Journal of Universal Computer Science, vol. 21, no. 2 (2015), 177-205
submitted: 20/5/14, accepted: 28/1/15, appeared: 1/2/15 © *J.UCS*

Safe Motor Controller in a Mixed-Critical Environment with Runtime Updating Capabilities

José Luis Gutiérrez-Rivas
(University of Granada, Granada, Spain
jlgutierrez@ugr.es)

Simon Holmbacka
(Åbo Akademi University, Turku, Finland
sholmbac@abo.fi)

Miguel Méndez-Macías
(Seven Solutions Inc., Granada, Spain
mmendez@sevensols.com)

Wictor Lund, Sébastien Lafond, Johan Lilius
(Åbo Akademi University, Turku, Finland
wictor.lund@abo.fi, sebastien.lafond@abo.fi, Johan.Lilius@abo.fi)

Javier Díaz-Alonso
(University of Granada, Granada, Spain
jda@ugr.es)

Abstract: Safety-critical systems and certification standards are the bare essential elements for the development process of avionics, automotive and industrial embedded systems. The necessity of including non-safety capabilities to reduce the price of these systems has resulted in a new type of critical systems, the mixed-criticality ones. These systems should be able to execute safety-critical applications but, at the same time, to run non-safety-critical functionalities without affecting the integrity of the safety-critical tasks. This paper presents a new system architecture which includes safety-critical and non-safety-critical parts in order to form a mixed-criticality system. The system consists of a reliable platform with a dual-core processor (implemented using a FPGA) architecture designed as open-hardware, running two isolated real-time operating systems which are connected through a safe core-to-core communication channel that executes the safety-critical applications. Moreover, the safety-critical system is connected to an external processor, an ARM9, which is used as an external sensing system. The ARM9 runs the non-safety-critical applications and allows the system to insert modifications updating without affecting the safety capabilities of the safety-critical part. This platform is described providing evidences of the isolation between safety-critical (SC) and non-safety-critical (NSC) applications, as well as describing an updating methodology for non-safety-critical applications. This system is validated using a complete and reliable application for safe emergency stop applications for industrial machinery.

Key Words: Mixed Critical, Safety Critical, FPGA, Runtime Updating Mechanism, Real-Time Operating System, Isolation
Category: D.2.11, B.1.3, C.3

1 Introduction

The simplification of the development process in safety-critical (SC) systems has been the center of attention of many companies and research centers working on avionics, automotive and industrial applications. The safety requirements impose significant time and cost overhead over conventional design procedures which many companies strive to minimize. In the past, processor trends were dominated by the increase of complex feature sets, higher clock speeds, growing thermal envelopes and power dissipation (super-scalar micro-controllers). This solution is not longer suitable because of hardware limitations and power/thermal dissipation issues caused by frozen clock speeds. However, markets and applications demand performance increase, safety and low energy consumption, which is closely related to critical embedded systems [Parkhurst et al. 2006].

Not many years ago, single-core was the most common hardware architecture for critical embedded systems but instead, there are many advantages of using multi-core/multi-processor systems: the single-core obsolescence and the lately business philosophy which aims to the increase of performance and the reduction of costs in the development process of these systems [Parkhurst et al. 2006]. Additionally, the introduction of multi-core processing platforms into these kind of systems poses an important challenge at different levels specially related to the management of shared resources. These issues mainly affect system-level scheduling [Zhuravlev et al. 2010], interferences between the low-critical and high-critical tasks [De Niz et al. 2009] as well as the architecture communication, which need to be solved at hardware and software level to properly guarantee that multiple processes running on the multi-core can fulfill the real-time constraints required by the SC applications without affecting each other [Abdelhalim et al. 2011].

A trend solution lately proposed has been the utilization of mixed-criticality architectures on the same processor, in which different levels of reliability and criticality can be achieved [Pellizzoni et al. 2009, Cuenca et al. 2011]. While traditionally mixed-criticality systems were based on spatial separation of SC and non-safety-critical (NSC) tasks using different hardware processors, current trends aim towards using different mechanisms as temporal isolation [Baruah et al. 2010] in order to share hardware resources. A goal is to find mechanisms that provide and prove isolation between NSC and SC parts. This allows the development of NSC applications of mixed-criticality systems in a simple and less expensive way and, additionally, to upgrade NSC parts of the system without requiring a re-certification process [Kelly et al. 2003]. These problems become even more crucial in critical systems which require a strong validation and verification process.

Different mechanisms are available to increase and achieve safety properties but the focus in this paper is the utilization of identical or diverse redundancy concepts for both hardware and software [Yeh et al. 2001, Powell et al. 2010]. Diverse redundancy refers to using two or even more different subsystems, which are built with different components, algorithms, electronics, design methodology, etc., to perform the same task. One of the benefits derived from the utilization of diverse redundancy is the increased capability to reduce common mode and systematic failures, such as those caused by design flaws. Software diversity was called into question for not being able to prevent system errors [Knight et al. 1990], for this reason, the development of a hardware-software architecture combining diversity techniques reduces the errors that can be correlated from software-like diversity solutions [Laprie et al. 1990]. This technique represents an effective defense against hidden dangerous faults, thus decreasing the probability of system failure in a safe state (Safe Failure Fraction). Logic solver technologies [IEC61511 2006], which use internal diverse redundancy have been developed for applications up to Safety Integrity Level (SIL) 3. This mechanism is one of the methodologies recommended by IEC61508 and IEC61511 standards in order to increase safety integrity of programmable electronic systems [IEC61508 2006].

The utilization of multi-core architectures involves the description of new requirements related to isolation and partitioning mechanisms that should be complied to achieve safety. First, a reliable communication connection (Core-to-Core communication) needs to be implemented in order not to lose the features that multi-core provides. In addition, a software system scheduling must be performed since different resources are shared by different processors, leading to system collisions and failures. As well as design methodologies, development tools should be adapted to these architectures since parallelism features and the possibility of concurrent processes are key points for multi-processors.

For the development we describe in this paper, different requirements have been taken into account. These requirements were extracted from the work accomplished in an Artemis JU project, RECOMP [RECOMP]. Our implementation complies with the development recommendations of three certification standards: IEC 61508, DO-254 [DO-254 2012] and DO-178C [DO-178C 2012]. Section 1.1 defines briefly these standards.

1.1 Certification Standards

By means of the industry domain, the IEC61508 describes requirements to prevent failures caused by hazardous events and to control failures by ensuring safety, even when faults are present. Additionally, the standard provides requirements for product's overall safety life-cycle. It specifies four discrete SIL levels of safety performance for a safety function. SIL 1 is the lowest level of safety integrity, and SIL 4 is the highest level.

Concerning avionics, DO-178B/C is a document used by the US Federal Aviation Administration (FAA) to determine the conditions in which software, that is required to be certified, is able to run, safely and reliable, in an airborne environment. This software standard can be accompanied by DO-254, which is used as a guide for the avionics hardware and electronics development process.

DO-254 standard is involved in the compliance for the design of complex electronic hardware of airborne systems. Complex electronic hardware includes devices like Field Programmable Gate Arrays (FPGAs) and Programmable Logic Devices (PLDs). This standard specifies the requirements for both design assurance and certification processes. Hardware design verification and validation need to be accomplished independently, which means that hardware designers should ensure that the design fulfills the defined system functionality and the verification team should verify that all of the derived requirements from the standard are met.

The integration and evolution of new technologies into aerospace, automotive and industrial domains have created the need for the adaptation of the certification process as soon as possible. One of these new trends is the introduction of multi-core systems on industrial and avionics HW/SW architectures providing the possibility of running multiple partitions concurrently on a computing platform, increasing the flexibility of the system, easing system upgrades and customization possibilities [RECOMP]. Next Section 1.2 introduces the certification issues for multi-core systems and how they can be undertake as part of our work.

1.2 Adapting Certification Standards from Single-core to Multi-core

The main issue related to multi-core is that certification guides are clearly defined for single-core but not for multi-core, leading to new challenges and problems that must be addressed. In most of the cases, critical functions must run independently on a single-core with dedicated resources but in other cases, they are exposed to coexist together with non-critical functions (mixed-critical applications). When this occurs, allocation technologies, partitioning and isolation methodologies must be extended from single-core standards to justify the spatial and temporal independence between each partition in multi-core systems [Abdelhalim et al. 2011].

The use of multi-core technologies is also a challenge for mixed-critical systems. The main issue is to make possible the coexistence of different criticality levels on the same computing platform. In these systems, low-critical and high-critical applications must share processing resources and time maintaining criticality properties. Unfortunately, this makes even more expensive and complicated the certification process, including the adaptation of certification standards, since it requires less criticality components to be certified at the highest criticality level.

In terms of costs, the certification process of SC applications is an expensive and complicated issue. This process normally increases development costs anywhere from 25 percent to 100 percent [IBM Software Rational 2010] unless isolation between NSC and SC parts becomes proven. NSC parts can offer user experience, flexibility and dynamism in software (graphical user interfaces, Ethernet broadcast messages, etc), which are highly valued from a user's point of view. Such functionalities focused on system monitoring can be implemented as NSC software. As long as this NSC software is isolated from the SC part, there is no risk for injury to the users or environment [RECOMP D.4.2b.1 2013], thus certification is not required for NSC. This leads to important cost savings in the certification process (0 percent for the isolated NSC part) of the whole system in case only the NSC application requires modification.

For this reason, the development of monitoring features as NSC applications is a low-cost solution compared to monitoring SC solutions. Moreover, developers may focus on the application features without taking care of safety limitations. Our work have been performed by following this alternative for the development of the NSC part of the mixed-criticality system using a multi-core architecture.

This paper presents a mixed-critical multi-core architecture in which different approaches have been developed to satisfy certification standard requirements by means of hardware, gateware and software. It is based on three elements:

- A multi-core open hardware platform capable of isolating fault propagation from the NSC part to the SC part while still providing communication [Mendez et al. 2013]. It includes a specific gateware specially developed for the work described in this paper.
- A SC application with compliances according to SIL3 level in the IEC61508 standard.
- A NSC application allowing communication with the SC part and ability to update software during runtime.

A diverse architecture implementation for hardware and software has been designed following the recommendations from certification guides for the implementation of the SC part of our mixed-critical system. The NSC component is based on a sensing application with run-time capabilities that is not crucial for correct execution of the system. To further increase the flexibility of the NSC software, it implements a runtime updating mechanism. Using this mechanism, software developers are able to patch the running software without restarting the system or application. This is a desired feature in for example complex machinery, pulping plants, mills etc. [Toivonen et al. 1990] since a system reboot can be very time consuming.

The rest of the sections of the paper are structured as follows: section 2 introduces the related work for mixed-critical systems. Section 3 describes the base of our work that involves the description of a case study: a safety emergency stop for industrial machinery. The hardware used for this implementation is described in Section 4.1 and Section 4.2 presents the software design and implementation. Finally, we state the results and conclusions of our work in Sections 5 and 6, as well as the future work that derives from the presented implementation.

2 Related Work

Mixed-criticality exists in several forms: certification wise (as in this paper), application wise (mapping based priority levels), processor wise (mapping based on processor type) etc [Shariful et al. 2009]. The fundamental commonality is, however, to secure the execution for the *higher priority* part independent of the low priority behavior. Wasicek et al. [Wasicek et al. 2010] present a SoC platform for executing mixed-criticality applications in which a Trusted Computing Base (TCB) is used to isolate a critical part from misbehaving components outside the TCB. Complimentary to this work, our mixed-criticality architecture can be set-up to manage for example data isolation with trusted memory spaces [Bate et al. 2003]; this is done rather than having a dedicated software part to intercept faulty behavior due to increased speed and less resource use.

To schedule mixed-criticality applications Mollison et al. [Mollison et al. 2010] suggested a multi-level scheduling mechanism for multi-core systems. The very scheduling technique is dedicated to a certain criticality level of the application used. High criticality applications are, for example, set to use only local static scheduling, while the lowest criticality levels use global best effort scheduling. A problem when using multi-core SMP scheduling in critical systems is to guarantee resources for critical applications in form of CPUs, OS resources, memory bandwidth when using inter-core locking.

One of the solutions proposed for the shared resources for multi-cores is virtualization. It consists in assigning access to shared resources by means of time or space. A shared source can be owned by a process for a slice of time or can be mapped only to a certain region of memory [RECOMP D.4.2b.1 2013]. The most popular solution for memory management in RTOS is the utilization of hypervisors.

In our architecture, using a hypervisor was not necessary because of the utilization of an asymmetric multi-core OS (AMP) which maps one time sharing scheduler on each core in separate memory blocks (FPGA). No problems caused by inter-core locking is therefore present in the OS since all OS resources are requested from the local-core OS and inter-core communication is done explicitly via two mailboxes.

A SoC design for mixed-criticality applications, in which hardware and functional isolation mechanisms are used to guarantee correct execution for critical applications in a pacemaker, is presented in [Pellizzoni et al. 2009].

The CPU provides memory protection for shared scratchpad memories and functional isolation is provided by online monitoring. In contrast to this platform, our shared resources are directly controllable by the safe FPGA and all non-safe applications are running on the external ARM9. The utilization of this ARM9 processor together with the AMP FPGA architecture provides this platform with diversity, which is helpful to increase reliability in terms of fault-avoidance [Lala et al. 1994]. In addition to this, this design provides isolation by design and allows the safe FPGA to operate completely independent of the external ARM9.

3 System Description

Having introduced safety-critical, mixed-criticality embedded systems and the certification process that these systems require, we describe the application scenario we have developed for the design of a mixed-criticality application. The main idea of our case studies is to create a system that includes both existing criticality systems: a SC part and a NSC part.

Certification standards stress that there are two procedures to include NSC elements in a SC application. One of them resides in certificating the NSC at highest priority level of the SC part, and the other is to isolate the NSC application from the SC part. The latter methodology reduces certification costs as no certification is required for the development of the NSC part. Updating only the NSC part would be exempt from certification too. For this reason, our design focuses on isolating between applications.

Our isolation design starts with the separation of hardware elements and mapping shared devices independently to ensure a correct access behavior to them. Hardware isolation is performed by allocating the SC and the NSC parts in different components of the platform. The SC application is allocated on two FPGA soft-processors and the NSC part runs on an external ARM9 processor with its private memory and OS as Fig. 5 shows.

With this configuration, isolation is assured by means of hardware but we need to rise this property up to software [Zhuravlev et al. 2010]. This ARM-FPGA solution provides the system with heterogeneous and diverse elements, since it is running on two different isolated processors interconnected with a shared memory [Lala et al. 1994]. The FPGA hardware architecture (Fig. 5) has been developed as a prototype for the first development stage. Next step is the migration of this design to a new implementation with physical processors.

The OS used for this system is a Real-Time Operating System (RTOS) for embedded systems, called OpenRTOS [OpenRTOS]. OpenRTOS is a RTOS de-

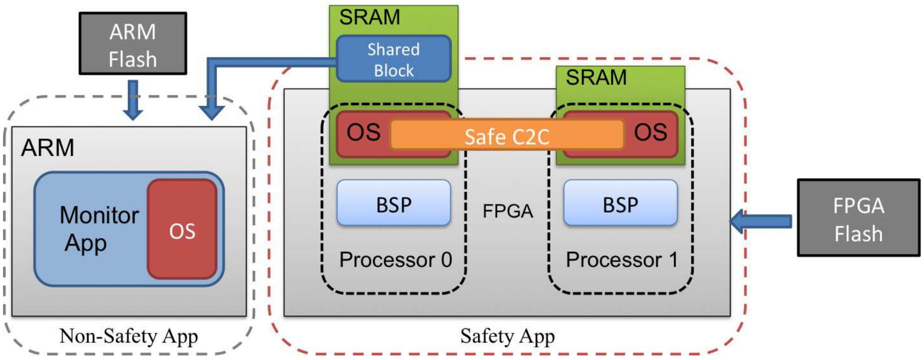

Figure 1: *System architecture of the complete application. Each FPGA processor runs an instance of a RTOS (OS) with its correspondent Board Support Package (BSP) configuration. Moreover, both processors are connected via a safe core-to-core (C2C) communication channel. The monitor application (ARM) runs another independent RTOS (OS) and reads data from the FPGA through a SRAM shared block.*

veloped by Wittenstein High Integrity Systems [Wittenstein]. The benefits of using this RTOS regards to the possibility of parallelizing functions/tasks in addition to a priority system that stands out for its simplicity and predictability. It is used for both SC and NSC parts and contains a simple scheduler which shares the execution time of tasks on the local CPU core.

Using a scheduler is necessary to parallelize access to shared memories and of course to make the tasks work "concurrently" for the emergency stop of the system, which involves up to 13 different OpenRTOS tasks at the same time. The utilization of a scheduler is also useful for parallelizing access to FPGA devices and the tasks that are involved in the core-to-core (C2C) communication between the two soft-processors that conform the SC part of the system and must be executed concurrently. The main objective of this application is the diagnostic of a safety function and the capture of system monitoring measurements. These values are later processed by the NSC part of the system, the sensing application.

Besides software, the hardware platform is also an important element in this system. The hardware platform on which we have decided to implement the mixed-critical multi-processor architecture, is called Avionic Computing Platform (ACP), and has been developed by Seven Solutions [Seven Solutions Inc.] in the framework of RECOMP project. This development platform is described in details in Section 4.1. Sections 3.1 and 3.2 describe the SC and NSC parts of our system in terms of design and specification in more detail.

3.1 Case Study: SC Emergency Stop

The SC part of the system has been developed according to the description and specification of a case study presented by Danfoss [Berthing 2012]. It is used as a basic, complete and real (rich in safety-critical and certification concepts) example for the RECOMP project.

This case study consists in the removal of the torque from an industrial motor. A misbehavior of the machinery could put human lives on risk. To prevent any further damage, these systems are provided with an emergency button that generates an *Emergency Stop* (ES) signal, which must be monitored and implemented following the industrial standard IEC61508. It describes the necessity of using a redundant architecture to process the ES signal and control the status of the system. IEC61508 recommends using a dual channel *1 out of 2* (1oo2) as the safety control architecture for this type of machinery. It minimizes the effect of dangerous failures using two independent processors.

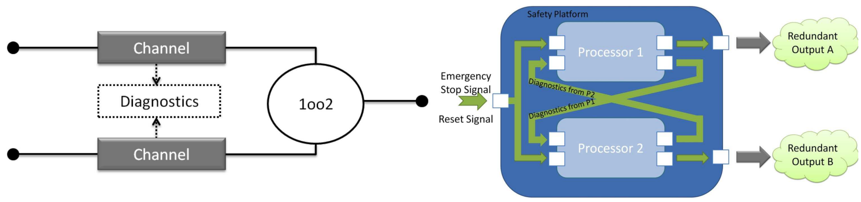

Figure 2: *Redundant and cross-comparison methodology used in a safe channel architecture 1oo2. Left: Concept design of a safe channel architecture 1oo2 extracted from IEC61508:2010. Right: Our design of a safe 1oo2 channel using two processors.*

Our implementation develops the 1oo2 channel using two MicroBlaze soft-processors that perform a cross-comparison diagnose of the ES signal as IEC61508 states (Fig. 2) for SIL3 applications. This process results in the activation of the safety function that performs the removal of the torque, the Safe Torque Off (STO) function. After the activation of the STO from any of the processors, the industrial motor removes the torque and the system halts.

The two processors of the 1oo2 architecture receive the same inputs: an ES signal and the data related to the STO function from the other processor. The data go through two diagnose processes, a cross-comparison function for local and external STOs, and another for the ES input and non-STO related variables as seen in Fig. 3. The diagnostics module includes also a liveness check routine implemented as a counter. The two cross-comparison modules together with the liveness check routine conform the failure detection system on both processors.

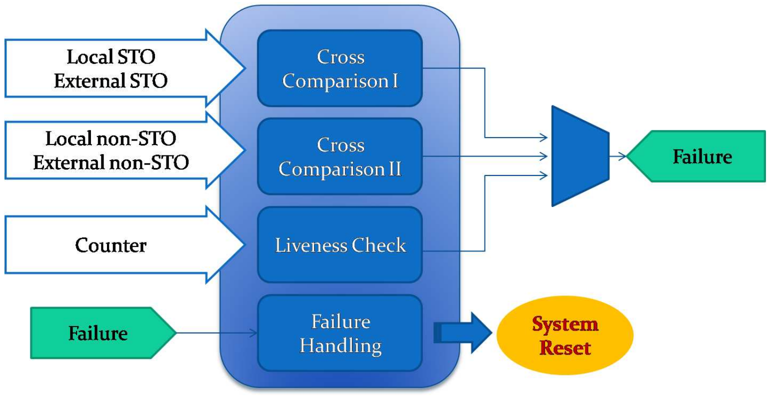

Figure 3: *Processor's diagnostic module. Each processor implements a diagnostic module to detect system failure. It consits in four components: two cross-comparison functions, one for the local and external STOs and another for non-STO related signals, an external counter to check the other processor is alive and a fourth function that detects when any of the other components rises an error signal.*

The functioning of the SC application starts with a power-up stage that checks the availability and correctness of the peripherals. After a successful power-up phase, the system runs normally. It executes several diagnostic tasks used to ensure the correct operation of the system and then, the sensing application captures and evaluates the data for monitoring. The behavior and measurement treatment of the NSC sensing application are described in Section 3.2.

In case of emergency, the ES button is pressed by a machine operator. Immediately after, each processor activates the STO function so that the torque of the motor is removed. Each processor evaluates the ES input, its local STO, and the external STO from the other processor. As Fig. 4 shows, the exchange of data between both processor is accomplished using a Core-to-Core (C2C) library guaranteeing SIL3 for the communication [C2C]. After the cross-comparison diagnose processes the STO from both processors are activated.

Finally, both STO outputs are connected to a signal analyzer through a CAN bus system, which is in charge of removing the torque of the industrial motor when the STO is activated. The implementation details of the SC application are described in Section 4.2. It should be noted that, CAN bus and the signal analyzer deployments are not in the scope of this paper.

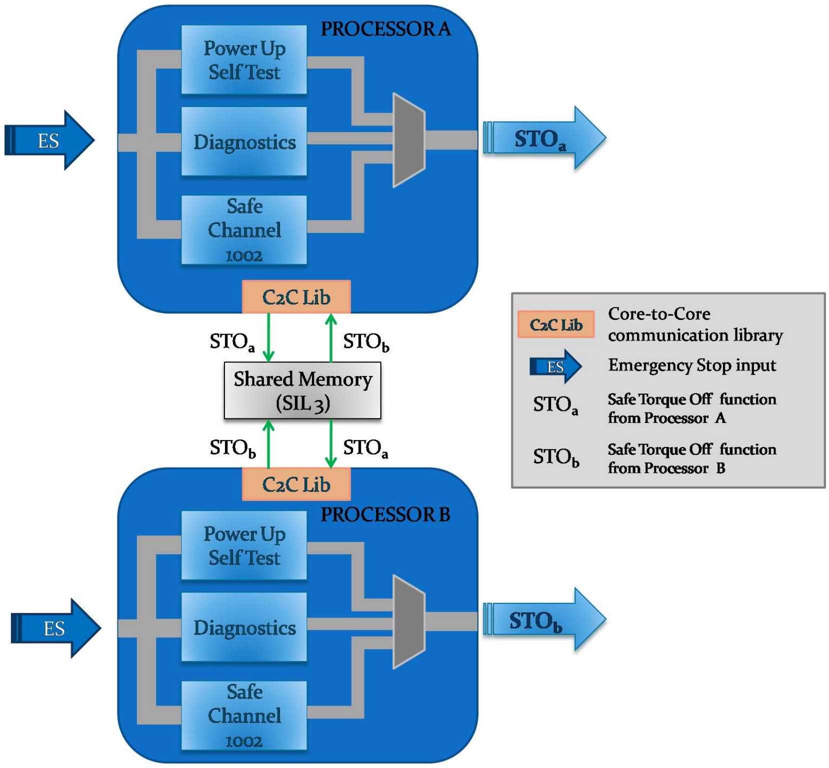

Figure 4: *Core-to-Core communication architecture. The ES signals are connected to each processor in which the STO activation is evaluated. The STO related values are sent and receive from the other processor using a C2C library developed at SIL3 to perform the cross-comparison diagnose phase.*

3.2 Case Study: NSC Sensing Application

NSC software is used to enhance the user experience and increase NSC functionality in a system. Since no certification is needed for this part due to the isolation methodology followed, its development and update do not increase certification costs.

This case study demonstrates a NSC sensing application for visualizing internal values of the SC FPGA platform. We present the implementation of the isolation mechanism, which enables communication between the SC FPGA and the NSC ARM9 while guaranteeing isolation from fault propagation. The sensing application runs on the external ARM9 and contains a functionality for presenting various measurement values from the sensor devices on the FPGA to the user. Values such as *temperature*, *voltage*, *safety function signals* and *error* *values* can be read by the FPGA and sent to the ARM9.

To stress the meaning of maintenance costs, the sensing application includes an update mechanism which can modify software during run-time, which enables the system to change the program code of the executing tasks. To be noticed is that this updating mechanism supplies the possibility to modify the behavior of the sensing application avoiding the necessity of powering the system off nor restarting the application [Wahler et al. 2009]. Similarly, the SC applications or the safe FPGA do not require a reboot when updating NSC software; which is beneficial for timing purposes. Moreover, no re-certification of the SC applications are needed when updating the NSC software since the platform guarantees correct behavior of the SC software independent of the NSC software.

In summary, we present a mixed-criticality system that is composed by two isolated applications: a SC part which evaluates the emergency stop of an industrial machine, and a NSC part that corresponds to the sensing application used for displaying measurement value and is able to insert new user necessities at run-time. Throughout this paper, we present the implementation that was necessary for the development of this system in terms of hardware, firmware, operating system and application.

4 Implementation

In this section we describe the implementation of the whole system in terms of hardware and software elements.

4.1 Hardware

The ACP platform is composed of two different boards connected through an external interface connector: the core board, in which processors and memories are included and the architecture is developed. This board is called AION. Below, it is connected to a peripheral board called RECOMP Sensor Board (RSB), which implements the required peripherals to fulfill safety-critical requirements and connections [Mendez et al. 2013]. The AION board is a dual-processor that provides dual and diverse processor-devices: an ARM9 single-processor and a Virtex-6 FPGA. Along with these processors, independent memory chips (two QDRII chips for the FPGA device, DDR2 for the ARM9 and flash memory chips for each), safety peripherals like watchdogs and isolated oscillators, are available for each processor (Fig. 5). More details about the platform development can be found in [ACP Manual 2012].

This platform covers the hardware requirements of this mixed-criticality system since it requires independent and duplicate system peripherals in order to implement an AMP architecture inside the FPGA as the safety part, whereas the non-safety part is implemented inside the ARM9 processor (Fig. 5)

[Shariful et al. 2009]. Furthermore, according to the case of study, we decided to implement an AMP dual processor architecture inside the FPGA provided within the ACP, whereas the ARM9 single-processor is used to run the NSC part of the system. Thanks to the utilization of the FPGA we can implement (and modify, if required) the hardware architecture using soft-processors in order to evaluate first the correct behavior and system functionality instead of being restricted to a single hardware architecture.

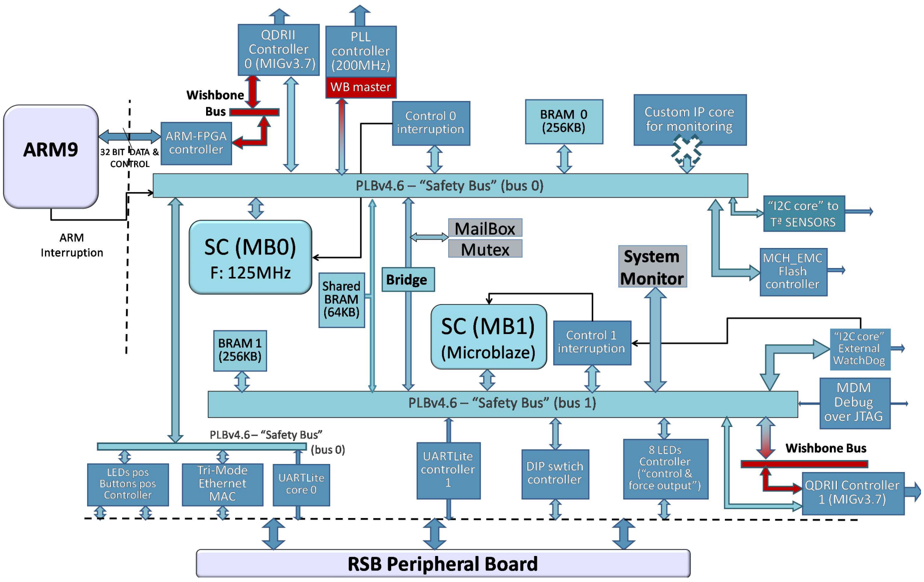

Figure 5: *ACP On-chip FPGA architecture [Adapted from [ACP Manual 2012]]*

We are using a heterogeneous AMP system, in which each software process is locked to a single core. This provides an execution environment similar to single-processor systems, allowing simple migration of legacy code. Moreover, it also allows developers to manage each core independently, implementing different OS and architecture in each one (memories, peripherals) if necessary. Nevertheless, the isolation of processors forces developers to implement a communication channel and its correspondent protocol for data transmission. We have implemented this communication channel also in the FPGA.

4.1.1 FPGA Hardware Architecture

The FPGA implements two independent Microblaze processors. The Microblaze processors are connected to the board peripherals and external memories through a PLB bus which will be the main bus of the system. IP-cores (the commonly used) and the Microblaze processor are being implemented in a safe and certifiable way by Xilinx in collaboration with third-party partners in the frame of critical projects. Therefore, although commercial versions of the IP cores are used, the migration of them to the qualified ones is very simple, which makes re-certification in future projects easier.

Due to the utilization of an AMP architecture, each processor must run independently from each other and only some peripherals are shared. So the architecture can be seen as two isolated soft-processors with different accessible peripherals that are connected as Table 1 describes. The processors are connected to each other through a mailbox and a mutex IP-core (Fig. 5) to share packets and instructions and a 64KB shared memory to exchange largest amount of data and information. These are the components that are used to provide the system with communication features.

AION Peripherals	**MB0**	**MB1**	**Shared**
QDRII External Memory	X	X	
Interruption Controller	X	X	
Internal Memory & Mailbox/Mutex Controller			X
External WatchDog Controller		X	
Flash Memory Controller	X		
SW Configuration Memory		X	
Temp. Sensors and FPGA Monitor	X		
LEDs and buttons	X	X	
JTAG Debug Controller			X
Serial Ports Controller	X	X	
LCD Controller	X		

Table 1: MicroBlaze processors connected peripherals.

Due to the criticality nature of the SC part of the system, we developed a controller for the shared QDRII memory between the ARM9 and the FPGA to ensure that the NSC application does not interfere with the SC one. This QDRII controller has been modified in order to avoid shared memory problems that may cause a wrong behavior of the SC part of the system. Hence, these changes ensure that the ARM9 is just able to read the data that the FPGA writes in the QDRII from a specific region address. Any other operation from the ARM9 is denied by the QDRII controller.

Note that the industry certification standards states that the safe channel architecture 1oo2 needs to be implemented using two cores in two different platforms with different power supplies in order to maintain the required SIL3 level. Our implementation uses two isolated processors in the same platform inside the FPGA that means that SIL3 cannot be ensured as IEC61508 describes. Nevertheless, we use first this FPGA platform to develop and perform the isolation and the communication mechanism that the hardware architecture and the application need. Once this development is successfully accomplished and as future work, we will separate both processors into two different platforms assuring the required SIL3 level.

The software implementations mapped to the FPGA and the ARM9 are described in next Sections 4.2 and 4.2.2 respectively.

4.2 Software

This section describes the software running on the hardware platform and compounds the case study that exposes the removal of the torque from an industrial motor, as well as the sensing application.

The developed software consists of to two different elements as Fig. 6 shows: the SC and the NSC applications. The SC software is mapped on the FPGA multi-core soft-processor. It contains the logic required to satisfy the case study requirements previously introduced in Section 3.1. The application requires the exchange of information regarding each processors' diagnose results compared to the other processor. In addition to this, the utilization of an AMP architecture requires of a safe mechanism to send and receive the data. The reason for this is to avoid inconveniences of using shared memories and to guarantee the robustness of the SC system. For this purpose, we have integrated a C2C communication library to provide processor communications with SIL3 [C2C].

As previously said, the chosen OS for the safety system is OpenRTOS and the programming language is C. Wittenstein provides a free license for FreeRTOS [FreeRTOS 2009], which is a successfully small and efficient embedded kernel and compatible with OpenRTOS. FreeRTOS and OpenRTOS are not certifiable operating systems as such, but a certifiable version of these RTOSes is available for SC systems: SafeRTOS [SafeRTOS 2012]. SafeRTOS is a certified pre-emptive RTOS that maintains the same features as the mentioned RTOSes and, in addition, contains additional features required for certification, such as a complete isolation system for SC tasks by the definition of Memory Protection Unit (MPU) regions per task.

By using a OpenRTOS, we can prioritize the execution order of tasks depending on the relevance of their work. In addition, this RTOS includes an specific

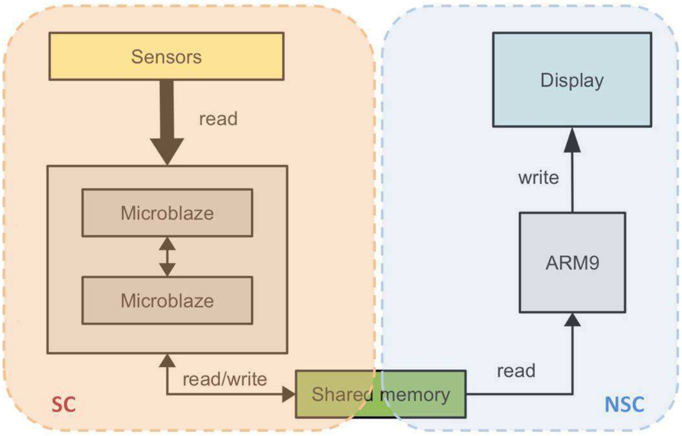

Figure 6: *SC-NSC system data-flow. Represents the flow of data from temperature and system monitor sensors (SC) connected to the FPGA to the display connected to the ARM9 (NSC)*

C2C communication library called WC2C library (WC2C) [C2C], which has been also developed by Wittenstein in the context of RECOMP. This library offers the possibility to exchange data between several processing elements connected through a shared memory and a mailbox. In addition to this, the WC2C library ensures safety features for the critical data exchanged between processors without putting at risk the reliability of the entire SC application as direct access to shared memories may cause.

4.2.1 SC Software Implementation

The most relevant part of the SC application is performed by each processor due to the tasks that it executes. Each processor runs different tasks in parallel to guarantee the correctness of the system. Three different tasks can be defined: STO diagnostic, system monitoring and communication. The STO diagnostic functions handle all tasks that evaluate and diagnose the system status. The SC application decides whether the STO must be activated or not. Moreover, these tasks must determine whenever the system reaches an undesirable state caused by an anomalous behavior, causing the system to a power cycle reset in order to regain the safe state.

The STO diagnostic tasks can be grouped into the following categories:

- Power up tasks: This task starts with a complete checking procedure during the power-on of the system. Once the initial test has been successfully accomplished, it remains activated waiting for the moment the safety function is activated, in order to contribute to the maintenance of the system.

- Diagnostics tasks: This is the main element among the diagnostic process. It processes both local and external STO function status after the power-on self test stage during the whole execution of the platform. The diagnostic of the state of the platform is transmitted to the other core through the C2C channel.
- Failure detection tasks: this task updates consecutively the system watchdog. When a failure occurs, it stops updating the watchdog causing the FPGA to reboot. The FPGA is then re-programmed with the initial content of the flash memory. The flash contains the FPGA firmware as well as the two OpenRTOS applications.
- Communication tasks: each processor runs also two tasks that are used to send and receive data from the safe C2C channel. One task is in charge of sending the local STO status values to the other processor, whereas the other one is reading.

These categories are composed by a certain number of tasks for each processor (Figs. 3 and 4). Table 2 summarizes the software implementation of these tasks.

Task Name	**Type**	**Description**
vPowerUpSelfTests()	Power Up	Performs a peripheral check at start-up
vCrossComparisonIAnalysis()	Diagnostic	Verifies the both local and external STOs
vCrossComparisonIIAnalysis()	Diagnostic	Verifies internal and external data different from STOs
vLivenessCheck()	Diagnostic	Checks whether the other process is alive or not
vFailureHandling()	Failure Handling	Diagnostics task. Launches a reboot command in case of failure
vSTO()	Diagnostic	Handles the local STO function for each processor
SafeChannel1oo2Write()	Communication	Uses the WC2C library to send data to the other processor
SafeChannel1oo2Read()	Communication	Uses the WC2C to read data from the other processor

Table 2: Main function tasks implemented in each processor. Tasks are divided into three type of functions: power-up, diagnostics and communication.

Owing to guarantee the correct behavior of the AMP platform and the schedulability of these tasks executing on each processor, FreeRTOS and OpenRTOS offer a routine that ensures the atomic execution of critical sections.

These routines are `portENTER_CRITICAL` and `portEXIT_CRITICAL` and are described in [OpenRTOS, FreeRTOS 2009]. Inside a critical region the scheduler will never extract the task from the processor during the execution of these lines by disabling hardware interrupts, avoiding undesirable and unpredictable reads/writes on peripherals and shared devices.

The monitoring tasks are used to measure values from internal sensors and registers for runtime monitoring. The SC application periodically reads all the sensor measurements, such as temperature sensors and system monitor values, as well as both local and external STO states, and writes values to a pre-defined region in the shared memory as unsigned 32-bit integer values. These values can be used to detect over heating of components or under voltage brown/blackouts. Furthermore, the SC part writes the STO signal for each processor, and error values for the STO function inconsistency in the shared memory. The STO signal is inconsistent if, for example, one processor signals STO:high while the other signals STO:low.

4.2.2 NSC Software Implementation

All NSC software containing the sensing application and the runtime updating mechanism is mapped on the external ARM9 processor. The goal of the sensing applications is to read measurement values from sensors connected to the SC FPGA and present the values on a display. The obtained values are partly derived from hardware sensors connected to the FPGA and partly from the STO applications earlier described. Since the sensing application is completely without certification, no actions apart from displaying values are taken from this part of the system.

The ACP platform allows communication between the FPGA and the ARM9 via a shared QDRII memory. Moreover, the NSC application has no guarantees for correct execution, and must be assumed to unexpectedly generate faults. These faults cannot propagate back to the shared memory and interfere with the SC part. The challenge is therefore to successfully isolate the SC part from the NSC part and to prohibit fault propagation to the SC part while allowing communication.

The sensing application is mapped to the ARM9 CPU described in Section 4.1. It is running as a task on top of a FreeRTOS [FreeRTOS 2009] port created for the ARM26EJ-S. Fig. 6 illustrates the data-flow from sensors to display via the shared QDRII memory. Initially, the sensor values are read by one of the MicroBlaze cores and written into shared memory. The ARM9 then polls the shared memory periodically to read the stored values. A read from the shared memory is performed simply by reading a 32-bit pointer value from the memory address associated with the shared memory. When a read is issued by the ARM9, a memory controller managed by the FPGA is called and fetches the data from the memory block and sends it back on the bus connecting the ARM9 and the FPGA. Currently, the available data from the SC written in the shared memory block of the QDRII are:

- Three external temperature sensors

– On-chip temperature sensor for the FPGA

– Internal and auxiliary voltage sensor for the FPGA

– STO function status from processor 0.

– STO function status from processor 1.

– Failure status errors occur by processor 0 undesirable behaviors.

– Failure status errors occur by processor 1 undesirable behaviors.

The ARM9 reads the shared memory completely autonomously and, in real-time, translates the values to relevant unit such as Celsius degrees ($^{\circ}C$) and millivolt (mV) depending on which value is read. These values are then displayed through the serial port to a terminal for the machinery operator.

– *Isolation mechanisms:*

One of the key features in mixed-criticality systems is the guaranteed isolation [Wasicek et al. 2010] between the safe and the non-safe part. As the software on the ARM9 is not certified and thus assumed unsafe, a miss behavior in the ARM9 cannot propagate to the safe FPGA. This means that the only communications channel – the shared memory – must be protected against misuse. Misuse can, in form of unintended faults, be originated from the non-safe ARM9 in form of:

1. Data overwrite: The ARM9 overwrites critical data in the shared memory.

2. Resource locking when writing: The ARM9 locks a memory space for an unpredicted time when writing.

3. Read flooding: The ARM9 floods the bus with reads and blocks the FPGA from writing.

We guarantee the isolation with the respective solutions:

1. ARM9 has read only access: A write will not change the value of the content.

2. ARM9 has read only access: A write will not lock the shared memory since the resources will never be accessed.

3. The QDR memory explained in Section 4.1 ensures the scheduling between read and write. This is provided at hardware level by the memory IP core controller developed in the FPGA.

The mentioned solutions for preventing fault propagation is possible since the FPGA is able to set read/write permissions for the shared QDRII memory. The ARM9 is not able to interfere with the FPGA in any other way, since the bus connecting the shared memory is the only physical connection between the two processors.

– *Run-time updating of NSC software:* To further improve the dynamism and flexibility of software, we have implemented a runtime updating mechanism for NSC software originating from the thesis [Lund 2012]. Run-time update of software is a process of replacing an existing part of software on a running platform with another part without shutting down the system or restarting the application. Reasons for updating software is usually due to version updates, bug fixes, algorithm optimization, and to keep the software more up-to-date with the users.

This procedure is fairly trivial as long as no SC software is running on the same platform and as long as the update is not executed at runtime. The shut down process and start up of a SC system can be very time consuming in complex machinery, mills, etc. This leads to NSC software updating possibilities only during complete system maintenance in which the whole system is brought to a stop. Since the uptime of large machinery is a crucial part of its efficiency, complete system shut downs should occur as infrequently as possible. Updating software online – on the other hand – does not require a reboot of the running system, nor does it require a restart of the application the update was performed on. A runtime updating mechanism will also enable remote updating of systems via the Internet, which reduces personnel expenses significantly. Bug fixes and other patches to NSC software could easily be distributed to systems in remote locations from one single location without the restart of the SC nor NSC part.

Run-time updating of NSC software on a mixed-criticality platform is therefore an interesting use-case since a) both the updated software and the updating mechanism itself are allowed to interfere with the SC part b) the updating of NSC software on a mixed-criticality platform has the potential to wastly reduce the development costs for NSC software and its integration.

The runtime updating mechanism is created for lightweight embedded systems running on FreeRTOS. It is capable of transferring the task state of any FreeRTOS task into the updated task version without system reset. As previously said, this NSC part including the sensing application and the updating mechanism, has been implemented and mapped onto the ARM, which is by hardware isolated from the safe FPGA. Hence, users can add new features to the system without interfering the correct behavior of the SC application. To this end, the modification performed on the shared QDRII block memory controller (described in Section 4.1) consolidates the avoidance of undesirable access from the NSC to the SC. This ensures the complete isolation between SC and NSC applications, thus demonstrating that the developed mixed-criticality system complies with the industrial certification standards that industrial machinery requires.

5 Results

In this section we describe results of the implemented system, which demonstrates the correct behavior of the heterogeneous multi-core mixed-critical system in which the SC application performs critical tasks correctly under all possible circumstances, and the NSC part does not interfere with it due to the isolation mechanisms here developed. Moreover, NSC-SC (ARM9-FPGA) isolation reduces significantly the certification cost. It is also included in our case study an example of on-the-fly update software in the sensing application.

5.1 Safety-critical Results

In order to verify the correct behavior of the SC system application, we have performed several tests. These tests evaluate the multi-core issues related to the certification process of this application: hardware and software redundancy, diversity, C2C communication channel architectures and isolation mechanisms. The testing methodology used is the following:

- Fault injection in the ES signal activation of the redundant STO activation process.
- Fault injection in the exchange of the STO signal through the C2C channel architecture.
- Fault injection in the cross-comparison functionality.
- Isolation controller to ensure the NSC part is not interfering the SC one.

Fault injection has been the evaluation technique to check redundancy correctness, detect errors and measure the response time of the case study. Our fault injection method consists in three different experiments. The fist one is the simulation of loss of information between hardware and software components. This loss should not hamper the system because of the redundant hardware and software architecture which guarantees the reception of the signal from two different paths. To simulate the loss of information, we have created a subroutine that disables the ES read function in one of the processors as i.e., a physical wire cut. When the ES button is activated, one of the processors' ES signal remains always inactive while the other one will read the correct ES activation and thus, generating the STO signal which will remove the torque. The reception of at least one of the ES signals is guaranteed by the redundant hardware architecture.

The second one injects wrong STO values to the C2C communication system to simulate that one of the processors is not processing the ES signal properly. This leads to a fake STO announcement from one of the processors to the other or the lack of it. In both cases this situation must be detected by the cross-comparison diagnostic functions (see Figure 3 and Table 2) in each OpenRTOS instance and start with the safety system halt. This halt has to ensure removing the torque from the motor although only one processor activates the STO signal since this means that a machine operator. The activation of the STO signal from any of the processors is guaranteed by the diverse software architecture and the 1oo2 communication channel architecture.

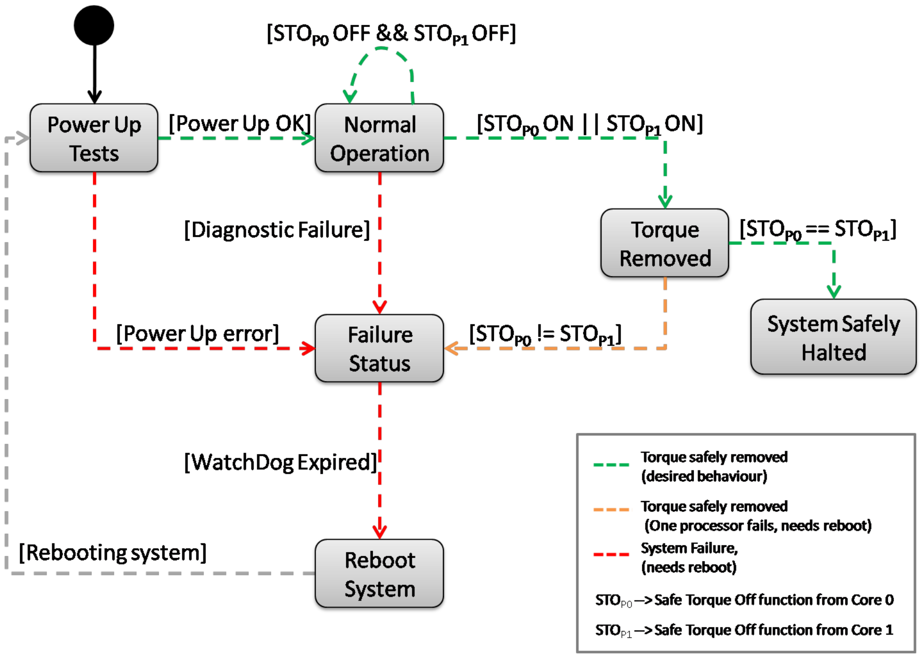

Figure 7: *State diagram of the SC system behavior. The green lines represent the correct status of the application. It ends with the expected removal of the torque preceded by the activation of both STO functions from processor 0 and 1 (STO_{P0} and STO_{P1} are ON). The yellow line represents the correct removal of the torque but the system needs reboot since STO_{P0} and STO_{P1} are not equal (safe state but undesirable). Red ones represent the expected actions the system performes when errors occur (failure)*

Fig. 7 presents the complete state diagram of the fault injection error scenario previously described and the response of our system to them. When the ES button is pressed, each processor receives the order to start the diagnostic process for the STO execution. After the activation of STO in one processor, this signal is sent to the other processor to perform the cross-comparison diagnostic. In case both STO signals are activated with no failure, the system removes the torque normally and it is halted (green lines in Figure 7). In case the two STO signals differ from each other, the system will remove the torque, but it automatically starts with the reboot sequence to guarantee a normal system behavior again (green lines). This information is stored in the shared memory for the NSC part.

The reboot sequence consists in the expiration of a watchdog. When the processors detect a system failure, they stop updating the watchdog and, two seconds after the watchdog expires, the system reboots, it resets the FPGA and loads both firmware and RTOS from the flash memory. This response time is completely customizable and depends on the system requirements. By this, we guarantee the correct behavior of the SC part in case of hardware connection faults.

The third fault injection simulates a software error that affects internal variables directly related to the local and external STO diagnostic phase. This method has been implemented as a function that modifies the local STO-dependent values randomly inside each of the OpenRTOS applications. The cross-comparison functions for both local and external STO data discover an incongruence data exchange (red lines). In this case, both processors request a hardware reset. Once again, the tasks updating the watchdog stop and the watchdog expires prompting a system reboot.

In terms of isolation, the variables that depend on the functionality of the mixed-criticality system are written in the shared memory in order to be read from the sensing system. The controller (Fig. 8) that was developed for the shared memory restricts the NSC part allocated on the ARM9 any possibility of writing. Moreover, read functions are performed by using a different HW bus (Wishbone) that the one used by the FPGA (PLB) to write in the memory. By this we ensure that the SC part is completed isolated from the NSC application preventing from any undesired behavior and thus, reducing the necessity of developing the NSC part to SIL3 (SC part) that leads to cost saving in the certification process.

The three fault injection tests together with the isolation controller show how the error prevention mechanisms work and, at the same time, how the system response flow is restricted as described in Fig. 7. Our SC implementation satisfies the following certification challenges for multi-core mixed-criticality architectures:

- Isolation of SC and NSC parts.
- Certification/re-certification of NSC updates with no additional cost.
- Individual memory mapping for both processors.
- Safety communication channel developed at SIL3.

– Scheduling capabilities using a RTOS.

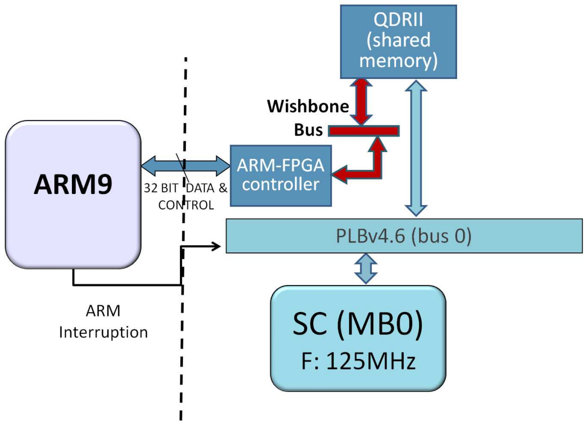

Figure 8: *HW isolation of the memory shared by the NSC and SC parts. The ARM-FPGA controller isolates and avoids any interference between the shared data from the SC part to the NSC application. The QDRII shared memory is accessible from the SC part (MB0) through the PLB bus and is able to write and read. The NSC part (ARM9) is only allowed to read the QDRII memory and it accesses the memory through the Wishbone bus.*

5.2 Non-safety-critical Results

As presented in Section 4.2.2, the NSC sensing application periodically reads sensor values from the shared memory provided by the safe FPGA. In this section we describe the results of the implemented system by showing the right behavior of the SC application and no interference of the NSC part when performing run-time updating. For this case study, we chose to update the software capable of displaying the measurement values on a display. This means that a failure in the display software is not critical for human safety, but degrades the user experience significantly. The reason for using runtime updatable software in display software is the ability to restructure the display layout, add more features, increase/decrease accuracy of measurement values, modify the update period, etc.

The first version of our software included two measurement values: On-chip temperature and internal chip voltage. The software also included two derived values: Max chip temperature and average chip temperature. These values are based on previous sensor values and is therefore part of the state context. We show by updating the software that the state of the application in transferred to the updated software.

A timer was used as decision maker for the runtime update. At a pre-defined time the timer called a callback function in the display task to signal its update request. The display task then entered the safe state and its context (max and average temperature) was saved. The newer version (Version 2.0) of the display software restored the context, and was able to continue registering maximum and average temperature based on the previously collected data. Fig. 9 shows both versions of the display software.

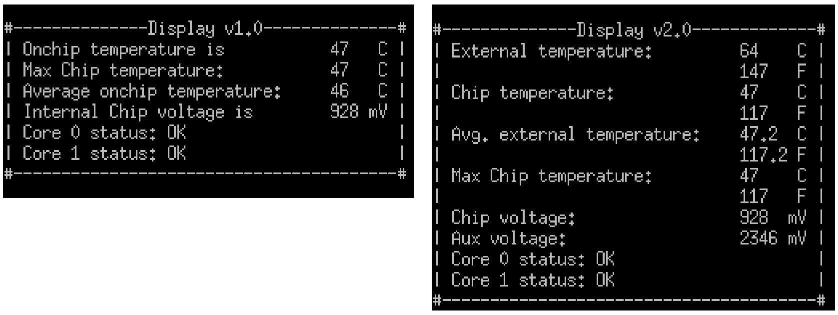

Figure 9: *Runtime update of display software. Version 1 to the left and Version 2 to the right*

The updated version of the display software allows more sensor values such as: external temperature, and auxiliary voltage levels. It also displays all temperatures in both Centigrade and Fahrenheit degrees – a feature added by the new software. It is worth mentioning that both display and run-time update mechanisms do not interfere in the SC system behavior because of the previously stated isolation between the ARM9 and the FPGA. In addition, any update/upgrade of the NSC application would not derive in any additional certification costs.

6 Conclusions

We have presented a complete and reliable mixed-criticality system which involves safe execution of an emergency stop button which removes the torque from a motor of an industrial machine. Furthermore, we have presented a NSC sensing application with run-time upgrade capabilities. The utilization of multicore has made possible to developed our specific isolation mechanisms for NSC and SC and also to implement a diverse and redundant solution for this industrial problem.

The utilization of a reliable hardware platform have been implemented by following certification standards. This provides a system with isolated and redundant peripherals. We have used a dual-core architecture for an AMP application using two MicroBlaze processors on a Virtex-6 FPGA to achieve the duplication of hardware required by the certification standard. These processors run two isolated instances of OpenRTOS which communicate via a safe C2C communication channel that has been implemented based on requirements from IEC61508 SIL3. The utilization of the WC2C communication library, joined with the isolated FPGA, has enabled one of the main requirements for this kind of SIL3 systems: the redundant channel architecture (1oo2), which is necessary for the cross-comparison diagnose stage of the system. This diagnose stage represents the redundant component which ensures the correct behavior of the system in response to the activation of the safety function in which human lives rely on.

In addition to this, we have included a complete sensing system application and a runtime updating mechanism which provides the possibility of upgrading the NSC part with new features depending on the user experience and needs at runtime. Due to the isolation between the FPGA and the ARM9 created in hardware we can upgrade the NSC part without interfering with the correct behavior of the SC part, and thus, reducing certification costs.

As already stated, several parts of the system have been implemented following certification standards. We have also shown the complete isolation between both SC and NSC parts of the system at hardware and software levels even when the updating mechanism is running. For the intercommunication between the processors the hardware itself provides the necessary isolation while still allowing inter-processor communication. Nevertheless, this inter-processor communication is supervised at software level and performs a safe communication system over shared resource inconveniences. Hence, hardware, firmware and the communications channel are close to a certifiable system.

It is worth mentioning that the flexibility of the platform in terms of reconfigurable hardware and software, has lead to an important point of our research, the introduction of open-hardware/open-source approaches regarding the certification process as open-boxed or systems. This helps to save time-to-market and development costs but the main feature obtained is that open platforms helps to improve the reliability of the overall system since reviewers, source code and safety evidences correctly documented, can be completely examined and verified by a wide engineering community [Mendez et al. 2013].

To fully achieve a certified product, we summarize the next required steps to cover all the safety standards requirements for every hardware and software components. In further versions of the system, we verify the feasibility of migrating current IP cores to qualified ones suitable for FPGA design. Additionally, proper PAR methods would be used in order to achieve a final DO-254 certified gate-ware. In the same manner, we need to improve safety in the execution of the SC application. Due to the fact that OpenRTOS is not certified, we need to integrate the safety version of this RTOS, SafeRTOS. By this, we cover the required safety properties for hardware, firmware, OS and communication between processors that are necessary for certifying a complete mixed-criticality system.

Following these improvements, we will ensure the certifiable property that our system would require in order to succeed in the exhaustive study that certification agencies realized to Avionics, Automotive, Nuclear Plants and Industry systems.

Acknowledgements

This work has been supported by the Andalusian Excellence Project VITVIR (grant number TIC-8120). We would like to acknowledge Wittenstein Hight Integrity Systems for their collaboration and support with the integration of FreeRTOS and OpenRTOS as well as the WC2C communication library. In addition, we would like also to thank Danfoss A/S and especially Jesper Berthing for the preliminary design of the case study, which was the point of departure for this work.

References

[Abdelhalim et al. 2011] Abdelhalim, M. B. and Habib, S. E.-D. : "An integrated high-level hardware/software partitioning methodology"; Journal of Design Automation for Embedded Systems, March 2011, 15, 1, (19-50), Springer, US.

[ACP Manual 2012] Seven Solutions Inc.: "D5.2 Reference design dual core Avionic Compute Platform for avionics applications"; Artemis JU RECOMP Project (2012).

[Baruah et al. 2010] Baruah, S. and Li, H. and Stougie, L. : "Towards the Design of Certifiable Mixed-criticality Systems"; 16^{th} IEEE Real-Time and Embedded Technology and Applications Symposium RTAS '10, Stockholm, Sweden.

[Bate et al. 2003] Iain, B. and Kelly, T. : "Architectural considerations in the certification of modular systems"; Journal of Reliability Engineering and System Safety, 81, 3, (303-324), Elsevier.

[Berthing 2012] Berthing, J. and Danfoss A.S.: "D5.6 Drive Controller"; Artemis JU RECOMP Project (2012).

[C2C] Wittenstein High Integrity Systems: "34-190-MAN-01 C2C Communication Library User Manual" (2012);

[Cuenca et al. 2011] Cuenca-Asensi, S. and Martinez-Alvarez, A. and Restrepo-Calle, F. and Palomo, F.R. and Guzman-Miranda, H. and Aguirre, M.A. : "Soft core based embedded systems in critical aerospace applications"; Journal of Systems Architecture (2011), 57, 10, (886-895).

[Danfoss] : "Danfoss Power Electronics"; `http://http://www.danfoss.com/`.

[De Niz et al. 2009] De Niz, D. and Lakshmanan, K. and Rajkumar, R. : "On the Scheduling of Mixed-Criticality Real-Time Task Sets"; Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE, Washington, DC, US.

[DO-178C 2012] : "Software Considerations in Airborne Systems and Equipment Certification".

[DO-254 2012] : "Design Assurance Guidance for Airborne Electronic Hardware Considerations in Airborne Systems and Equipment Certification";

[FreeRTOS 2009] : "FreeRTOS Reference Manual: API functions and Configuration Options"; Real Time Engineers Ltd.

[Gutierrez et al. 2012] Gutiérrez, J.L. and Berthing, J. and Fernández, D. and Díaz, J.: "Safety-Critical Platform Model Based on Certification Standards"; III Jornadas de Computación Empotrada, JCE '12, Elche, Spain.

[IBM Software Rational 2010] IBM Software Rational: "DO-178B compliance: turn an overhead expense into a competitive advantage"; IBM Software Rational White Paper for Aerospace and Defense, RAW14249-USEN-00, IBM Corporation.

[IEC61508 2006] : "Functional safety of electrical/electronic/programmable safety related systems".

[IEC61511 2006] : "Functional safety - Safety instrumented systems for the process industry sector - Part 1: Framework, definitions, system, hardware and software requirements".

[Kelly et al. 2003] Kelly, T.P. : "Managing Complex Safety Cases"; Proceedings of 11^{th} Safety Critical Systems Symposium, Bristol, UK.

[Knight et al. 1990] Knight, J.C. and Levenson, N.G. : "A reply to the criticisms of the Knight & Leveson experiment"; ACM SIGSOFT Software Engineering Notes New York, NY, USA

[Lala et al. 1994] Lala, J.H. and Harper, R.E.: "Architectural principles for safety-critical real-time applications"; Proc. IEEE, Vol. 83, (Jan 1994) 25-40.

[Laprie et al. 1990] Laprie, J.C and Arlat, J. and Beounes, C. : "Definition and Analysis of Hardware- and Software-Fault-Tolerant Architectures"; Computer, (39-51)IEEE Computer Society, IEEE.

[Lund 2012] Lund, W.: "A Unified Run-time Updating and Task Migration Mechanism"; Master Thesis; Institution of Information Technologies. Åbo Akademi University, Finland.

[Mendez et al. 2013] Méndez, M. and Gutiérrez, J.L. and Fernández, D. and Díaz, J: "Open Platform for Mixed Criticality Applications"; Workshop on Industry-Driven Approaches for Cost-effective Certification of Safety-Critical Mixed-Criticality Systems, WICERT '13, Grenoble, France.

[Mollison et al. 2010] Mollison, M.S. and Erickson, J.P. and Anderson, J.H. and Baruah, S.K. and Scoredos, J.A.: "Mixed-Criticality Real-Time Scheduling for Multicore Systems"; IEEE 10^{th} International Conference on Computer and Information Technology (CIT), (1864-1871), Bradford, UK.

[OpenRTOS] : "OpenRTOS"; Wittenstein High Integrity System `http://www.highintegritysystems.com/openrtos/`.

[Parkhurst et al. 2006] Parkhurst, J. and Darringer, J. and Grundmann, B.: "From Single Core to Multi-Core: Preparing for a new exponential"; Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design, ICCAD '06, (67-72), San Jose, California, USA.

[Pellizzoni et al. 2009] Pellizzoni, R. and Meredith, P. and Nam, M.Y. and Sun, M. and Caccamo, M. and Sha, L. : "Handling mixed-criticality in SoC-based real-time embedded systems"; Proceedings of the seventh ACM international conference on Embedded software (EMSOFT '09), (235-244), Grenoble, France.

[Polakovic et al. 2007] Polakovic, J. and Mazare, S. and Stefani, J.B. and David, P.C.: "Experience with safe dynamic reconfigurations in component-based embedded systems"; Proceedings of the 10th international conference on Component-based software engineering, CBSE'07, Medford, MA, USA.

[Powell et al. 2010] Powell, D. and Arlat, J. and Deswarte, Y. and Kanoun, K : "Tolerance of design faults"; Dependable and Historic Computing, Cliff B. Jones and John L. Lloyd (Eds.), Springer-Verlag, Berlin, Heidelberg.

[RECOMP] : "Artemis JU RECOMP Project"; `http://www.recomp-project.eu/`.

[RECOMP D.4.2b.1 2013] Artemis FP7 JU RECOMP Project: "Recommendations for use of multi-core in certifiable applications for Avionics". RECOMP Workpackage 4 Deliverable, Artemis.

[SafeRTOS 2012] : "SafeRTOS"; Wittenstein High Integrity Systems `http://www.highintegritysystems.com/safertos/`.

[Segal et al. 1993] Segal, M.E. and Frieder, O. : "On-the-fly program modification: systems for dynamic updating"; Journal Software IEEE March 1993, 53 -65.

[Seven Solutions Inc.] : "Seven Solutions Inc."; `http://www.sevensols.com`.

[Shariful et al. 2009] Shariful, I. and Neeraj, S. and Andrs, B. and Gyrgy, C. and Andrs, P. : "An optimization based design for integrated dependable real-time embedded systems"; Journal of Design Automation for Embedded Systems, December 2009, 13, 4, (245-285), Springer, US.

[SIL] H. Schbe: "Definition of Safety Integrity Levels and the Influence of Assumptions, Methods and Principles Used"; TUV InterTraffic (2004) `http://www.tuv.com/`.

[Toivonen et al. 1990] Toivonen, H. T. and Tamminen, J. : "Minimax robust LQ control of a thermomechanical pulping plant"; Journal of Automatica (347-351), 26, 2, Pergamon Press, Inc., Tarrytown, NY, USA.

[Wahler et al. 2009] Wahler, M. and Richter, S. and Oriol, M.: "Dynamic software updates for real-time systems"; Proc. of the 2^{nd} International Workshop on Hot Topics in Software Upgrades; ACM (2:1-2:6), New York, NY, USA.

[Wasicek et al. 2010] Wasicek, A. and El-Salloum, C. and Kopetz, H. : "A System-on-a-Chip Platform for Mixed-Criticality Applications"; Proceedings of the 2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing; ISORC '10, (210-216), Washington, DC, USA.

[Wittenstein] : "Wittenstein High Integrity Systems" `http://www.highintegritysystems.com/`.

[Yeh et al. 2001] Yeh, Y.C. : "Safety critical avionics for the 777 primary flight controls system "; Proceedings of 11^{th} Safety Critical System Symposium DASC '01, (14-18), Stockholm, Sweden.

[Zhuravlev et al. 2010] Zhuravlev, S. and Blagodurov, S. and Fedorova, A. : "Addressing shared resource contention in multicore processors via scheduling"; Proceedings of the 15^{th} edition of ASPLOS on Architectural support for programming languages and operating systems (ASPLOS '10), (129-142), New York, USA.