the data in the CL message queue. Similar to the message buffer in an RMC, the CLI message queue also implements a sequential FiFo memory.

Message Transceiver

The processes of the second domain implement the CLI message transceiver. As already explained in the system design, two independent CLI channels are implemented in order to increase fault tolerance against transmission faults in one channel. So the transceivers are implemented by separate VHDL processes. To implement full-duplex capability, separate TX and RX processes are also implemented for each CLI channel. These implement the serialization of a CL message over the physical interface or receive incoming bits and assemble them into a message. The TX processes read the CL message to be sent directly from the message queue, calculate the CRC checksum for the message, and append it as a trailer. The CRC checksum is 16 bits long in each CLI module and is generated according to the CRC16_MODBUS version. This CRC implementation includes reflection of the inputs and outputs, the initial value of the checksum is 0xFFFF and the CRC polynomial is 0x8005. These values are implemented as local constants in the module and can therefore be configured by the framework at design time. For successful CRC generation and evaluation, however, they must be configured identically for all communication partners. The VHDL code example below shows the implementation of calculating a 16-bit CRC checksum for a CL message with a total length of 96 bits.
This involves iterating over each byte of the CL message and calculating the corresponding part for the checksum. The synthesized logic of this function can later be completely executed on the hardware in one clock cycle. The loops are unrolled during synthesis so that the data is ultimately used only once by the logic circuit.
To enable the evaluation of a CRC checksum of a received CL message, an additional evaluation process is implemented for each RX process. These functions have been separated from each other during implementation so that the RX process is ready for the next message as soon as one is received. This is to prevent an RX process from missing the start of the next CL message. During CRC check, an evaluation process calculates the checksum of the CL message header and the payload and compares it with the checksum that was attached to the message. The results of the evaluation are then forwarded to the CLI management process.

```
1  crc_polynom := x"8005"
2  crc         := x"FFFF";
3  -- iterate over the first 10 CL message bytes
4  for x in 0 to 9 loop
5      byte    := cl_message((96-1)-x*8 downto (96-8)-x*8);
6      -- reflect the input
7      for i in 15 downto 8 loop
8          buffer(i)       := byte(15-i);
9      end loop;
10     buffer(7 downto 0)  := x"00";
11     crc := (buffer xor crc);
12     for y in 0 to 7 loop
13         if (crc(15) = '1') then
14             crc   := shift_left(crc, 1) xor crc_polynom;
15         else
16             crc   := shift_left(crc, 1);
17         end if;
18     end loop;
19 end loop;
20 -- reflect the output
21 for i in 15 downto 0 loop
22     crc_result(i)  := crc(15-i);
23 end loop;
```

Listing 5.2: VHDL code for CRC16 computation

Initial timing tests of the message transceiver implementation have shown that the serial transmission of CL messages takes considerably more time than the rest of the data processing. This confirms the assumption made in the design section that message transfer represents the timing bottleneck of the entire RM system. The data rate of the CLI transfer must be high enough so that it creates not too much delay compared to the software execution in the processor, and so that the performance requirements 1-16 and 1-36 can be met. The timing of each bit during serial transmission is measured by counting clock

cycles. The frequency of the module's clock signal and the desired data rate determine how many clock cycles a bit takes. In order to achieve high data rates, the duration of a bit must be reduced and the clock frequency increased. A bit duration must be at least three clock cycles to be correctly detected by the receiver. The achievable data rate can be calculated using the following formula:

$$\text{data rate} = \frac{\text{clock frequency}}{\text{bit duration}} \tag{5.1}$$

With a clock frequency of 100 MHz and a bit duration of three clock cycles, a data rate of about 33 Mbit/s could be achieved. In requirement 1-23 (see section 7.2.4), however, a CL data rate of 200 Mbit/s is required. This data rate can be achieved with a clock frequency of 1 GHz and a bit duration of five clock cycles:

$$200\,\frac{\text{Mbit}}{\text{s}} = \frac{1000\,\text{MHz}}{5\,\frac{1}{\text{bit}}}$$

However, not all FPGA modules can be run at 1 GHz. For example, the AXI modules are limited to a maximum clock frequency of 300 MHz. One solution might be to supply individual modules with different clock frequencies. As a consequence, this can lead to asynchronous behavior, especially when exchanging data between modules, and would increase the complexity of the implementation. An alternative solution could be to supply all modules with a lower clock frequency (e.g. 100 MHz), so that the clock synchronization at module level is still guaranteed. In addition, the CLI module can be supplied with a second clock signal that provides a much higher frequency (e.g. 1 GHz). This second clock signal would then be used exclusively by the high performance processes (message transceivers). This alternative would limit the problem of asynchrony between different clock domains to a single module, while allowing faster execution of time-consuming sequential processes.
A single clock frequency of 100 MHz is used in the first version of the redundancy management system. The implementation of high-performance processes is planned for a later version of the framework.

Management Process

The management process is the third domain of the CLI module. It checks the results of the CRC evaluation of both CLI channels and initiates the forwarding of the received CL message to the corresponding RMC, provided that the CRC and thus the transmission is correct. The message from the first CLI channel is preferred. However, if the CRC check of channel 1 indicates a transmission error, the CL message of channel 2 is used provided it contains a correct message. If the CRC check identifies a fault in both channels, none of the messages will be forwarded to any RMC. To forward a CL message, the ID of the target RMC is read from the CL message header. According to the CRC check, the management process composes a status message (see Table 7.2 in the appendix section), which is sent to the RM management unit. The status message indicates whether or not a fault or a failure has occurred during transmission in one of the channels.

5.5.3 Management Unit

The Management Unit (MU) is responsible for evaluating the status messages of all RMCs and the CLI in the redundancy management system. Based on this information, the MU can detect faults in the modules and can take appropriate action to contain the faults. It does not matter if the fault event is caused by the software, external systems, or random faults. In addition, the MU is responsible for resetting all RM modules if necessary. Also, the MU distributes the current mode of the lane to all RM modules. The mode of a lane consists of an 8-bit vector and encodes both the channel activity status and the lane configuration. The channel activity status indicates whether the channel is in an active or standby state. This information is particularly important when multiple redundant channels are present in the avionics system. When a channel is in the active state, the RM modules would regularly process the TX and RX messages. In the standby state, the channel's redundancy management system would also process the messages, but would not send any TX messages. The lane configuration indicates whether the lane is in commanding or monitoring mode. This information is hard-coded into the FPGA as a global constant, accessible to every FPGA module.
A separate process is implemented in the MU to evaluate the status information for each RMC as well as for the CLI module. These processes read the corresponding status

message to check whether a fault has occurred in the module. If so, an appropriate fault counter is incremented. The RMC status messages are evaluated separately for the TX and RX processes. As long as the number of faults in the MP-TX or MP-RX processes of the RMC is below a configured threshold, the RMC is still considered to be correct. The functional state of the corresponding RMC is set to failed only if the threshold is exceeded in one of the processes. In the basic configuration, the MU interprets an RMC as failed if one of the processes (MP-TX or MP-RX) generates or receives an incorrect message three times in a row. The RMC is also considered failed if a total of 10 faulty messages occur in a total of 1000 messages (1 % error rate). These thresholds are implemented in the MU as local constants and can therefore als be adjusted by the framework at design time. Since a unique evaluation process is implemented for each RMC, only one generic process is implemented in the MU building block and packed into a *for-generate* statement. This allows the final number of evaluation processes to be implemented flexibly. The number of RMCs is transferred to the *for-generate* statement when the FPGA IP is generated by the framework. During the synthesis, the *for-generate* statement then creates as many of these evaluation processes as there are RMCs instantiated in the system.
In the CLI evaluation process, a functional state is defined for each CLI channel as well as for the entire module. When a CLI channel reports a faulty transmission, the MU sets that channel to the failed state. However, this decision can be reversed if the channel shows correct operation in the subsequent messages. Once at least one CLI channel is correct, the entire CLI module remains correct. The CLI is only be classified as failed if both channels fail. This state cannot be reversed at runtime.
An additional process monitors the functional status of all RMCs and the CLI simultaneously to determine the overall status of the lane. Once an RMC or the CLI goes into the failed state, the entire channel is classified as failed. The resulting fault containment measures include resetting all RM modules, which stops all processing. It also disables the channel's external interfaces so that it no longer has access to the avionics communication system. These actions prevent a detected fault from propagating beyond the FCR boundaries to other channels or peripheral systems. In this version of the redundancy management system, a channel passivation can no longer be reversed at program runtime. Only a power cycle would allow a channel to return to an active state again.

In parallel to the evaluation and fault containment logic, an AXI interface is also implemented in the MU, which allows the exchange of status information from the RM system to the processor. The 32 bit status message contains the functional and activity states of all RMCs as well as those of the CLI, the lane, and the channel. It also indicates whether the external interfaces are disabled or enabled by the lane. The structure of the MU status message is shown in the appendix section (see Table 7.3). This status information should enable the processor to track the safety status of the lane and the channel.

5.5.4 Resource Utilization of FPGA Implementation

As you can see from the previous sections, most of the modules in the redundancy management system consist of processes that run simultaneously. Each of these processes therefore requires separate logic components in the FPGA, which increases the utilization of FPGA resources. In addition, there are a number of parallel buffer memories that require additional resources. However, the number of CLBs is constant for each FPGA and therefore represents the limit of how much logic can be placed on the chip. Ultimately, a compromise may have to be found between processing performance and resource utilization.
Table 5.23 shows the resource utilization of a redundancy management system with 4 RMCs supporting four partitions, which is implemented and synthesized for a Zynq Ultrascale+ XCZU3EG SoC.

Resource Type	Used	Available	Utilization
CLB LUTs	14199	70560	20.12 %
CLB Flip Flop Registers	41350	141120	29.30 %
F7 Multiplexer	4351	35280	12.33 %
F8 Multiplexer	2057	17640	11.66 %

Table 5.23: Redundancy Management IP (4 partitions) resource utilization on a Zynq Ultrascale+ ZU3EG SoC

The results show that the RM system requires around 20 % of the Lookup Tables (LUTs), 29 % of the flip-flop registers and around 12 % of the F7 and F8 multiplexers. However, the RMCs consume the most resources with 17,20 % of the LUTs, 27,28 % of the CLB

registers, and 23,60 % percent of the F7 and F8 multiplexers. This is acceptable at first, but should be taken into account if further RMCs or even additional logic (independent to the RM system) is to be implemented in the FPGA.

Module	CLB LUTs	CLB Registers	F7 Multiplexer	F8 Multiplexer
MPU	0.78 %	0.79 %	0.01 %	0.00 %
AXI	0.14 %	0.17 %	0.09 %	0.00 %
TX Buffer	1.68 %	2.93 %	1.45 %	1.45 %
RX Buffer	1.70 %	2.93 %	1.45 %	1.45 %

Table 5.24: RMC module resource utilization on a Zynq Ultrascale+ ZU3EG SoC

Tests with various configurations have shown that the buffer memory accounts for the majority of resource usage (see Table 5.24). In this regard, it would be feasible to reduce the number of memory locations in each memory in order to reduce resource utilization. However, the performance of the partitions and the peripherals must be taken into account. The amount of memory must not be so small that data is in danger of being lost if the partition produces data faster than the peripheral system can process and respond to it.

5.6 Building Block Verification

The verification of the redundancy management building blocks checks whether the system is functionally correct and fulfills the requirements. Verification is performed using design reviews, simulations, and hardware tests. The verification considers only the requirements for the FPGA-based redundancy management system.

5.6.1 Design Review

The design review should consider requirements that cannot be verified by laboratory testing. The following table lists all the requirements for the RM system that are verified by a system design review.

ID	Name
1-1	Generic design
1-2	Hardware Description Language
1-3	RM modes
1-4	RM configurability
1-5	PS-PL interfaces
1-6	PS-PL interface configuration
1-7	Cross-Lane interface
1-8	Peripheral interface ports
1-9	Simultaneous processing
1-10	Peripheral interface configuration
1-11	Peripheral interface enable
1-13	Provide channel status
1-19	Provide passivation status
1-20	Synchronize lanes
1-33	Exact consensus
1-34	Approximate consensus
1-37	Target hardware

Table 5.25: RM requirements to be verified by analysis

As explained in the design and implementation section, the redundancy management system IP is developed for integration into the FPGA of an AMD Zynq Ultrascle+ SoC in the hardware description language VHDL. In addition, the system has a modular structure and is composed of generic modules. The specification of each module is implemented using either global or local static variables. This allow the individual customization of all RM IP modules according to the avionics system requirements. The IP also implements an internal mode that can put the system into a commanding-monitoring and active-standby configuration at the same time. For communication, the RM system has two customizable processor FPGA interfaces and configurable ports for peripheral interfaces. The number of peripheral interface ports is also configurable. However, at least 15 interfaces are available. In addition, the RM IP also monitors the health status of the lane and enables/disables the peripheral interfaces as needed. The IP uses this information to build

a status message that is sent to the processor. This status message includes information about the operating status (active or passive) of the lane and channel. Furthermore, the individual modules consist of parallel processes to increase the performance of the system and reduce the required execution time. To ensure data integrity, the RM IP checks each message generated by the processor or received from a sensor and compares it to the other lane. The goal of the comparison is to reach a consensus on the data. The integrity logic of the RM system supports both exact and approximate data comparison. Data is exchanged between the lanes via a redundant CLI, which also serves to synchronize the two lanes by communicating with each other.

Result

The design review has shown that the majority of the requirements listed in Table 5.25 are met by the developed system. Only requirements 1-13 and 1-19 (see section 7.2.1) are only partially fulfilled, as the current implementation does not yet support data communication with other channels.

5.6.2 Simulation Tests

Simulating VHDL modules is a very convenient way to test implemented logic early on in the development process. Simulation can be used to test individual functions as well as entire modules or even multi-module systems. For this purpose, the Device Under Test (DUT) is instantiated in a separate VHDL testbench file. Finally, variables are created in the testbench to feed the input variables of the DUT during the simulation. A simulation of a VHDL module is therefore a Software-in-the-Loop (SiL) test. In a simulation process, values are assigned to the simulation variables in order to simulate the behavior of a system adjacent to the DUT. The desired temporal behavior can be configured in the simulation by integrating *wait for* and *wait until* methods. Simulation tests are performed in the Vivado design suite. This makes it possible to set brake points in the DUT during the simulation and to execute the code step by step, which is especially useful for debugging. During the verification of the redundancy management system, simulation tests were performed on the RMC message buffer, the MPU, the CLI, and the entire redundancy management IP.

Message Buffer Simulation

The message buffer simulation should verify that the module stores data correctly and outputs it in the correct order. The correct behavior of the internal counter and index variables is also verified. During the test, data is written to and read from the buffer several times in succession.
The results of the test can be viewed in the Vivado waveform viewer. This has shown that the buffer module stores the transferred data correctly and also outputs it in the correct order. Internal flags, message counters, and status signals are also set correctly. The test has been successfully completed.

MPU Simulation

A total of two simulation tests are performed to verify the Message Processing Unit.
The first test checks the processing of several consecutive TX messages. To do this, the testbench generates appropriate TX messages and simulates the signals of a CLI. This test pays particular attention to proper lock handling for access to the CLI and the integrity process. The simulation of this test has shown that the MPU processes the TX message correctly. Only during the output of the CL message a minor error occurred which caused the CL message output port to be driven by several processes. After this error was corrected by an internal circuit, the test was successfully completed. The lock handling was also performed as intended.
In the second test, a parallel incoming RX message was simulated in addition to the TX message. Once again, the CLI is realized using simulated signals. The purpose of this test is to verify that the processes within the MPU are running correctly in parallel, and that lock handling is correct for concurrent requests. The test is also used to verify the integrity check logic. The goal is to verify that the data can be correctly compared to ensure input and output consensus. The verified requirements are 1-21, 1-22, 1-30, and 1-31 (see section 7.2.1). This test was also successfully completed. The processes of the TX and RX paths were executed correctly without influencing each other. The lock handling also worked properly. No violation of access rights to shared resources was detected.

CLI Simulation

To verify the CLI, appropriate simulation signals were created in the testbench to simulate an RMC using the CLI module. In addition, the serial ports of the CLI interface transceivers were connected (TX-1 to RX-1 and TX-2 to RX-2), creating a communication loop.
In the first test, a lock was requested from the simulated RMC and a CL message was provided. The purpose of this test is to verify that the CLI correctly assigns the lock grant, processes the CL message as expected, sends it, receives it, and forwards it back to the RMC. The test showed that the CLI issued a lock grant to the correct RMC. The data was then correctly stored and sent through the interface. The communication loop has ensured that the CLI received its data back directly. The previously generated CRC was successfully evaluated and the CL message was returned to the original RMC. The status message issued by the CLI reflects error-free execution.
The second test of the CLI simulates multiple RMCs sending multiple CL messages. Various CL message types (request and response) are also simulated. The purpose of this test is to demonstrate that the CLI will assign the lock grant to only one RMC, even in the case of simultaneous lock requests from multiple RMCs. This test was also successful. The CLI only ever granted the lock to one RMC at a time, and processed its CL messages correctly. The CL messages received were also routed to the correct RMC.

MU Simulation

The Management Unit (MU) tests simulates the status messages of the four RMCs and the CLI. The purpose of this tests is to verify that the MU uses the status messages to correctly determine the status of the partitions, the lane, and the channel, and to perform the appropriate fault containment actions. These tests are largely responsible for verifying compliance with safety and fault tolerance requirements. The requirements verified by these tests are: 1-12, 1-14, 1-15, 1-16, 1-17, 1-18, and 1-19 (see section 7.2).
In the first MU test, status messages from all RMCs and the CLI were simulated, showing the correct behavior of the modules. Status messages were alternately sent sequentially or simultaneously to the MU. The result of the test shows that the MU interprets the status messages correctly and does not detect any misbehavior in any of the modules. The MU

has sent this information to the processor as a status message and enabled the external interfaces.
The second test simulates some status messages that indicate an error in an RMC. However, no more than two faulty status messages were simulated in a row per RMC and fewer than 10 in total. The execution of the test shows that the MU detects and tracks the fault flags in the status messages. However, the MU does not classify any of the RMCs as failed because the failure thresholds are not met. The test passes.
The third test simulated three erroneous status messages in sequence, followed by 15 messages, of which 11 indicated a fault. This test triggers the configured fault indication thresholds. The MU also operates correctly in this test. It detects the three consecutive status messages with errors and then classifies the corresponding RMC as failed. As a further consequence, the MU passivates the lane and the entire channel and disables the external interfaces. The MU shows the same behavior with a total of 11 erroneous messages. This proved that the logic of the MU correctly evaluates the status messages from the RMCs and correctly passivates the lane and the channel in case of a detected fault. A logic analyzer measurement has shown that the fault containment process requires a total of 4 clock cycles from fault detection to reset all RM modules, passivate the lane and the channel, and disable the external interfaces. At a clock frequency of 100 MHz, this corresponds to a duration of 40 ns. This also satisfies requirement 1-15 regarding the required fault handling performance.
The fourth test checks the behavior of the MU in case of faulty CLI channels. First, a CLI status message is simulated indicating that both channels are operating correctly. The status message is then changed to simulate an error in one channel. The faulty CLI channel is swapped multiple times. Finally, the case is simulated where both CLI channels are faulty. The test showed that if the CLI status message is correct, the MU takes no further action. As soon as one of the two CLI channels has an error, the MU detects this, but leaves the CLI in the active state. Only if both CLI channels are simulated as failed, the MU will classify the entire CLI as failed and then passivate the lane and the channel. The external interfaces are also disabled.
The behavior of the MU observed in all four simulation tests is consistent with the requirements of the RM system. Therefore, the tests have been passed and the above requirements are satisfied.

Redundancy Management System Simulation

As a final SiL test, the entire redundancy management system is assembled from all modules into an IP block and simulated. Only the AXI interfaces must be excluded from this test. These are verified during hardware testing. Previously, individual modules were always tested with simulated adjacent modules. Now that the functionality of the individual modules has been verified, the next step is to test their operation together. The test focuses on the correct interaction between the modules.
This test simulates a TX message written to the TX buffer by the testbench. Once the message is in the buffer, the RM system takes over processing. After the TX message is processed and sent, an RX message is simulated. The test showed that the modules of the RM system work together without indicating errors. The simulated TX message was processed correctly and finally sent. The same applies to the processing of the RX message.

Simulation Test Results

Overall, it can be said that the simulation tests were successfully completed and most of the RM requirements were satisfied. A few minor implementation errors were discovered during debugging. These have been fixed.

5.6.3 Hardware Test

Finally, the hardware tests are designed to prove that the RM system also works on the SoC hardware in conjunction with a test software on the processor. As part of the hardware tests, both the AXI interfaces and the entire RM system are tested on a development board. Ultimately, tests with two development boards should realize dual-lane operation in a commanding-monitoring channel.

AXI Interface Test

Since the AXI interface implements the hardware communication between the processing system and the FPGA in a SoC, simulation testing is hardly possible. For this reason, the AXI module is first tested separately on the development hardware which is why

only the AXI module is instantiated in the FPGA configuration. For test purposes, the registers of the TX and RX messages are also connected together on the FPGA side to create a communication loop. Sending and writing of messages to the TX registers and reading of the RX registers is done by test software on the processor. This is a bare metal program written in C that initializes the AXI module on the processor side and finally writes TX data to the corresponding registers slv_reg2, slv_reg3, and slv_reg4. The software then reads the RX registers slv_reg5, slv_reg6, and slv_reg7. The readout data is sent through the debug interface and displayed on the console of a lab PC. This test is repeated with 5 different TX messages. The test showed that any TX message sent via the AXI interface could be read out correctly via the RX register. So this test proved that the AXI interface is working correctly.

RM Single-Lane Test

The Redundancy Management system single-lane test instantiates the entire RM system in an FPGA configuration and deploys it on the development hardware. However, this test uses only one software partition and therefore only one RMC. The test software written for the previous test can be reused here with minor modifications. In this test, the external communication ports of a lane (TX and RX messages, cross-lane TX and RX) are interconnected by hardware pins on the development board. This creates a communication loop that virtually simulates the interaction of two identical lanes.
In the first test case, the test software writes four TX messages to the TX message register of the RMC AXI interface. The console output on the host PC shows that the RM system has correctly processed the messages in the TX and RX paths. Because of the communication loop, TX and RX messages are identical. You can see from the RMC and the Management Unit (MU) status messages that all status flags are set to '1'. This means that no errors have been detected and all modules are correct and operational.
The external interfaces are enabled when the program starts and remain enabled for the duration of the test
The second Redundancy Management (RM) hardware test simulates a partial failure of the software partition. The test software sends the first two TX messages with an incorrect CRC. This monitors whether the RM system identifies the errors and sets the status flags accordingly. The console output of the software is shown in Figure 7.2 in the appendix

section. It is shown that the RMC correctly detects the faulty CRC checksum. It sets bits 10, 9, and 8 to '0' in its status message and indicates an invalid CRC in the TX message. When the third and fourth TX messages are processed (correct CRC), the flags are set back to '1'. When reading the RX register, it is noticeable that only the third and fourth messages can be read. This means that the RMC did not sent the first two erroneous messages. The fault containment process was therefore successful at RMC level. Since the fault detection threshold was not reached with only two faulty messages, the status of all RM modules remains active-correct (see MU status message in the console output). The external interfaces were also enabled throughout the test. The results show that this test is passed.
The third and fourth tests simulate reaching the fault detection threshold. First, three erroneous messages are simulated in sequence. 1000 messages are then sent by the test software, 10 of which have an incorrect CRC. The console output (see Figure 7.3 in the appendix section 7.7) shows that the RM identifies the three erroneous TX messages. As we have already seen in the second test, the status flags will be set in accordance with a faulty CRC. You can also see that after the third erroneous message, the RMC resets itself to the idle state (all status flags set to '0'). In addition, the test shows that the MU performs appropriate fault containment procedures immediately after detecting the third erroneous TX message. This resets the RMC, disables the external interfaces (see Figure 5.7), and sets the MU status flags.

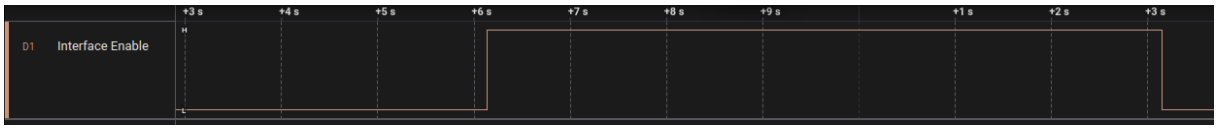


Figure 5.7: RM single-lane hardware tests 3 and 4 - measurement of interface enable pin

The MU status message indicates that the TX path of the RMC-1 has failed (bit 0 is set to '0'). Accordingly, the RMC-1 is classified as passive (bit 2 is set to '0'). In addition, bits 15, 17, and 18 indicate the passivation of the lane and the channel and the deactivation of the external interfaces. In the fourth test, the RM system shows the same behavior once the tenth erroneous message is transmitted.
These tests show that the RM system functions correctly in single-lane operation on real hardware. Error-free TX messages are processed and transmitted correctly, and their

responses are received and evaluated correctly. When a fault is detected in an RMC and the fault threshold is reached, the MU initiates the fault-containment measure and passivates the computer to prevent the fault from propagating outside the FCR.

RM Dual-Lane Test

Previously, all hardware tests were performed on a single development board, and the interfaces to a second lane were bridged by onboard circuitry. This procedure is sufficient for the functional verification of the Redundancy Management (RM) system of one lane. However, tests with two independent hardware lanes (development boards) are also required, as it would be the case in the DLR FCC. Due to the asynchronous execution on both lanes, these tests check the robustness of the RMC modules and CLI interfaces with respect to the timing of data exchange between the two lanes. In addition, all four RMCs are now instantiated in each lane's RM system to support four software partitions for this test. The test software is adapted to operate all four RMC AXI interfaces from one partition. No multicore software with multiple partitions will be developed as part of this work.
As in the previous tests, correct TX messages are first sent to the RMCs to test the normal operation of the system. Then, erroneous messages are simulated, which should eventually trigger the passivation of both lanes.
The test setup for the RM dual-lane hardware test is shown in Figure 5.8. To allow debugging with external pins, two development boards are used instead of one FCC. These boards are stacked and act as two processing lanes. Both are supplied by an individual power source and have separate debug USB connections. The CLI transceiver pins are connected between the boards to allow physical CL communication. In addition, a Saleae logic analyzer is used to capture the physical CLI communication (see Figure 5.9).

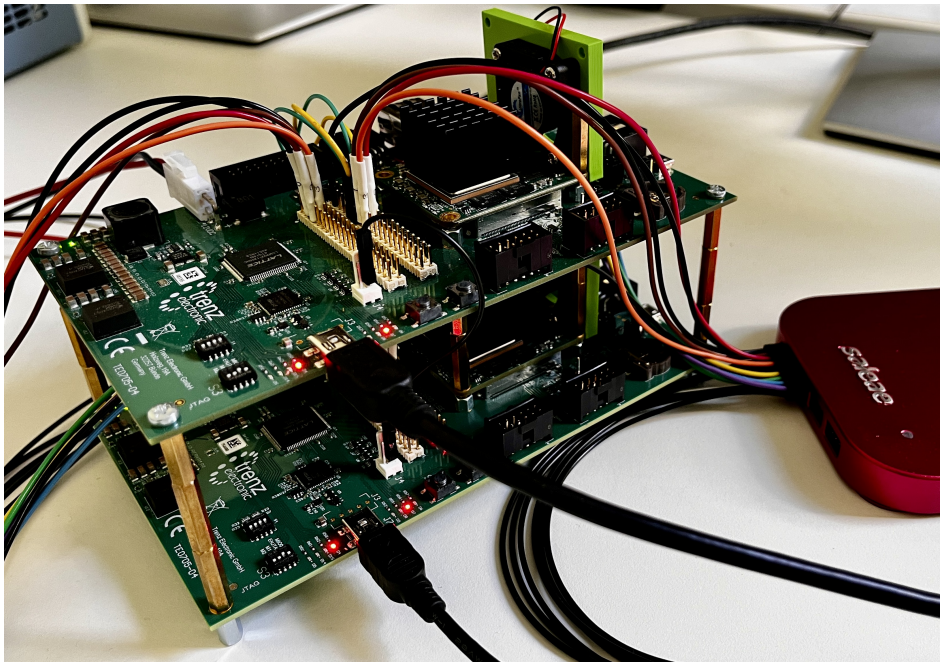


Figure 5.8: RM dual-lane hardware test setup

During the initial tests, problems with the asynchrony of the two lanes became immediately apparent. One of the two lanes always passivated shortly after the program started. It is suspected that there is a timing problem with the CLI interface data exchange. As a result, the CLI counter in the RMCs of both lanes was gradually increased. This counter indicates how long an RMC waits for a response from its counter part on the other lane. The first successful tests were performed after a waiting time of just over 5 ms.

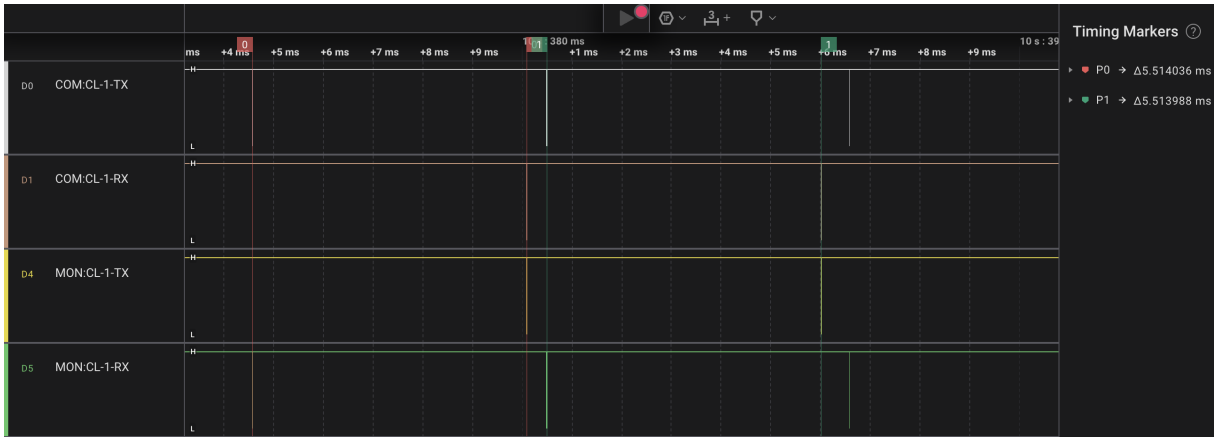


Figure 5.9: RM dual-lane hardware test - measurement of CLI transceiver pins

As can be seen in Figure 5.9, the time difference between the first transmitted CL message between the commanding and monitoring lane is about 5,5 ms. This difference is due to an asynchronous power cycle and boot process. However, it has been measured that at runtime this difference is always the same and therefore both lanes have a constant execution time. The console output (see Appendix section 7.8) also shows that the messages from all RMCs are processed correctly and no errors are identified by the RM system.
Finally, a test is performed in which all TX messages from the RMC-3 show an error (incorrect CRC). The console output (see the figures in the appendix section 7.8) also shows that the RM detects the errors in RMC-3 and sets the appropriate status flags. After the third erroneous message in RMC-3, the MU initiates the expected fault containment actions and passivates the lanes and thus the channel. This includes deactivation of the external interfaces as well (see Figure 5.10).

Figure 5.10: RM dual-lane hardware test - measurement of interface enable pins

Hardware Test Results

Ultimately, these hardware tests prove that the functional requirements of the RM system are met in single and dual lane operation in a commanding-monitoring configuration. Due to the long latency for communication between lanes, the current version of the system is not very performing and therefore does not meet all performance requirements. To solve this problem, an initialization routine could be implemented in RM that first synchronizes with the RM system on the other lane before starting to process messages. In such a routine, the two lanes would communicate their availability and then begin executing the actual RM logic together. This eliminates initial asynchrony caused by a delay in powering up or booting, and improves system performance.

6 Conclusion and Outlook

At the beginning of this work, the main faults to be considered in safety-critical systems and possible measures for the development of fault-tolerant avionics systems were identified. It turns out that avionics systems must be designed to survive a certain number of failures before they can fail. Byzantine Faults and Common-Mode Failures pose the greatest challenge to fault prevention and tolerance.
Based on this knowledge, a framework was designed to support the development of a redundancy management system for modular avionics systems. Requirements for the framework and the generated redundancy management system were also derived.
The developed framework provides a configuration environment to map the avionics architecture and define the required fault tolerance measures. To facilitate this process, the framework also provides generic patterns for various fault tolerance mechanisms. The main part of the framework consists of an FPGA building block portfolio. These are generic VHDL modules, each of which implements different redundancy management tasks. Depending on the configuration of the avionics system, the framework generates an appropriate redundancy management system from these building blocks and configures it for the particular use case. The result is a fully configured FPGA module optimized for use in hybrid SoCs in modular avionics computers.
The implementation and especially the tests of the redundancy management system have demonstrated that embedding it in the FPGA part of a SoC has significant advantages. This means that the system can be developed and executed separately from the actual avionics software application. The framework uses the configuration file to define a clear interface between the application and the redundancy management that makes this possible. On the other hand, the FPGA enables simultaneous execution of multiple logic modules, which significantly reduces the processing overhead of redundancy management mechanisms compared to conventional processors. The implementation of this first release supports dual-lane avionics channels with multiple parallel application partitions on each lane. Fault detection is implemented using a commanding-monitoring configuration. For this purpose, the framework provides FPGA IP for a full-duplex CL interface. The com-

parison of the replica data can be performed in the respective lanes according to the exact or approximate consensus principle. Each redundancy management system generated by the framework also includes a central management module that monitors the status of all partitions in the lanes and activates fault-containment measures if a fault is detected.
The hardware tests shows that the asynchronous execution of both lanes can be a problem for CL communication. In particular, the delayed boot process proved to be a significant source of error. An upcoming enhancement to the redundancy framework will include an initialization routine that synchronizes both lanes in a channel to a common program start. This should allow the CL data rate to be increased to satisfy the performance requirements.
There are also plans to expand the framework to include the already designed CCI. This is intended to implement coordination between multiple channels in an avionics system so that multi-channel avionics systems can be supported by the framework. Redundancy management will initially be customized for active-standby configurations. This capability allows a standby channel to take over the role of a failed channel. In this way, fail-operational behavior can ultimately be achieved.
Another planned enhancement is the support of redundant replicas at the software level to better utilize hybrid multicore SoCs. The idea here is that a commanding-monitoring pair is implemented inside an SoC within a physical lane. In a dual-lane channel, a dual-duplex system could be configured with fail-operational capabilities.
Finally, dynamic reconfiguration of partitions and related redundancy management elements will be considered for future releases of the framework. This can be achieved through runtime reconfiguration of partial FPGA bitstreams. The redundancy framework would thus support more and more approaches of the IMA concept.

7 Appendix

7.1 Redundancy Framework Requirements

This section includes all redundancy framework related requirements.

7.1.1 Functional Requirements

0-1 Generate Redundancy Management System IP

The Framework must be able to generate an avionics redundancy management system implemented in an FPGA IP.
Rationale:
The generation of the upfront configured redundancy management system is the key feature of this framework. The framework is intended to support SoC based avionics computers, hence the resulting redundancy management system is implemented as FPGA IP to benefit from the configurability and simultaneous computing capabilities of such chips.
Means of Compliance: Analysis/Test
Traces to upper level requirements: N/A

0-2 Avionics Architectures Support

The framework must be designed generic to support several avionics architectures.
Rationale:
Such framework is only useful for avionics developers when supporting several different avionics architectures. Providing a generic approach increases the number of applicable use-cases.
Means of Compliance: Analysis
Traces to upper level requirements: N/A

0-3 Configurability

The framework must evolve configurability to adapt the redundancy management according to safety requirements.
Rationale:
This is a key feature to support modular avionics concepts like IMA.
Means of Compliance: Analysis
Traces to upper level requirements: N/A

0-4 Fault Tolerance Pattern

The framework shall provide pattern implementing several fault tolerance mechanisms.
Rationale:
These pattern allow a flexible configuration of the final redundancy management system design according to the actual needs.
Means of Compliance: Analysis
Traces to upper level requirements: N/A

0-5 Configuration Process

The framework must allow redundancy management configuration using a configuration file.
Rationale:
Applying the configuration with a designated configuration file and defining the corresponding semantic has proven to be an applicable method of interfacing the framework with the developer.
Means of Compliance: Analysis/Test
Traces to upper level requirements: N/A

0-6 Partition Reconfiguration

The framework shall support reconfiguration of application partitions.
Rationale:
The IMA concept introduces the reconfiguration of application partitions between the processing modules within an avionics system. In order to support this functionality, the

framework shall consider this.
Means of Compliance: Analysis/Test
Traces to upper level requirements: N/A

0-8 Configurable Communication Protocols

The framework must support several low- and high-level data communication protocols. The framework must be able to configure the interfaces accordingly.
Rationale:
Depending on the use-case and application, several different low- and high-level communication protocols may be used in an avionics computer. The framework therefore must be able to provide a generic and configurable template to support them.
Means of Compliance: Analysis/Test
Traces to upper level requirements: N/A

0-10 Commanding-Monitoring Configuration

The framework must provide support for a commanding-monitoring channel configuration.
Rationale:
This is a common fault-tolerance mechanism on channel level to support fault identification.
Means of Compliance: Analysis
Traces to upper level requirements: N/A

0-11 Fail-Operational Behavior

The framework must provide support to achieve fail-operational behavior.
Rationale:
Safety-critical avionics avionics computers must not fail due to a single fault [4].
Means of Compliance: Analysis
Traces to upper level requirements: N/A

0-12 Redundant Data Interfaces

The framework must provide support for simplex and duplex data interfaces.
Rationale:
To support fault-tolerant data communication.
Means of Compliance: Analysis
Traces to upper level requirements: N/A

0-13 Intra-Channel Interface

The framework must provide support for a redundant intra-channel communication interface.
Rationale:
Required by the commanding-monitoring configuration to ensure channel integrity.
Means of Compliance: Analysis
Traces to upper level requirements: N/A

0-14 Inter-Channel Interface

The framework must provide support for a redundant inter-channel communication interface.
Rationale:
Required to achieve fail-operational behavior. Channels need fault tolerant communication interface to obtain consensus about their status.
Means of Compliance: Analysis
Traces to upper level requirements: N/A

0-16 Communication Protocol Checksum

The framework must provide support to use CRC checksums to assure data communication.
Rationale:
Required to check integrity of computed and received data.
Means of Compliance: Analysis
Traces to upper level requirements: N/A

0-17 Input Consensus

The framework must provide support to achieve input consensus between replicated input values.

Rationale:

Required to achieve output consensus.

Means of Compliance: Analysis

Traces to upper level requirements: N/A

0-18 Output Consensus

The framework must provide support to achieve output consensus between replicated output values.

Rationale:

Output consensus and agreement on the computed values is a key requirement to provide correct data and ensure fault containment.

Means of Compliance: Analysis

Traces to upper level requirements: N/A

0-19 Exact Consensus

The framework must provide support to achieve exact (bit-wise) consensus between replicated data values.

Rationale:

Some values/results require exact consensus.

Means of Compliance: Analysis

Traces to upper level requirements: N/A

0-20 Approximate Consensus

The framework shall provide support to achieve approximate consensus between replicated data values.

Rationale:

Some values/results require approximate consensus. E.g. sensor values displayed in physical quantities cannot be compared bit-wise.

Means of Compliance: Analysis
Traces to upper level requirements: N/A

0-21 Approximate Consensus Threshold Configuration

The framework shall enable the configuration of the approximate consensus threshold by the designer.
Rationale:
Approximate comparison of values/results requires a predefined threshold which the deviation of the compared values is allowed to be max to be considered as consent.
Means of Compliance: Analysis
Traces to upper level requirements: N/A

0-22 Redundancy Management System Dissimilarity

The framework shall provide support to generate/implement dissimilar replicas of the configured redundancy management system.
Rationale:
Dissimilarity may be required in avionics systems to avoid common-mode-failures due to an identical replication of logic/code.
Means of Compliance: Analysis
Traces to upper level requirements: N/A

0-23 Redundant Channels

The framework must provide support for redundant channels within an avionics system.
Rationale:
This is a key requirement to achieve fail-operational behavior.
Means of Compliance: Analysis
Traces to upper level requirements: N/A

0-24 Redundant Lanes

The framework must provide support for redundant lanes within a channel.
Rationale:

This is a key requirement for a command-monitoring architecture.
Means of Compliance: Analysis
Traces to upper level requirements: N/A

0-25 Fault Containment

The framework must provide support to achieve fault containment within the avionics systems channels.
Rationale:
Key requirement to achieve fault tolerance. A fault must not propagate and affect/corrupt other modules in an avionics system.
Means of Compliance: Analysis
Traces to upper level requirements: N/A

0-26 Channel Status Consensus

The framework must provide support to achieve consensus between the avionics systems channels about their actual status (active, standby, passive).
Rationale:
If several redundant channels are integrated in an avionics system, they must keep track on their own and each others operational status. Consensus ensures that all channels agree on their status and on the channel which is currently the active one.
Means of Compliance: Analysis
Traces to upper level requirements: N/A

0-27 Channel Status Determination

The framework must provide support for each channel to determine its status of functionality (operational, faulty, failed, passive).
Rationale:
Required to achieve status consensus between all redundant channels.
Means of Compliance: Analysis
Traces to upper level requirements: N/A

0-28 Input Integrity

The framework must provide support to achieve input integrity between replicated input values.
Rationale:
Ensuring integrity of input data is a key requirement to ensure fault identification.
Means of Compliance: Analysis
Traces to upper level requirements: N/A

0-29 Output Integrity

The framework must provide support to achieve output integrity between replicated output values.
Rationale:
Ensuring integrity of output data is a key requirement to ensure fault identification.
Means of Compliance: Analysis
Traces to upper level requirements: N/A

7.1.2 Non-Functional Requirements

0-7 HDL Support

The framework shall support the IP generation in VHDL language.
Rationale:
VHDL is the current language used by the developer team.
Means of Compliance: Analysis
Traces to upper level requirements: N/A

0-9 Target Hardware

The framework must support heterogeneous System-on-Chip hardware platforms including a microprocessor and a programmable FPGA part.
Rationale:
Heterogeneous platforms enable the physical separation from application execution and

the redundancy management system. Providing the redundancy management own computing resources, reduces/eliminates the computational overhead on the microprocessor site and therefore the potential affection of applications.
Means of Compliance: Analysis
Traces to upper level requirements: N/A

7.2 Redundancy Management IP Requirements

This section includes all Redundancy Management IP related requirements.

7.2.1 Functional Requirements

1-6 PS-PL Interface Configuration

The RM IP must be able to configure the amount of AXI interfaces and corresponding registers.
Rationale:
Further interfaces may be instantiated for further application partitions.
Means of Compliance: Analysis
Traces to upper level requirements: 0-3; 0-9

1-10 Peripheral Interface Configuration

The RM IP must be able to configure the amount of peripheral bidirectional interfaces.
Rationale:
The amount depends on the number of partitions and used peripheral interfaces. The amount of instantiated ports/drivers must therefore be configurable by the developer according to the actual use-case.
Means of Compliance: Analysis
Traces to upper level requirements: 0-2; 0-3

1-11 Peripheral Interface Enable

The RM IP must implement dis- and enable functionality for each peripheral interface.
Rationale:
Some hardware interface drivers require an enable signal to transmit data. Further, a disable by the RM IP is required to avoid transmission of faulty data over the interface. This is part of the fault containment strategy.
Means of Compliance: Analysis/Test
Traces to upper level requirements: 0-25

1-12 Status Determination

The RM IP must be able to determine the status (operational, faulty, failed, passive) of a RM instance.
Rationale:
The status is required to determine the overall channel status and represents important information for other channels within the avionics system.
Means of Compliance: Test
Traces to upper level requirements: 0-27

1-13 Provide Channel Status

The RM must be able to provide its internal status to other RM instances on other lanes or even other channels. It must also be able to obtain the status from other RM instances.
Rationale:
Required to achieve avionics system status consensus among redundant channels.
Means of Compliance: Analysis
Traces to upper level requirements: 0-11; 0-26

1-14 Fault Handling

The RM must perform channel internal fault identification and treatment.
Rationale:
These are key functionalities for fault-tolerant systems. In order to ensure and maintain fault tolerance, faults must be detected and treated.

Means of Compliance: Test
Traces to upper level requirements: 0-11

1-16 Fault Containment

The RM must prevent fault propagation through the channel boundaries.
Rationale:
A fault in a channel must not affect other channels in the system.
Means of Compliance: Test
Traces to upper level requirements: 0-11; 0-25

1-17 Channel Passivation

The RM must passivate a channel if a failure is detected.
Rationale:
The passivation is part of the fault/failure treatment strategy. It is assumed that a channel showing a failure event would not remain correctly operational.
Means of Compliance: Test
Traces to upper level requirements: 0-25

1-19 Provide Passivation Status

The RM must provide the passivation action to the software layer of each processing lane and other channels within the avionics system.
Rationale:
The software may report this event to a higher level management instance. The other channels need to be informed that a standby channel can take over the operational role (if available).
Means of Compliance: Analysis/Test
Traces to upper level requirements: 0-11; 0-26

1-20 Synchronize Lanes

The RM shall synchronize the processing lanes of a channel.
Rationale:

Synchronization is required to compare the correct replica values. These values must be from the same age, they must result from a corresponding computing cycle.
Means of Compliance: Analysis/Test
Traces to upper level requirements: 0-10; 0-24

1-21 Input Consensus

The RM must achieve consensus between replicated processing lane inputs.
Rationale:
To ensure that both lanes use the same values for the computation. This is required to achieve output consensus.
Means of Compliance: Test
Traces to upper level requirements: 0-17

1-22 Output Consensus

The RM must achieve consensus between replicated processing lane outputs.
Rationale:
To ensure that both lanes have computed the same values from the given input values. The lanes must agree on these values.
Means of Compliance: Test
Traces to upper level requirements: 0-18

1-26 Channel Activation

The RM must switch the channel mode from standby to active if required.
Rationale:
To take over the operational role in the system in order to achieve fail-operational behavior.
Means of Compliance: N/A
Traces to upper level requirements: 0-11

1-27 Channel Activation Consensus

The RM must achieve consensus between lanes on a channel activation switching event.
Rationale:
Both lanes must agree on the decision to switch from standby to active mode. Otherwise this decision could result from a fault in one lane (e.g. misinterpretion of a message).
Means of Compliance: N/A
Traces to upper level requirements: N/A

1-29 Internal Status Agreement Not applicable anymore

The RM instances in the processing lanes of a channel must agree on each others internal channel status.
Rationale:
In order to determine that the lanes in a channel operate correctly or not.
Means of Compliance: N/A
Traces to upper level requirements: N/A

1-30 Input Integrity

The RM must check the integrity of received input values.
Rationale:
Ensuring integrity of input data is a key requirement to ensure fault identification.
Means of Compliance: Test
Traces to upper level requirements: 0-28

1-31 Output Integrity

The RM must check the integrity of the output values to be transmitted.
Rationale:
Ensuring integrity of output data is a key requirement to ensure fault identification.
Means of Compliance: Test
Traces to upper level requirements: 0-29

1-33 Exact Consensus

The RM must provide an exact (bit-wise) consensus algorithm to cross-check the processing lane data interfaces.
Rationale:
Some values/results require exact consensus.
Means of Compliance: Analysis
Traces to upper level requirements: 0-19

1-34 Approximate Consensus

The RM shall provide an approximate consensus algorithm to cross-check the processing lane data interfaces.
Rationale:
Some values/results require approximate consensus. E.g. sensor values displayed in physical quantities cannot be compared bit-wise.
Means of Compliance: Analysis
Traces to upper level requirements: 0-20

7.2.2 Non-Functional Requirements

1-1 Generic Design

The RM IP must have a generic and configurable design.
Rationale:
To enable the support of different avionics systems without the need to modify the IP templates or even the framework.
Means of Compliance: Analysis
Traces to upper level requirements: 0-2

1-2 Hardware Description Language

The RM IP shall be developed in VHDL.
Rationale:
VHDL is the current used HDL by the FPGA developer team.

Means of Compliance: Analysis
Traces to upper level requirements: 0-7

1-3 RM Modes

The RM IP shall include several modes implementing different redundancy mechanisms.
Rationale:
To be flexible regarding the actual use-case and various avionics architectures and concepts.
Means of Compliance: Analysis
Traces to upper level requirements: N/A

1-4 RM Configurability

The RM IP must have configurable parameters which allow to configure the IP block(s) according to the specific use-case.
Rationale:
The IP must be configurable that an avionics developer must not adjust the IP by adjusting the code.
Means of Compliance: Analysis
Traces to upper level requirements: 0-3

1-18 Failure Definition

A channel shall be defined to be failed if the processing lanes determining a disagreement in one interface three times in a row or ten times in 1000 messages.
Rationale:
The first value disagreement would not necessarily indicate a fault. Therefore three disagreements in a row are required to classify a vaule as faulty.
Means of Compliance: Test
Traces to upper level requirements: N/A

1-37 Target Hardware

The RM IP must be developed to be embed into the FPGA part of an AMD system-on-chip.
Rationale:
The heterogeneous AMD SoCs are the main processing platform of DLR's current avionics computers. The FPGA part provides sufficient processing power while preventing the RM system from interfering with application partitions on the processor side.
Means of Compliance: Analysis
Traces to upper level requirements: 0-9

7.2.3 Interface Requirements

1-5 PS-PL Interfaces

The RM IP must have at least two PS-PL interfaces.
Rationale:
PS-PL interfaces are required to communicate between processor and FPGA part. At least two are required for a control and status interface and one for a application partition.
Means of Compliance: Analysis
Traces to upper level requirements: 0-9

1-7 Cross-Lane Interface

The RM IP must have a redundant bidirectional interface to communicate between the processing lanes.
Rationale:
Required to enable the commanding-monitoring channel architecture and provide a reliable communication interface between the lanes.
Means of Compliance: Analysis
Traces to upper level requirements: 0-10; 0-13

1-8 Peripheral Interface Ports

The RM must provide bidirectional ports for at least 10 data interfaces in a processing lane.
Rationale:
Required to obtain input data and transmit output data for multiple software partitions.
Means of Compliance: Analysis
Traces to upper level requirements: 0-14

1-24 Cross-Channel Interface

The RM must provide a redundant cross-channel communication interface.
Rationale:
Required to achieve fail-operational behavior. Channels need fault-tolerant communication interface to obtain consensus about their status and to communicate activation and passivation actions.
Means of Compliance: Analysis
Traces to upper level requirements: 0-14; 0-23

7.2.4 Performance Requirements

1-9 Simultaneous Processing

The RM must process all peripheral interfaces of a processing lane simultaneously.
Rationale:
To minimize the temporal overhead of the redundancy management tasks.
Means of Compliance: Analysis
Traces to upper level requirements: N/A

1-15 Fault Handling Time

The fault identification and treatment must not take longer than 1 ms.
Rationale:
To avoid fault propagation and to remain the continuous systems functionality, fault identification and treatment must be performed in a small time window. This value is

the length/duration of a 1 kHz time window.
Means of Compliance: Test
Traces to upper level requirements: N/A

1-23 Cross-Lane Interface Data Rate

The cross-lane communication interface shall provide a data rate of at least 200 Mbit/s.
Rationale:
To minimize the temporal overhead due to the required data exchange.
Means of Compliance: Test
Traces to upper level requirements: N/A

1-25 Cross-Channel Interface Data Rate

The cross-channel communication interface shall provide a data rate of at least 460,8 kbit/s.
Rationale:
Especially passivation event messages must be transmitted fast and with low latency that an available standby channel is able to react and activate itself rapidly.
Means of Compliance: Test
Traces to upper level requirements: N/A

1-28 Mode Switch Time

The switching from standby to active mode must take no longer than 1 ms.
Rationale:
In the event of a failure and the corresponding passivation of one channel, the take over of a standby channel must be performed rapidly. The 1 ms value represents the duration of one 1 kHz time window.
Means of Compliance: Test
Traces to upper level requirements: N/A

1-36 RM Time Offset

The RM shall not produce an execution time offset of more than 10 ms.

Rationale:

The RM must not affect the application processing schedule.

Means of Compliance: Test

Traces to upper level requirements: N/A

7.3 Redundancy Management IP Architecture

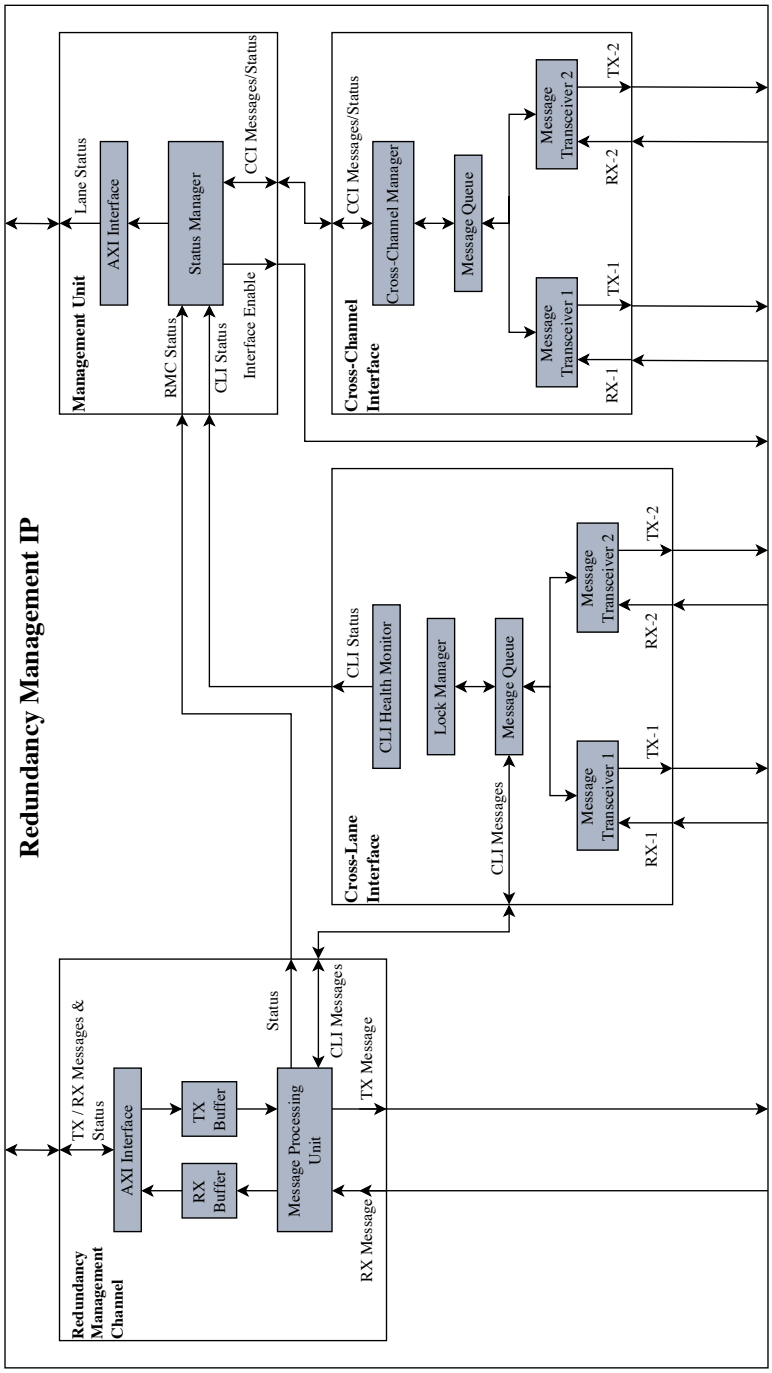


Figure 7.1: Redundancy Management IP architecture

7.4 RMC Status Message

The value '1' of a bit indicates that an item check result is true, correct, or valid. A value of '0' indicates a false, invalid, or incorrect check result of an item.

Bit	Description
0	TX Buffer full
1	TX Buffer empty
2	Reserved
3	RX Buffer full
4	RX Buffer empty
5	Reserved
6	TX CL-Response received
7	TX CL-Request received
8	TX CRC valid
9	TX CRC-2 (from other lane) valid
10	TX CRC-1 (from own lane) valid
11	TX Integrity valid
12	TX-2 Integrity (from other lane) valid
13	TX-1 Integrity (from own lane) valid
14	TX CL-Response Header valid
15	TX CL-Request Header valid
16	Reserved
17	RX Message received
18	RX CL-Response received
19	RX CL-Request received
20	RX CRC valid
21	RX CRC-2 (from other lane) valid
22	RX CRC-1 (from own lane) valid
23	RX Integrity valid
24	RX-2 Integrity (from other lane) valid
25	RX-1 Integrity (from own lane) valid
26	RX CL-Response Header valid
27	RX CL-Request Header valid
28-31	Reserved

Table 7.1: RMC status message structure

7.5 CLI Status Message

Bit	Description
0	CLI Channel-1 correct and operational
1	CLI Channel-2 correct and operational
2-31	Reserved

Table 7.2: CLI status message structure

The value '1' of a bit indicates that an item (CLI channel) is correct and operational. A value of '0' indicates that an item is incorrect and has failed.

7.6 Management Unit Status Message

Bit	Description
0	RMC 1 TX correct
1	RMC 1 RX correct
2	RMC 1 active
3	RMC 2 TX correct
4	RMC 2 RX correct
5	RMC 2 active
6	RMC 3 TX correct
7	RMC 3 RX correct
8	RMC 3 active
9	RMC 4 TX correct
10	RMC 4 RX correct
11	RMC 4 active
12	CLI Channel 1 correct
13	CLI Channel 2 correct
14	CLI active
15	Lane 1 active
16	Lane 2 active
17	Channel active
18	Interfaces enabled
19-31	Reserved

Table 7.3: MU status message structure

The value '1' of a bit indicates that an item (RMC/CLI/lane/channel) is correct or active. A value of '0' indicates that an item is failed or passive. Bit 18 indicates if the external interfaces are enabled (value '1') or disabled (value '0').

7.7 RM Single-Lane Hardware Test Results

```
Redundancy management System Test Softwa

Redundancy Management Status: ffffffff
Read RMC 1 status register: 12

TX message 1:
Header: dd10
Payload: 1010101
CRC: 3942

Redundancy Management Status: ffffffff
Read RMC 1 status register: f8d2

TX message 2:
Header: dd10
Payload: 2020202
CRC: 8cf2

Redundancy Management Status: ffffffff
Read RMC 1 status register: f8d2

TX message 3:
Header: dd10
Payload: 3030303
CRC: e063

Redundancy Management Status: ffffffff
Read RMC 1 status register: ffeffc2

TX message 4:
Header: dd10
Payload: 4040404
CRC: a790

Redundancy Management Status: ffffffff
Read RMC 1 status register: ffeffc2

Received RX message 1 Header: dd10
Received RX message 1 Payload: 3030303
Received RX message 1 CRC: e063

Redundancy Management Status: ffffffff
Read RMC 1 status register: ffeffc2

Received RX message 2 Header: dd10
Received RX message 2 Payload: 4040404
Received RX message 2 CRC: a790

Redundancy Management Status: ffffffff
Read RMC 1 status register: ffeffd2

Received RX message 3 Header: 0
Received RX message 3 Payload: 0
Received RX message 3 CRC: 0

Redundancy Management Status: ffffffff
Read RMC 1 status register: ffeffc2

Received RX message 4 Header: 0
Received RX message 4 Payload: 0
Received RX message 4 CRC: 0

Redundancy Management Status: ffffffff
Read RMC 1 status register: ffeffc2

Redundancy Management Status: ffffffff
Read RMC 1 status register: ffeffc2

End of Test Software
```

Figure 7.2: RM single-lane hardware test 2 - console output

```
Redundancy management System Test Software

Redundancy Management Status: ffffffff
Read RMC 1 status register: 12

TX message 1:
Header: dd10
Payload: 1010101
CRC: 3942

Redundancy Management Status: ffffffff
Read RMC 1 status register: f8d2

TX message 2:
Header: dd10
Payload: 2020202
CRC: 8cf2

Redundancy Management Status: ffffffff
Read RMC 1 status register: f8d2

TX message 3:
Header: dd10
Payload: 3030303
CRC: e062

Redundancy Management Status: ffffffff
Read RMC 1 status register: 12

TX message 4:
Header: dd10
Payload: 4040404
CRC: a790

Redundancy Management Status: fff97ffa
Read RMC 1 status register: 12

Received RX message 1 Header: 0
Received RX message 1 Payload: 0
Received RX message 1 CRC: 0

Redundancy Management Status: fff97ffa
Read RMC 1 status register: 12

Received RX message 2 Header: 0
Received RX message 2 Payload: 0
Received RX message 2 CRC: 0

Redundancy Management Status: fff97ffa
Read RMC 1 status register: 12

Received RX message 3 Header: 0
Received RX message 3 Payload: 0
Received RX message 3 CRC: 0

Redundancy Management Status: fff97ffa
Read RMC 1 status register: 12

Received RX message 4 Header: 0
Received RX message 4 Payload: 0
Received RX message 4 CRC: 0

Redundancy Management Status: fff97ffa
Read RMC 1 status register: 12

Redundancy Management Status: fff97ffa
Read RMC 1 status register: 12

End of Test Software
```

Figure 7.3: RM single-lane hardware test 3 - console output

7.8 RM Dual-Lane Hardware Test Results

```
Redundancy management System Test Software

Redundancy Management Status: ffffffff

Read RMC 1 status register: 12
Read RMC 2 status register: 12
Read RMC 3 status register: 12
Read RMC 4 status register: 12

write 5 TX messages to each RMC

RMC 1 TX
TX message 1:
Header: dd10
Payload: 1010101
CRC: 3943

Redundancy Management Status: ffffffff

Read RMC 1 status register: ffeffc2

RMC 2 TX
TX message 2:
Header: dd10
Payload: 2020202
CRC: 8cf3

Redundancy Management Status: ffffffff

Read RMC 2 status register: ffeffc2

RMC 3 TX
TX message 3:
Header: dd10
Payload: 3030303
CRC: e063

Redundancy Management Status: ffffffff

Read RMC 3 status register: ffeffc2

RMC 4 TX
TX message 4:
Header: dd10
Payload: 4040404
CRC: a790

Redundancy Management Status: ffffffff

Read RMC 4 status register: ffeffc2
```

Figure 7.4: RM dual-lane hardware test 1 - console output 1

```
Read the RX messages from each RMC

Redundancy Management Status: ffffffff

RMC 1 RX
Received RX message 1 Header: dd10
Received RX message 1 Payload: 1010101
Received RX message 1 CRC: 3943
Read RMC 1 status register: ffeffc2

RMC 2 RX
Received RX message 2 Header: dd10
Received RX message 2 Payload: 2020202
Received RX message 2 CRC: 8cf3
Read RMC 2 status register: ffeffc2

RMC 3 RX
Received RX message 3 Header: dd10
Received RX message 3 Payload: 3030303
Received RX message 3 CRC: e063
Read RMC 3 status register: ffeffc2

RMC 4 RX
Received RX message 4 Header: dd10
Received RX message 4 Payload: 4040404
Received RX message 4 CRC: a790
Read RMC 4 status register: ffeffc2

Redundancy Management Status: ffffffff

RMC 1 RX
Received RX message 1 Header: dd10
Received RX message 1 Payload: 1010101
Received RX message 1 CRC: 3943
Read RMC 1 status register: ffeffc2

RMC 2 RX
Received RX message 2 Header: dd10
Received RX message 2 Payload: 2020202
Received RX message 2 CRC: 8cf3
Read RMC 2 status register: ffeffc2

RMC 3 RX
Received RX message 3 Header: dd10
Received RX message 3 Payload: 3030303
Received RX message 3 CRC: e063
Read RMC 3 status register: ffeffc2

RMC 4 RX
Received RX message 4 Header: dd10
Received RX message 4 Payload: 4040404
Received RX message 4 CRC: a790
Read RMC 4 status register: ffeffc2

Redundancy Management Status: ffffffff
```

Figure 7.5: RM dual-lane hardware test 1 - console output 2

```
RMC 1 RX
Received RX message 1 Header: dd10
Received RX message 1 Payload: 1010101
Received RX message 1 CRC: 3943
Read RMC 1 status register: ffeffd2

RMC 2 RX
Received RX message 2 Header: dd10
Received RX message 2 Payload: 2020202
Received RX message 2 CRC: 8cf3
Read RMC 2 status register: ffeffd2

RMC 3 RX
Received RX message 3 Header: dd10
Received RX message 3 Payload: 3030303
Received RX message 3 CRC: e063
Read RMC 3 status register: ffeffd2

RMC 4 RX
Received RX message 4 Header: dd10
Received RX message 4 Payload: 4040404
Received RX message 4 CRC: a790
Read RMC 4 status register: ffeffd2

Redundancy Management Status: ffffffff

RMC 1 RX
Received RX message 1 Header: 0
Received RX message 1 Payload: 0
Received RX message 1 CRC: 0
Read RMC 1 status register: ffeffd2

RMC 2 RX
Received RX message 2 Header: 0
Received RX message 2 Payload: 0
Received RX message 2 CRC: 0
Read RMC 2 status register: ffeffd2

RMC 3 RX
Received RX message 3 Header: 0
Received RX message 3 Payload: 0
Received RX message 3 CRC: 0
Read RMC 3 status register: ffeffd2

RMC 4 RX
Received RX message 4 Header: 0
Received RX message 4 Payload: 0
Received RX message 4 CRC: 0
Read RMC 4 status register: ffeffd2

Redundancy Management Status: ffffffff

Read RMC 1 status register: ffeffd2
Read RMC 2 status register: ffeffd2
Read RMC 3 status register: ffeffd2
Read RMC 4 status register: ffeffd2

End of Test Software
```

Figure 7.6: RM dual-lane hardware test 1 - console output 3

```
Redundancy management System Test Software

Redundancy Management Status: ffffffff

Read RMC 1 status register: 12
Read RMC 2 status register: 12
Read RMC 3 status register: 12
Read RMC 4 status register: 12

write 5 TX messages to each RMC

RMC 1 TX
TX message 1:
Header: dd10
Payload: 1010101
CRC: 3943

Redundancy Management Status: ffffffff

Read RMC 1 status register: ffeffc2

RMC 2 TX
TX message 2:
Header: dd10
Payload: 2020202
CRC: 8cf3

Redundancy Management Status: ffffffff

Read RMC 2 status register: ffeffc2

RMC 3 TX
TX message 3:
Header: dd10
Payload: 3030303
CRC: e062

Redundancy Management Status: ffffffff

Read RMC 3 status register: f8d2

RMC 4 TX
TX message 4:
Header: dd10
Payload: 4040404
CRC: a790

Redundancy Management Status: ffffffff

Read RMC 4 status register: ffeffc2
```

Figure 7.7: RM dual-lane hardware test 2 - console output 1

```
RMC 1 TX
TX message 1:
Header: dd10
Payload: 1010101
CRC: 3943

Redundancy Management Status: ffffffff

Read RMC 1 status register: ffeffc2

RMC 2 TX
TX message 2:
Header: dd10
Payload: 2020202
CRC: 8cf3

Redundancy Management Status: ffffffff

Read RMC 2 status register: ffeffc2

RMC 3 TX
TX message 3:
Header: dd10
Payload: 3030303
CRC: e062

Redundancy Management Status: ffffffff

Read RMC 3 status register: 12

RMC 4 TX
TX message 4:
Header: dd10
Payload: 4040404
CRC: a790

Redundancy Management Status: fff97ebf

Read RMC 4 status register: 12

RMC 1 TX
TX message 1:
Header: dd10
Payload: 1010101
CRC: 3943

Redundancy Management Status: fff97ebf

Read RMC 1 status register: 12

RMC 2 TX
TX message 2:
Header: dd10
Payload: 2020202
CRC: 8cf3

Redundancy Management Status: fff97ebf

Read RMC 2 status register: 12
```

Figure 7.8: RM dual-lane hardware test 2 - console output 2

```
Read the RX messages from each RMC

Redundancy Management Status: fff97ebf

RMC 1 RX
Received RX message 1 Header: 0
Received RX message 1 Payload: 0
Received RX message 1 CRC: 0
Read RMC 1 status register: 12

RMC 2 RX
Received RX message 2 Header: 0
Received RX message 2 Payload: 0
Received RX message 2 CRC: 0
Read RMC 2 status register: 12

RMC 3 RX
Received RX message 3 Header: 0
Received RX message 3 Payload: 0
Received RX message 3 CRC: 0
Read RMC 3 status register: 12

RMC 4 RX
Received RX message 4 Header: 0
Received RX message 4 Payload: 0
Received RX message 4 CRC: 0
Read RMC 4 status register: 12

Redundancy Management Status: fff97ebf

RMC 1 RX
Received RX message 1 Header: 0
Received RX message 1 Payload: 0
Received RX message 1 CRC: 0
Read RMC 1 status register: 12

RMC 2 RX
Received RX message 2 Header: 0
Received RX message 2 Payload: 0
Received RX message 2 CRC: 0
Read RMC 2 status register: 12

RMC 3 RX
Received RX message 3 Header: 0
Received RX message 3 Payload: 0
Received RX message 3 CRC: 0
Read RMC 3 status register: 12

RMC 4 RX
Received RX message 4 Header: 0
Received RX message 4 Payload: 0
Received RX message 4 CRC: 0
Read RMC 4 status register: 12

Redundancy Management Status: fff97ebf
```

Figure 7.9: RM dual-lane hardware test 2 - console output 3

Bibliography

[1] SAE International. *"Guidelines for Conducting the Safety Assessment Process on Civil Aircraft, Systems, and Equipment: ARP4761A"*. Dec. 20, 2023. DOI: https://doi.org/10.4271/ARP4761A.

[2] Radio Technical Commission for Aeronautics (RTCA). *"DO-297: Integrated Modular Avionics (IMA) Development Guidance and Certification Considerations"*. Nov. 8, 2005.

[3] B. Lüttig. "Selective Middleware: An approach Towards a Stepwise Integration of a Fault-Tolerant, Distributed Avionics-Platform for Applications in Large Aircraft Using Integrated Modular Avionics". Thesis in partial fulfillment of the requirements for the degree of Doctor of Engineering Sciences. Stuttgart: University of Stuttgart, 2022.

[4] SAE International. *"Guidelines for Development of Civil Aircraft and Systems: ARP4754A"*. Dec. 21, 2010. DOI: https://doi.org/10.4271/ARP4754A.

[5] W. G. Bouricius et al. "Reliability Modeling for Fault-Tolerant Computers". In: *IEEE Transactions on Computers* C-20.11 (1971), pages 1306–1311. DOI: 10.1109/T-C.1971.223132.

[6] J. H. Lala and R. E. Harper. "Architectural principles for safety-critical real-time applications". In: *Proceedings of the IEEE* 82.1 (1994), pages 25–40. DOI: 10.1109/5.259424.

[7] E. F. Hitt and D. Mulcare. *"Fault-Tolerant Avionics"*. 2001.

[8] D. M. Johnson. "A review of fault management techniques used in safety-critical avionic systems". In: *Progress in Aerospace Sciences* 32.5 (1996), pages 415–431. DOI: 10.1016/0376-0421(96)82785-0.

[9] J. Hartfiel. *"TE0821 Technical Reference Manual"*. Edited by Trenz Electronic. July 2021.

[10] Advanced Micro Devices. *"Ultrascale+ MPSoC Data Sheet: DS891"*. Nov. 7, 2022.

[11] Advanced Micro Devices. *"Ultrascale+ Architecture"*. Jan. 10, 2024.

[12] S. D. Brown. *"Field-Programmable Gate Arrays"*. Volume v.180. The Springer International Series in Engineering and Computer Science Ser. New York, NY: Springer, 1992.

[13] E. Monmasson and M. N. Cirstea. "FPGA Design Methodology for Industrial Control Systems—A Review". In: *IEEE Transactions on Industrial Electronics* 54.4 (2007), pages 1824–1842. DOI: 10.1109/TIE.2007.898281.