



Distributed Real-time Architecture for Mixed Criticality Systems

Cross Domain Mixed-Criticality Pattern D 5.3.1

Project Acronym	DREAMS	Grant Number	Agreement	FP7-ICT-2013.3.4-610640	
Document Version	1.0	Date	05/07/2016	Deliverable No.	[D 5.3.1]
Contact Person	Imanol Martinez	Organisation		IK4-IKERLAN	
Phone	+34 943712400	E-Mail		imartinez@ikerlan.es	

Version History

Version No.	Date	Change	Author(s)
0.0	09.11.2015	First review and Draft.	IKL, TUV
0.1	22.01.2016	Extended definition of patterns	IKL, UPV, FENTISS
0.2	04.06.2016	Implementations and results of patterns are included	IKL, UPV, FENTISS
0.3	30.06.2016	Reviewed and improved version.	IKL, TUV, UPV, FENTISS
1.0	13/07/2016	Feedbacks of deliverable's internal revision. Minor modifications.	IKL

Contributors

Name	Partner	Part Affected	Date
Martínez, Imanol	IKL	All.	
Larrucea, Asier	IKL	All.	
Geven, Arjan	TTT	All.	
Steiner, Wilfried	TTT	All.	
Zwirschmayr, Jakob	TTT	All.	
Haugen, Øystein	SINTEF	All.	
Trapman, Ton	ALSTOM	All.	
Crespo, Alfons	UPV	4.1.1 and 4.1.2	
Simó, José	UPV	4.1.1 and 4.1.2	
Brocal, Vicent	FENTISS	4.1.1 and 4.1.2	
Coronel, Javier	FENTISS	4.1.1 and 4.1.2	
Heinen, Robert	TUV	All.	
Klaes, Gernot	TUV	All.	
Bouwer, Gebhard	TUV	All.	

Index

Version History.....	2
Contributors.....	2
1 Executive Summary.....	5
2 Introduction	6
2.1 Mixed-Criticality.....	6
2.2 Virtualization.....	6
2.3 COTS devices.....	6
2.4 Mixed-Criticality Networks	7
2.5 Certification Standards	7
2.6 Cross-Domain Patterns	8
2.6.1 Cross Domain Pattern Representation	8
3 System Architecture.....	11
3.1 COTS Multi-Core devices.....	13
3.1.1 Shared Memory	14
3.1.2 Coherency Management Unit.....	14
3.1.3 Interconnection Management Unit	15
3.1.4 Interrupt Controller.....	15
3.1.5 Programmable Logic	16
3.2 Hypervisor	16
3.3 Mixed-Criticality Network.....	17
4 Cross-Domain Mixed Criticality Patterns	18
4.1 Hypervisor	19
4.1.1 NoC Accessible Critical Memory Area Diagnosis Pattern	19
4.1.2 Critical Partition Diagnosis Pattern	21
4.1.3 Digital I/O Server Pattern.....	27
4.1.4 Communication I/O Server Pattern	32
4.2 COTS processor	35
4.2.1 Shared Memory Diagnosis Pattern	35
4.2.2 Cache Coherency Management Unit Diagnosis Pattern	45
4.2.3 Inter-Connection Management Unit Diagnosis Pattern	54
4.2.4 Interrupt Controller Diagnosis Pattern	57
4.3 Mixed-criticality Network	59
4.3.1 NoC Pattern.....	59
5 Conclusions	64
6 List of Open Points (LOP)	65
Abbreviated terms	75

Bibliography	77
--------------------	----

1 Executive Summary

This document includes the definition and implementation of cross-domain mixed-criticality patterns. These patterns aim to guide and support engineers towards solutions that solve commonly occurring problems in the development of mixed-criticality products, from design to verification and validation. The cross-domain patterns which are defined in this deliverable have been identified as the result of the analysis of the IEC 61508 safety-related standard and its application for today's mixed-criticality systems. The identified cross-domain patterns are selected based on previous projects such as GENESYS [1], results from European FP7 TERESA project [2], the expertise of industrial partners and tasks of FP7 DREAMS project, including the state-of-the-art in the validation, verification and certification of mixed-criticality systems [3], the modular safety cases (MSCs) from T5.1 [4-6], the requirements of DREAMS case studies (WP6, WP7 and WP8) and the deliverables of WP1 [7]. On the other hand, the cross-domain patterns defined in this deliverable applicable to the wind-turbine, avionic and healthcare demonstrators of WP6, WP7 and WP8. Figure 1 shows the inputs and outputs of this deliverable regarding to European FP7 DREAMS project.

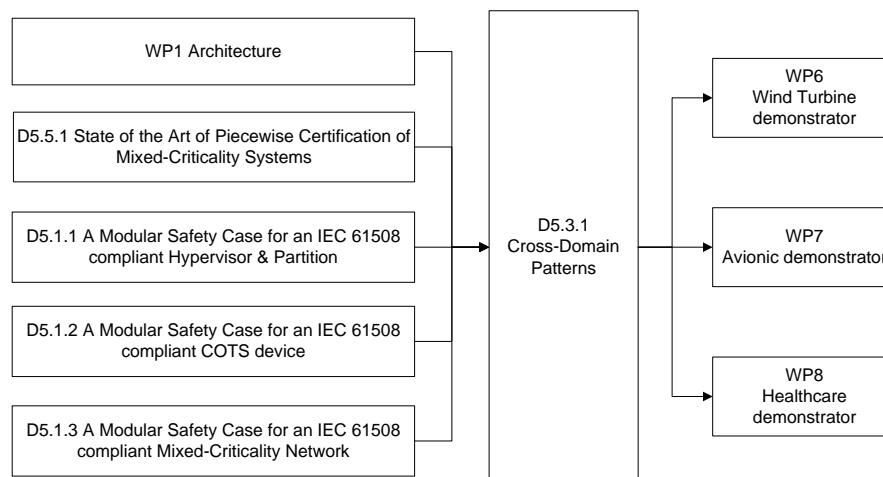


Figure 1: Linkage of DREAMS technologies.

Section 2 sets out to introduce the basic concepts mentioned throughout this deliverable. Section 3 defines the system architecture on which this deliverable is based. Section 4 defines the most remarkable cross-domain patterns that are currently the goal for many embedded system developers. Section 6 contains the results of the functional safety assessment (e.g., List of Open Points (LOP)).

2 Introduction

This section defines the basic concepts mentioned throughout this deliverable.

2.1 Mixed-Criticality

The architecture of embedded systems in multiple domains commonly follows a federated architecture paradigm where a system is composed of interconnected subsystems with well-defined functionalities. The ever increasing demand for additional functionalities is a growing trend in several domains such as transportation and industrial control. For example, the control systems of the modern off-shore wind turbines manage up to three thousand inputs / outputs and several hundred functions. The integration of additional functionalities with different criticality also leads to the increase in the number of subsystems, connectors and wires, increasing the overall cost-size-weight factor and reducing the overall reliability of the system. For example, in automotive domain, between 30 - 60 % of electrical failures are attributed to connector problems.

A mixed-criticality system is referred as the integration of the HW (HW), operating system, middleware services and application software (SW) with different levels of criticality into the same embedded computing platform [8]. The integrated approach improves scalability and reliability by reducing the amount of systems-wires-connectors, which in turn reduces the overall cost-size-weight factor. However, safety certification according to industrial standards poses many challenges as provide sufficient evidences to demonstrate that the resulting system is safe for its purpose [8, 9].

2.2 Virtualization

A virtual machine (VM) is a SW implementation of a machine (computer) that executes programs like a real machine. Hypervisor (also known as virtual machine monitor (VMM)) is a layer of SW or a combination of SW and HW that allows running several independent execution environments in a single computer platform. Several terms are used as synonym of independent execution environments: guest operating system, virtual machine, partition or domain. The key difference between hypervisor technology and other kind of virtualizations (such as Java virtual machine or software emulation) is the performance.

In real-time embedded applications, the predictability and efficiency are requirements to be considered. The virtualization techniques such as hypervisor can be used to achieve the temporal and spatial isolation jointly with real-time constraints require strict design methods and efficient solutions to guarantee the system behavior. Hypervisor technology is a promising solution for the development and certification of safety critical embedded systems. For example, XtratuM [10] is a virtualization solution based on hypervisor technology. It runs directly over the HW and abstracts it creating several runtime environments, also called partitions, where applications with different criticality level can be executed (e.g., Safety Integrity Level (SIL) 1 – 4 in accordance with the IEC 61508 safety standard).

2.3 COTS devices

The use of multi-core Commercial Off-The-Shelf (COTS) processors is gaining popularity among different embedded systems domains driven by the demand of low cost, increased complexity solutions and shortened time to market, which in turn is driven by the physical limits of single-core architectures [11-14]. However, the implementation of COTS devices in safety critical and mixed-criticality systems is hindered by numerous drawbacks that can compromise the safety feature. E.g., shared resources, limited service history or hidden properties.

One of the main reasons for that is that multi-core COTS devices usually are designed for offering maximum average performance by increasing the complexity of the underlying architecture [11, 14]. E.g., P4080 [15], ZYNQ [16], Hercules [17] or MPC5643L [18]. This is somehow in conflict with the common practice in safety critical systems that aim to employ simple, predictable and proven-in-use processors. Consequently, the integration of applications with different criticality (such as safety, real-time or security) into COTS multi-core processors leads to several challenges related with their certification (e.g., the assurance of the temporal independence).

2.4 Mixed-Criticality Networks

In distributed systems, processing and data are spread out over multiple systems (e.g., multi-core processors) over communication networks (e.g., TTEthernet). The broad trend of the integration of functionalities with different criticality on a single embedded computing platform requires the usage of communication media systems with different criticality. These communication systems, which are also called as mixed-criticality networks, shall be capable of supporting a safe and a predictable message exchange between distribution application subsystems (DAS) with different criticality. Mixed-criticality networks are targeted as the natural replacement of traditional legacy buses due to the increasing amount of data that is required to be exchanged, the decrease of cost factor, the higher speed and the integration with existing network infrastructures. They can be divided into off-chip and on-chip networks with real-time and non-real time features. Off-chip networks are used to connect different devices that may be located far away (physically) from each other. On the other hand, on-chip networks provide communication between the elements of the device (e.g., cores, memories and peripherals). For example, EtherCAT is a real-time industrial Ethernet on-chip network. The use of internal network-on-chip systems shifts the problems associated with traditional networks into the chip.

However, the shift towards mixed-criticality networks poses many challenges related to increasing demand for real-time, safety and security features in different application domains such as automotive and railway. To cope with those challenges *white channel* and *black channel* network approaches are defined by the IEC 61508-2 safety standard [19]. *White channel* shall be designed, implemented and validated according to the IEC 61508-2-3 [19, 20] and IEC 61784-3 [21] or IEC 62280 [22] safety-related standards. Instead, in the case of the *black channel* it is assumed that not all parts of the communication channel are designed and validated according to the IEC 61508 safety standard. In that case, the safety-related subsystems or elements that compose the communication channel shall implement IEC 61784-3 [21] or IEC 62280 [22] compliant measures and diagnostic techniques to ensure the failure performance of the communication process.

2.5 Certification Standards

Certification is the process that ensures the compliance of a product, system, subsystem or element with respect to a specific standard (e.g., IEC 61508 or ISO 26262). In safety domain, the certification process assess that a product, system or element is safe enough for its purpose, with a given confidence level and in a given environment.

IEC 61508 [19, 20, 23] is an international standard for Electrical, Electronic and Programmable Electronic (E/E/PE) safety-related systems. This safety standard is used as the reference standard by multiple domain specific standards such as machinery, industry process, automotive and railway. IEC 61508 [19, 20, 23] defines the concept of Safety Integrity Level (SIL) as a discrete risk reduction level provided by a safety-related system with values in range between 1 and 4, where 4 is the highest level and 1 the lowest. As a rule of thumb, higher SIL level means higher certification cost.

IEC 61508 does not directly support nor restrict the certification of mixed-criticality systems. Whenever a system integrates safety functions of different criticality, sufficient independence of implementation must be shown among the functions, otherwise all integrated functions will need to

meet the highest integrity level. Sufficient independence of implementation is established when the probability of a dependent failure between the higher and lower integrity parts is sufficiently low in comparison with the highest SIL [8, 9].

2.6 Cross-Domain Patterns

Cross-domain patterns are widely used universal approaches for describing and documenting recurring solutions for design problems of systems, subsystems and elements. They are used to guide and support engineers towards solutions that solve commonly occurring problems in the development of mixed-criticality products (from design to verification & validation).

2.6.1 Cross Domain Pattern Representation

This section presents the cross-domain pattern representation approaches, including traditional, commonly used and our custom pattern representation.

Traditional pattern representation is derived from the definition of design patterns. In general, the traditional pattern representation consists of four essential elements:

- **Name:** A meaningful name for the pattern.
- **Context:** Describes the preconditions or the situation in which the pattern can be used to solve the problem.
- **Problem:** Describes the problem that is intended to solve by the pattern.
- **Solution:** Defines the solution to the problem.

Commonly used patterns [24-28], which are based on traditional patterns, differ from those last ones in terms of different element naming or additional elements. For instance, *Context* element is called *Applicability* or *Preconditions* in some common pattern representation.

On the other hand, patterns for mixed-criticality systems increase some levels of safety on pattern representation. Most popular works are defined in [29, 30]. In addition, in [31] a SW architecture design method for safety-related systems is presented.

In order to represent the cross-domain patterns described throughout this deliverable, a pattern representation based on elements of traditional pattern approach (grey elements) and custom elements (black elements) is used.

- **Pattern ID:** A collection of characters to identify the pattern PAT-AAAA-XX.
 - AAAA – Pattern ID.
 - XX – Pattern version.
- **Pattern Name:** A meaningful name to describe the pattern.
- **Related patterns:** The closely related design patterns to this pattern.
- **Type:** Gives the classification of the design pattern into:
 - HW: when the pattern contains HW.
 - SW: when the pattern contains SW.
 - Combination of HW and SW: other cases.
- **Context:** The general situation in which the design pattern can be applied.
- **Problem:** This part gives a summary of the problem which is addressed and solved by this pattern.

- **Solution under consideration:** A short description of the solution to be implemented.
- **Board Name:** The board/processor/system where the pattern is implemented.
- **Implementation:** This part gives the aspects, hints and techniques that should be taken into consideration when implementing the pattern.
- **Results:** Includes results of implementation.
- **Additional Consideration:** Includes additional considerations that are relevant for the pattern that is implemented (e.g., linkage to a modular safety case (MSC)).
- **References:** Bibliographical references.

Pattern ID:	
Pattern Name:	
Related pattern:	
Type:	
Context:	
Problem:	
Solution under consideration:	
Board Name:	
Implementation:	
Results:	

Additional Considerations:
References:

Table 1: Customized cross-domain pattern - Overview.

3 System Architecture

Embedded systems have commonly followed a federated architecture paradigm in which each Distributed Application Subsystem (DAS) is implemented on its own stand-alone distributed HW base with a well-defined functionality. The soaring demand for high performance and increasing functionality jeopardizes the viability of this approach, leading to the trend of moving towards integrated architectures [32]. As a consequence, applications with different criticality level can be integrated on a single embedded computing system. The trend towards multi-core processors has further contributed to this tendency. Multi-core devices provide benefits in terms of cost, size and weight reduction as well as improved scalability by reducing the amount of wires and connectors. Commercial Off-The-Shelf (COTS) multi-core processors are designed to offer maximum average performance at the cost of increasing complexity. The use of these processors is gaining popularity in different embedded systems domains (e.g., railway, automotive and elevation). However, the shift towards multi-core processors is hindered by numerous drawbacks (e.g., shared resources) that can compromise the safety of the mixed-criticality systems. One of the main reasons for that is that multi-core COTS devices are designed with the objective of offering maximum average performance that is usually achieved by increasing the complexity of the underlying architecture. E.g., P4080 [15], ZYNQ [16], Hercules [17], MPC5643L [18].

To tackle challenges described before, partitioning mechanisms such as hypervisors are commonly used solutions to limit the impact of changes to reduced areas (partitions) of the system, enabling in turn the reusability of those areas and reducing the system's complexity. The resultant partitions can be designed, developed and certified individually with different level of criticality (e.g., SIL 1 to 4 according to the IEC 61508 safety standard).

In distributed systems, processing and data are spread out over multiple systems (e.g., multi-core processors) over networks (e.g., TTEthernet). The broad trend of the integration of functionalities of different criticality on a single embedded computing platform requires the use of communication media systems with different criticality. These communication media systems, also called to as mixed-criticality networks, support safe and predictable message exchanges between DAS of different criticality. Mixed-criticality networks are targeted as the natural replacement of legacy fieldbuses due to the increasing amount of data required to be exchanged, the decrease of cost factor, the higher speed and the integration with existing network infrastructures. However, the shift towards mixed-criticality networks poses many challenges related to increasing demand for real-time, safety and security in different application domains (e.g., automotive or railway). The use of network on-chip systems brings the problems associated with traditional networks into the chip. These networks are not only used to connect the processing cores but also to interconnect all the sub-modules, peripherals, memories, interrupt controllers, cache controllers and others.

Figure 2 presents the system architecture that is used as a pillar in this deliverable for identifying, describing and implementing the remarkable mixed-criticality cross-domain patterns. This system architecture is based on a COTS multi-core device that contains a dual-core processing system (PS) and a programmable logic (PL) where a single or multiple soft-core processors may be implemented. In addition, this architecture provides, among others, private cache memories for each ARM processor, a shared cache memory, memories such as the on-chip memory (OCM) and DDR, a memory coherency and interconnection management unit (SCU) and a generic interrupt controller (GIC) with different levels of priority and two levels of security (secure or none secure). In addition, as shown in Figure 3, the multi-core device is partitioned by means of XtratuM hypervisor [10], thus enabling the implementation of a wide set [0:N] of functionalities with different criticality (e.g., SIL1 to SIL4 according to the IEC 61508 safety standard). On the other hand, in order to provide a safe communication interface among partitions and avoid as much as possible the interferences which may be caused by the shared resources of the multi-core COTS device, a black channel network approach is implemented. The black channel network is based on the STMicroelectronics network-

on-chip (STNoC) where on top of it a safety communication layer (SCL) is implemented. The SCL implements the safety-related functionalities of the communication system and measures and diagnoses the STNoC network.

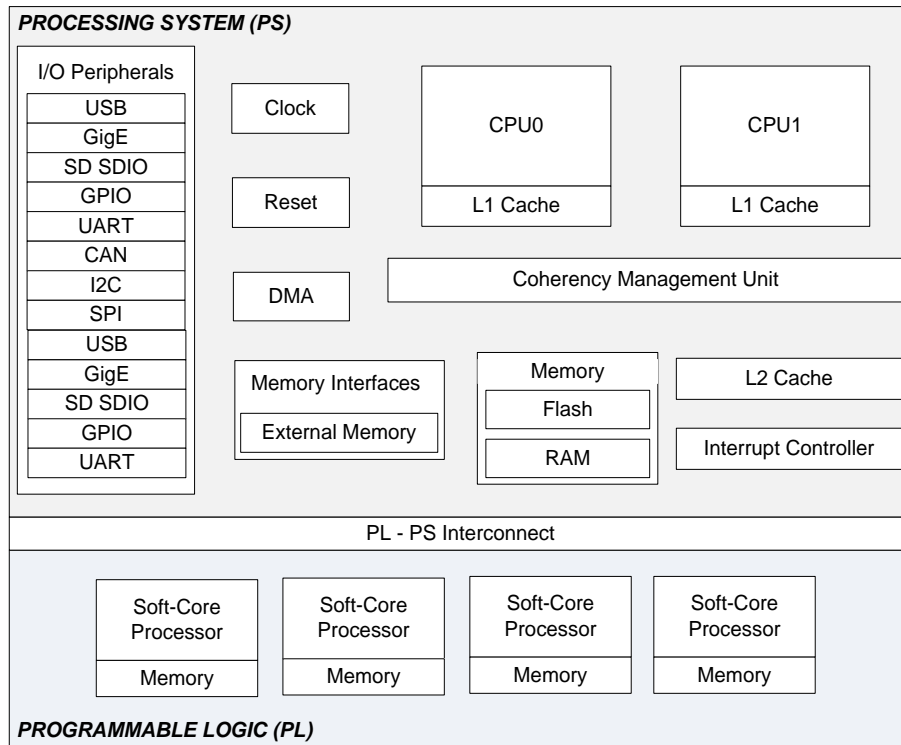


Figure 2: System Architecture - Overview.

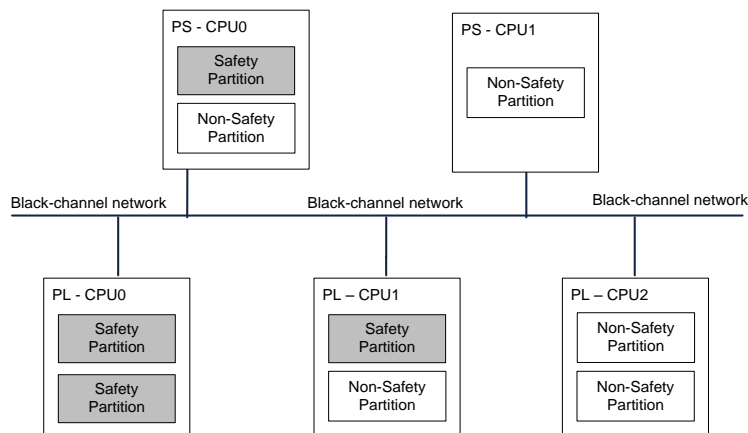


Figure 3: System Partitioning & Communication - Example.

In previous deliverables D5.1.1, D5.1.2 and D5.1.3 the MSCs for an IEC 61508 compliant hypervisor, partition, COTS device and mixed-criticality network are defined. Those MSCs and the analysis of the IEC 61508 compliant safety standard have given rise to the identification of common sources of issues related to the mixed-criticality system architectures and its components (e.g., COTS device, hypervisor, partitions and mixed-criticality network). In the following subsections the identified issues for mixed-criticality systems are analysed. Section 3.1 analyses the challenges related to COTS multi-core devices. Section 3.2 analyses the issue of hypervisor based virtualization mechanisms, instead, Section 3.3 analyses the issues related to the mixed-criticality networks implemented on today's mixed-criticality systems.

3.1 COTS Multi-Core devices

COTS multi-core devices are low cost and complex solutions with short time-to-market. They are commonly used devices in real-time embedded computing systems. These devices contain components which may cause drawbacks and may jeopardize the safety of the system (e.g., simultaneous running of tasks or/and the sharing of the resources between more than one component). Figure 4 and Figure 5 show the remarkable challenging components (highlighted in grey) of the ZYNQ 7000 and P4080 multi-core devices, including the shared memory (L2 cache), the interrupt controller (GIC), the interconnection management unit, the coherency management unit (SCU, CoreNet) and the direct memory access (DMA).

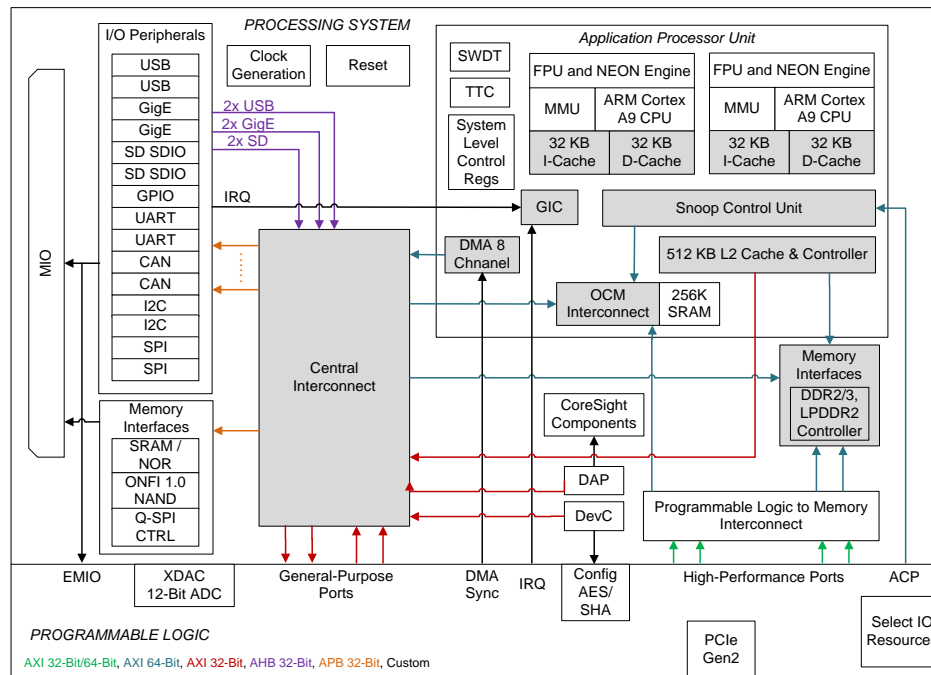


Figure 4: ZYNQ 7000 – Sources of interference. (Source [16])

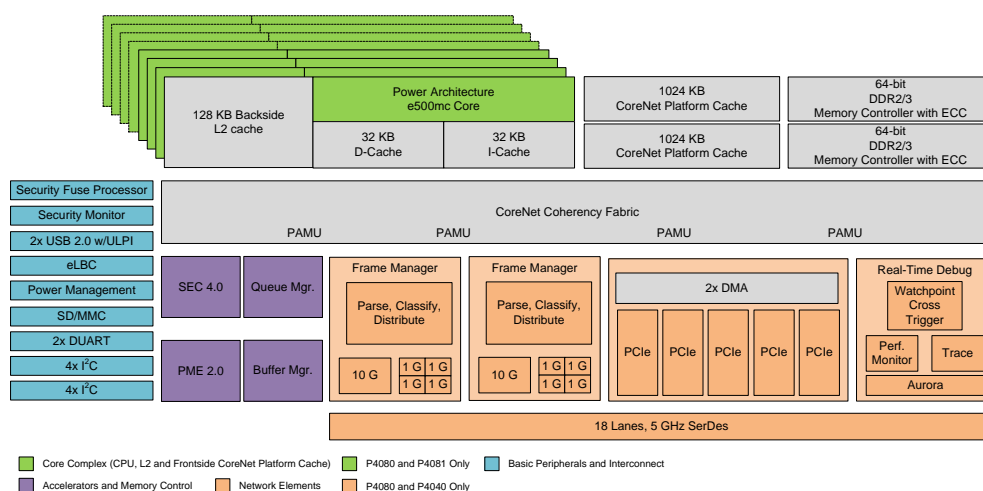


Figure 5: P4080 – Sources of interference. (Source [15])

In the following sections there are described and analysed the components of today's multi-core COTS devices which are identified such as challenging.

3.1.1 Shared Memory

Modern multi-core devices contain different cache memories which can be applied for private use (e.g., L1 cache) or can be shared between the components of the device (e.g., L2 cache). Secondary cache or L2 cache memory is commonly used to improve the performance of the system when significant data traffic is generated by the processor. The use of L2 cache assumes the presence of a primary cache or L1 cache, which is usually coupled or internal to the processor (See Figure 3 and Figure 5). However, the use of the secondary cache memory implies ever greater complexity of the system and a new source of interferences which may lead to an undesirable behaviour of the system. For example, two cores may access (write/read) to the same shared memory at the same time, causing the blocking effect, which may cause the failure of the overall system.

In single-core architecture domain, the IEC 61508 safety standard covers the failures caused by memory sharing such as the causal factors of the execution interference between elements of a single computer platform (See Annex F of IEC 61508-3 [20] *Techniques for achieving non-interference between SW elements on a single computer*). In addition, this safety standard recommends a set of measures and diagnostic techniques to detect the random failures of both variable and invariable memories. However, the measures and diagnostic techniques recommended by this standard are focused on single-core architectures where a resource (e.g., memory region) cannot be shared by more than one component at the same time. Instead, in multi-core architectures the sharing of a resource between more than one component (e.g., two cores) is a usual task. Therefore the measures and diagnostic techniques which are recommended by the IEC 61508 safety standard are not directly applicable to multi-core architectures, but have to be extended according to the given conditions.

Different research studies propose techniques to solve or reduce as much as possible the interferences caused by the shared memories. For example, techniques to improve the performance of the system by reducing the memory interferences of the applications [33-35] are proposed. These techniques focus in scheduling policies which provide request prioritization and reduce the inter-partition interferences. Other solutions aim to control the mapping of application's data to memory channels [36].

In short, the shared memories can be considered such as a recurrent source of interferences in nowadays mixed-criticality systems based on multi-core devices. In this line of thought, the shared memories and their failures must be deeply analyzed to provide new measures and diagnostic techniques to detect their associated interferences and act in consequence. In Subsection 4.2.1 a cross-domain pattern for detecting the failures of the shared memories is proposed.

3.1.2 Coherency Management Unit

The coherency management unit is responsible for notify all the cores, memories (e.g., L1 and L2 caches and the On-chip memory (OCM)), the programmable logic (PL) and others of changes to shared values, ensuring that all copies of the data are consistent. For example, in multi-core architectures coherency related issues usually arise with inconsistent data. For example, Core A (see Figure 6) has a copy of a memory block from a previous read and Core B changes the memory block. Therefore, Core A could be left with an invalid cache of memory, thus generating an inconsistency.

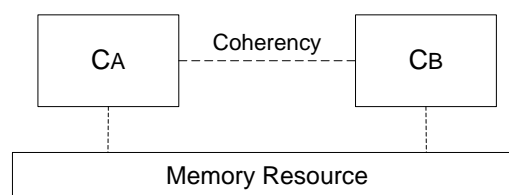


Figure 6: Memory Coherency – Overview.

Coherency mechanisms include the directory-based, snooping and snarfing techniques, although the two most common techniques are directory-based and snooping. In directory-based technique, the data to be shared is placed in a common directory that manages and maintains the coherency. Instead, in snooping the memory is monitored by all the devices which require coherency. This technique assumes that each processor has its own cache and that there is a shared main memory

Summarizing, the coherency management unit is required in today's multi-core devices for managing the coherency of the memory and the processors. Therefore, in the case that the coherency fails and the data that is stored in the memory is changed, the updated data will not be spread among the cores, leading to inconsistencies which may cause that the system behave incorrectly. Subsection 4.2.2 presents a diagnostic technique to detect coherency faults in multi-core architectures.

3.1.3 Interconnection Management Unit

The interconnection management unit is responsible for managing the interconnection between the cores, peripherals, memories and the PL of the COTS multi-core devices. In multi-core architectures, as shown in Figure 4 and Figure 5, the interconnection management unit is referred to as the SCU and the CoreNet, respectively. These units manage the communication between the most remarkable components of the devices, where a failure of these units may lead to the occurrence that the data will not be spread over the device and its direct consequence, the possible failure of the system. Subsection 4.2.3 presents a diagnostic technique to detect the failures in the interconnection management unit.

3.1.4 Interrupt Controller

The interrupt controller is a key component that manages the prioritization of the tasks in today's COTS multi-core devices. Figure 7 presents the interrupt controller unit of the ZYNQ 7000 device. This unit, which is also called such as the generic interrupt controller (GIC) [37] is composed of a single distributed block and [1:N] core interface blocks. The number of the GIC's distributed blocks is dependent of the amount of the device's cores.

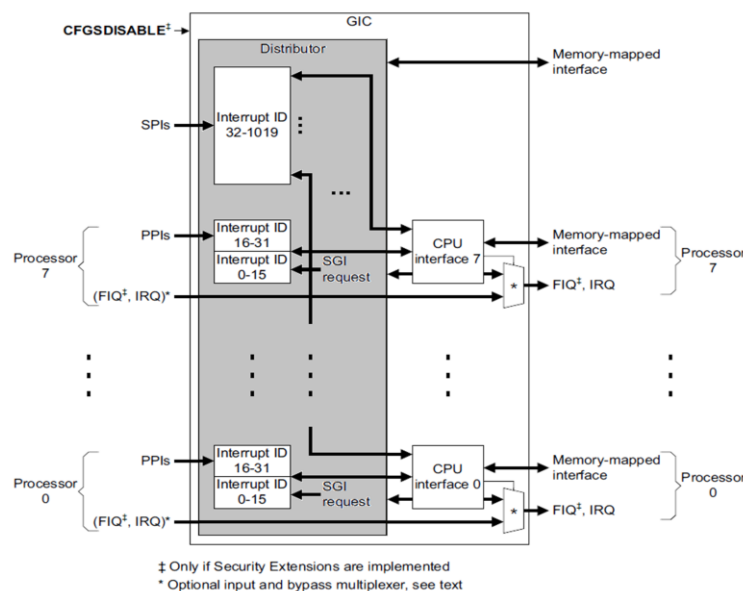


Figure 7: Generic Interrupt Controller (GIC). (Source [37]).

The *interrupt distributor* centralizes the sources of interrupts before dispatching the one with the highest priority level to an individual core. The interrupt controller ensures that an interrupt

targeted to several cores can only be taken by one core at a time. The sources of interrupts are identified by a unique interrupt ID number, a configurable priority and a list of the cores which are targeted. The interrupts that can be handled by the interrupt controller can be originated in cores (private peripheral interrupts (PPIs)), the PL and the PS (Shared Peripheral Interrupts (SPIs)) and the PS (SW Generated Interrupts (SGIs)). On the other hand, the *core interfaces* perform the interrupt priority masking and preemption handling for the cores of the device. Each core interface block provides an interface for each processor that operates within the GIC. In the case that the security extension is implemented by the core interface, the IRQ (non-secure) and FIQ (secure) signals may be used. In addition, the write protection lock mechanism is also provided by the GIC for preventing unauthorized accesses to the critical configuration registers.

3.1.5 Programmable Logic

In Section 3 the major features of networked and partitioned multi-core system architecture are presented. As shown in Figure 2, this system architecture is composed by a COTS device with a processing system (PS) and a programmable logic (PL). In addition, this system architecture is partitioned by a hypervisor to enable the implementation of a wide variety of functionalities with different criticality on the same embedded computing platform. These partitions can be integrated indistinctly in the PS or the PL. In the PS, the partitions may be implemented into the cores of the device, whereas the PL supports the implementation of a single or multiple soft-core processors with independent memory (e.g., BRAM). These soft-core processors and the PL itself can be vulnerable to single event upsets (SEUs). SEUs are usually caused by the collision of cosmic particles and atoms and they may result in the failure of the BRAM, the electronic devices or the PL. Xilinx provides a solution to these failures, providing the soft error mitigation (SEM) IP core [38] for detecting and correcting SEUs in configuration memory of Xilinx FPGAs. SEM IP does not prevent soft-errors; however, it provides methods to manage their effects at system level.

3.2 Hypervisor

A hypervisor is a layer of SW or a combination of SW and HW that allows running several independent execution environments, also called partitions, in a single embedded computing platform. Partitions are logical divisions of memory with static or dynamic cycle and execution time. They can have assigned one or more peripherals and can be developed for different level of criticality (e.g., SIL1 to SIL4 according to the IEC 61508 safety standard). The implementation of a hypervisor for partitioning the system can give rise to spatial independence, temporal independence and real-time constraints.

The broad trend of partitioning the system into different execution environments where functionalities with different criticality can be implemented requires from inter-partition communication. In multi-core architectures, the partitions can communicate through shared memories such as the L2 cache or by using network-on-chip (NoC) communication media systems. The implementation of NoCs for communicating the partitions avoids the challenges caused by the implementation of the shared memories. However, it increments the complexity of the system, and implies several challenges which must be managed to achieve compliance with a safety-related standard (e.g., IEC 61508).

On the other hand, the measures and diagnosis techniques recommended by the current safety-related standards such as the IEC 61508 safety standard are not fully applicable to today's mixed-criticality systems that consist of multi-core architectures, hypervisors, partitions and mixed-criticality networks. The major reason for that is that these standards are geared to single-core architectures where a resource (e.g., memory) cannot be shared between more than one component. Instead, in multi-core architectures a resource is usually shared between more than one element. Therefore, during this deliverable it is assumed that there are some scenarios where the

recommendations, measures and diagnostic techniques of these standards are not applicable to mixed-criticality systems that include COTS devices, virtualization mechanisms, mixed-criticality networks and etc. For example, diagnose that a mixed-criticality network, which is implemented over a partitioned mixed-criticality system, cannot access to the critical memory areas of the partitions. Or diagnose that the memory areas of critical partitions are not accessed or cannot be modified by non-safety-related partitions. In Subsection 4.1.1 it is defined a diagnostic technique to detect whether the hypervisor or attached partitions access to the memory area of the mixed-criticality network and Subsection 4.1.2 presents the critical partition diagnostic pattern.

On the other hand, in mixed-criticality systems there are resources which are implemented very frequently to perform safety and non-safety-related activities. These resources can be managed by safety and non-safety-related partitions to provide partition based centralized solutions which enable reducing the implementation time and allows reusability. Section 4.1.3 defines the centralized partition for digital I/Os (DIOS) while Section 4.1.4 defines the centralized partition for I/O communication.

3.3 Mixed-Criticality Network

Mixed-criticality networks are commonly used bespoke solutions to the traditional legacy fieldbuses due to their benefits in terms of low-cost, high-speed, higher bandwidth and easy integration within network infrastructures. They provide communication between devices and communicate the functionalities with different-criticality of mixed-criticality architectures. In the latter case, they manage the traffic with different criticality levels such as the time-triggered (TT), rate-constrained (RC) and the best-effort (BE) traffic. However, today's mixed-criticality network does not support the simultaneous use of the three traffic classes. For example, TTNoC network does not support the transmission of RC and BE messages and AEtherreal NoC does not support the transmission of RC messages.

Section 4.3 defines a generic cross-domain pattern for managing traffic with different priority and criticality levels and support mixed-criticality systems and hard real-time applications.

4 Cross-Domain Mixed Criticality Patterns

This section contains the definition, implementation and results of remarkable cross-domain mixed-criticality patterns which are identified in Section 3. These patterns are defined following the pattern representation defined in Subsection 2.6 and they aim to provide solutions that solve commonly occurring problems in the development of mixed-criticality systems based on virtualization mechanisms, COTS devices, functionalities with different criticality level and mixed-criticality networks.

The cross-domain patterns that are analysed, defined and implement (not all) during this section are the following:

- *Hypervisor*
 - NoC accessible critical memory area diagnosis (see Subsection 4.1.1). *Implemented*
 - Critical partition diagnosis (see Subsection 4.1.2). *Implemented*
 - Digital I/O Server (see Subsection 4.1.3). *Implemented*
 - Communication Server (see Subsection 4.1.4).
- *COTS device*
 - Shared memory diagnosis (see Subsection 4.2.1). *Implemented*
 - Cache Coherency diagnosis (see Subsection 4.2.2). *Implemented*
 - Inter-connection management unit diagnosis (see Subsection 0).
 - Interrupt Controller diagnosis (see Subsection 0).
- *Mixed-criticality network*
 - Priority based NoC (see Subsection 4.3.1). *Implemented*

4.1 Hypervisor

4.1.1 NoC Accessible Critical Memory Area Diagnosis Pattern

Pattern ID:		PAT – NACMAD – 00
Pattern Name:	NACMAD	
Related pattern:		PAT – CPD – 00
Type:		HW / SW
Context:		
<p>The typical interconnection between two functionally independent systems inside a chip is usually done through memory area registers. The registers are accessed to communicate or interchange data. By construction, registers are portions of continuous memory that can be accessed and modified (write/read) by any on-chip subsystem which is physically connected to the memory area. Critical memory areas shall be protected in a way that non authorized agents could not read nor write data.</p>		
Problem:		
<p>In multi-core mixed criticality systems, network-on-chip (NoCs) are widely implemented communication systems to avoid point-to-point (P2P) individual communication paths between the components of the system and enable the creation of logic paths to interchange data. On-chip networks may access to critical memory areas in use by the components that compose a system. Non authorized memory accesses of a NoC may imply errors that could jeopardize the safety of the system. The most significant impact of the memory access that can be performed by a NoC communication system is the breaking of the temporal isolation. Furthermore, the temporal isolation can also be endangered in the case that the amount of traffic in the NoC is so high that the incoming/outgoing data transfers among the components of the network (e.g., buffers, I/Os, processing cores) are delayed.</p>		
Solution under consideration:		
<p>The solutions proposed below aim to provide robust measures and diagnosis techniques to detect failures in the critical memory areas which are accessible by the NoC communication media systems. The following design solutions can be considered depending on the hardware available.</p> <ul style="list-style-type: none"> Provide complete hardware isolation of the NoC subsystem <p>This solution proposes the assignment of a dedicated memory to the NoC for incoming/outgoing message buffers and for its internal operations. For that purpose, in order to achieve a complete isolation from the processing cores, the NoC shall not be attached to the same bus as the processing cores. This solution scheme can be implemented by means of a dual-port RAM memory, where one port is accessible by the NoC and the other port is connected to the bus where the processing cores are connected.</p> I/O Memory Management Unit (MMU) <p>MMU controls the access of direct memory access (DMA) transfers programmed by the bus-master capable I/O devices. Consequently, the DMA transfers do not overwrite or read from the restricted memory addresses. In the case of safety critical systems, the memory addresses may contain the code and data of safety critical tasks. An I/O MMU enforces the spatial isolation and avoids the overwritten of the safety sensitive memory regions by the NoC.</p> Additional monitoring mechanisms <p>Additional monitoring mechanisms are described in the next design pattern 'PAT-CPD-00'.</p> 		

Board Name:	ZYNQ zc706
Implementation:	
<ul style="list-style-type: none"> Provide complete hardware isolation of the NoC subsystem <p>The NoC communication networks are a passive element that waits for the processor to initiate the data transfer. However, the NoC implementation of the harmonized platform does not wait for the processor for transferring data and therefore, we cannot consider that isolation exists at hardware level. In this particular case, due to limitations of the hardware, it is not possible to implement this solution.</p> I/O Memory Management Unit (MMU) <p>No I/O MMU is available in the current harmonized platform. Therefore the implementation of this cross-domain pattern is not applicable. However, this solution pattern could be implemented in other hardware architectures that support I/O MMU.</p> Additional monitoring mechanisms <p>The implementation of the monitoring mechanisms discussed in the next cross-domain pattern (PAT-CPD-00) can be also implemented for NoC-related diagnostic purposes as follows:</p> <ul style="list-style-type: none"> (a) The detection of temporal interference caused by the NoC when accessing the bus can be considered as equivalent to the interference due to the contention of the bus access which is caused by the competing cores (<i>Limit the concurrency</i>). In the case of NoC communication systems, an estimation of the interference can be calculated based on the expected amount of traffic. (b) The detection of write access from the NoC to critical memory areas. They can be seen as similar to failures in the spatial isolation due to errors in the hypervisor in the sense that the perceived effect is the same. <p>Therefore, the implementation and results provided in the next pattern (PAT-CPD-00) are considered as representative results of the current pattern.</p>	
Results:	
See PAT- CPD- 00.	
Additional Considerations:	
This cross-domain pattern defines a diagnosis technique that it is related to the safety arguments of the modular safety case for an IEC 61058 compliant generic mixed-criticality network [5] and hypervisor [6].	
References:	
DREAMS, "Distributed Real-Time Architecture for Mixed-Criticality Systems - A Modular Safety Case for Mixed-Criticality Network," in <i>D5.1.3</i> , ed, 2015. DREAMS, "Distributed Real-Time Architecture for Mixed-Criticality Systems - A Modular Safety Case for Hypervisor," in <i>D5.1.1</i> , ed, 2015.	

4.1.2 Critical Partition Diagnosis Pattern

Pattern ID:		PAT – CPD – 00
Pattern Name:	CPD	
Related pattern:		N/A
Type:		SW
Context:		
Partitioned mixed-criticality architecture limits the impact of changes, provides reusability of their parts and reduces the complexity of the system. Partitions can be designed, developed and certified individually with different levels of criticality (e.g., SIL1 – SIL4 according to IEC 61508). If a partition contains a safety critical function that it is considered critical by the system, it should be protected against interferences (temporal and spatial) caused by other partitions.		
Problem:		
When dealing with partitioned systems with different criticality, failures caused by the interchange of information are quite probable. The lower criticality functionalities can lead to interferences on the higher criticality functionalities. Therefore, it must be guaranteed that partitions with different criticality level do not influence each other. Two possible sources of interferences can be considered: <ul style="list-style-type: none">I) Temporal interferences generated by multiple accesses in parallel to the shared memory (e.g., by the cores of a multi-core device). The concurrent accesses will compete for accessing to the shared memory cache, which will lead to interferences in temporal domain.II) Failures in the spatial isolation provided by the hypervisor. It is also found in mono-core architectures.		
Solution under consideration:		
This pattern aims to provide a generic diagnosis pattern to detect interferences on critical partitions. It provides a scalable set of measures and diagnosis techniques to detect and control failures of critical partitions and guarantee the system’s temporal and spatial independences. <ul style="list-style-type: none">Limit the concurrency<p>This cross-domain pattern proposes a solution to limit the amount of inter-core temporal interferences. The basic idea is that critical tasks are executed without concurrency. Therefore, when a critical task is running in a certain core the other cores do idle only for the duration of this task, thus avoiding contention. The limitation of concurrency can be achieved by appropriately configuring the partition execution windows for all the cores at design time.</p><p>The loss in performance can be leveraged by tuning the amount of time that a core (running a critical task) executes without concurrency. The maximum amount of interference suffered by one core due to accesses to shared memory and the bus bandwidth used by the other cores can be calculated by means of off-line analyses. The concurrent execution can be guaranteed up to a certain safe time limit based on the temporal constraints of the safety critical tasks and the maximum amount of interferences.</p>Assess the spatial and temporal isolation<p>The following solutions define a way to diagnose the <i>spatial</i> and <i>temporal isolation</i>.<ul style="list-style-type: none"><u>Spatial isolation</u><p>A monitoring mechanisms or a diagnosis partition can be implemented to periodically check the data of the critical memory areas, including the hypervisor’s code and the code and data of</p></p>		

partitions. Checksum and similar mechanism recommended by the IEC 61508 safety standard are perfect candidates to ensure that no accidental modification of code or data takes place. These measures and diagnostic techniques can be individually implemented by the partitions for checking their own code and data. However, the code of XtratuM shall be checked by at least one partition to ensure that is not unexpectedly modified. It shall be taken into account that the hypervisor takes advantage of memory management unit (MMU) hardware to enforce the spatial isolation and that this proposed solution intends to diagnose errors in the configuration of the MMU which can be caused by errors in the hypervisor or the hardware.

- Temporal isolation

The measures and techniques recommended by the IEC 61508 safety standard can also be implemented to diagnose the temporal isolation. For instance, the “Program temporal sequence monitoring” technique (see Section A.9 of IEC61508-7 [39]) can be implemented to monitor the execution of safety critical tasks in terms of temporal response and to ensure that the temporal isolation is not compromised due to a failure of the hypervisor. This proposed solution aims to diagnose failures in the hypervisor or in the configuration of the partition execution windows that may jeopardise the temporal isolation of the partitions. Therefore, the task/process scheduling inside the partitions is out of the scope of this solution.

On the other hand, in deliverables D2.2.2 [40] and D2.3.4 [41] the deadline overrun and QoS services are defined. These services improve the isolation execution of critical applications by introducing internal deadline monitoring and interrupt best effort applications and they are applicable for mixed-criticality systems with partitioning.

Board Name:

N/A

Implementation:

This section presents the implementation of the solutions defined before in the XtratuM hypervisor.

- **Limit the concurrency**

This implementation aims to determine the worst-case conditions that a partitioned system can suffer. For that purpose, this implementation scenario implements the model of inter-core interferences when partitions in different cores access shared memory (e.g., DDR). This model is available in [40].

The worst case scenarios that can be considered are:

- Two partitions access simultaneously to the shared DDR memory from different cores.
- The shared DDR memory regions are configured as non-cacheable.
- The percentage of overlapping between the partition slots is 100%.

In the above worst case scenario, the measured performance decreases to a 69% [40]. This overhead is defined as the maximum temporal interference that can suffer any partition, including any critical partition. Therefore, overhead values below 69% are detected as unsafe.

On the other hand, when the counting performance decreases below the 69% threshold, which means that it goes from the nominal value of 391618 counts per 100ms with no concurrent access to the DDR memory and down to 270216 per 100ms when there is complete concurrent access to the DDR memory, a safety action is executed, issuing a warning message.

- **Assess the spatial and temporal isolation**

- Spatial isolation

This partition diagnosis pattern implements memory checksum technique (IEC61508-7 [42]) to detect modifications of the critical memory regions of the system. This implementation defines non-cacheable critical memory regions where the contents of these memory areas cannot be

modified during the system execution. For demonstration purposes, this implementation selects the text code sections of XtratuM hypervisor and critical partitions as the critical memory areas which are protected by means of a checksum.

- I) The hypervisor's text code is contained between `[_sdata, _spdata]` symbols of the XtratuM hypervisor's executable and linkable format (.ELF) and corresponds to physical addresses `[0x20000000, 0x2001d3b8]`. Since the memory area where the hypervisor resides is protected from partitions by means of the *MMU*, the diagnosis partition that computes the checksum shall have system management rights in the XtratuM configuration file (XMCF) to have to read-only access to the hypervisor's text code and to compute the checksum technique.
- II) The critical partition's text code is contained between `[_sguest, _sdata]` symbols of the partition ELF and corresponds to physical addresses `[0x10100000, 0x101036c8]`. Since the critical partition memory can reside in a separate partition, the diagnosis partition shall have read-only access to the critical partition memory configured in the XMCF.

The critical memory regions defined above are protected by means of a *Cyclic Redundancy Check (CRC32)* checksum IEEE 802.3. The CRC32 is periodically checked each 1000ms, at the start of each critical partition temporal slot. The frequency of the check is configured through the partition scheduling period, which is defined in the XtratuM Configuration File (XMCF).

Note that the selection of the CRC32 algorithm has been performed for demonstration purposes. In operational systems the checksum algorithm shall be determined depending on the size of memory that is required to be protected and the expected error rate value of the operational hardware.

○ Temporal isolation

The partition diagnosis pattern implements the temporal sequence monitoring technique (IEC61508-7 [42]) to detect temporal errors. This technique is implemented by XtratuM hypervisor. Specifically, the critical partition configures a virtual interrupt (extended interrupt in the XtratuM terminology) that is triggered at the beginning of the execution slot of each partition (`XM_VT_EXT_CYCLIC_SLOT_START`) and executes the following measures and diagnoses [41].

- Check that the partition slot starts according to the scheduling plan

This diagnosis partition check is implemented evaluating the invariant

$$|currentSlotStart - expectedSlotStart| < CFG_SCHDDRIFT_THRESHOLD$$

where

- *currentSlotStart* is the measured time when the slot started by issuing an `XM_get_time(XM_HW_CLOCK, ¤tSlotStart)`.
- *expectedSlotStart* is the expected time when the partition slot should start. This value is defined off-line at system design time and it is used to configure the partition execution windows in the XMCF.
- *CFG_SCHDDRIFT_THRESHOLD* is the maximum scheduling drift that is considered safe under nominal operation conditions. For demonstration purposes, this value is set to 100us. Therefore, if a value above 100us is detected, it is considered that a temporal interference occurs, which leads to the execution of a safety action that consists in emitting a warning of the scheduling drift.

- Check that the partition slot period is according to the scheduling plan

This diagnosis partition check is implemented evaluating the invariant

$$|currentPeriod - expectedPeriod| < CFG_SCHDDRIFT_THRESHOLD$$

where

- *currentPeriod* is the measured period of partitions that is computed as the difference between the current and the previous slot start times.
- *expectedPeriod* is the expected period of partitions that is computed from the partition scheduling plan defined off-line in the XMCF.
- *CFG_SCHEDDRIFT_THRESHOLD* is the maximum scheduling drift (100us).

Results:

- **Limit the concurrency**

- Temporal interference

The detection capabilities of the concurrency monitoring implementation are validated on-line by means of a scenario where a faulty partition progressively causes an increasing amount of temporal interferences. Validation refers to the fact that it is ensured by means of test cases that the interference is indeed detected, thus generating the following exception:

«[P0-readerPSM.c:45] DetectMulticoreInterference counter 270414 below 270216 threshold»

These interferences are sourced by accessing the shared un-cached DDR memory area. The amount of interference is controlled by means of different scheduling plans which cause an increasing percentage of temporal overlapping between the critical partition (P0) and the fault partition (P2). The temporal overlapping percentage takes the values: 0%, 20%, 40%, 60%, 80%, and 100%.

This diagnosis scenario is setup according to the worst case scenarios defined in the implementation section, where two partitions accesses to a matrix located in the non-cacheable shared DDR memory [40]. However, it does not define how to avoid the impairment of the safety function. For that purpose, the partition developer can make use of the services provided by the hypervisor such as stopping the interfering partition. This service can be achieved by using the hypervisor's *XM_suspend_partition()* hypercall.

Trace of the execution of the critical partition :

```
Plan 0: Overlap 0
Counter: 391532  391359

Plan 1: Overlap 20
Counter: 367537  367373

Plan 2: Overlap 40
Counter: 343291  343103

Plan 3: Overlap 60
Counter: 319001  318837

Plan 4: Overlap 80
Counter: 294698  294539

Plan 5: Overlap 100
Counter: 270414  270407
```

[P0-readerPSM.c:45] DetectMulticoreInterference counter 270414 below 270216 threshold

First, the two partitions execute according to the scheduling plan 0 that has 0% temporal overlapping, in this case the counter values take their maximum nominal value. Next, the plan is changed to scheduling plan 1 that causes a 20% temporal overlap that causes small performance degradation where performance drops from 100% to 94%. When the two partitions are executed according to the scheduling plan 5, then temporal slots overlap completely as shown in the code listing (line Plan 5: Overlap 100) then the critical partition (P0) detects that the access times to the shared DDR memory have increased above the maximum threshold defined in the model and issues

a safety warning message.

- **Assess the spatial and temporal isolation**

- Spatial isolation

The detection capabilities of the memory checksum implementation are validated at run-time by means of a scenario where a faulty partition randomly injects memory errors in the critical memory sections. These errors are injected at random instants in time and are randomly distributed over the whole critical memory regions. This diagnosis ensures by means of test cases that the breach in spatial isolation is indeed detected, thus generating the following exception.

«[P0-critical.c:49] ChecksumDetect mismatch computed 204014F expected FCA9BE35»

This diagnosis pattern does not define how to avoid the impairment of the hypervisor's safety functions. For such purpose, the partition developer can make use of the services provided by the hypervisor such as redundant partitions to detect the corruption in the partition, bring the system to a safe state and restart the complete system by means of the *XM_reset_hypervisor()* hypercall.

P2 Faulty partition pseudo code:

```

loop
    randTime = select random instant to start injecting memory errors
    if (currentTime == randTime)
        error_addr = select random address from critical memory region
        *error_addr = random 32-bit word
    endif
endloop

```

Execution trace:

```

[P0-critical.c:98] ChecksumInit partition 0x10100000 CRC32 0xFCA9BE35
[P0-critical.c:Execut99] ChecksumInit hypervisor 0x20000000 CRC32 0xE5B846AC
[P2-faulty.c:19] TriggerPartitionCorruption at 0x1010000D = 0
[P0-critical.c:49] ChecksumDetect mismatch computed 204014F expected FCA9BE35
[P2-faulty.c:19] TriggerPartitionCorruption at 10100013 = EA
[P0-critical.c:49] ChecksumDetect mismatch computed 26C06CDF expected FCA9BE35

```

The above listing depicts the execution trace of partitions' spatial isolation assessment scenario, where, first, the critical partition (P0) computes the initial *CRC32* checksum during the initialization phase.

- The first *ChecksumInit* computes the *CRC32* checksum of the critical partition memory region starting at 0x10100000.
- The second *ChecksumInit* computes the *CRC32* checksum of the hypervisor memory region starting at 0x20000000.

Then, the faulty partition (P2) starts its execution and issues a *TriggerPartitionCorruption()* at a random address 0x1010000D in the operation phase. Hereafter, line shows the critical partition issues the periodic checksum verification and issues the *ChecksumDetect()* function that detects the mismatch between the current checksum and the initial checksum, and issues a warning that alerts that the memory region has been modified.

- Temporal isolation

The detection capabilities of this solution are validated on-line by means of a scenario where a faulty partition causes random temporal interferences to the critical partition. Validation refers to the fact that we guarantee by means of test cases that the interference is indeed detected, thus generating the following exception:

«[P0-critical.c:45] SlotStart drift detected SlotStart 2000607 expected 2000000»

A faulty partition (P2) is used to simulate a failure in the spatial isolation provided by the hypervisor. This approach waits to the end of the faulty partition slot and issues a long XtratuM hypercall that is not completed in the remaining slot time. This event forces the time consumption of the next partition by the hypercall (XtratuM hypercalls are not pre-emptive and therefore XtratuM cannot perform a partition context switch until the hypercall has finished). However, this diagnosis pattern does not define how to avoid that the impairment of the safety function. For such purpose, the partition developer can make use of the services provided by the hypervisor, like for example stopping the interfering partition using the hypervisor's *XM_suspend_partition()* hypercall.

P2 Faulty partition pseudo code

```

loop
  randTime = select random instant to start injecting memory errors
  if (currentTime == randTime)
    wait for the end of the current partition slot
    issue a long hypercall to interfere with next slot
  endif
endloop

```

Execution trace

```

[P0-critical.c:65] WatchdogInit
[P2-faulty.c:58] TriggerTemporalInterference 2000000
[P0-critical.c:45] SlotStart drift detected slotStart 2000607 expected 2000000
[P2-faulty.c:58] TriggerTemporalInterference 4000000
[P0-critical.c:45] SlotStart drift detected slotStart 4000607 expected 4000000

```

First, during the initialisation phase, the critical partition issues a call to the *WatchdogInit()* service that sets up an extended interrupt to be received at the start of each partition slot as detailed in the implementation section. Then, the faulty partition (P2) issues a *TriggerTemporalInterference* call. This call waits for the end of the current slot and issues a long hypercall. As a result, the start of the slot of the critical partition is delayed by 607 us. When the critical partition starts executing, it measures the current *slotStart* time and checks if it differs from the expected *slotStart* computed from the *XMCF*. Since the expected *slotStart* is 2000000us and the measures *slotStart* is 20000607us, a 607us difference is detected which is greater than the 100us (*CFG_SCHEDDRIFT_THRESHOLD*) defined in the implementation section, and causes the critical partition to emit a warning that a scheduling drift has been detected.

Additional Considerations:

This cross-domain pattern defines a diagnosis technique that it is related to the safety arguments of the modular safety case for an IEC 61058 compliant generic hypervisor and partition [6].

References:

DREAMS, "Distributed Real-Time Architecture for Mixed-Criticality Systems - A Modular Safety Case for Hypervisor," in *D5.1.1*, ed, 2015.

DREAMS, "Distributed Real-Time Architecture for Mixed-Criticality Systems - Hypervisor adaptation and drivers for local resource manager," in *D2.3.4*, ed, 2016, p. 56.

DREAMS, "Distributed Real-Time Architecture for Mixed-Criticality Systems - Report on monitoring, local resource scheduling and re-configuration services for mixed criticality and security with implementation (source code) of low- and high-level monitors, scheduling, security and re-configuration services supporting mixed criticality and adaptation," in *D2.2.2*, ed, 2015, p. 46.

4.1.3 Digital I/O Server Pattern

Pattern ID:		PAT –DIOS – 00
Pattern Name:	DIOS	
Related pattern:	N/A	
Type:	HW/SW	
Context:		
Digital I/Os are widely used among different system architectures for communication purposes. They can be managed by partitions with different criticality (e.g., safety, non-safety and real-time) but not at the same time. The simultaneous access to a digital I/O by more than one partition usually leads to an error that jeopardizes the system.		
Problem:		
Partitions with different criticality level usually require commanding digital I/Os. This is a common requirement among different system architectures, where a digital I/O can be requested by more than one partition at the same time, thus causing a conflict that could lead to an error. The DIOs may be corrupted physically or in a register level which could cause a failure in safety-related subsystem. As a general rule, a digital I/O cannot be assigned to more than one component, unless a voting mechanisms or equivalent is used. On the other hand, from a product line perspective, the number DIOs which may be requested by a product might change, leading to scalability problems.		
Solution under consideration:		
The proposed solution in this section is based on the implementation of a Digital I/O server partition, which manages the digital I/Os of the mixed-criticality system. The Digital I/O Server (DIOS) is a consistent concurrent manager of digital I/Os that is abstracted from platform and hypervisor details to assure reusability, enabling its integration on different system architectures without major changes, simplifying the system design. In addition, it includes a set of measures and diagnostic techniques to assess random and systematic failures. Figure 8 shows an example of the digital I/O server cross-domain pattern which is integrated on a partitioned multi-core architecture and manages the requests for digital I/Os from partitions with different criticality level (e.g., safety and non-safety).		
<div><div><div>Digital I/Os</div><div>CPU 0</div><div><div>Communication Server</div><div>Safety Partition</div></div></div><div>CPU 1</div><div><div>Non-Safety Partition</div><div>Safety Partition</div><div>Non-Safety Partition</div></div></div>		
Figure 8: DIOS cross-domain pattern - Example.		

The digital I/O server periodically updates the values of the inputs to refresh the information of the partitions where the inputs are required. Afterwards, the diagnostic techniques explained in the following paragraphs are executed, so that, if a failure is detected, the outputs will be refreshed with the safe value instead of with the value provided by the partitions. In the case that different partitions try to update the same output with different values, the partitions will be moved to a safe state and the outputs will be updated with their default value.

The conditions, measures and diagnosis techniques which are implemented by the digital I/O server partition to assess that the digital I/Os controlled by safety-related partitions don't cause a failure that can affect to other partitions are the following:

- The safety-related outputs have associated inputs with the same or opposite values. The values vary depending on the configuration.
- The cyclic redundancy codes (CRCs) of the values of the registers associated to the digital inputs and outputs are periodically compared against the values already stored by the digital I/O server. The comparison period is determined by the minimum refreshing period of the digital outputs.
- Check that the partitions in charge for updating the digital outputs refresh the values of DIOS partition. For that purpose, this solution implements a token that is updated every time that the communication is refreshed, always agreeing with the expected values in DIOS. This solution may also be applied in the remaining partitions, but with the inputs to assure that the communication among partitions and DIOS keeps working.
- Every time that the values of the digital outputs change, shall be checked that the register values match with the values supported by the DIO Server.
- Each digital input shall be checked to detect whether their values are able to be changed. These checks shall be executed under a pre-configured timeout. If the timeout value is not specified, the default value will be used (a month) and the developer will be forced to integrate an output to change the values of the inputs in a controlled non-safety way for testing purposes.

In safety-related applications, the configuration of the digital inputs and outputs of the server like the configuration of partitions shall be established by means of off-line qualified tools. This pattern relies on the safety-related arguments (e.g., resource virtualization, exclusive access to peripherals) defined in the modular safety case (MSC) for an IEC 61508 compliant generic hypervisor [6] and the safe communication between partitions provided by a IEC-61508 compliant hypervisor.

Board Name:

XILINX ZYNQ-7000 zc702

Implementation:

In order to test the proposed solution pattern, the use case shown in Figure 9 composed of two safety-related partitions (both of them execute the same code), two non-safety related partitions and a digital I/O server partition is implemented. The digital I/O server implements three inputs and two outputs (one of them safety) for communicating purposes.

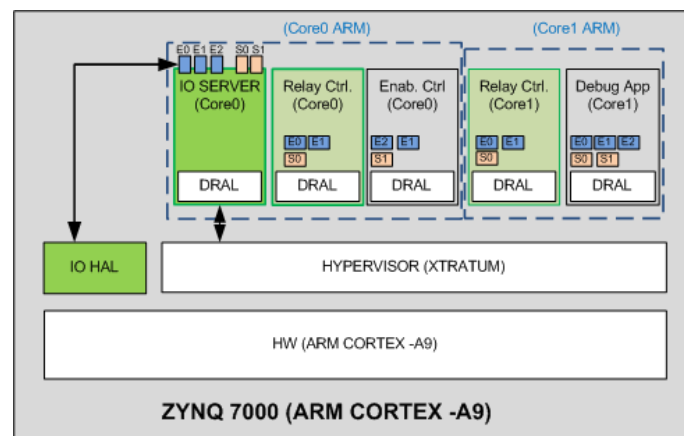


Figure 9: DIOS - Implementation architecture.

The first step of the implementation is to define the *configuration file*. The configuration file enables managing a configurable number of digital I/Os, thus assuring the scalability of the system. In addition, during this section it is assumed that the digital I/Os shall include the following set of variables for diagnosis purpose.

- Digital Inputs (DI)

- *Name*: Identification of the digital input.

For continuous mode inputs, where the state is changed frequently (similar like frequency or like a hard-beat):

- *Change Frequency*: This parameter describes how often the inputs have to change.
 - *Timeout*: The timeout before considering that an input is faulty will be defined with this parameter and depends on its application.
 - *Configuration Unique ID*: This is an identification to associate in the HAL each input to a concrete hardware input.
 - *Partition and sampling frequency*: The identification of the partitions that requires that input and the frequency with what they need the updated value.

High demand and low demand inputs are tested using test-pulses generated by a test-pulse generator. The diagnostic circuit sends a test-pulse to the test-pulse generator reads the test-pulse via the input and checks whether the test-pulse has been detected in the right time.

- *Test-pulse period*: This parameter defines the period of testing the input e.g. once a hour.
 - *Test-pulse duration*: This parameter defines the width of the test-pulse. The width depends on load assignment of the wiring.

- Digital Outputs (DO)

- *Name*: Identification of the digital output.
 - *Configuration Unique ID*: This is an identification to associate in the hardware abstraction layer (HAL) each output to a concrete hardware output.
 - *Safety Value*: The value to which the output should be updated when the system has to go to the safe state.
 - *Safety Input*: This parameter identifies if the output has an input associated to check that the actual value of the output is the correct one.
 - *Safety Input Logic*: This parameter identifies if the safety input associated to a safety output has the same or opposite value of the actual output.
 - *Safety Input timeout*: This parameter identifies the maximum time required by the safety input to update its value when the output changes. E.g., 8-10ms.
 - *Partition and sampling frequency*: The identification of the partitions which require an update an output and the update's frequency value.

The configuration of this solution is collected in the xml file (*xm_cf.arm.xml*) of XtratuM hypervisor that aims to configure the partitions of the hypervisor. This configuration file is checked by means of the *IOServerGenerator* T3 off-line qualified tool (see IEC 61508-3) that is also responsible for developing the I/O database required by the I/O Server pattern. Figure 10 shows the verification procedure executed in this implementation.

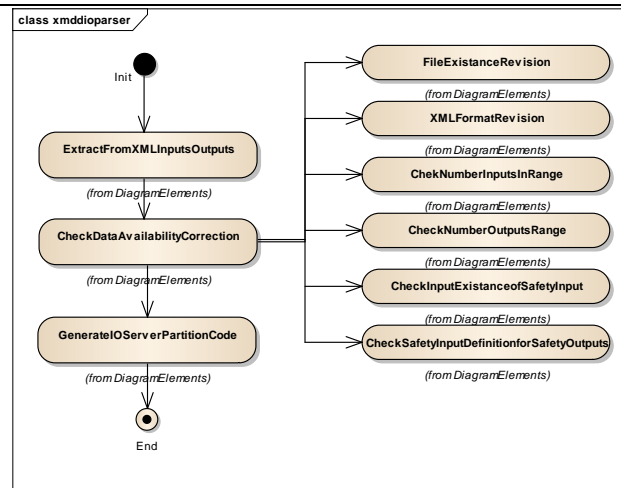


Figure 10: Configuration file - Verification process.

The current implementation of this pattern is based on the concept of associate the digital inputs and outputs to communication ports which may be accessed by the safety-related partitions. Therefore, when defining the configuration file, apart from defining the digital inputs and outputs; their associated ports shall also be defined. In application domain, if the hypervisor's abstraction layer (DRAL) is used, the digital inputs and outputs will be used as ports. The following class diagrams (Figure 11 and Figure 12) show the main blocks that compose the system architecture implemented in this section and the content of the digital I/O server.

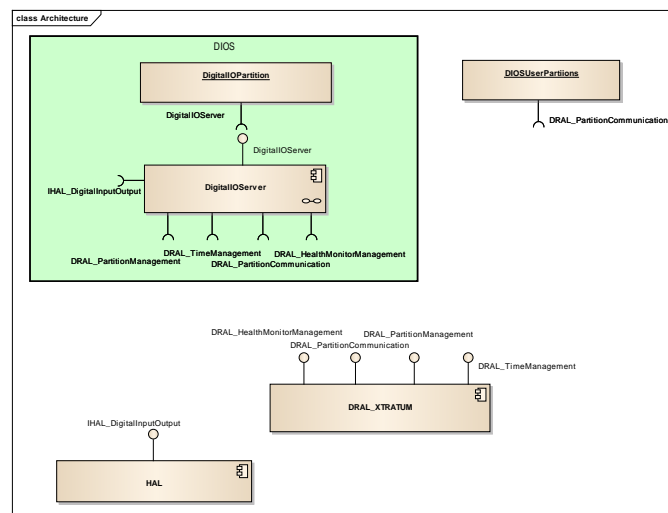


Figure 11: DIOS System architecture - Overview.

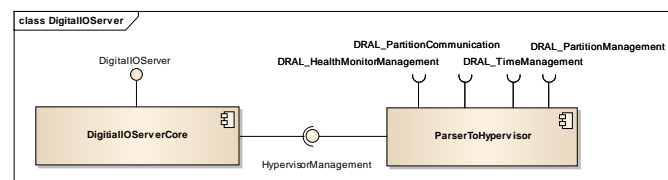


Figure 12: Digital I/O Server - Overview.

The following diagram presents the interfaces which are used for implementing the digital I/O serve.

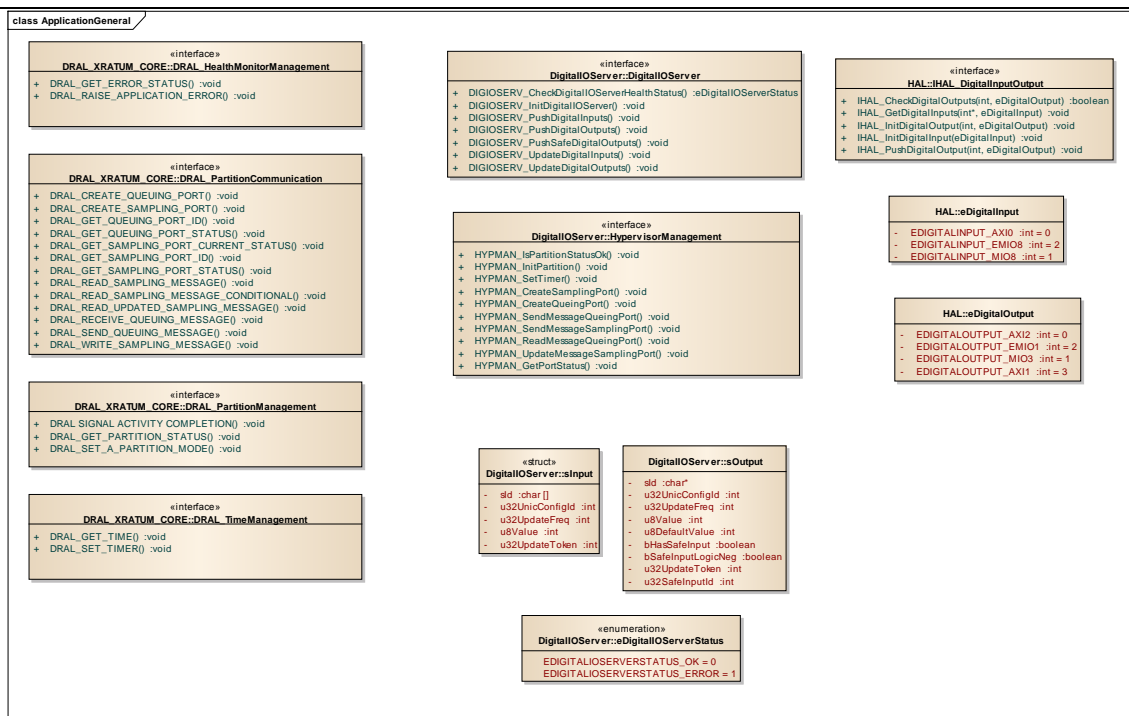


Figure 13: DIOS Interfaces - Overview.

The digital I/O server (DIOS) and the digital I/O partitions defined in this pattern are generic solutions which may be implemented in different system architectures without major changes. The *DigitalInputOutput* class shall be updated according the characteristics of the HW platform and/or hypervisor where this solution is implemented.

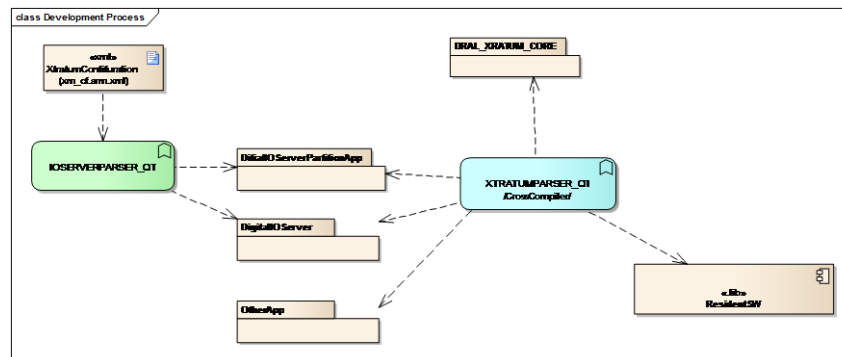


Figure 14: Development process of DIOS - Overview.

Results:

The following screenshots summarize the satisfactory results of this pattern's implementation. The test cases which are executed to validate this use case consider that everything works correctly. In addition, further test cases are executed to check the fault adaptability of DIOS, where if a fault is detected the digital outputs go to safe state.

```

- <DigitalInputOutput serverPartitionID="0">
-   <DigitalInput>
-     <Input name="e1" ChangeFreq="10000000" configUnitID="1">
-       <Sampling freqUpdate="10" partition="0"/>
-       <Sampling freqUpdate="10" partition="1"/>
-       <Sampling freqUpdate="20" partition="2"/>
-       <Sampling freqUpdate="10" partition="3"/>
-       <Sampling freqUpdate="50" partition="4"/>
-     </Input>
-     <Input name="e2" ChangeFreq="10000000" configUnitID="2">
-       <Sampling freqUpdate="10" partition="0"/>
-       <Sampling freqUpdate="20" partition="2"/>
-       <Sampling freqUpdate="50" partition="4"/>
-     </Input>
-     <Input name="e0" ChangeFreq="10000000" configUnitID="0">
-       <Sampling freqUpdate="10" partition="0"/>
-       <Sampling freqUpdate="10" partition="1"/>
-       <Sampling freqUpdate="10" partition="3"/>
-       <Sampling freqUpdate="50" partition="4"/>
-     </Input>
-   </DigitalInput>
-   <DigitalOutputs>
-     <Output name="s1" configUnitID="1" safetyInputLogic="0" safetyInputTimeOut="" safeValue="1" safetyInput="e2" safety="0">
-       <Sampling freqUpdate="10" partition="0" isOrigin="0"/>
-       <Sampling freqUpdate="20" partition="2" isOrigin="1"/>
-       <Sampling freqUpdate="50" partition="4" isOrigin="0"/>
-     </Output>
-     <Output name="e0" configUnitID="0" safetyInputLogic="0" safetyInputTimeOut="50" safeValue="0" safetyInput="e0" safety="1">
-       <Sampling freqUpdate="10" partition="0" isOrigin="0"/>
-       <Sampling freqUpdate="10" partition="1" isOrigin="1"/>
-       <Sampling freqUpdate="10" partition="3" isOrigin="1"/>
-       <Sampling freqUpdate="50" partition="4" isOrigin="0"/>
-     </Output>
-   </DigitalOutputs>
- </DigitalInputOutput>
-/SystemDescription

```

```

~/opt/xilinx/SDK/2014.4/gnu/arm/lib/bin/arm-xilinx-eabi-gcc -march=armv7-a -mcpu=cortex-a9 -mfpu=vfpv3 -x c -O2 -Wall -I/home/pelo/sdk-arm/xn/user/lib/include -I/home/pelo/xn-sdk-arm/jn/include -nostdlib -nostdinc -include xn_inc/config.h -include xn_lib/arch_types.h a.c.xnc -o objf -Wl,-entry=0x00000000,-fdls81228p
}
command=[/opt/Xilinx/SDK/2014.4/gnu/arm/lib/bin/arm-xilinx-eabi-obcopy -O binary objf xn_cf.bin.xnc
]
~/opt/ubuntu:/xn-sdk-arm/xal-examples/IOS make xn_cf.bin.xnc
/home/pelo/xn-sdk-arm/jn/bln/mcparser -o xn_cf.bin.xnc xn_cf.arm.xml
command=[
/opt/Xilinx/SDK/2014.4/gnu/arm/lib/bin/arm-xilinx-eabi-gcc -march=armv7-a -mcpu=cortex-a9 -mfpu=vfpv3 -x c -O2 -Wall -I/home/pelo/sdk-arm/xn/user/lib/include -I/home/pelo/xn-sdk-arm/jn/include -nostdlib -nostdinc -include xn_inc/config.h -include xn_lib/arch_types.h a.c.xnc -o objf -Wl,-entry=0x00000000,-fdls81228p
]
command=[/opt/Xilinx/SDK/2014.4/gnu/arm/lib/bin/arm-xilinx-eabi-obcopy -O binary objf xn_cf.bin.xnc
]
~/opt/ubuntu:/xn-sdk-arm/xal-examples/IOS make resident_sw
/home/pelo/xn-sdk-arm/jn/bln/xmeformat build -n xn_cf.bin.xnc -c -o xn_cf.xef.xnc
corisa3ff93cab8bf7b9375dc4db xn_cf.xef.xnc
/home/pelo/xn-sdk-arm/jn/bln/smack check xn_cf.xef.xnc -h /home/pelo/xn-sdk-arm/jn/lib/xn_core.xef:xn_cf.xef.xnc -p 0:partition0.xef -p 1:partition1.xef -p 2:partition2.xef -p 3:partition3.xef -p 4:partition4.xef
> /home/pelo/xn-sdk-arm/jn/lib/xn_core.xef ... ok
> partition0.xef ... ok
> partition1.xef ... ok
> partition2.xef ... ok
> partition3.xef ... ok
> partition4.xef ... ok
/home/pelo/xn-sdk-arm/jn/bln/smack build -k /home/pelo/xn-sdk-arm/jn/lib/xn_core.xef:xn_cf.xef.xnc -p 0:partition0.xef -p 1:partition1.xef -p 2:partition2.xef -p 3:partition3.xef -p 4:partition4.xef container.bin
> /home/pelo/xn-sdk-arm/jn/lib/xn_core.xef ... ok
> partition0.xef ... ok
> partition1.xef ... ok
> partition2.xef ... ok
> partition3.xef ... ok
> partition4.xef ... ok
Done [container]
/home/pelo/xn-sdk-arm/jn/bln/rswbuild container.bin resident_sw
Created by "pelo" on "ubuntu" at "Sat Jun 4 14:26:11 CEST 2016"
xm path: "/home/pelo/xn-sdk-arm/jn/"
Xstratum Core:
Version: "2.0.7"
Arch: "arm"
File: "/home/pelo/xn-sdk-arm/jn/lib/xn_core.xef"
Sha1: "d8be30197db401257a3adcf45a57dc4afab3cf1d40bc9"
Changed: ""
Xstratum Library:
Version: "2.0.7"
File: "/home/pelo/xn-sdk-arm/jn/lib/libbm.a"
Sha1: "3cf1d456e728b0b6cf45a57dc4afab3cf1d40bc9"
Changed: ""
Xstratum Tools:
File: "/home/pelo/xn-sdk-arm/jn/bln/mcparser"
Sha1: "97265733b691074551093d7220fa9ec2fafadd"

```

4.1.4 Communication I/O Server Pattern

Pattern ID:		PAT – CIOS – 00
Pattern Name:	CIOS	
Related pattern:		PAT – DIOS – 00
Type:		HW/ SW
Context:		
<p>Safety and non-safety-related partitions of partitioned mixed-criticality systems usually require communicating. For that purpose, multi-core architectures usually implement shared memories and communication media systems. These communication interfaces have its pros and cons. For instance, shared memories are common sources of interferences in architectures with more than one core. On the other hand, in order to overcome issues related to the shared memories, the communication networks are a bespoke solution to provide internal and external communication. In addition, it is assumed in product family domain, that external communication networks may suffer modifications, or may be replaced owing to changes in the</p>		

safety-related requirements of the network. This gives rise to scalability issues. These external communication systems are implemented over a specific architecture design.

Problem:

The communication between partitions with different criticality (e.g., SIL1 to SIL4 in accordance with the IEC 61508 safety standard) can be carried out through on-chip (e.g., STNoC) and off-chip mixed-criticality networks (e.g., TTEthernet) with real-time capabilities or not. For example, EtherCAT is a real-time industrial Ethernet off-chip network. Off-chip networks may be used to connect different devices that may be located far away (physically) from each other. Instead, on-chip networks (NoC) may be implemented for communicating the internal components of the system. For instance, as shown in Figure 15, on-chip networks (NoC) can be implemented for communicating the CPUs of a multi-core mixed-criticality embedded computing system.

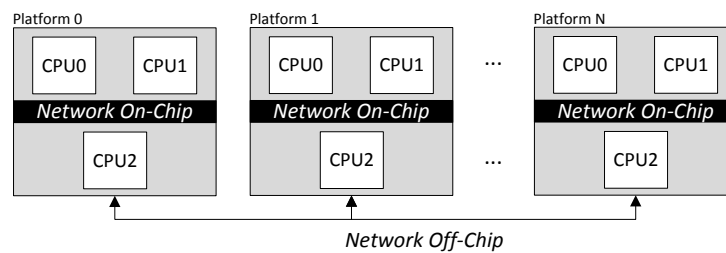


Figure 15: Network on-chip and off-chip.

The NoCs shifts the problems associated with traditional networks into the chip due to their low-cost, high-speed and easy integration with existing networks infrastructures. However, although they provide benefits in terms of spatial and temporal segregation, they lead to certification challenges (e.g., assure temporal independence).

On the other hand, the number of functions of a mixed-criticality product which require to communicate tends to raise, mixing the communication requirements of different criticality (e.g., safety, real-time and security) and hampering the development and certification of mixed-criticality networks. As a result, the underlying mixed-criticality communication media systems require an adaptation process or shall be modified to cover new requirements, leading to higher engineering and certification cost.

Solution under consideration:

The proposed solution focuses on the development of a communication server that simplifies the system design and development, and reduces the cost of certification. The communication server aims at managing the communication between partitions and external elements of the system (see Figure 16).

In addition, it is assumed that this server is logically abstracted from the processor control of the communication network (e.g., using partition ports) and that it manages the assignment of the peripherals to the partitions, implementing the exclusive access to peripherals technique.

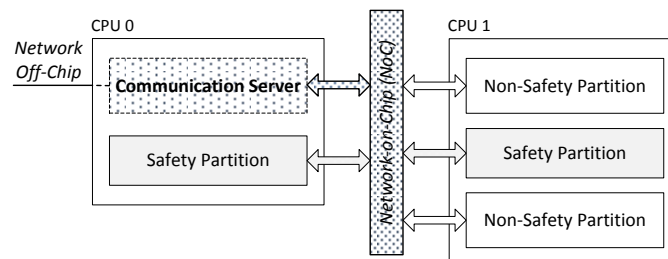


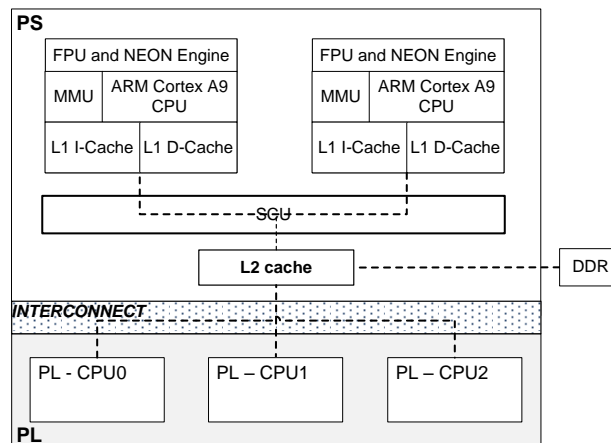
Figure 16: Communication Server - Example.

This communication server shall be compliant to the IEC 61508 safety standard with a SIL up to SIL3 and it shall support *black channel* and *white channel* network approaches [19, 21]. In the case that the communication server manages a white channel network, it should rely on the hypervisor's safety-related functions.

Board Name:	N/A
Implementation:	
N/A	
Results:	
N/A	
Additional Considerations:	
This pattern is related to the modular safety case for an IEC 61508 compliant generic Hypervisor [6] and Mixed-Criticality Network [5].	
References:	
DREAMS, "Distributed Real-Time Architecture for Mixed-Criticality Systems - A Modular Safety Case for Mixed-Criticality Network," in <i>D5.1.3</i> , ed, 2015. DREAMS, "Distributed Real-Time Architecture for Mixed-Criticality Systems - A Modular Safety Case for Hypervisor," in <i>D5.1.1</i> , ed, 2015.	

4.2 COTS processor

4.2.1 Shared Memory Diagnosis Pattern

Pattern ID:		PAT –SMD – 00
Pattern Name:	SMD	
Related pattern:	PAT – CCMU – 00 PAT – ICMUD – 00	
Type:	HW/SW	
Context:		
<p>The transition from conventional federated architectures to integrated architectures enables the integration of functionalities with different levels of criticality (such as safety, security and real-time) on the same embedded computing platform. This trend is supported by the transition from single-core to multi-core and many-core architectures. Multi-core architectures provide benefits in terms of cost, size, weight reduction as well as improved scalability. However, they imply certification challenges, among others; due on their shared resources (e.g., memory and peripherals) which can lead to interferences in general that can influence the behaviour of the safety-related (e.g., in temporal domain).</p>		
Problem:		
<p>The sharing of resources is a habitual implementation in today’s multi-core mixed devices for improving the performance. These resources can be accessed at the same time from multiple components of the device (e.g., cores and soft-core processors) through regular memory operations and requests. These accesses may cause interferences in general that can imply deviations in the behaviour of the system. The IEC 61508 safety standard recommends a set of measures and diagnostic techniques to detect the random failures of variable and invariable memories (see Tables A.5 and A.6 of the IEC 61508-2 [19]). However, these measures and diagnostic techniques are focused on single core architectures where, as a general rule, a resource cannot be accessed by more than one component at the same time. Instead, in multi-core architectures, it is common that a resource (e.g., memory or peripheral) can be accessed by two or more components (e.g., two CPUs) at the same time, which may lead to the failure of the system.</p> <p>In our particular case, the ZYNQ-7000 [16] and the P4080 [15] multi-core COTS devices implements a shared memory as the secondary layer memory for providing communication between their components. For example, as shown in Figure 17, the ZYNQ 7000 device implements the shared memory for communicating the cores with the DDR memory, the PL and etc.</p>		
		
Figure 17: Shared memory - Overview (ZYNQ-7000 device).		
Solution under consideration:		

This pattern aims to provide a generic diagnostic technique to detect failures in the shared memories of multi-core devices. Although the terms used throughout this pattern are exclusive from the ZYNQ device (e.g., SCU and GIC), they can be replaced by the terms used by any device, provided that they follow similar architecture design (e.g., SCU (ZYNQ device) – CoreNet (P4080)). In this section, the following two possible solution approaches to detect, evict and manage the failures of shared memories are defined.

- **Limit the use of shared memories**

Shared memories are implemented in today's multi-core COTS devices for provide communication of the components and improve the performance. However, the use of these memories may lead to temporal and spatial independence issues. For that purpose, this solution proposes to limit as much as possible the use of shared memories and in the case that they are implemented to control the access to them to avoid parallel accesses.

For example, in the ZYNQ-7000 multi-core device, the shared memory can be disabled for avoiding interferences. Instead, as shown in Figure 18, the shared memory can be replaced by a network-on-chip (NoC) communication media system for communicating the components of the device and avoiding interferences caused by the shared memory. NoC networks provide benefits in terms of spatial and temporal segregation.

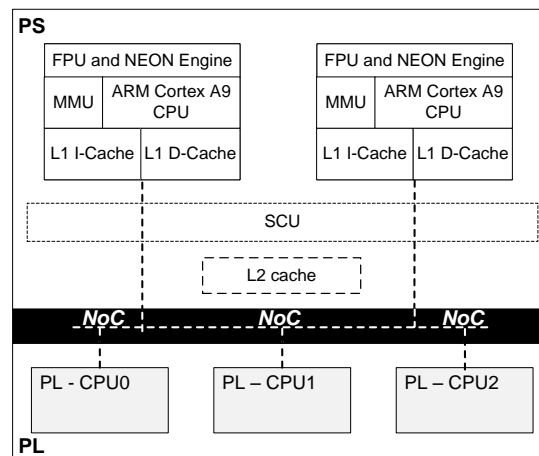


Figure 18: Shared memory diagnostic cross-domain pattern - Solution 1.

- **Cyclic redundancy check with comparison**

This solution implements a cyclic redundancy check (CRC) based diagnostic with comparison to detect failures of the shared memory. The application data that is sent through the shared memory is used to calculate a CRC which is stored in memory (e.g., DDR). In addition, a golden CRC of the data that is sent is calculated and stored in the memory (e.g., OCM) by each core. This golden CRC is used to perform the comparison with the CRC value of the data that is sent through the shared memory and determine if the shared memory is source of failures. The calculation of the CRCs can be realized at the beginning or at the end of the execution of the tasks. In the case that the CRCs are calculated at the beginning, a synchronization mechanism may be required to synchronize the calculation and comparison of the CRCs.

This technique assumes that:

- The cores and the soft-core processors of the multi-core COTS device are checked in advance.
- The programmable logic (PL) and its associated components are checked in advance (e.g., BRAM).
- The timers of the device are checked in advance.
- The interrupt controller is checked in advance (e.g., GIC).
- The coherency management unit is correctly configured and checked in advance (e.g., SCU).

- The interconnection management unit is checked in advance.
- The memories are checked in advance (e.g., DRAM and OCM).

This solution considers the following two implementation scenarios where the technique proposed in this section may be implemented to diagnose the shared memory. The first scenario considers that the multi-core COTS device is provided as it is by the device manufacturer. The term *as it is*, is referred that the device provided without modifications that alter its properties (e.g., safety-related). Instead, the second scenario considers that the COTS device is provided with changes that enable the integration of a wide set of functionalities with different criticality (e.g., SIL1 to SIL4 according to the IEC 61508 safety standard). Virtualization solutions such as hypervisors are bespoke solution for that aim.

○ **Scenario 1: Non-partitioned system**

In this scenario it is assumed that the multi-core COTS device is provided without modifications (e.g., without partitioned), where each processor executes a single functionality that can be the same or not. This opens up new considerations (sub-scenarios) where the proposed CRC and comparison based diagnostic technique can be applied.

▪ **Scenario 1.1: Non-partitioned system with diverse functionality**

In this sub-scenario it is considered that the CPUs of the device execute different functionalities and that the overall solution presented at the beginning of this section is supported. All the necessary steps for implementing the proposed solution approach are defined down below. Furthermore, Figure 19 presents the main blocks and the overall scheme for implementing the proposed solution in this sub-scenario.

Step A) CPU0 and CPU1 execute different functionalities. Before sending data to the memory, the gold CRC of the data is calculated and stored in the memory. The gold CRC per each CPU is stored in different memory areas (Mem.1).

Step B) CPU0 and CPU1 write data in the memory (e.g., DDR). Each CPU has its own memory region (grey and white).

Step C) The data stored in the memory is read each 50ms and it is calculated their CRC (CRC32). The calculation of the CRCs can be performed at the beginning or at the end of the task which is executed by each CPU.

Step D) The comparison of the gold CRC and new CRCs is executed. The comparator of CPU0 reads the CPU0's gold CRC which is stored in the memory1 and compares it against the current CRC. If the CRCs matched, the execution continues. Otherwise, a fault-tolerance technique or a safe-state is executed.

Step E) Once the CRCs are compared and it is checked that the data is not corrupted by the shared memory, the current CRCs (one CRC per CPU) are stored in the memory1 as the new gold CRCs.

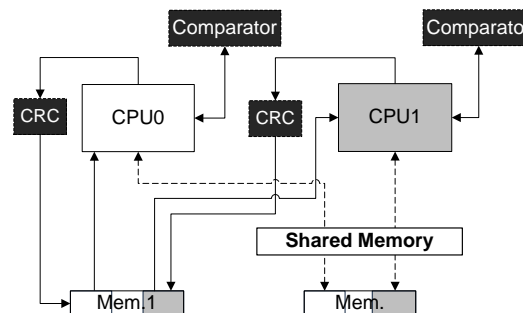


Figure 19: Shared memory diagnosis pattern - Solution2 - Scenario 1.1a: Non partitioned-system with different functionality.

▪ **Scenario 1.2: Non partitioned-system where each processor executes the same**

functionality

This sub-scenario considers a device where the CPUs execute the same functionality and where the overall solution which is presented at the beginning of this section is implemented. In this scenario (see Figure 20), the proposed solution works as follows.

Step A) CPU0 and CPU1 write data to the memory. Each CPU has its own memory region.

Step B) The data that is stored in the memory is read each 50 ms and the CRCs of the data are calculated. The calculation of the CRCs can be performed at the beginning or at the end of the task which is executed by each CPU.

- If the execution of the CPUs' task is synchronous, steps **c**, **d** and **f** shall be followed.
- Otherwise, if the execution of the tasks is asynchronous, synchronization mechanisms shall be required to synchronize the execution of the tasks and associated CRC calculations. In that case, once a synchronization mechanism is implemented, steps **e** and **f** shall be followed.

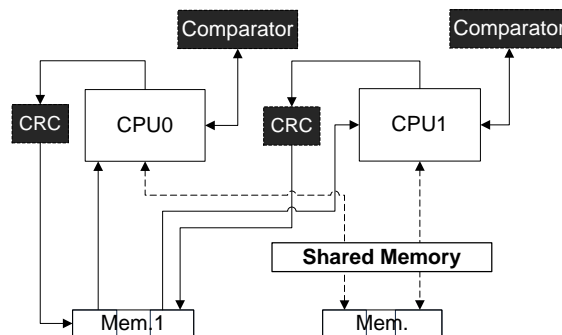


Figure 20: Shared memory diagnosis pattern - Solution 2 - Scenario 1.2a: Overview.

Step C) The CRCs of read data from the memory are calculated. During the calculating process, the CRCs can be compared *on the fly* against the golden CRCs which are stored in the memory1 (see Figure 21).

Step D) The CRCs of read data from the memory are calculated. During the calculating process, the CRCs can be compared on the fly against the same golden CRC which is stored in the memory 1. This is a particular case where it is assumed that the golden CRCs of each CPU are the same because the functionalities and data of the CPUs are the same and therefore, the resulting CRCs must be the same (see Figure 21).

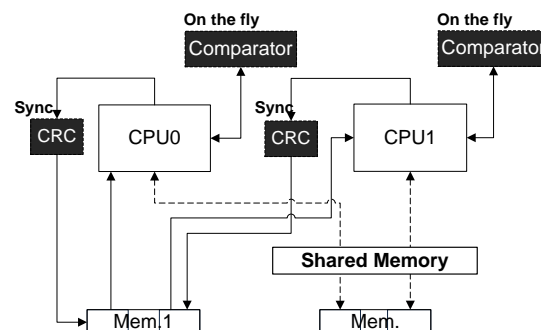


Figure 21: Shared memory diagnosis pattern - Solution 2 - Scenario 1.2b: On the fly comparison.

Step E) The comparison of the CRCs is executed. CPU0 reads the golden CRC that is stored in the memory1 and it compares it against the current CRC. If the CRCs matched, the execution continues. Otherwise, a fault-tolerance technique or a

safe-state should be executed. The same is applicable for CPU1.

Step F) Once the CRCs are compared and it is checked that the data has not being corrupted, the current CRCs are stored in the mem.1 as the new golden CRCs.

- **Scenario 1.3: Non partitioned-system where the functionalities are executed by a CPU and a soft-core processor**

In this third sub-scenario it is assumed that the device is composed of a processing system (PS) and a programmable logic (PL) where [1:N] soft-core processors are implemented. The diagnosis coverage provided by this architecture is better than the provided ones by the other diagnosis scenarios because the PL can be considered as additional independent hardware, where its failure modes, causes and effects differ from the ones of the PS. This scenario considers that the functionalities which are implemented on the soft-core processors may vary from the executed in the PS. Independently of the scenario, the steps defined in the scenarios 1.1 and 1.2 shall be followed to diagnose the shared memory. The major differences between these scenarios lie in the execution environments of the tasks and in the access methods to memories (e.g., OCM and DDR) such as shown Figure 22 and Figure 23.

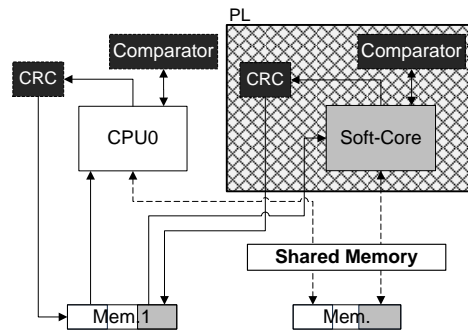


Figure 22: Shared memory diagnosis pattern - Solution 2 - Scenario 1.3a: PS-PL.

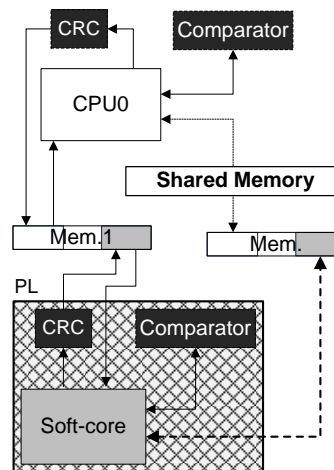


Figure 23: Shared memory diagnosis pattern - Solution 2 - Scenario 1.3b: PS-PL (direct access to memory).

- **Scenario 2: Partitioned system**

Mixed-criticality systems implements virtualization mechanisms such as hypervisors (e.g., XtratuM) for partitioning the systems into different execution environments or partitions where functionalities with different criticality can be implemented. This second scenario considers the diagnosis of the shared memory in a realistic environment where the processors (CPUs) of the device are partitioned, thus enabling the implementation of functionalities with different criticality. From a safety perspective, the proposed diagnosis scenario can be improved if the comparison is given between two partitions which are located in different cores (see Figure 28) or if hardware redundancy is implemented.

Nevertheless, during this scenario it is assumed that the partitions are scheduled by the hypervisor, thus providing bounded execution time of the tasks. Partitions can be configured with different schedules (see Figure 19) and executed by different CPUs.

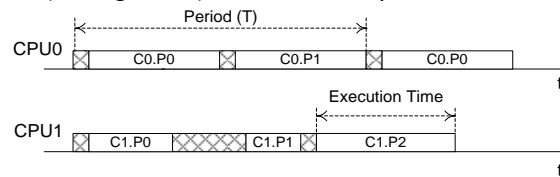


Figure 24: Partition scheduling - Example.

- **Scenario 2.1: Partitioned system where all safety-related partitions execute different functionalities**

In sub-scenario 1.1 the overall solution for a device where each CPU executes a different functionality is presented. That solution can also be applied to partitioned multi-core devices where functionalities of different criticality level are executed on the same CPU (see Figure 25). This sub-scenario shall follow the steps stated in the sub-scenario 1.1, taking into account that instead of having two CPUs, it is only available a single CPU.

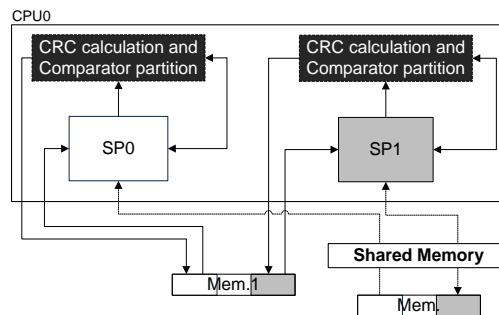


Figure 25: Shared memory diagnosis pattern - Solution 2 - Scenario 2.1: Partitioned system with different functionalities.

- **Scenario 2.2: Partitioned system where some safety-related partitions execute the same functionality**

In sub-scenario 1.2 the overall solution for a device where two CPUs execute the same functionality is presented. That solution can also be applied to the partitioned multi-core device shown in Figure 26. This scenario shall follow the steps stated in sub-scenario 1.2, taking into account that instead of having two CPUs, we have a single partitioned CPU.

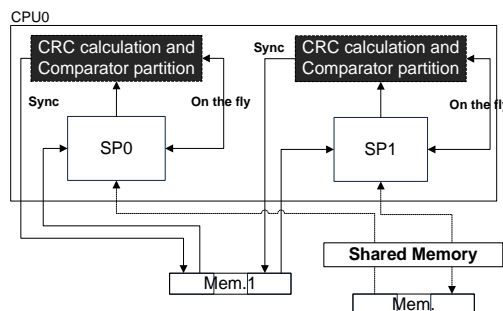


Figure 26: Shared memory diagnosis pattern - Solution 2 - Scenario 2.2: Partitioned system with same functionality.

- **Scenario 2.3: Partitioned-system where the partitions are executed on a CPU and a soft-core processor.**

In sub-scenario 1.3 the overall solution for a multi-core COTS device where a CPU and a soft-core processor execute the same or different functionality is presented. That solution can be also applied to a partitioned multi-core device such as shown in Figure 27. This scenario shall follow the steps stated in the sub-scenario 1.2, taking into account that

instead of having two CPUs, we have a partitioned CPU and a soft-core processor.

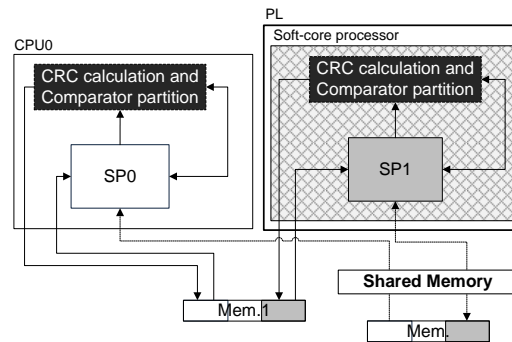


Figure 27: Shared memory diagnosis pattern - Solution 2 - Scenario 2.3: Partitioned system - PS-PL.

▪ **Scenario 2.4: Additional considerations and scenarios**

In the preceding scenarios, several solution approaches for shared memory diagnosis are presented. In addition, as stated at the beginning of Scenario 2, these diagnosis scenarios can be improvised in some cases. For example, as presented in

Figure 28, partition redundancy can be implemented to achieve this goal. In this case, the comparison of the CRCs can be carried out in different cores. On the other hand, Figure 29 presents an additional diagnosis scenario where it is implemented a redundant HW architecture. This scenario provides the comparison of CRCs at partition level and at system level. The comparison can be realized internally by a SW comparator or externally by a HW comparator. In the case that a SW based comparator is implemented, a communication network shall be required to spread of the resulting CRCs through the entire system. This approach enables to detect whether the shared memory fails due to a failure of some component of the device or the device itself.

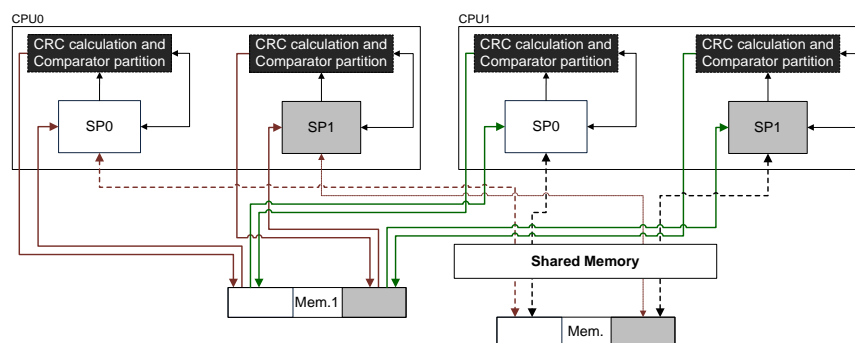


Figure 28: Shared memory diagnosis pattern - Solution 2 - Scenario 2.4a: Additional considerations and solutions.

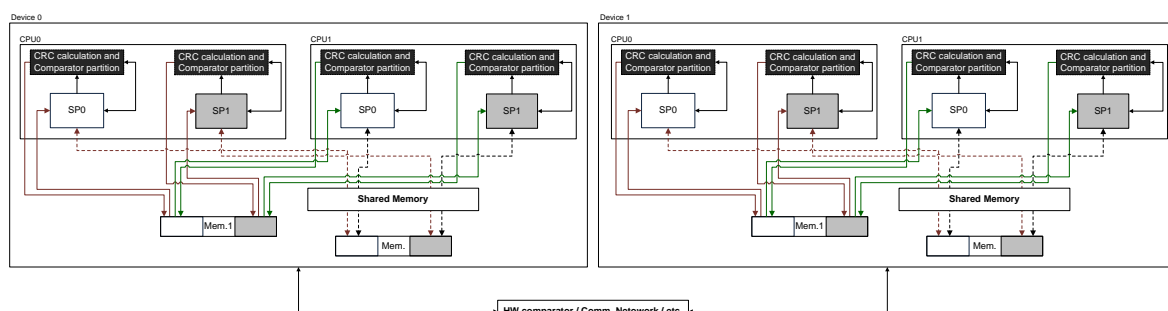


Figure 29: Shared memory diagnosis pattern - Solution 2 - Scenario 2.4b: Additional considerations and solution.

Board Name:	XILINX ZYNQ-7000 zc706
Implementation:	
This section presents the implementation of the pattern proposed before for the ZYNQ-7000 ZC706 multi-core device (See Section 3.1). More specifically, this section includes the implementation of the scenarios	

described before 1.1 and 1.2. Sub-scenario 1.3 and Scenario 2.x are not implemented, although the proposed solution is also applicable to them. Therefore, during the implementation of scenarios selected, the PS layer of the ZYNQ device is used. The PL is not used due to we do not implemented the scenario 1.3. However, indications to implement this scenario are included during this section.

Implementing scenarios 1.1 and 1.2:

The implementation of these scenarios follows the indications defined in the section before. For this purpose, we use the OCM and DDR memory of the ZYNQ device. The DDR memory is used for writing and reading the data of the CPUs. Instead, the OCM memory is used for storing the values of the CRCs which are calculated by the CPUs. In addition, as shown in

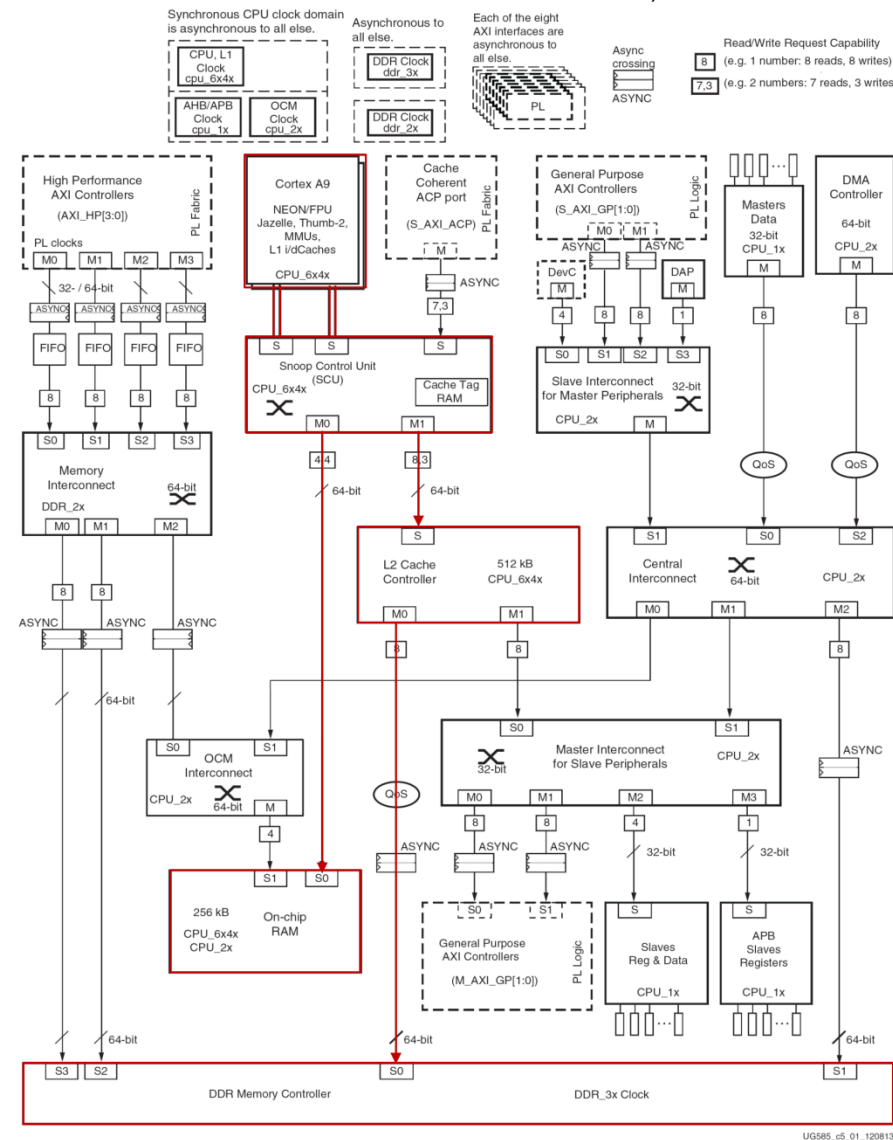


Figure 30, the DDR memory of the ZYNQ device can be accessed by the CPUs through the snoop control unit and the shared memory or L2 cache, while the OCM memory can be accessed through the SCU without going through the L2 cache. The snoop control unit is analyzed in pattern PAT- CCMU –XX (see Section 4.2.2).

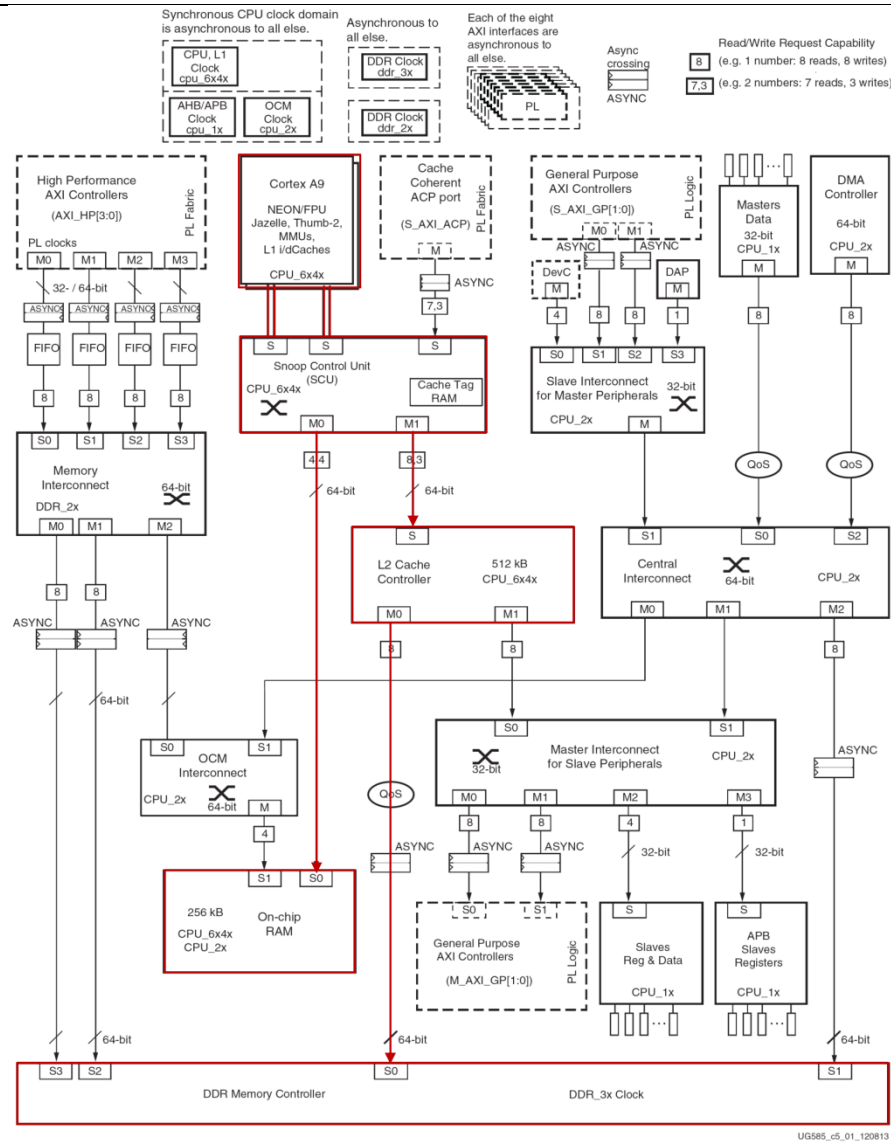


Figure 30: ZYNQ 7000 – Interconnecting the OCM and DDR memory and the CPUs (Source [16]).

This multi-core device is proprietary of Xilinx. Therefore, proprietary software tools such as Vivado and SDK are used for implementing these two scenarios (see Figure 31). Vivado software is used for defining the system architecture of the device, whereas the SDK software is used for implementing the application code. In this case, we use the SDK software for implementing the application code in C for the two CPUs of the ZYNQ device. The two CPUs implement the same functionality. The main difference is the CPU where the application is executed.

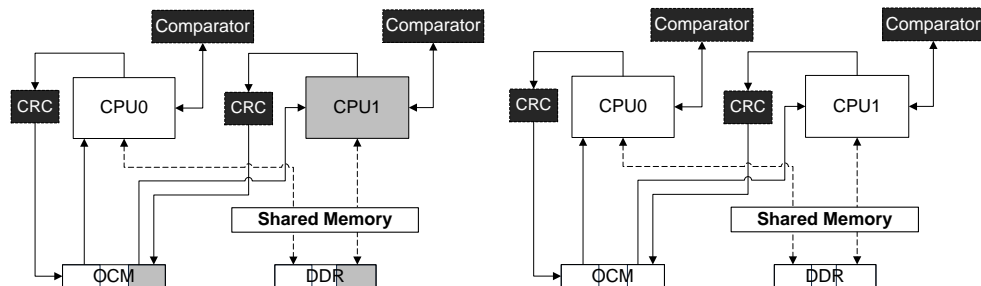
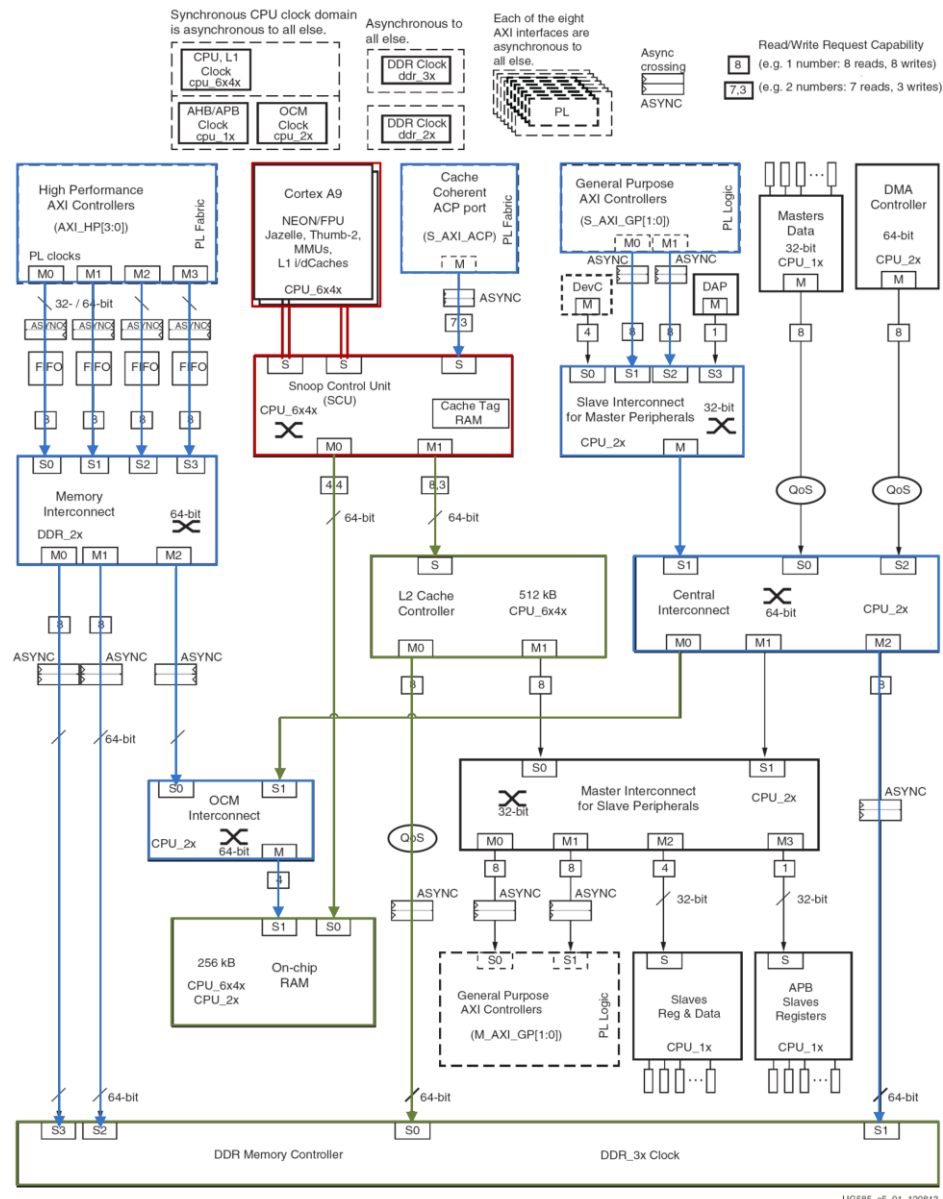


Figure 31: Implementation of scenarios 1.1 and 1.2 for the ZYNQ-7000 device.

Implementing scenario 1.3:

This scenario is an extension of the scenarios 1.1 and 1.2 that implements the proposed CRC and comparator

based solution in both the processing layer and the programmable logic of a multi-core device. In our case, the PS and the PL layers of the ZYNQ device are used for implementing this scenario. As defined before, the steps defined in the scenarios 1.1 and 1.2 are followed by this scenario to diagnose the shared memory. The major differences between these three scenarios lie in the execution environments of the applications and in the access methods to memories.



In the case of the ZYNQ device, as shown in

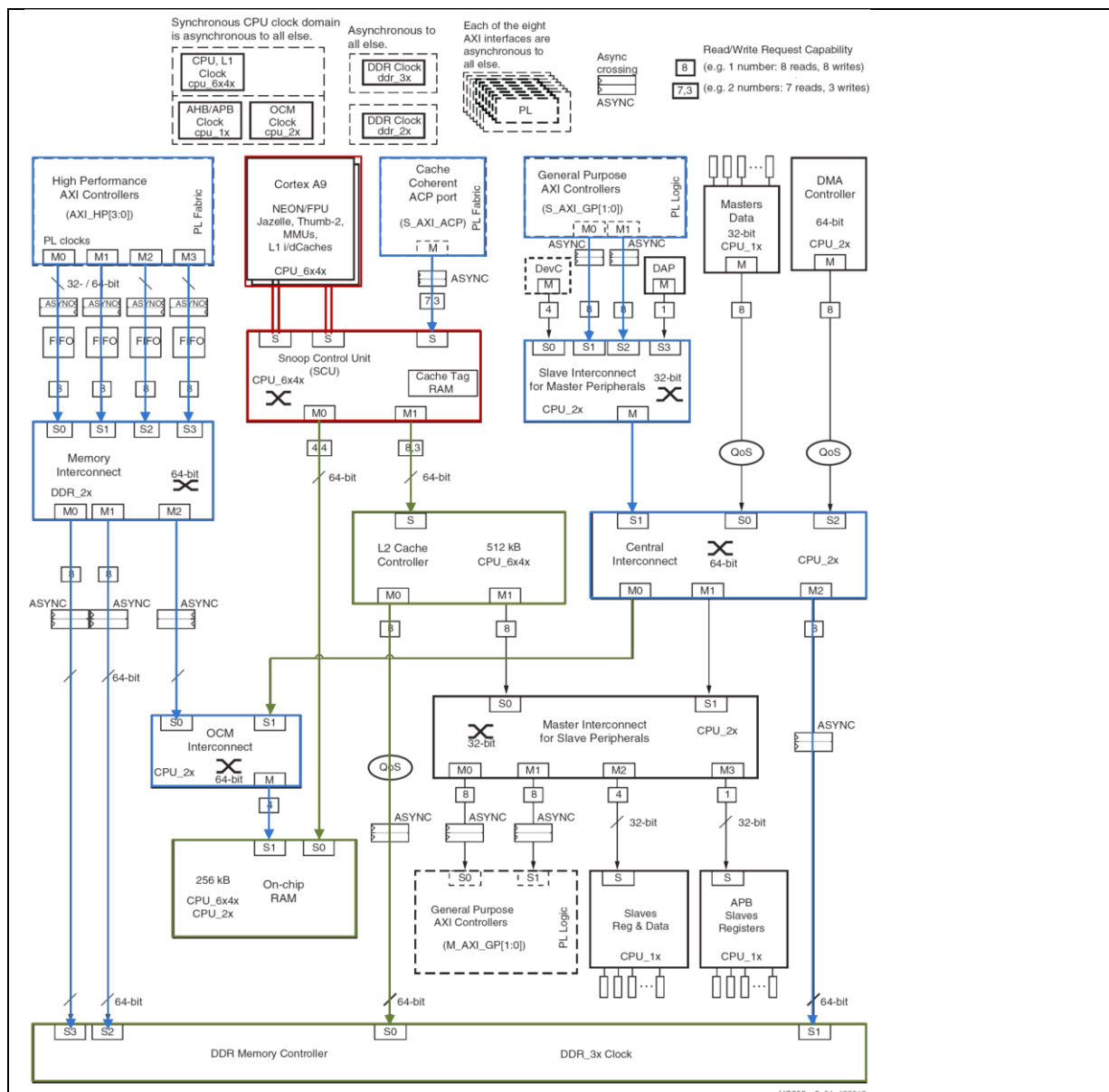


Figure 32, the OCM memory can be accessed by the CPUs of the PS through the snoop control unit, while the DDR memory can be accessed through the snoop control unit and the L2 cache. On the other hand, the PL can access to the DRR directly through the AXI_HP and the AXI_GP buses or through the shared memory using the AXI-ACP coherency bus. In the same vein, the OCM memory can be also accessed directly through the AXI_HP bus or through the snoop control unit using the AXI_ACP bus. Therefore, as defined in section before, this scenario can be implemented in different ways, depending on the intercommunication buses which are selected to access to the OCM and the DDR memories.

Results:

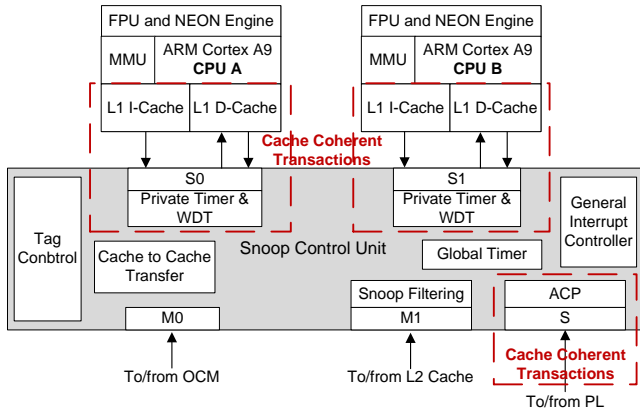
The following sequences present the results from the execution of the diagnostic technique Scenario 1.1. The right column defines the execution sequence followed by CPU0. Instead, the left column defines the sequence followed by CPU1.

Disable cache on OCM
 Disable cache on FSBL
 Initialize the SCU Interrupt Distributed (ICD)
 CPU0 – writing start address for CPU0
 Golden CRC calculated and stored in memory

Disable cache on OCM
 Disable cache on FSBL
 Initialize the SCU Interrupt Distributed (ICD)
 CPU1 – writing start address for CPU1
 Golden CRC calculated and stored in memory

Write data to DDR Read from DDR Comparing... Successful comparison Writing new golden CRC to memory Waiting... CPU0 – Cycle 1 Disable cache on OCM Disable cache on FSBL Initialize the SCU Interrupt Distributed (ICD) CPU0 – writing start address for CPU0 Golden CRC calculated and stored in memory Write data to DDR Read from DDR Comparing... Unsuccessful comparison	Write data to DDR Read from DDR Comparing... Successful comparison Writing new golden CRC to memory Waiting... CPU1 – Cycle 1 Disable cache on OCM Disable cache on FSBL Initialize the SCU Interrupt Distributed (ICD) CPU1 – writing start address for CPU1 Golden CRC calculated and stored in memory Write data to DDR Read from DDR Comparing... Successful comparison
Additional Considerations:	
This cross-domain pattern defines a diagnosis technique that it is related to the safety arguments of the modular safety case for an IEC 61058 compliant generic COTS processor [4].	
References:	
IEC, "IEC 61508-2 Functional safety of electrical/electronic/programmable electronic safety-related systems - Part 2: Requirements for electrical/electronic/programmable electronic safety-related systems," ed: IEC, 2010. F. Semiconductor, "P4080 Development System User's Guide," Freescale Semiconductor August 2010. XILINX, "ZYNQ-7000 All Programmable SoC: Technical Reference Manual," September 2014. DREAMS, "Distributed Real-Time Architecture for Mixed-Criticality Systems - A Modular Safety Case for COTS device," in <i>D5.1.2</i> , ed, 2015.	

4.2.2 Cache Coherency Management Unit Diagnosis Pattern

Pattern ID:		PAT – CCMU – 00
Pattern Name:	CCMU	
Related pattern:		PAT – ICMUD – 00 PAT – SMD – 00
Type:		HW/SW
Context:		
<p>Cache coherency is the consistency of shared resource data that ends up stored in multiple local caches (e.g., L1 cache and L2 cache). For example, it stores the copies of data saved in several caches. When one copy of data is modified, the other copy shall be changed, otherwise an inconsistency shall arise. Here is where this cross domain pattern is focused, ensuring that changes of data are propagated through the device and if not, detecting whether a coherency failure occurs. There are three main coherency mechanisms (Directory based, Snooping and Snaffling) which are usually used to provide coherency of memories.</p>		
Problem:		
<p>In today's mixed-criticality systems based on multi-core devices, the coherency management unit is implemented for managing, among others, the coherency of the processors, the memory and the programmable logic (PL). For example, as shown in Figure 33, the ZYNQ 7000 multi-core device implements the snoop control unit (SCU) which manages the coherency by means of the snooping coherency technique. In addition, this device considers that the accesses to the memory, the peripherals and etc., which are not routed through the SCU, are non-coherent accesses. In those cases, it is assumed that the coherency and the synchronization between the components of the device shall be handled by SW [16].</p> 		
<p>Figure 33: ZYNQ device - Coherency block-diagram (Source [16]).</p>		
<p>The bus sniffing or bus snooping technique assumes that each processor of the device has its own cache (e.g., L1 cache) and that a shared main memory is available (e.g., L2 cache). These cache memory architectures usually lead to coherency inconsistency issues that may arise with inconsistent data (a common case in multi-core architectures). For example, Core A of Figure 33 has a copy of a memory block from a previous read and Core B changes the memory block. Consequently, in the case that the coherency management unit fails, the data of Core A is not updated, leading to the inconsistency of data, which can cause the failure of the mixed-criticality system. Therefore, although multi-core devices implement coherency management mechanisms, coherency related failures may arise due to the failures analysed in the next section.</p>		
Solution under consideration:		
<p>This cross-domain pattern aims to provide a generic diagnosis technique that enabled the detection and control of the coherency-related faults. This cross-domain pattern assumes that:</p> <ul style="list-style-type: none"> - The cores of the device are checked in advance. 		

- The L1 and the L2 cache memory and the OCM memory are checked in advance.
- The interconnection management unit is checked in advance.
- The PL and its associated components (e.g., BRAM) are checked in advance.
- The timers of the device are checked in advance.
- The interrupt controller (e.g., GIC) is checked in advance.

This cross-domain pattern proposes the following three possible approaches where the coherency management unit is analyzed from safety perspective.

- **Check the configuration of the coherency management unit**

The configuration of the coherency management unit shall be chosen in a reasonable manner for providing minimum possible interferences between the resources (components) that are connected to the coherency management unit. Wrong or incorrect configuration of the coherency management unit may lead to the loss of coherency and the resultant failure of the system. Therefore, in this first solution the periodic checking of the coherency management unit's configuration is proposed, comparing it with the expected configuration or the last valid configuration set. In addition, this solution assumes that the chosen configuration shall be free of systematic faults, ensuring that it is protected against unexpected configuration changes. The configuration-related failure modes are defined in the FMEA and FMECA analyses defined in the deliverable D5.1.2 "A modular safety case for COTS device" [4].

- **Failures caused by unexpected behaviour of coherency management unit (Random faults)**

Software has the ability to manage the memory regions which are shared among certain sets of coherent masters. In addition, it ensures that the shareability mappings between the types of masters are consistent to avoid unexpected behaviours or inconsistencies. For instance, protection mechanisms such as the Memory Management Unit (MMU) can be used to control the memory, manage permissions to blocks of the memory and translate the virtual addresses to physical addresses. Furthermore, this solution considers the following measures and diagnostic techniques to detect and control the faults of the coherency management unit such as the wrong addressing, partial update or single bit errors faults.

- A watchdog timer (WDT) can be implemented to detect message order violations in a fixed communication network.
- A sequence number can be used to detect the correct reception of messages, where if a message is not received it is considered that a fault occurs.
- CRC with comparison (see cross-domain pattern PAT-SM-00), ECC and/or parity bit diagnostic technique can be implemented to detect data consistency violations, including partial update or single bit error failures.

These faults and diagnostic techniques are defined in the FMEA analysis included in the deliverable D5.1.2 "A modular safety case for COTS device" [4].

- **Failures caused by external influences (Systematic faults)**

The coherency management can be affected by systematic faults which can be caused by HW design, environmental stress or influences or operational failures. This solution considers the implementation of the measures and diagnostic techniques recommended in tables A.15 to A.17 of IEC 61508-2 for detecting and controlling the systematic faults of the coherency management unit. Furthermore, it is assumed that the selection of measures and diagnostic techniques depends on the HW platform or the SW that is supported by the system architecture. Therefore, the selection of measures and diagnostic techniques for this purpose may vary. Systematic faults of the coherency management unit are analysed in DREAMS deliverable D5.1.2 "A modular safety case for COTS processor" [4] by means of a FMECA analysis.

In addition, the following measures for **fault avoidance** and **fault control** in systems using cache coherency

may be included:

- **Fault avoidance:**
 - Limit shared memory usage to an absolute minimum required for operation
 - Limit the use of multiple threads and tasks for one safety function to a minimum required
 - Make sure that per potential cache line there is only one task/process allowed to write and all other may only read (only 1:N communication allowed). The assignment should be defined statically.
- **Fault control:**
 - Implement communication protocol with additional messaging between sender and receiver of the information. For example:
 - *Order violation detection:* Flags to indicate whether the information is updated and received.
 - *Data consistency violations:* Extra coding information (e.g., CRC/ECC or Parity Information) in the same memory block as the updated information is stored. The flags must be updated as last write action to the shared memory.
 - It is safe to assume that a HW implemented ECC/Parity on caches may have bugs (e.g., ARM: 751475—Parity error may not be reported on full cache line access (eviction / coherent data transfer / cp15 clean operations).
 - Implement data structures that match the cache architecture (e.g., maximum size of one cache line for optimal performance) and allow additional diagnostics:
 - Cache memory ECC and scrubbing, if applicable.
 - Implement timing expectations and error detection for the shared memory communication.
 - Implement other typical communication error related measures (e.g., sequence number, addressing (could be done by different CRC codes), coding and/or timing expectation).
 - Automatic invalidation of cache lines after a defined period of time to ensure that caches are flushed periodically.

Board Name:

XILINX ZYNQ-7000 zc706

Implementation:

This section defines the implementation of the solutions defined before for detecting failures of the coherency management unit. For that purpose, during this section the ZYNQ 7000 zc706 device is used as the reference multi-core COTS device. This device implements the coherency between its components through the snoop control unit (SCU). SCU implements the snooping coherency protocol for that purpose. In addition, as shown in Figure 34, the SCU unit is the core for accessing any component through the ARM Cortex A9 core and to access the shared memory or L2 cache from the PL using the AXI_ACP coherency bus.

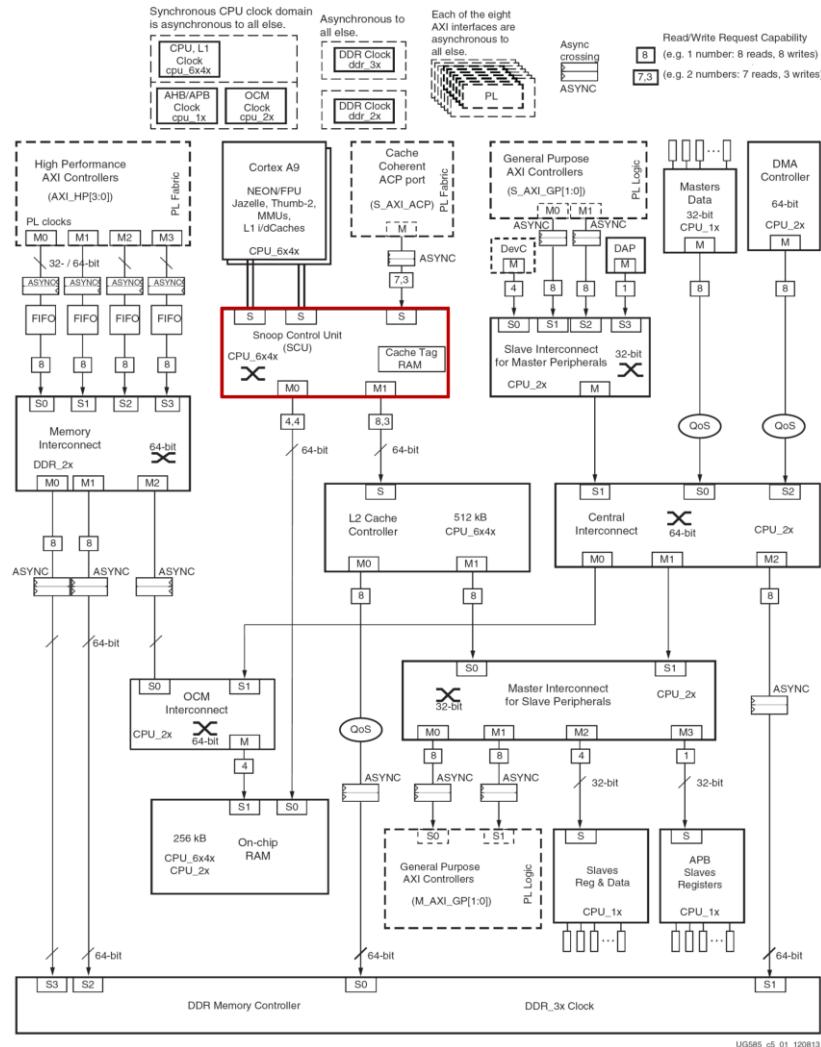


Figure 34: ZYNQ 7000 - SCU Interconnect (Source [16]).

In the following paragraphs it is defined the implementation in the ZYNQ device of the solutions defined before.

- **Diagnostic of configuration errors**

The coherency management unit of a multi-core COTS device shall be safely configured. This mean that the configuration registers of the coherency management unit shall be actively diagnosed to detect configuration errors. Table 2 shows the registers associated to the coherency management unit of the ZYNQ device. These registers are periodically checked and compared against the values expected to detect whether the configuration values change. For that purpose, we implement a diagnostic application that reads the registers related to the coherency management unit and that compare them against expected values (pre-defined values at design time). If the configuration registers match, the device continues working, otherwise a fault is asserted and the device goes to safe-state.

Control register bit assignment		
Bit	Name	Description
[2]	SCU RAMs Parity	1 = Parity on.

	Enable	0 = Parity off. This bit is always zero.
Configuration register bit assignment		
<i>Bit</i>	<i>Name</i>	<i>Description</i>
bit [7:4]	CPUs SMP	Defines the Cortex A9 processors that are in SMP or AMP 0: in AMP mode not taking part in coherency or not present 1: in SMP mode taking part in coherency bit [7]: CPU3 bit [6]: CPU2 bit [5]: CPU1 bit [4]: CPU0
SCU CPU Power Status Register bit assignment		
<i>Bit</i>	<i>Name</i>	<i>Description</i>
bit[25:24]	CPU status	Power Status of the Cortex A-9 processor b00: Normal mode. b01: Reserved b10: the Cortex A9 process is about to enter (or is in) dormant mode. No coherency request is sent to the Cortex A9 processor b11: the Cortex A9 process is about to enter (or is in) powered-off mode, or is non-present. No coherency request is sent to the Cortex A9 processor
AXI USER attributes encodings		
<i>Bit</i>	<i>Name</i>	<i>Description</i>
bit[0]	ARUSERMx	Shared bit 1 Coherent request 0 Non-coherent request
bit[0]	AWUSERMx	Shared bit 1 Coherent request 0 Non-coherent request

Table 2: SCU registers – Coherency (Source [42]).

As defined before, the coherency management unit or SCU is accessible from the PL through the ACP AXI bus. AXI_ACP is a full AMBA 3 AXI slave coherency interface bus, with the exception of the following transfers which are not supported:

- Coherent exclusive read and write transfers
- Coherent locked read and write transfers
- Optimized coherent read and write transfers when byte strobes are not all set.

In deliverable DREAMS D5.1.2 “A modular safety case for COTS processor” [4], the random and systematic failures of the coherency management unit are analysed by means of a FMEA and a FMECA.

- **Diagnostic of random faults**

The failures of the coherency management unit are diagnosed by means of the following measures and diagnostic techniques:

- CRC with comparison:

This technique is defined and implemented in pattern below (PAT-SM-XX).

- Error Correcting Code (ECC):

The ZYNQ 7000 device supports ECC technique in half-bus width (16bit) data width configuration. ECC provides single error correction and dual error detection. When ECC is enabled, a write operation computes and stores an ECC code along with the data, and a read operation reads and checks the data against the stored ECC code. It is therefore possible to receive ECC errors when reading uninitialized memory locations. To avoid this problem, all memory locations must be written before being read.

On the other hand, the errors detected by ECC based diagnosis can be classified into correctable

and uncorrectable errors. For correctable ECC errors, there is no error actively signalled via an interrupt or AXI response. Instead, for uncorrectable ECC errors, the controller returns a signal response back to the re-requesting AXI bus master. In both cases, information regarding the error (such as column, row and bank error address, error byte lane, etc.) is logged in the controller register space. In the case that the controller detects a correctable ECC error

When the controller detects a correctable ECC error, it automatically corrects the error and sends the correct data to the bus master. Instead, when the controller (e.g., DDRC) detects an uncorrectable ECC error, it returns a signal response to the bus master with the uncorrectable data. In that point, if the L2 cache or shared memory is disabled, the signal response is directly received by the CPU, causing data abort. Otherwise, if the shared memory is enabled, the ECC error is reported to the CPU by means of an interrupt caused by the shared memory.

The ECC based diagnostic technique which is implemented in this section implements the following order of execution:

- a) Disable the cache
- b) Read ECC registers
- c) Initialize data on DDR memory
- d) Disable the ECC
- e) Depending on the errors which are required to be detected, this pattern injects uncorrectable or correctable errors on the DDR.
- f) Enable the ECC
- g) Read ECC registers
- h) If the cache memory shall be implemented, enable the cache, set up an interrupt for reporting the ECC errors to the CPU and read data from DDR.
- i) If uncorrectable ECC errors are detected, an interrupt is generated by the cache memory to report the ECC errors to the CPU.
- j) Otherwise, the ECC error is directly transmitted to the CPU.

This solution considers the following four scenarios that depend on the ECC error detection requirements (e.g., detection of correctable or uncorrectable errors) and the availability of the shared cache memory.

- Correctable error detection with shared memory enable
- Correctable error detection with shared memory disable
- Uncorrectable error detection with shared memory enable
- Uncorrectable error detection with shared memory disable

▪ Parity bit:

In the case of the ZYNQ device this technique is supported by almost all memories implemented by the multi-core device.

- *DDR memory:*

The DDR memory controller of the ZYNQ device supports parity detection. It can be enabled or disabled through the configuration of the registers of the DDR controller (Register *ddrc ECC_scrub* [4:0] with relative resolution address 0x000000F4 and absolute address 0xF80060F4, bit [2:0] in "010").

- *L1 and L2 caches:* The parity bit of the L2 cache can be enabled by configuring the registers of the shared memory. By default the parity bit of the L2 cache is disabled. In our case, we implement a testing environment to detect parity bit error of the L2 cache. For that purpose it is defined a testing scenario where a parity bit error is generated, providing a

data abort exception and an interrupt. This software implements the following steps:

- a) Disable L2 cache
- b) Disable the parity
- c) Enable L2 cache
- d) Write data
- e) Disable L2 cache
- f) Enable parity
- g) Enable L2 cache
- h) Read data

- *SCU controller register:*

See Table 2.

- *OCM memory:*

OCM memory supports both single and multiple bit parity bit errors. In the event that a parity error is detected, an interrupt is asserted and the parity bit error of the OCM memory is returned or provided.

- a) Disable D cache of L1 and L2 cache memories.
- b) Disable I cache of L1 and L2 cache memories.
- c) Configure the OCM_PARITY_CTRL register to enable the AXI read and the use of interrupts for reporting the parity error to the CPU.
- d) Write data to OCM to generate a parity error
- e) Read data from OCM.

- **Diagnostic of systematic faults**

In this section we assume that the other components of the ZYNQ device have been diagnosed in advance to detect and control systematic faults.

- **Fault avoidance**

During the implementation of this pattern it is indented to minimize the use of the shared memory to an absolute minimum required for operation. For that purpose, during the diagnosis of the cache coherency unit the shared memory of the ZYNQ device has been disabled.

- **Fault control:**

On the other hand, in order to control faults that can occur in the cache coherency unit, several measures and diagnostic techniques such as the ECC, Parity and CRC with comparison have been implemented, thus detecting and controlling data consistency violations.

Although the terms used throughout this pattern are exclusive from the ZYNQ device (e.g., SCU and GIC), they may be replaced by the terms used by any device, provided that they follow similar architecture (e.g., SCU (ZYNQ device) – CoreNet (P4080)).

Results:

This cross-domain diagnostic pattern implements a set of measures and diagnostic techniques to detect faults related to the coherency management unit. The execution of these measures and diagnostic techniques result on the following results which evidence the implementation and applicability of the proposed solutions.

- **Diagnostic of configuration errors**

As defined before, this diagnostic solution read the configuration registers of the coherency management unit and compares them against expected register values (by default values). If the registers match, this diagnostic is executed each 50ms. Otherwise an exception is triggered and the system goes to a safe-state.

```
#### Checking configuration errors
#### Reading registers of SCU Controller....
10001000001101111000000001011010
#### Checking if read values match with the expected configuration values
#### Successful Comparison
#### Summary of SCU controller's registers:
    SMP mode activated for CPU0
    AMP mode activated for CPU1
    AMP mode activated for CPU2
    AMP mode activated for CPU3
#### Reading registers of the SCU CPU power status....
0000000000000001101001000011101
#### Checking if read values match with the expected configuration values
#### Unsatisfactory Comparison
#### Restarting...
```

- **Diagnostic of random faults**

- CRC with comparison

See PAT-SM-00.

- Parity Bit

The parity bit error diagnostic technique is implemented and executed to check parity bit error in the OCM and L2 cache memories. As shown below, this pattern's execution results on the following execution sequence where an error is inserted for checking the application.

```
#### OCM parity error Test
#### An exception processed
IRQ No.35 OCM interrupt processed
#### OCM parity error test is run successfully run
```

In the same vein, the parity bit error for L2 cache results in the following execution sequence.

```
#### L2 cache parity error Test
#### An exception processed
IRQ No.34 L2 cache interrupt processed
#### L2 cache parity error test is run successfully run
```

- ECC

The sequence that is defined below presents the result from the execution of the ECC diagnostic technique.

```
#### Disable L1 and L2 Cache
#### Read ECC registers
DDRC.CHE_CORR_ECC_LOG_REG_OFFSET:00000000 ( No Correctable Error )
```

```

DDRC.CHE_UNCORR_ECC_LOG_REG_OFFSET:00000000 ( No Uncorrectable Error )
DDRC.CHE_ECC_STATS_REG_OFFSET:00000000 ( 0 Correctable Error(s), 0 Uncorrectable Error(s) )
#### Initialize Data on DDR3
00100000: 00000000
00100004: 00000000
00100008: 00000000
0010000C: 00000000
00100010: 00000000
00100014: 00000000
00100018: 00000000
0010001C: 00000000
#### Disable ECC
ADDR: 0x000000F4 W: 0x04
#### Insert Correctable Errors (1bit error) on DDR3
00100000: 00000001
00100004: 00000000
00100008: 00000000
0010000C: 00000000
00100010: 00000000
00100014: 00000000
00100018: 00000000
0010001C: 00000000
#### Enable ECC
ADDR: 0x000000F4 W:0x04, ADDR:0x000000C4 W:0x03, W:0x00
#### Read ECC registers
DDRC.CHE_CORR_ECC_LOG_REG_OFFSET:00000000 ( No Correctable Error )
DDRC.CHE_UNCORR_ECC_LOG_REG_OFFSET:00000000 ( No Uncorrectable Error )
DDRC.CHE_ECC_STATS_REG_OFFSET:00000000 ( 0 Correctable Error(s), 0 Uncorrectable Error(s) )
#### Enable Cache
#### Enable D-Cache (L1 & L2)
#### Enable I-Cache (L1 & L2)
#### Set Up Interrupt
#### Read Data from DDR3 again
00100000: 00000000
#### Read ECC registers
DDRC.CHE_CORR_ECC_LOG_REG_OFFSET:00000007 ( Correctable Error Detected )
DDRC.CHE_CORR_ECC_ADDR_REG_OFFSET:00040000 ( Correctable Error: Bank=0x0, Row=0x40,
Column=0x0 )
DDRC.CHE_UNCORR_ECC_LOG_REG_OFFSET:00000000 ( No Uncorrectable Error )
DDRC.CHE_ECC_STATS_REG_OFFSET:00000100 ( 1 Correctable Error(s), 0 Uncorrectable Error(s) )

```

Additional Considerations:

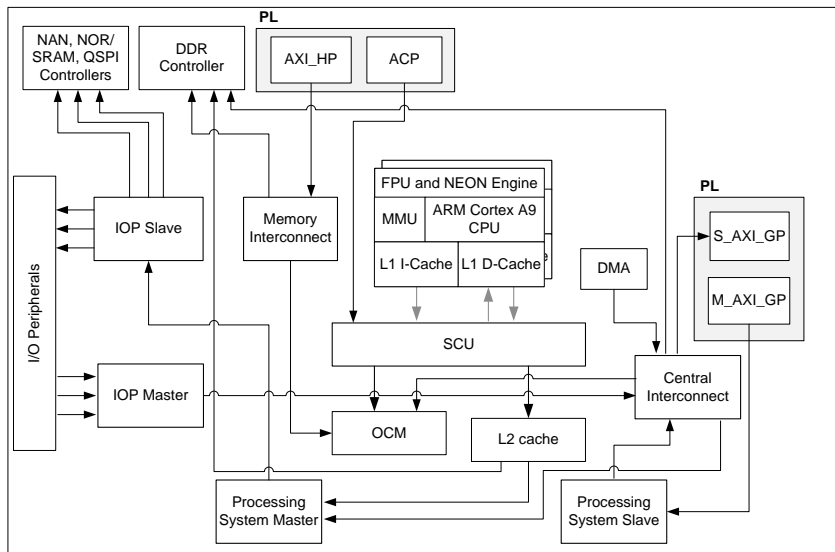
This cross-domain pattern defines a diagnosis technique that it is related to the safety arguments of the modular safety case for an IEC 61058 compliant generic COTS processor [4].

References:

F. Semiconductor, "P4080 Development System User's Guide," Freescale Semiconductor August 2010.

DREAMS, "Distributed Real-Time Architecture for Mixed-Criticality Systems - A Modular Safety Case for COTS device," in *D5.1.2*, ed, 2015.

4.2.3 Inter-Connection Management Unit Diagnosis Pattern

Pattern ID:		PAT – ICMUD – 00
Pattern Name:	ICMUD	
Related pattern:		N/A
Type:		HW/SW
Context:		
<p>Multi-core processor architectures implement interconnection management units for managing the transactions between their components. These units are prone to failures and uncertainties related to their expected behaviour (e.g., lack of information). In the event that the interconnection management unit fails, the interconnections, the arbitrations and the communications among the subsystems and elements of the device will fail, leading to a general failure of the system.</p>		
Problem:		
<p>COTS multi-core devices include a wide variety of components which usually require for communicating. For that purpose interconnection buses are usually implemented. The interconnection buses switch the traffic through different components of the device. For example, the ZYNQ 7000 device implements an interconnection manager that is composed of a set of interconnect blocks or switches that manages, among others, the communication among the cores, the memories, the peripherals and the PL. Figure 35 shows the block-diagram of the interconnections inside the ZYNQ device.</p>		
<div></div> <p>Figure 35: ZYNQ device - Interconnect.</p>		
<p>The interconnect blocks or switches which are implemented by the ZYNQ device are the following:</p> <ul style="list-style-type: none">- <i>Interconnect master</i> (ACP, AXI_HP, AXI_GP, DMA, IOP, etc.).- <i>Snoop Control Unit (SCU)</i>.- <i>Central interconnect</i> – is the core of the interconnect switches.- <i>Master interconnect</i> – switches the low and medium speed traffic from the central interconnect to		

M_AXI_GP ports, IOP and etc.

- *Slave interconnect* – switches the low and medium speed traffic from S_AXI_GP ports to the central interconnect.
- *Memory interconnect* – switches high speed traffic from the AXI_HP ports to DDR DRAM and on-chip RAM (OCM) through another interconnect.
- *OCM interconnect* – switches high speed traffic from the central interconnect and the memory interconnect.

The communication between the components of the ZYNQ device such as the CortexA9 processors, the peripherals (IOP), the memory and the PL is carried out through bus switches. In the event that a switch or interconnection block fails, the communication might fail, which may lead to an unexpected behaviour of the device. For example, imagine that the CortexA9 processors (PS) require communicating with the soft-core processor of the PL. In that event, as shown in Figure 36, the PL could be accessed by the PS through AXI_HP ports and/or AXI_GP ports, going through several interconnect blocks such as the central interconnect, OCM, SCU, memory interconnect, IOP and others. Therefore, in the case that the interconnect unit fails or an unexpected fault is provided in this unit, the communication of the system may fail or may influence on the safety-related behavior of the system. In addition, there are also common the interferences between the traffic from the CPUs (through L2 cache), the DMA and IOP masters and the traffic from the PL.

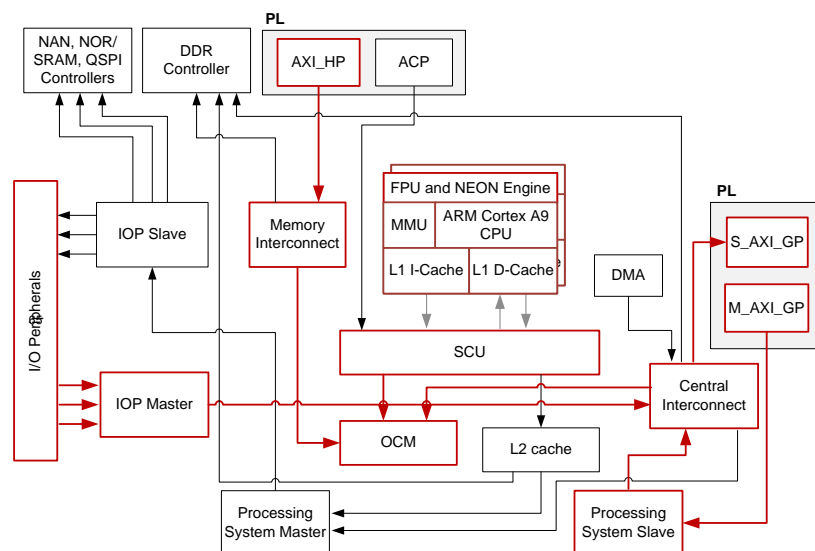


Figure 36: Interconnect - Example.

Solution under consideration:

The interconnection scheme of each COTS device is unique. For instance, the P4080 and ZYNQ 7000 devices implement different interconnection management units. This cross-domain pattern aims to define a generic solution or set of solutions for measuring and detecting the faults of interconnection management units of multi-core devices. This cross-domain pattern assumes that:

- The cores of the device are checked in advance.
- The L1 and L2 cache memories and the OCM memory are checked in advance.
- The PL and associated components (memories, etc.) are checked in advance.
- The timers of the device are checked in advance.
- The interrupt controller is checked in advance.

The proposed solution considers the following three solution approaches for testing the interconnection management unit.

- **Check the configuration of the interconnect management unit**

The interconnect management unit shall be configured in a reasonable manner to provide minimum possible interferences. The components or blocks that compose the interconnect manager are configured by means of registers. The configuration of their registers will be used to manage their behaviour, thus leading in an erroneous or partial behavior. Therefore, in order to detect whether the configuration of the interconnection management unit changes, this solution proposes the implementation of periodic readback check with comparison of the interconnect manager's configuration registers.

- **Failures caused by unexpected behaviour of the interconnection management unit (Random Failures)**

In the case that a maximum latency is required by the implemented system, the quality of service (QoS) modules can be used to ensure expected throughput and latency in the system design. The modules regulate the masters that do not guarantee maximum latency (e.g., CPU, DMA and IOP). In addition they can be used to resolve issues related to contention by means of two-level arbitration abstraction scheme. The first scheme is based on priority indicated by the QoS register. The highest QoS value has the highest priority. The second scheme is based on a least recently granted scheme and is used when multiple request are pending with the same QoS signal value.

In addition, the interconnect manager shall provide the following set of measures and diagnostic techniques to detect random faults. Among other, this unit shall consider measures and diagnostic techniques for typical faults such as wrong addressing or wrong data forwarding, including partial transmissions or single bit error. For that purpose, this solution considers the following measures and diagnostic techniques:

- A watchdog timer (WDT) can be implemented to detect temporal deviations.
- CRC with comparison (see cross-domain pattern PAT-SM-00), ECC and/or parity bit diagnostic technique can be implemented to detect data consistency violations, including partial update or single bit error failures.

These faults and diagnostic techniques are defined in the FMEA analysis included in the deliverable D5.1.2 "A modular safety case for COTS device" [4].

- **Failures caused by external influences (Systematic Failures)**

The interconnect manager can be affected by systematic faults which can be caused by the HW design, environmental stress or influences or operational failures. This solution considers the implementation of the measures and diagnostic techniques recommended in tables A.15 to A.17 of IEC 61508-2 for detecting and controlling the systematic faults of the interconnection management unit. Furthermore, it is assumed that the selection of measures and diagnostic techniques depends on the HW platform or the SW that is supported by the system architecture. Therefore, the selection of measures and diagnostic techniques for this purpose may vary. On the other hand, the possibility of systematic errors in the configuration of the interconnection management unit shall be addressed by these techniques. Table 24 of DREAMS deliverable D5.1.2 [4] analyses the possible systematic failures in the configuration process by means of an FMCA analysis. In addition, Chapter 4.2.11.2.2 (Tables 41 and 43) of DREAMS deliverable D5.1.2 "A modular safety case for COTS processor" [4] analyses the systematic faults of the coherency management unit by means of FMECA analyses.

Board Name:	XILINX ZYNQ-7000 zc706
Implementation:	
N/A	
Results:	
N/A	
Additional Considerations:	
This cross-domain pattern defines a diagnosis technique that it is related to the safety arguments of the	

modular safety case for an IEC 61058 compliant generic COTS processor [4].

References:

F. Semiconductor, "P4080 Development System User's Guide," Freescale Semiconductor August 2010.
DREAMS, "Distributed Real-Time Architecture for Mixed-Criticality Systems - A Modular Safety Case for COTS device," in *D5.1.2*, ed, 2015.

4.2.4 Interrupt Controller Diagnosis Pattern

Pattern ID:		PAT – ICD – 00
Pattern Name:	ICD	
Related pattern:		N/A
Type:		HW/SW
Context:		
<p>The interrupt controller is an integral part of today’s multi-core COTS devices that is implemented to manage the events of the device. An interrupt is a signal that causes the stop of the ongoing task and figures what to do next. For example, in operating systems the use of interrupt handlers is usual procedure to prioritize the interrupts.</p>		
Problem:		
<p>The interrupt controllers manage the execution of the tasks of the cores of multi-core devices. Therefore, at the event that the interrupt controller fails or that the request for an interrupt or the assignment of an interrupts fails, the execution of the processor’s shall be affected. In addition, in multi-core mixed-criticality systems where applications with different criticality level are integrated into the same device, the interrupt controller shall guarantee and manage the execution of the functionalities with different criticality level. For that purpose, the interrupt controller shall manage interrupts with different criticality level. In Table A.1 of IEC 61508-2 [19] there are defined the requirements for faults that shall be detected and measured in order to guarantee the safety of the interrupt handling. However, this standard is focused by single-core architectures where a resource cannot be shared between more than one component, and therefore, the measures and diagnostic techniques recommended by this standard are not at all applicable to interrupt controllers which are shared or used for managing the execution of functionalities with different criticality.</p>		
Solution under consideration:		
<p>The interrupt controller or interrupt manager is commonly used unit for controlling the execution of tasks in multi-core device. These units can be differently called and can be composed of different functionalities, depending on the multi-core device. For example, the interrupt controller of the ZYNQ 7000 device implements an interrupt controller called “Generic Interrupt Controller (GIC)” for managing the execution of the processors’ tasks. In this section it is assumed that:</p> <ul style="list-style-type: none">- The cores of the device are checked in advance.- The L1 and L2 cache memories and the OCM memory are checked in advance.- The PL and associated components (memories, etc.) are checked in advance.- The timers of the device are checked in advance.- The interconnection management unit is checked in advance.- The coherency management unit is correctly configured and checked in advance. <p>The proposed solution in this section considers the following three solution approaches for testing the</p>		

interrupt controller.

- **Check the configuration of the interrupt controller unit**

In Section 3.1.4 the architecture overview of an interrupt controller is presented, where it composes of a distributor and one or more CPU interfaces. These components can be configured independently by means of registers. The configuration registers of these components which are listed in document "ARM Generic Interrupt Controller – Architecture Specification" [37], shall be periodically checked to detect whether the configuration of the interrupt controller is modified. Therefore, this solution defines a periodic checking of the configuration registers of the interrupt controller, which are analyzed by means of a FMEA analysis in DREAMS D5.1.2 deliverable [43].

- **Failures caused by unexpected behaviour of the interrupt controller (Random failures)**

The interrupt controller component can be the subject of unexpected internal failures which can be caused by direct-current (DC) faults, drift and oscillations and reset-related faults. In Table A.1 of IEC 61508-2 techniques and measures for diagnostics and recommended maximum levels of diagnostic coverage for an interrupt controller are defined.

- **Failures caused by external influences (Systematic failures)**

Tables A.15 to A.17 of IEC 61508-2 [19] recommend techniques and measures for controlling systematic failures, including techniques and measures to control systematic failures caused by HW design, environmental stress or influences or operational failures. These techniques shall be implemented to detect systematic faults that can occur in the GIC. For example, the possibility of systematic errors in the configuration of the interconnection management unit shall be addressed by means of these techniques. Table 24 of DREAMS deliverable D5.1.2 [4] analyses the possible systematic failures in the configuration process by means of an FMCA analysis. In addition, in Chapter 4.2.11.2.4 (Table 46) of DREAMS deliverable D5.1.2 [4] the systematic failures of the interrupt controller are analyzed by means of FMEAs.

Board Name:

N/A

Implementation:

N/A

Results:

N/A

Additional Considerations:

This cross-domain pattern defines a diagnosis technique that it is related to the safety arguments of the modular safety case for an IEC 61058 compliant generic COTS processor [4].

References:

DREAMS, "Distributed Real-Time Architecture for Mixed-Criticality Systems - A Modular Safety Case for COTS device," in D5.1.2, ed, 2015.

IEC, "IEC 61508-2 Functional safety of electrical/electronic/programmable electronic safety-related systems - Part 2: Requirements for electrical/electronic/programmable electronic safety-related systems," ed: IEC, 2010.

4.3 Mixed-criticality Network

4.3.1 NoC Pattern

Pattern ID:		PAT –PNoC – 00
Pattern Name:	PNoC	
Related pattern:	N/A	
Type:	SW	
Context:		
<p>Mixed-criticality networks supports communication between the subsystems and elements of the systems. In domains such as avionics, railway, automotive, industrial control and medical systems, where functions of different criticality are integrated on a single embedded distributed computing platform, mixed-criticality networks are used for communication purposes. Protection mechanisms are a prerequisite for the integration of subsystems and elements with different criticality, thus avoiding interferences in spatial and the temporal domains. Furthermore, the communication among subsystems, elements and functionalities with different criticality level usually leads to issues related to interferences. For instance, a non-safety communication may cause interferences on safety-related communication.</p>		
Problem:		
<p>A mixed-criticality system can integrate functionalities with different criticality levels which may require communicating. Furthermore, the integration of functionalities with different criticality level can lead to issue interferences in general. For example, a non-safety subsystem can lead to a failure in a safety related subsystem.</p> <p>Different NoCs are suitable for safety-critical applications, providing support of TT, RC or BE traffic. The shift towards the use of NoC communication subsystems for mixed-criticality systems leads to recurrent challenges related to supporting of multiple types of communication as well as supporting applications with different criticality level. For instance, TTNoC networks do not support the transmission of event-triggered messages, whereas AEtheral NoC does not support the transmission of RC messages.</p>		
Solution under consideration:		
<p>This pattern aims to manage the prioritization of different criticality subsystem communication that performs scheduling, routing, traffic shaping and error detection. Figure 37 shows the integration of this pattern on a multi-core device, where it is located on top of a NoC.</p> <div><div><div>Partition 1 Safety</div><div>Partition 2 Non-Safety</div><div>Partition 3 Non-Safety</div></div><div>Hypervisor</div><div><div>Processor Core 0</div><div>Processor Core 1</div></div><div><div>Local On-Chip Memory</div><div>PNoC</div><div>Network Interface</div></div></div>		
<p>Figure 37: PNoC integration in a partitioned mixed-critical device. (Source [44])</p> <p>In accordance with the IEC 61508 safety standard, this pattern can be taken as a SCL network which is implemented on top of a black channel network. Therefore, it is assumed that parts of the communication channel (NoC) cannot be designed, implemented and validated according to a safety standard. Instead, the PNoC (SCL) shall be compliant to a safety standard (e.g., IEC 61508 and IEC 61784-3). It must fulfil the safety</p>		

requirements which are defined in the MSC for an IEC 61508 (IEC 61784-3) compliant generic mixed-criticality network [5].

The priority based NoC pattern shall provide the following requirements in order to schedule, route, shape traffic and detect errors:

- I) Multiple traffic types: TT and ET (BE and RC) traffic types shall be supported by this pattern.
 - Periodic transmission of TT messages offers predictable timing with minimal latency and no jitter.
 - BE messages do not have timing restrictions and fulfil requirements of non-safety applications.
 - RC messages offer a reasonable trade-off between resource reservation and latency.
- II) Compatibility to a wide range of NoCs: This pattern shall be integrable on a wide range of NoCs, enabling the system to support TT and ET communications, despite only event triggered (ET) transmission are supported by the underlying network.
- III) Support of hard-real time applications: This pattern shall ensure that messages of the system meet the pre-specified deadlines in all situations defined in [45]. For this purpose, this pattern shall provide a scheduler that enables to achieve deterministic communication.
- IV) Support of mixed-criticality system: The communication of applications with different criticality level that interact and coexist on a shared computing platform requires protection mechanisms that establish chip-wide segregation. The use of partitioning mechanisms such as hypervisors is not enough because non-safety partitions can influence to safety-related ones. Therefore, this approach shall provide rigid temporal and spatial partitioning by establishing a chip-wide partitioning.

In addition, a set of diagnosis techniques in compliance with IEC 61508-2 and IEC 61784-3 shall be provided in order to assure that all safety-related failures are detected and controlled. The safety-related requirements and diagnosis for a mixed-criticality network are defined in DREAMS D5.1.3 deliverable "A modular safety case for an IEC 61508 compliant generic mixed-criticality network" [5].

Board Name:

XILINX ZYNQ-7000 zc706

Implementation:

This pattern is implemented as additional HW layer on a networked and partitioned mixed-criticality multi-core device. The multi-core device is partitioned by means of XtratuM hypervisor [10], though other hypervisors can be used for the same purpose (e.g., PikeOS or Wind-River Hypervisor). The partitions generated by XtratuM (e.g., safety and non-safety) are integrated among the cores of the device. As stated in Section 3, the HW architecture used during this deliverable is based on a harmonized platform composed of a processing system (PS) and a programmable logic (PL). The PS is composed of two ARM Cortex A9 processors; instead the PL can be composed of a single or multiple soft-core processors. In addition, as stated in Section 4.2.1, the communication among partitions may be carried out through a shared memory, although in order to evict issues related to those memories (e.g., interferences), the STNoC is implemented. Figure 38 shows the implementation of the PNoC on the partitioned multi-core ZYNQ device.

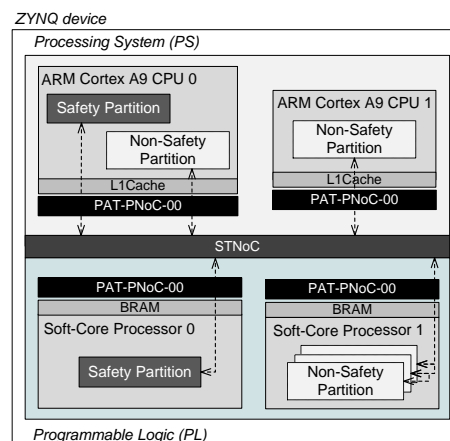
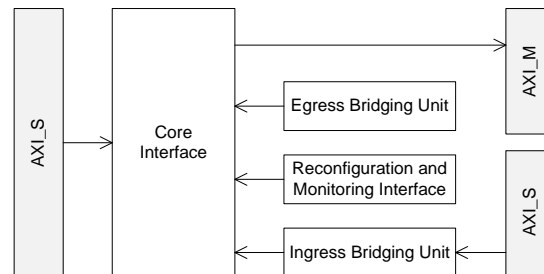


Figure 38: PNoC implementation on the ZYNQ device.

The PNoC which is implemented in the ZYNQ device makes use of AXI interfaces to communicate with the cores of the PL and the PL. In addition, it contains several clock domains which are used for the AXI transactions, to operate in different clock domains and synchronize the communication activities. Figure 39 shows the main building blocks of this pattern that are required to provide the requirements stated in previous section ‘Proposed solution’.

**Figure 39: Block diagram of PAT-PNoC-00. (Source [46])**

- **Core Interface:**

The primary purpose of the core interface is to provide buffers for storing the messages in both directions using I/O ports. Ports are self-contained I/O units which include on-chip memory (e.g. BRAM in the FPGA-based implementation) for storing the messages, registers for the configuration and status values, and a control unit for the operation of the port.

Moreover, the core interface plays an important role for managing the ET messages to be injected to the NoC, in order to simplify operation of the egress bridging unit. It contains priority queues and the configuration parameters which are useful to map the ports to the queues. For instance, one an RC or BE port signals the arrival of a new message, this unit reserves memory for the core within the respective priority queue. Thereafter, when an ET message is allowed to interleave between TT messages, the core interface checks the queues and triggers a dequeued signal of the port belonging to the highest priority. In this way the core interface guarantees the lowest delay for the ET messages of higher priority.

- **Egress Bridging Unit (EBU):**

The egress bridging unit assures the timely injection of TT messages into the NoC and facilitates interleaving of ET messages between TT messages. The EBU is composed of a TT scheduler and an ET interleaver. The TT scheduler controls the injection of TT messages by triggering the respective ports at predefined instants. In case of ET messages, they use the same priority as for the TT. The ET interleaver manages the interleaving of those messages in such a way that no TT message is affected by the ET messages. As shown in Figure 38, the STNoC that is employed as an underlying NoC on the DREAMS ZYNQ platform supports two prioritized virtual networks and guarantees bounded impact of low priority messages on high priority ones. Moreover, each priority owns its own PAT-PNoC-00 and NoC and uses separate AXI interfaces.

- **Reconfiguration and Monitoring Interface (RMI)**

The reconfiguration and monitoring interface is responsible for (re-) configuring the priority based NoC and act as the interface for reading the status of the pattern. The reconfiguration can be given at runtime.

- **Ingress Bridging Unit (IBU)**

The ingress bridging unit dispatches the incoming messages to the corresponding ports once a new message arrives at the AXI_S at the NoC side. The operation of the IBU is the same as the operation of the port selector within the core interface. The IBU employs a look-up table which maps the AXI write addresses with the port IDs.

Despite the STNoC is used for communicating safety and non-safety-related components, this cross-domain pattern can be used in conjunction with different NoCs, provided that the requirements stated before are

fulfilled.

Results:

Figure 40 shows the integration of the NoC cross-domain pattern in a partitioned multi-core mixed-criticality system, where this pattern is implemented by each processor core (ARM CPUs and micro-blaze processors) for enabling the communication of partitions with different criticality level through time-triggered and event-triggered messages.

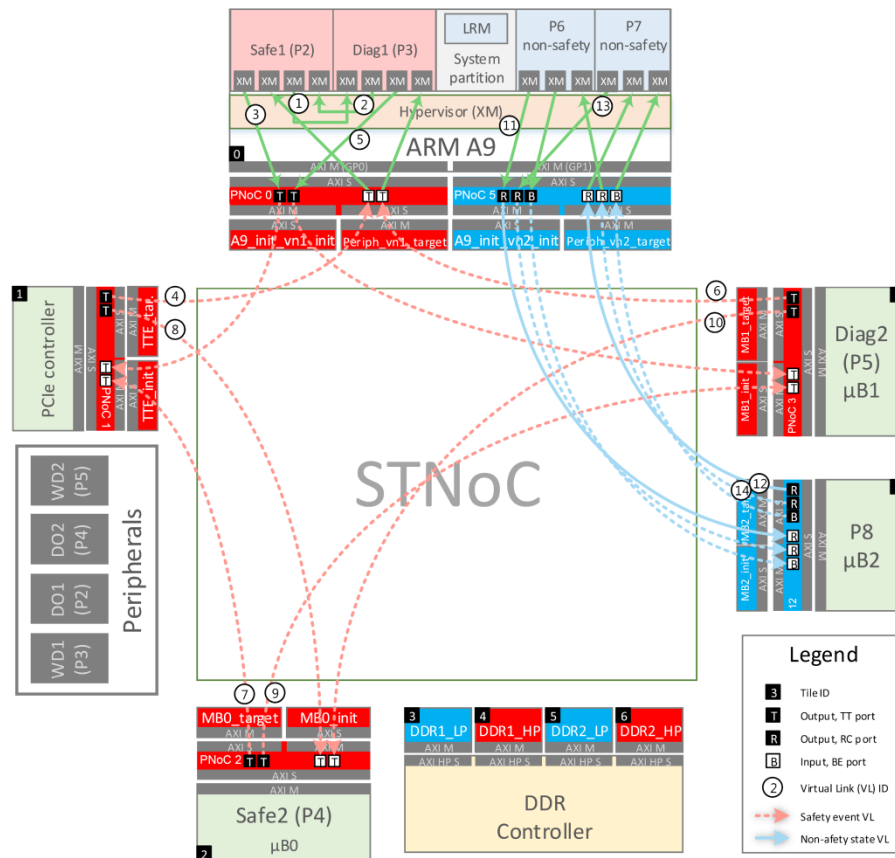


Figure 40: PNoC cross-domain pattern - Implementation system architecture.

During this implementation, PNoC 0 to PNoC 3 communicate through TT messages with a period of 976us. The remaining PNoCs are communicated through ET messages. These safety-related networks are configured as follows:

PNoC0				
Port ID	Type	Direction of the port	Destination PNoC ID	Destination port ID
0	TT	OUT	1	2
1	TT	OUT	7	2
2	TT	IN	1	0
3	TT	IN	7	0
PNoC1				
Port ID	Type	Direction of the port	Destination PNoC ID	Destination port ID
0	TT	OUT	0	2
1	TT	OUT	2	2
2	TT	IN	0	0
3	TT	IN	1	0
PNoC2				
Port ID	Type	Direction of the port	Destination PNoC ID	Destination port ID
0	TT	OUT	1	3
1	TT	OUT	7	3
2	TT	IN	1	1
3	TT	IN	7	1
PNoC3				
Port ID	Type	Direction of the port	Destination PNoC ID	Destination port ID

0	TT	OUT	0	3
1	TT	OUT	2	3
2	TT	IN	0	1
3	TT	IN	0	1
PNoC4				
Port ID	Type	Direction of the port	Destination PNoC ID	Destination port ID
0	RC	OUT	0	3
1	RC	OUT	0	4
2	BE	OUT	0	5
3	RC	IN	0	0
4	RC	IN	0	1
5	BE	IN	0	2
PNoC5				
Port ID	Type	Direction of the port	Destination PNoC ID	Destination port ID
0	RC	OUT	8	3
1	RC	OUT	8	4
2	BE	OUT	8	5
3	RC	IN	8	0
4	RC	IN	8	1
5	BE	IN	8	2

Figure 41: Configuration of PNoCs.

Additional Considerations:

This pattern is related to the modular safety case for an IEC 61508 compliant generic Mixed-Criticality Network [5] and D2.1.2 [46] and D2.1.3 [47] deliverables of DREAMS.

References:

DREAMS, "Distributed Real-Time Architecture for Mixed-Criticality Systems - A Modular Safety Case for Mixed-Criticality Network," in *D5.1.3*, ed, 2015.

5 Conclusions

Today's mixed-criticality systems based on multi-core architecture are composed of a wide variety of complex components that lead to an increase in their development and certification. On the other hand, today's safety-related standards such as the IEC 61508 standard do not consider measures and diagnostic techniques for these kinds of systems where functionalities with different criticality levels can be integrated into the same system.

The objective of this deliverable is to provide generic reusable solutions, measures and diagnostic techniques that ease the development and certification of multi-core mixed-criticality systems composed of virtualization mechanisms like hypervisors, COTS multi-core devices and mixed-criticality networks. Due to time limitations, we have only implemented some of the patterns defined in this deliverable. These patterns will be integrated in the wind-turbine demonstrator of WP7.

6 List of Open Points (LOP)

LOP – No.	Item	Title of document / Document Name / Version No. / Author / Date of issue / Index of Changes Description	inserted (date)	brought up by	Status	Comments by Customer	Comments by TUV
File A		D5 3 1 Cross Domain Mixed-Criticality Patterns_v0_0.docx, Version 0.0 dated 2015-11-09					
A1	chapter 3, page 12	"This chapter identifies common source of certification challenges in the development ..." Suggestion: "..... challenges among others in the development..."	30/11/2015	TUV-kg	3	02/12/2015 al: OK. Modified.	2016-06-07 TUV-Hei OK, completely modified. The whole document has to be re-reviewed.
A2	chapter 3.1, page 13	"... interconnection coherency management units..." Is this the generic expression for SCU / CoreNet? If so, please use it continuous within this document.	30/11/2015	TUV-kg	3	02/12/2015 al: OK. Modified.	2016-06-07 TUV-Hei OK, completely modified. The whole document has to be re-reviewed.
A3	chapter 3.1.1, page 14	"... , which are not applicable to multi-core systems...." Better: "..., which are not directly applicable to multi-core systems, but have to be extended according to the given conditions."	30/11/2015	TUV-kg	3	02/12/2015 al: OK. Modified.	2016-06-07 TUV-Hei OK, completely modified. The whole document has to be re-reviewed.
A4	chapter 3.1.2, page 14	"... the memory coherency between CPUs, L1 cache memories and L2 shared memories." Better: "...the data memory coherency between cores, L1 cache, L2 cache and (external) shared memories." Suggestion: We should use the word "core" instead of "CPU", just to use the same word for the same item.	30/11/2015	TUV-kg	3	02/12/2015 al: OK. Modified.	2016-06-07 TUV-Hei OK, completely modified. The whole document has to be re-reviewed.
A5	chapter 3.1.5, page 15	"...which may be diagnosed..." Better: "...which must be diagnosed..."	30/11/2015	TUV-kg	3	02/12/2015 al: OK. Modified.	2016-06-07 TUV-Hei OK, completely modified. The whole document has to be re-reviewed.
A6	chapter 3.2.2	Typo: underplaying => underlaying	30/11/2015	TUV-kg	3	02/12/2015 al: OK. Modified.	2016-06-07 TUV-Hei OK, completely modified. The whole document has to be re-reviewed.
A7	chapter 3.3,	"...network in order to correct and continue running." Suggestion: Delete "in order to correct and continue running", because this may be one option. Another	30/11/2015	TUV-kg	3	02/12/2015 al: OK. Deleted.	2016-06-07 TUV-Hei OK, completely modified. The whole document has to be re-reviewed.

		option is in safety applications to shut-down the network.					
A8	chapter 4.1.1,	"Critical memory areas should be protected..." Better: "Critical memory areas must be protected..."	30/11/2015	TUV-kg	3	02/12/2015 al: OK. Modified.	2016-06-07 TUV-Hei OK, completely modified. The whole document has to be re-reviewed.
A9	chapter 4.1.1	"This solution aims to avoid the issues related to the memory sharing..." It is not the intension of this pattern to avoid shared memory, but to provide "robust measures" and "diagnosis techniques" to detect any unauthorized access. Please remove "avoid the issues related to the memory sharing".	30/11/2015	TUV-kg	3	02/12/2015 al: OK. Modified.	2016-06-07 TUV-Hei OK, completely modified. The whole document has to be re-reviewed.
A10	chapter 4.1.2	Typo: "...for critical partitions, with could..." => "...for critical partitions, which could..."	30/11/2015	TUV-kg	3	02/12/2015 al: OK. Modified.	2016-06-07 TUV-Hei OK
A11	chapter 4.1.2	"This pattern aims to provide ...non-critical partitions." Suggestion to rephrase: "This pattern aims to provide a generic diagnosis pattern to detect interferences on criticality partitions that might be caused by non-critical partitions."	30/11/2015	TUV-kg	3	02/12/2015 al: OK. Modified.	2016-06-07 TUV-Hei OK
A12	chapter 4.1.3	"The DIOs can be accessed by several partitions at the same time, leading to errors." Suggestion to rephrase: "Several partitions may have access to DIOs at the same time, which can lead to errors."	30/11/2015	TUV-kg	3	02/12/2015 al: OK. Modified.	2016-06-07 TUV-Hei OK, completely modified. The whole document has to be re-reviewed.
A13	chapter 4.1.3	"The I/O server partition manages ...diagnosis, etc." Suggestion to rephrase: "The I/O server partition manages a configurable number of DIOs, each of them shall be commanded by one communication port. In safety applications the I/O server partition shall provide diagnostics according to the requested fault model, e.g. table A.1 IEC 61508-2." Only this way there is a real benefit of implementing the DIO Server as a generic pattern.	30/11/2015	TUV-kg/bo	3	02/12/2015 al: OK. Modified.	2016-06-07 TUV-Hei OK, completely modified. The whole document has to be re-reviewed.
A14	chapter 4.1.4	"... into account that	30/11/2015	TUV-kg	3	02/12/2015 al: OK.	2016-06-07 TUV-kg

		the communication networks..." -> the external communication				Modified.	Ok, added in v0.2
A15	chapter 4.1.4	"The communication of safety-related..." => "The communication between safety-related..."	30/11/2015	TUV-kg	3	02/12/2015 al: OK. Modified.	2016-06-07 TUV-Hei OK, completely modified. The whole document has to be re-reviewed.
A16	chapter 4.1.4	"This leads to association failures, because..." Suggestion to rephrase: "This may lead to communication errors,..."	30/11/2015	TUV-kg	3	02/12/2015 al: OK. Modified.	2016-06-07 TUV-Hei OK, completely modified. The whole document has to be re-reviewed.
A17	chapter 4.1.4	"... the underlying communication network of..." => "... the underlying communication layer of..."	30/11/2015	TUV-kg	3	02/12/2015 al: OK. Modified.	2016-06-07 TUV-Hei OK, completely modified. The whole document has to be re-reviewed.
A18	chapter 4.1.4	"... abstracted from the underlying platform architecture ." => "... abstracted from the underlying communication layer ."	30/11/2015	TUV-kg	3	02/12/2015 al: OK. Modified.	2016-06-07 TUV-Hei OK, completely modified. The whole document has to be re-reviewed.
A19	chapter 4.1.4	"... network approaches (see IEC 61508)." Better: "...network approaches (see IEC 61508 and IEC 61784-3)."	30/11/2015	TUV-kg	3	02/12/2015 al: OK. Modified.	2016-06-07 TUV-Hei OK, completely modified. The whole document has to be re-reviewed.
A20	chapter 4.2	"... which is normally coupled or internal to the..." Suggested rephrase: "... is normally coupled to the ..."	30/11/2015	TUV-kg	3	02/12/2015 al: OK. Modified.	2016-06-07 TUV-Hei OK, completely modified. The whole document has to be re-reviewed.
A21	chapter 4.2	"Therefore, two or more cores of a multicore processor..." This is exactly the reason why e.g. SCU is implemented. It seems that the described problem is already solved!	30/11/2015	TUV-kg	3	02/12/2015 al: OK. Modified.	2016-06-07 TUV-Hei OK, completely modified. The whole document has to be re-reviewed.
A22	chapter 4.2.1	"... shared memory is free of interferences, new measure and diagnosis techniques are required." As mentioned before measures are already implemented to control /mitigate exactly this fault scenario (SCU). But the question is: How can we be sure that the SCU works as expected? Which diagnostic techniques have to be implemented to cover a DC of 60, 90, 99%?	30/11/2015	TUV-kg	3	02/12/2015 al: OK. We have to think about it. 27/06/2016 al: Yes. You are right. All the references to the DC have been deleted. On the other hand, in relation to the diagnosis of the SCU, this pattern assumes that the SCU was checked in advance (PAT-CCMU-00 and PAT-ICMUD).	2016-06-07 TUV-heikg Clause deleted in v0.2. However, an answer is still open. 2016-06-24 TUV-boheikg: Basically the achieved DC is not documented in any of the pattern. From our point of view the corresponding evidences can be generated anyhow only based on a real implementation and should be exported to the system

							architect. This way this item may be closed.
A23	chapter 4.2.2	"...many copies of any one instruction operand saved in several caches." Suggested rephrase: "... to have many copies of data saved in several caches." Reason: To my knowledge, cache coherency will be only supported for data and not for instructions.	30/11/2015	TUV-kg	3	02/12/2015 al: OK. Modified.	2016-06-08 TUV-Hei OK, completely modified. The whole document has to be re-reviewed.
A24	chapter 4.2.3	"... , the interconnection management unit..." Is "interconnection management" the same as "CoreNet Coherency Fabric" stated in figure 4? If so, please use the same name.	30/11/2015	TUV-kg	3	02/12/2015 al: OK. Modified.	2016-06-08 TUV-Hei OK, completely modified. The whole document has to be re-reviewed.
A25	chapter 4.2.4	"...measures and techniques listed in IEC 61508 are..." Better: "... listed in IEC61508-2, table A.4..."	30/11/2015	TUV-kg	3	02/12/2015 al: OK. Modified.	2016-06-08 TUV-Hei OK, completely modified. The whole document has to be re-reviewed.
A26	chapter 4.3.1	"...could lead to issues related to temporal and spatial interferences" Suggestion to rephrase: "...could lead to issue interferences in general."	30/11/2015	TUV-kg	3	02/12/2015 al: OK. Modified.	2016-06-08 TUV-Hei OK, completely modified. The whole document has to be re-reviewed.
A27	chapter 4.3.1	"...in order to avoid unintended interferences in both temporal and spatial domains" Suggestion to rephrase: "... in order to ensure that the right data will be received by the right participant right in time."	30/11/2015	TUV-kg	3	02/12/2015 al: OK. Modified.	2016-06-08 TUV-Hei OK, completely modified. The whole document has to be re-reviewed.
A28	Abbreviated terms	Missing terms. Please add the following terms: BRAM, SEU, AXI, SEM	01/12/2015	TUV-bo	3	02/12/2015 al: Ok. Added.	2016-06-08 TUV-Hei OK
A29	chapter 4.2.2 - suggested fault avoidance measures	Example list of measures for fault avoidance in systems using cache coherence systems which may be included: - Limit shared memory usage to an absolute minimum required for operation o Limit the use of the use of multiple threads and tasks for one safety function to a minimum required o - Make sure that per potential cache line there is only one	01/12/2015	TUV-bo	3	02/12/2015 al: Ok. Included. 27/06/2016 al: OK. The fault-avoidance and fault-control techniques are included in Section 4.2.2. In addition, the techniques which are implemented by this pattern have been identified in Section "implementation".	2016-06-24 Bo: NOK: I do not see the fault avoidance measures listed in the section. I believe it is crucial for safety applications to keep the use of shared memories to a minimum and to be very clear on the used in the corresponding documentation of the project. Only where the complexity of the "problem" or limitation of

		task/process allowed to write and all other may only read (only 1:n communication allowed). The assignment should be defined statically.					computing resources require the use of shared resources/memories this should be allowed to be used.
A30	chapter 4.2.2 - suggested fault control measures	Example list of measures for fault control in systems using cache coherence systems which may be included (DCs to be discussed): - Implement communication protocol with additional messaging between sender and receiver of the information (e.g. o Order violation detection: Flags to show indicate updated information, and Flags to indicate that this information was received., o Data consistency violations: additional coding information like CRC/ECC or Parity Information in the same memory block where the actual updated information is stored. The flags must be updated as last write action to the shared memory). (It is safe to assume that also a HW implemented ECC/Parity on caches may have bugs (e.g. ARM: 751475—Parity error may not be reported on full cache line access (eviction / coherent data transfer / cp15 clean operations))) - Implement data structures that match the cache architecture (e.g. have max. size of one cache line - for optimal performance) and allow additional diagnostics (see above and below). - Facilitate Cache memory ECC and facilitate cache scrubbing, if applicable. - Implement timing expectations and error detection for the shared memory communication - Implement Other typical communication error related measures	01/12/2015	TUV-bo	3	02/12/2015 al: OK. Included. 27/06/2016 al: OK. The fault-control techniques are included in Section 4.2.2.127	2016-06-24 Bo: NOK. I do see the items only partially included in the "solution under consideration". I believe this section should be independant of the actual Hardware used and thus should list all possible solutions. In the "Implementation" section not all items may be required since there are other measures provided which already provide the corresponding diagnostics. In the v0.2 of the document previously added text of the document version from mid of march is missing. Please add this again. The section is starting with "In addition, the following measures for fault avoidance and fault control in systems using cache coherency may be included:"

		like: Sequence number, Addressing (could be done by different CRC codes as well), coding, timing expectation - Automatic invalidation of cache lines after a defined period of time in order to make sure the caches are flushed periodically.				
A31	chapter 4.1.1,	The naming of the pattern seems very DREAMS specific. Actually as far as I can see this chapter is talking about some kind of Memory server which is accessible via a safety communication protocol. It seems a good idea to have this chapter/pattern to be renamed in a more generic manner.	01/12/2015	TUV-bo	3	02/12/2015 al: OK. Changed, STNoC accessible memory area diagnosis pattern --> NoC accessible critical area diagnosis pattern
					3	
	File A1	D5 3 1 Cross Domain Mixed-Criticality Patterns_v0_2.docx, Version 0.2 dated 2016-06-07				
A1_1	Chapter 4.1.2, Critical Partition Diagnosis Pattern, Temporal interference	<p><u>The detection capabilities of the concurrency monitoring implementation are validated by means of a scenario where a faulty partition progressively causes an increasing amount of temporal interferences.</u></p> <p>When will the validation take place? At development time? Or is this also a diagnosis of the diagnosis during run-time? How is assured that the actual safety function is not impaired?</p>	TUV-Hei	08/06/2016	3	<p>27/06/2016 VB: The detection of the temporal interference is performed on-line as a diagnosis technique. «Validation» refers to the fact that we validated, by means of test cases, that the interference is indeed detected (a message «[P0-readerPSM.c:45] DetectMulticoreInterference counter 270414 below 270216 threshold» is generated).</p> <p>The diagnosis pattern does not define how to avoid that the safety function is impaired. However, for such purpose the partition developer can make use of the services provided by the hypervisor, like for example stopping the interfering partition using the XM_suspend_partition() hypercall.</p> <p>We can include the previous paragraph as clarification.</p>
A1_2	Chapter 4.1.2, Critical Partition Diagnosis Pattern, Spatial isolation	<p><u>The detection capabilities of the memory checksum implementation are validated by means of a scenario where a faulty partition randomly injects memory errors in the critical memory sections.</u></p> <p>When will the</p>	TUV-Hei	08/06/2016	3	<p>27/06/2016 VB: The detection of a problem in the spatial isolation is performed on-line as a diagnosis technique. «Validation» refers to the fact that we validated, by means of test cases, that the breach in spatial isolation is indeed detected (a message «[P0-critical.c:49] ChecksumDetect mismatch</p>

		validation take place? At development time? Or is this also a diagnosis of the diagnosis during run- time? How is assured that the actual safety function is not impaired?				<p>computed 204014F expected FCA9BE35» is generated).</p> <p>The diagnosis pattern does not define how to avoid that the safety function is impaired. However, for such purpose the partition developer can make use of the services provided by the hypervisor, like for example having a redundant partitions that detect the corruption in the other one, bring the system to a safe state and then restart the complete system by means of the XM_reset_hypervisor () hypercall.</p> <p>We can include the previous paragraph as clarification.</p>	
A1_3	Chapter 4.1.2, Critical Partition Diagnosis Pattern, Temporal isolation	<p><u>The detection capabilities of this solution are validated by means of a scenario where a faulty partition causes random temporal interferences to the critical partition.</u></p> <p>When will the validation take place? At development time? Or is this also a diagnosis of the diagnosis during run- time? How is assured that the actual safety function is not impaired?</p>	TUV-Hei	08/06/2016	3	<p>27/06/2016 VB: The detection of the temporal interference is performed on-line as a diagnosis technique. «Validation» refers to the fact that we validated, by means of test cases, that the interference is indeed detected (a message «[P0-critical.c:45] SlotStart drift detected slotstart 2000607 expected 2000000» is generated).</p> <p>The diagnosis pattern does not define how to avoid that the safety function is impaired. However, for such purpose the partition developer can make use of the services provided by the hypervisor, like for example stopping the interfering partition using the XM_suspend_partition() hypercall.</p> <p>We can include the previous paragraph as clarification.</p>	
A1_4	Chapter 4.1.3, Digital I/O Server Pattern	<p><u>Each digital input shall be checked to detect whether their values change.</u></p> <p><i>Each digital input shall be checked to detect whether their values are able to be changed.</i> I assume that you are not speaking about the normal sampling of the digital inputs but about the testing/diagnosis of the digital inputs.</p>	TUV-Hei	08/06/2016	3	<p>27/06/2016 al: OK. The sentence is corrected.</p>	
A1_5	Chapter 4.1.3, Digital I/O Server Pattern	<p><u>Change Frequency: Parameter that describes how often the input changes. The</u></p>	TUV-Hei	08/06/2016	3	<p>27/06/2016 al: OK. It is specified that the change frequency parameter is only suitable in high or</p>	

		<p><u>timeout before considering that an input is faulty will be four times the figure associated to this characteristic.</u></p> <p>The definition "four times the figure" is not suitable in all cases. E.g. an Emergency Stop has no change frequency at all because it is a low-demand operation (and not high demand or continuous mode of operation).</p>				continuous mode of operation and that it is not suitable in low mode of operation.	
A1_6	Chapter 4.2.2, 4.2.2 Cache Coherency Management Unit Diagnosis Pattern, "Solution under Consideration"	<p>It is not clear to me how a watchdog timer can detect message order violations in general. This may only be the case if you have a fix and defined communication schedule.. A simple measure would be to have a sequence number implemented.</p>	TUV-Bo	24/06/2016	3	27/06/2016 al: OK. It is specified that the WDT can only be used with fixed and defined communication schedule. In addition, we have included the sequence number technique that you propose.	
A1_7	Chapter 4.2.3, Solution under consideration	<p>Beside others it is assumed that: "The <u>coherency management unit</u> is correctly configured and checked in advance."</p> <p>Since the "coherency management unit" is part of SCU and SCU is part of Inter-connection units, it seems that the condition "checked in advance" is not correct, because checking the inter-connection units is exactly subject of this pattern chapter.</p>	TUV-kg	10/06/2016	3	27/06/2016 al: Ok. This statement has been deleted.	
A1_8	Chapter 4.2.3, Solution under consideration	<p>Check the configuration of the interconnect management unit</p> <p>"... configuration of their registers can be used to manage ..."</p> <p>Better: "... configuration of their registers will be used to manage ..."</p>	TUV-kg	10/06/2016	3	27/06/2016 al: OK. The sentence is corrected.	
A1_9	Chapter 4.2.3, Solution under consideration	<p>Check the configuration of the interconnect management unit</p> <p>A "inexpertly" modification of configuration registers is a systematic aspect, which is part of the verification activities during the development.</p>	TUV-kg	10/06/2016	3	27/06/2016 al: OK. The sentence is corrected.	

		Therefore please delete "which may be inexpertly modified".				
A1_10	Chapter 4.2.3, Solution under consideration	<p>"Systematic faults of the coherency management unit are analyses in DREAMS deliverable D5.1.2 "A modular safety case for COTS processor" [4] by means of a FMECA analysis."</p> <p>Which chapter in D5.1.2 is meant here? Please add a reference accordingly.</p>	TUV-kg	10/06/2016	3	<p>27/06/2016 al: OK. It is included the chapter where the FMECA analysis is defined in DREAMS deliverable D5.1.2. "...in Chapter 4.2.11.2.2 (Tables 41 and 43) of DREAMS deliverable D5.1.2 "A modular safety case for COTS processor" [4] by means of a FMECA analysis"</p>
A1_11	Chapter 4.2.3, Solution under consideration	<p>The possibility of systematic errors in the configuration of the interconnection management unit configuration should be addressed.</p> <p>A simple way would be to add a precondition ("...pattern assumes that:") like:</p> <ul style="list-style-type: none"> - The configuration of the interconnection management unit is sufficiently free of systematic faults due to the used development process. 	TUV-bo	24/06/2016	3	<p>27/06/2016 al: OK. The possibility of systematic errors in the configuration of the interconnection management unit is included. In addition, a reference to chapter 4.2.8.3 of deliverable D5.1.2 where the FMECA analysis of the configuration process is included.</p>
A1_12	Chapter 4.2.4, Solution under consideration	<p>Which chapter in D5.1.2 is meant here? Please add a reference accordingly.</p> <p>D5.2.1 covers in chapter 4.2.11.2.4.1, table 44 and 45 only random faults.</p>	TUV-kg	10/06/2016	3	<p>27/06/2016 al: OK. It is included the chapter where the FMECA analysis is defined in DREAMS deliverable D5.1.2. "...in Chapter 4.2.11.2.4 (Table 46) of DREAMS deliverable D5.1.2 [4] by means of FMECA's."</p>
A1_13	Chapter 4.2.4, Solution under consideration	<p>The possibility of systematic errors in the configuration of the interconnection management unit configuration should be addressed.</p> <p>A simple way would be to add a precondition ("...pattern assumes that:") like:</p> <ul style="list-style-type: none"> - The configuration of the interrupt controller is sufficiently free of systematic faults due to the used development process. 	TUV-bo	24/06/2016	3	<p>27/06/2016 al: The possibility of systematic errors in the configuration of the interrupt controller unit is included. In addition, a reference to chapter 4.2.8.3 of deliverable D5.1.2 where the FMECA analysis of the configuration process is included.</p>
A1_14	Chapter 4.3.1, Solution under consideration	<p>"... (SCL) shall be compliant to a safety standard (e.g., IEC 61508)."</p> <p>I recommend to reference also the IEC 61784-3, chapter 5.3,</p>	TUV-kg	10/06/2016	3	<p>27/06/2016 al: OK. The reference is included.</p>

		which lists all communication errors.				
A1_15	Chapter 4.3.1, Solution under consideration	<p>"... in compliance with IEC 61508 and IEC 61784 shall..."</p> <p>Better: "... IEC 61508-2 and IEC 61784-3..."</p>	TUV-kg	10/06/2016	3	27/06/2016 al: Ok. The references re modified to "IEC 61508-2 and IEC 61784-3".
A1_16	Chapter 4.3.1, Solution under consideration	<p>"It must fulfil the safety requirements which are defined in the MSC for an IEC 61508 (IEC61784-3) compliant generic mixed-criticality network"</p> <p>The communication errors as listed in IEC 61508-2, 7.4.11 and IEC 61784-3, 5.3 are not considered or addressed in Implementation or Results. Why they are not addressed?</p> <p>According to my understanding Implementation / Results covers the handling of TT and ET messages, but again, do not address the communication error scenarios.</p>	TUV-kg	10/06/2016	3	27/06/2016 al: OK. The communication errors listed in IEC 61508-2 and IEC 61784-3 are theoretically considered by this pattern. However, they are not implemented by the pattern. This pattern is under continuous development process and therefore, we expect that the communication errors listed in those standards will be implemented in a posterior version.

Abbreviated terms

AHB	Advanced High-performance Bus
APB	Advanced Peripheral Bus
ASB	Advanced System Bus
AXI	Advanced eXtensible Interface
AXI_HP	AXI High Performance
AXI_GP	AXI General Purpose
AXI_ACP	AXI Accelerator Coherency Port
BRAM	Block RAM
BE	Best-Effort
COTS	Commercial Off-The-Shelf
CRC	Cyclic Redundancy Check
DAS	Distributed Application Subsystem
DC	Direct Current
DIO	Digital I/O
DIOS	Digital I/O Server
DMA	Direct Memory Access
DREAMS	Distributed Real-time Architecture for Mixed Criticality Systems
E/E/PE	Electrical/Electronic/Programmable Electronic
EBU	Electronic Bridging Unit
ECC	Error Correcting Code
ET	Event Triggered
FIQ	Fast Interrupt Request
FMEA	Failure mode and effect analysis
FMECA	Failure mode, effect and criticality analysis
FMEDA	Failure mode, effect and diagnostic analysis
FPGA	Field Programmable Gate Array
GENESYS	GENeric Embedded SYStem Platform
GIC	Generic Interrupt Controller
HM	Health Monitoring
HW	HW
I/O	Input/Output
IBU	Ingress Bridging Unit
IOP	Input/Output Peripheral
IRQ	Interrupt Request
LOP	List of Open Points
MCS	Mixed-Criticality System
MMU	Memory Management Unit
NoC	Network-on-Chip
OCM	On-Chip Memory

PAT	Pattern
PL	Programmable System
PNoC	Priority based Network-on-Chip
PPI	Private Peripheral Interrupt
PS	Processing System
RC	Rate-Constrained
RMI	Reconfiguration and Monitoring Interface
SCL	Safety Communication Layer
SCU	Snoop Control Unit
SEM	Soft Error Mitigation
SEU	Single Event Upset
SGI	Software Generated Interrupt
SIL	Safety Integrity Level
SPI	Shared Peripheral Interrupt
SW	Software
TERESA	Trusted Computing Engineering for Resource constrained Embedded System Applications
TT	Time-Triggered
TTE	Time-Triggered Ethernet
TTNoC	Time-Triggered Network-on-Chip
VM	Virtual Machine
VMM	Virtual Machine Monitor
V&V	Verification and Validation
WP	Work Package
XM	XtratuM
P2P	Point to Point
XMCF	XtratuM Configuration File
ELF	Executable and Linkable Format
HAL	Hardware Abstraction Layer

Bibliography

- [1] GENESYS. (2008). *GENeric Embedded SYStem*. Available: <http://www.genesys-platform.eu/results.htm>
- [2] F. TERESA. (2011). *Trusted Computing Engineering for Resource Constrained Embedded System Applications*. Available: <http://www.teresa-project.org/>
- [3] DREAMS, "Distributed Real-time Architecture for Mixed Criticality Systems - State of the Art of Piecewise Certification of Mixed Criticality Systems," in *D5.5.1*, ed, 2014.
- [4] DREAMS, "Distributed Real-Time Architecture for Mixed-Criticality Systems - A Modular Safety Case for COTS device," in *D5.1.2*, ed, 2015.
- [5] DREAMS, "Distributed Real-Time Architecture for Mixed-Criticality Systems - A Modular Safety Case for Mixed-Criticality Network," in *D5.1.3*, ed, 2015.
- [6] DREAMS, "Distributed Real-Time Architecture for Mixed-Criticality Systems - A Modular Safety Case for Hypervisor," in *D5.1.1*, ed, 2015.
- [7] DREAMS, "Distributed Real-time Architecture for Mixed Criticality Systems: Architectural Style of DREAMS D 1.2.1," July 2014.
- [8] J. Perez, D. Gonzalez, S. Trujillo, A. Trapman, and J. M. Garate, "A safety concept for a wind power mixed-criticality embedded system based on multicore partitioning," in *Functional Safety in Industry Application, 11th International TÜV Rheinland Symposium*, Cologne, Germany, 2014, p. 36.
- [9] J. Perez, D. Gonzalez, C. F. Nicolas, T. Trapman, and J. M. Garate, "A safety certification strategy for IEC-61508 compliant industrial mixed-criticality systems based on multicore partitioning," *Euromicro DSD/SEAA*, vol. Verona, Italy, August 2014.
- [10] FENTISS. (2014, February). *Hypervisor*. Available: <http://www.fentiss.com/en/products/hypervisor.html>
- [11] D. Dasari, B. Akesson, V. Nelis, M. A. Awan, and S. M. Petters, "Identifying the sources of unpredictability in COTS-based multicore systems," in *Industrial Embedded Systems (SIES), 2013 8th IEEE International Symposium on*, Porto, Portugal, 2013, pp. 39-48.
- [12] L. M. Kinnan, "Use of multicore processor in avionics systems and its potential impact on implementation and certification," in *Digital Avionics Systems Conference, 2009. DASC '09. IEEE/AIAA 28th*, Orlando, Florida, 2009, pp. 1.E.4-1 - 1.E.4-6.
- [13] P. Radojkovic, S. Girbal, A. Grasset, E. Quiñones, S. Yehia, and F. J. Cazorla, "On the evaluation of the impact of shared resources in multithreaded COTS processors in time-critical environments," *CM Transactions on Architecture and Code Optimization (TACO) - HIPEAC Papers*, vol. 8, January 2012.
- [14] J. Bin, S. Girbal, D. Gracia Perez, A. Grasset, and A. Merigot, "Studying co-running avionic real-time applications on multi-core COTS architectures," presented at the *Embedded Real Time Software and Systems (ERTS), Conference on*, Toulouse, France, 2014.
- [15] F. Semiconductor, "P4080 Development System User's Guide," Freescale Semiconductor August 2010.
- [16] XILINX, "ZYNQ-7000 All Programmable SoC: Technical Reference Manual," September 2014.
- [17] T. Instruments, "Safety Manual for TMS570LS31x and TMS570LS21x Hercules ARM Safety Critical Microcontrollers - User Guide," April 2013.
- [18] Freescale, "Safety Manual for Qorivva MPC5643L," April 2013.
- [19] IEC, "IEC 61508-2 Functional safety of electrical/electronic/programmable electronic safety-related systems - Part 2: Requirements for electrical/electronic/programmable electronic safety-related systems," ed: IEC, 2010.

- [20] IEC, "IEC 61508-3 Functional safety of electrical/electronic/programmable electronic safety-related systems - Part 3: Software requirements," ed, 2010.
- [21] IEC, "IEC 61784-3 Industrial communication networks – Profiles – Part 3: Functional safety fieldbuses – General rules and profile definitions," ed, 2010, p. 132.
- [22] IEC, "IEC 62280-1 Railway applications – Communication, signalling and processing systems – Part 1: Safety-related communication in closed transmission systems," ed, 2002, p. 36.
- [23] IEC, "IEC 61508-1 Functional safety of electrical/electronic/programmable electronic safety-related systems - Part 1: General Requirements," ed: IEC, 2010.
- [24] V. S. Alagar and R. Missaoui, "Object-Oriented Technology for Database and Software Systems," ed, 1995, pp. 295-312.
- [25] B. Rubel, "Patterns for generating a layered architecture," in *Pattern languages of program design*, J. O. Coplien and D. C. Schmidt, Eds., ed: ACM Press/Addison-Wesley Publishing Co., 1995, pp. 119-128.
- [26] D. Riehle and H. Züllighoven, "A pattern language for tool construction and integration based on the tools and materials metaphor," in *Pattern languages of program design*, J. O. Coplien and D. C. Schmidt, Eds., ed: ACM Press/Addison-Wesley Publishing Co., 1995, pp. 9-42.
- [27] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*.: Wesley, Addison, 1994.
- [28] S. S. Adams, "Functionality ala carte," in *Pattern languages of program design*, O. C. James and C. S. Douglas, Eds., ed: ACM Press/Addison-Wesley Publishing Co., 1995, pp. 7-8.
- [29] B. P. Douglass, *Doing hard time: developing real-time systems with UML, objects, frameworks, and patterns*: Addison-Wesley Longman Publishing Co., Inc., 1999.
- [30] B. P. Douglass, *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems*: Addison-Wesley Longman Publishing Co., Inc., 2002.
- [31] W. Wu and T. Kelly, "Safety tactics for software architecture design," in *Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International*, 2014, pp. 368-375.
- [32] R. Hammett, "Flight-Critical Distributed Systems - Design Considerations [avionics]," *Aerospace and Electronic System Magazine*, IEEE, vol. 18, pp. 30-36, June 2003.
- [33] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The Mälardalen WCET benchmarks – past, present and future," in *Worst-Case Execution Time Analysis (WCET), International Workshop on*, Dagstuhl, Germany, 2010, pp. 136-146.
- [34] M. Paolieri, E. Quiñones, F. J. Cazorla, and M. Valero, "An Analyzable Memory Controller for Hard Real-Time CMPs," *Embedded Systems Letters, IEEE*, vol. 1, pp. 86-90, December 2009.
- [35] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter, "A scalable and high-performance scheduling algorithm for multiple memory controllers," in *High-Performance Computer Architecture (HPCS), International Symposium on*, Bangalore, India, 2010, p. 12.
- [36] H. Shah, K. Huang, and A. Knoll, "Timing Anomalies in Multi-core Architectures due to the Interference on the Shared Resources," in *Design Automation (ASP-DAC), Asia and South Pacific Conference on*, Singapore, 2014, pp. 708-713.
- [37] ARM, "ARM Generic Interrupt Controller (GIC): Architecture Specification v1.0," September 2008.
- [38] J. Hussein and G. Swift, "Mitigating Single-Event Upsets," in *WP395*, ed, 2012, p. 10.
- [39] IEC, "IEC 61508-7 Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 7: Overview of techniques and measures," ed: IEC, 2010.

-
- [40] DREAMS, "Distributed Real-Time Architecture for Mixed-Criticality Systems -Preliminary Assessment Report Related to Improving or Calibrating the Technological Results," in *D7.3.1*, ed, 2014.
 - [41] FENTISS, "XtratuM Hypervisor for ARM CORTEX-A9: Volume 2: User Manual," ed, 2016.
 - [42] ARM, "Cortex - A9 MPCore: Technical Reference Manual," 2.0 ed, 2009, p. 122.
 - [43] DREAMS, "Distributed Real-Time Architecture for Mixed-Criticality Systems: Modular Safety Case for COTS processor," in *D5.1.2*, ed, 2015.
 - [44] H. Ahmadian and R. Obermaisser, "Time-triggered extension layer for on-chip network interfaces in mixed-criticality systems," presented at the Digital System Design (DSD), Euromicro Conference on, Madeira, Portugal, 2015.
 - [45] H. Kopetz, *Real-Time Systems: Design principles for distributed embedded applications*: Springer, 2011.
 - [46] DREAMS, "Distributed Real-time Architecture for Mixed Criticality Systems System-level executable specifications of a) virtualization and memory interleaving support of the Spidergon STNoC backbone at the network interface layer and b) a bus-to-noc bridge macro-architecture for seamlessly interconnecting Spidergon STNoC and network gateways from WP3 D 2.1.2," in *D2.1.2*, R1-0 ed, 2015, p. 34.
 - [47] DREAMS, "Distributed Real-time Architecture for Mixed Criticality Systems RT-level design specifications of a) virtualization and memory interleaving support of the Spidergon STNoC backbone at the network interface layer and b) a bus-to-noc bridge for seamlessly interconnecting STNoC to the network gateway from WP3 D 2.1.3," in *D2.1.3*, ed, 2015, p. 51.