International Journal of Safety and Security Engineering
Vol. 11, No. 1, February, 2021, pp. 21-34
Journal homepage: http://iieta.org/journals/ijsse

# ISO-26262 Compliant Safety-Critical Autonomous Driving Applications: Real-Time Interference-Aware Multicore Architectures

Abdullah El-Bayoumi[1,2]

[1] TTTech Auto Iberia, TTTech Group, Barcelona 08029, Spain
[2] Electronics and Electrical Communications Engineering Department, Cairo University, Giza 12613, Egypt

Corresponding Author Email: abdullah.elbayoumi@pg.cu.edu.eg

https://doi.org/10.18280/ijsse.110103

**Received:** 12 September 2020
**Accepted:** 26 December 2020

***Keywords:***
*real-time operating system, multicore architecture, multi-processor, freedom from interference, functional safety, fault-tolerance, reliability, worst case execution time*

**ABSTRACT**

Over the decades, autonomous vehicles have been developed and qualified using variant single-core architectures. With the evolutionary trend of safety critical applications, innovative safety design methodologies have raised present requirements constraints and limitations to mitigate such design complexity deviations. The main objectives of this work are to investigate, evaluate and introduce an efficient safety-critical multi-cache multicore architecture, that is fully compliant with methods and principles of ISO 26262. Moreover, this paper presents new safety design choices applied to timing monitoring, temporal protection, runtime monitoring and services protection to overcome multicore processor challenges in runtime that eventually decay the worst case execution time and the interconnections (symmetric and asymmetric processors, critical timing, data coherency and synchronization predictability, core interconnects, etc.), as well as to tolerate real-time interference faults.

## 1. INTRODUCTION

The decay of the semiconductor scaling [1] during the past decade marked the end of the gigahertz era, whereas the current shift rises towards multicore designs due to their more favorable performance-power ratio [2]. Moreover, there is no need to have a higher clock speed, as discussed in the research [3]. Optimizing inter-core resource sharing distributed among software application components, presented by Schliecker et al. [4], minimizes the computing power by avoiding concurrent accesses of wait-states to the shared resources with the expense of independent data processing and parallelization losses [5]. Thus, system architectures experiencing high-performance data processing and computation have been trending to be real-time mixed-criticality multicore processor platforms, as interpreted in Figure 1.

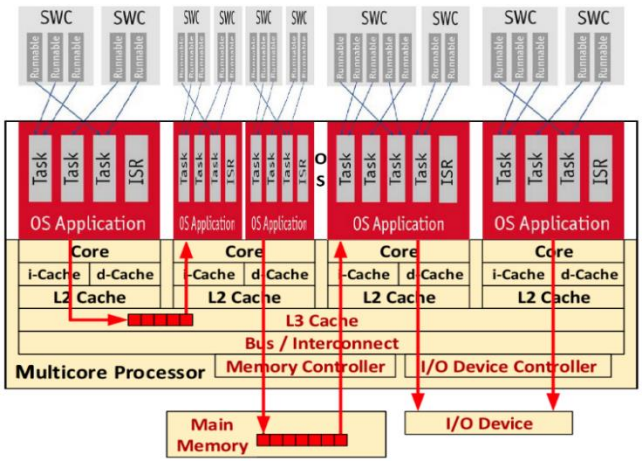


**Figure 1.** Multicore architecture block diagram

These platforms target complex automotive applications such as Advanced Driver Assistance Systems (ADASs), which target reliable recognition of moving objects to provide decision-making algorithms. Autonomous driving imposes significant challenges at various levels as it mostly depends on technology fusion of one or more of the following: radar, high-resolution camera, laser, and Light Detection and Ranging (LiDAR), examined in the studies [6, 7]. Software applications run with different criticality such as scheduling, sharing computation, communication delays, communication links, and communication resources. These issues become challenges at an operating system (OS) level in today's multicore environments [8, 9].

Autonomous driving is both a rapidly advancing technology as it will ensure a better future with increased safety on the roads, and a subject of controversy due to automotive hacking incidents, and the risks of fatal crashes. There are 5 levels of autonomous driving [8], developed by the Society of Automotive Engineers (SAE), spanning from driver assistance to fully autonomous cars without considering the level zero that correlates to having no automation and instead complete human control of the vehicle.

In level 1 which named as driver assistance, It´s a fail-safe system where the vehicle manages to detect the fault, but a human driver is responsible for all tasks associated with operating the car and to react to such a fault (i.e. normally stopping the operation). There is a driving automation system in the car that helps with either steering or accelerating, but not both.

In level 2 which named as partial automation, the automation system in the car can assist with both steering and acceleration, while the driver is still responsible for most of the safety-critical functions and environment monitoring. Currently, the level 2 autonomous vehicles are by far the most

common on the roads.

In level 3 which named as conditional automation, the car itself monitors the environment by utilizing autonomous vehicle sensors and performs other dynamic driving tasks, such as braking. It can also react partially to the undesirable event, by operating in a degraded mode with the help of the safety mechanisms. The human driver must be prepared to intervene if a system failure occurs or other unexpected conditions arise while driving.

In level 4 which named as high automation, it is a fail operation where it correlates to a high level of automation. The car can complete an entire journey without any intervention from the driver and react to all hazardous events due to the sufficient level of redundancy. However, there are some restrictions: the driver can switch the vehicle into this mode only when the system detects that the traffic conditions are safe and there are no traffic jams.

Finally, in level 5 which named as full automation, automakers are striving to achieve this level where the driver simply specifies their destination, and the vehicle takes complete control and responsibility for all driving modes. Therefore, level 5 cars will have no provisions for any human control, such as steering wheels or pedals.

One of the major metrics that certifies the project street allowance is the functional safety (i.e., catastrophic consequences absence that affect the user(s) and the environment). Usually, safety-critical functions are subject to timing requirements. The criticality concept controls and potentially impacts the functional safety, informally refers to the system application. A more formal definition of a criticality (level) illustrated in ISO 26262 [10] for road vehicles which defined the design and development processes for the safety-critical embedded system (hardware and software).

The criticality level results from performing Failure Mode, Effect and Criticality Analysis (FMECA) process, discussed by Tobias [11] which requires:

(1) Defining the functionality and failure modes,

(2) analyzing failure causes and effects,

(3) assigning severities to the failure modes according to the failure effects,

(4) identifying the existing compensating provisions, and

(5) assign criticality categories and recommendations.

ISO 26262 regulates mixed safety-critical systems in both design and integration. Although, mixed functionalities are defined in both the spatial and temporal domains, the whole system is developed according to the highest level of criticality.

By the time, the Automotive Safety Integrity Level (ASIL) of such systems raises. The main cause is that ASIL, resulted from the hazard analysis and risk assessment [12], illustrates the frequency and severity of a failure mode and assign the corresponding safety requirements to the probability of failure, architectures, and design processes. Meantime, the state-of-the-art ADAS functionalities are usually Quality Management (QM) and subject to the driver control and responsibility. Wherefore, autonomous driving transfers this responsibility to complex ADAS multicore critical systems depending on the autonomy level that results in producing highly safety-critical functions with high-performance requirements, as they have been introduced in the literature [8, 13, 14]. Moreover, traditional safety-critical mechanical features, such as antilock braking, have moved towards new autonomous driving solutions, such as the electronic stability program with the help of networked layered architecture systems. Furthermore, customers (i.e. ultimately passengers) constantly demand for fully autonomous vehicles.

AUTOSAR illustrated in Ref. [15] supports an abstracted layered architecture in a runtime environment that resolve the tremendous development efforts performed to provide a new software if a component has become obsolete or outdated. So, the number of different messages revealed from various communication types provided on automotive networks has grown much faster than the number of implemented functions. As a result, the software integration becomes more complex, in addition to having a huge increase in the consumed computing power. This pushes the processing performance to its limits.

AUTOSAR and OSEK/VDX OS utilize allowing controlled communication between partitions and uses time division multiple access (TDMA) scheduling for fixed time partitioning, in preparation of achieving timing independence as in FlexRay communication protocol. This is mandated by ISO 26262 as in memory and time partitioned system. Moreover, ISO 26262 permits static priority scheduling (as in CAN communication protocol), with higher priorities assigned to multiple critical tasks, and without considering inversion effects [16].

ISO 26262 guarantees Freedom from Interference (FFI) in which the separation mechanism must always adhere to the highest ASIL involved. The main goal is that a safety code execution cannot be corrupted by a non-safety code. This means assuring the critical signals flow through software components with being protecting from lower ASIL or QM interfering software components that would affect the data correctness. Software architectures including communication interfaces (i.e. FlexRay, CAN, LIN, Ethernet, I2C, SPI, etc.) must be developed accordingly. A disruptive challenge of functional safety reveals in the system efficiency in which there is a performance loss, at least for the critical tasks by going to a safe state (i.e. degraded mode) and aborting fault propagation in case of failure. By increasing multicore system dynamics complexity, TDMA scheduling limitations increase [17, 18].

Although multicore CPUs have huge potential to produce efficient and sophisticated functionalities with a high return of its investment, ISO 26262 implicates many architectural and design requirements to assure the system operates in safe state in a time less than fault time tolerant interval (FTTI) if erroneous values affects critical signals (even related to calibration data). There are means of FFI corruption methods affect the safety-related Software Components (SWCs) such as: information exchange interference, memory interference, real-time interference, and shared peripheral interference. Functional safety methods and mechanisms reduces the functional system efficiency especially if it is a multicore architecture.

This work is unprecedented and sets the basis for future development and discussions. This paper presents optimized safety-related configuration, and enhanced safety mechanisms protection for complex multicore architectures to seize and react to real-time faults and to let the system behave in a safe way. This work is fully compliant with ISO 26262 methods for Aurix and Renesas multicore microcontrollers.

The rest of the paper is organized as follows. In Section 2, functional safety constraints of multicore architectures metrics that mostly deteriorate the safety-related WCET are discussed. While Section 3 represents the analysis of freedom from real-time interference challenges for runtime faults. Whereas

Section 3 illustrates the proposed software safety mechanisms for real-time cutting-edge challenges encountered in multicore processors with proposed software safety-related configuration. Meanwhile future work is delineated for a full research scope in Section 5. Finally, a conclusion is provided in Section 6.

## 2. SAFETY-CRITICAL MULTICORE ARCHITECTURES CHALLENGES DESIGN AND ANALYSIS

Multicore processors integrate independent cores into a single Integrated Circuit (IC) that runs at lower clock frequencies with a lower power consumption and a higher performance. This performance is not multiples of single-core processor performance due to the exhibition of the required parallelism needed, by extra software, to have concurrent running cores. In this section, safety-relevant design measures featured in variant multicore architectures are presented.

### 2.1 Symmetric and asymmetric multi-processors challenges

A multicore processor is defined as homogenous or heterogeneous (usage of non-identical cores). While symmetric multi-processors involve utilizing a single OS running across multicore processor to reduce the IC footprint. Limitation of either shared-memory performance, or core performance, or input/ output performance impact the symmetric multicore scalability. In symmetric multi-processors, as shown in Figure 2(a), the OS kernel executes application processes and threads scheduling across multiple cores. Even though, an affinity OS as a safety mechanism ties these processes and threads to individual cores to enhance real-time performance.

On the contrary, number of cores in symmetric multi-processors is limited to protect critical shared resources that ensure serialized access. Consequently, application performance of such processors is limited. Many independent applications that are running simultaneously on different cores may require an inter-partition communication (IPC) as a safety mechanism. However, exploiting symmetric multi-processors experiences losses in determination as the consumed time to access a shared critical resource is not predictable due to depending on an activity in another core that attempts to gain another access to a shared critical resource. Furthermore, there are execution timings variances among rescheduled tasks on the multicore due to caching and interconnection effects as represented in the state-of-the-art cache-interference management [3, 19-22].

Asymmetric multi-processors, symbolized in Figure 2(b), employ a hypervisor instead of a framework to implement control and inter-core communication. This offers more flexibility and control and a higher level of security. They are distinguished processors as they treat each core as an individual processing unit in a way to replicate instances of the same application and operate on separate data sets across the multicores. Wherefore, they permit more operating systems (i.e. real-time OS, Linux, etc.) to operate on variant cores. The running operating systems may propagate faults from a core to another as the individual cores uses L2 cache and memory buses to share critical hardware resources. Thereby, utilizing supervision or virtualization to the multicore architecture increases as a defensive safety mechanism so as not to violate a safety goal by preventing multicore applications from contending for shared critical resources to provide more isolation among running QM applications on a core, and running safety-critical applications on another core (i.e. software partitioning) [23].

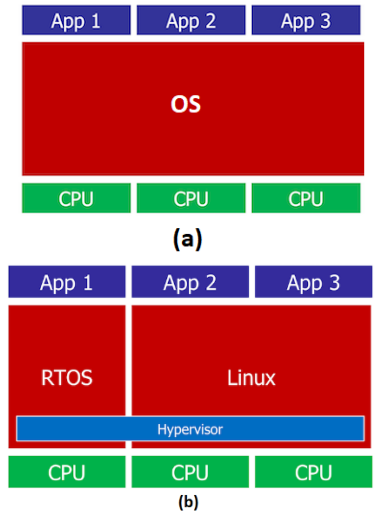


**Figure 2.** Multicore architecture configuration (a) Symmetric multi-processor (b) Asymmetric hypervisor multi-processor

However, exploiting asymmetric multi-processor indices potential barriers due to the increased coupling among application on the multicores due to sharing critical resources, memory controllers, caches, and hardware peripherals. Thus, deadlocks could be produced. By using semaphores, spinlocks, to protect shared safety critical resources from QM SWCs as safety mechanisms, it resolves the challenge of either having many running applications at the same/ different criticality levels with adjusting the tasks properties as well.

### 2.2 Timing challenges

If a safety-related task misses its deadline, it means the system will not go to the safe state (i.e. a software reset). This leads to a safety goal violation. So, the task must terminate before it reaches its deadline [24] (e.g. its, Worst Case Execution Time, WCET). There are many challenges to measure the WCET. Firstly, the WCET can be blocked or preempted if the OS is multi-tasking. The term Worst Case Response Time (WCRT), which include the WCET in addition to preemption/ blockage time jitter, is more accurate to depend on. In practice, using non-preemptive scheduling for higher-ASIL short tasks shall reduce latency of safety critical outputs. In addition, tasks with long WCET should be preemptive to reduce latency of critical outputs.

Secondly, because of the processor caching and pipe-lining effects, the timing sequence of an instruction represents part of previously executed instructions. Abstract interpretation is one of the safest method, which is processed during either static testing or fault injection to check timing, infeasible paths and how instructions flows in pipelining with consideration of cache hit/miss [3] for safety-related tasks. It is based on a semantics procedure mapped to an abstracted model, which provide faster computation. Predicting timing of tasks allows abstract interpretation to measure the WCET and WCRT maximum execution time for critical tasks (without exceeding program execution time) if they are performed during system

scheduling analysis.

Lastly, resource sharing by concurrent accesses among tasks on either the core level or the multi cores level make the corresponding WCET and WCRT become variable. This results from accesses to shared caches, shared flash memory pre-fetch buffers, or shared memory controllers. Highly recommended safest solutions are to configure the access rights of software components especially for safety-related tasks as in the Memory Protection Unit (MPU), also to activate the hardware safety mechanisms such as instruction caches, branch history table, out-of-order pipelining, or static/dynamic branch prediction. However, this could make the local WCET (revealed as cache miss) not to be part of critical global WCET. In other words, the WCET is significantly less than the cache hit due to scheduling effects of processor. Additionally, configuring the lock-step mode achieves a predictable performance.

### 2.3 Predictability challenges

As the WCET prediction is complex, queues, represented before the caches for buffering and neglecting cache misses, are a suitable safety mechanism for load/ store operations in multicore architectures. Queue inter-connections are based on faster data flow of concurrent accesses into cache lines that are requested by ongoing instructions, in which their data might be available in the same core or another one. Precise memory addresses requests decrease WCET measurements for multiple scenarios. Hence, efficient Transactions on the architecture bus are maintained. In particular, hardware mechanisms of branch prediction and history tables provide extra bits that increase memory consumption.

There are many safe, precise and efficient queue predictable procedures target processor caches. The Least-Recently-Used (LRU) procedure depends on classifying memory read access. Less performed read accesses make LRU procedure more accurate in WCET determination than First-In-First-Out (FIFO) and Pseudo-LRU (PLRU) procedures. On the other side, there are two-write procedures. The first procedure is write-through, in which an operation storage is written in the memory hierarchy level. The second procedure is write-back which follows write-through procedure if the memory field is freed from the cache. Due to the cache analysis uncertainties, as well as, increasing the cache levels, the write-back procedure analysis becomes more difficult.

In other words, FIFO is a queue with new elements are inserted at the front, while evicting elements are at the end of the queue. In contrast, LRU hits do not change the queue. Their implementations utilize a round-robin replacement counter for each set pointing to the cache line to replace next. This counter is increased if an element is inserted into a set, while a hit does not change this counter. Moreover, PLRU is a tree-based approximation of the LRU policy. It arranges ways in a tree bits pointing to the line to be replaced. It is much cheaper to implement than true LRU in terms of storage requirements and update logic which reduces predictability. PLRU also tracks invalid lines. On a cache miss, invalid lines are filled from left to right, ignoring the tree bits. The tree bits are still updated.

The same predictability procedures are followed for external devices connected with the caches over the system bus. These devices as static/ dynamic memory controllers (or communication controllers. On the other hand, WCRT shall be set with other overheads for asynchronous events of program executions. This strategy is followed for safety-related interrupts, Direct-Memory-Access (DMA), Error Correcting Code (ECC) in Random Access Memory (RAM) and hardware exceptions.

### 2.4 Core interconnect challenges

Figure 1 shows a 4-core architecture block diagram similar to Intel Core I7 with some changes. It consists of four physical cores connected with high-speed communication path illustrated as Interconnect with a shared L3 cache (shared with all physical/logical cores) for high power and performance efficiencies. Each physical core has two logical cores and an individual L2 cache. For a faster simultaneous multi-threading OS, each logical core has its private instruction and data L1 cache, as well as shared memory controllers placed among all physical cores. Moreover, there is no inherited timing interferences among the cores. Each core has redundant nine banks of five registers (control, status, address and error information registers) linked to hardware safety units. Therefore, the architecture includes a safe hardware error reporting mechanism for: uncorrected errors, uncorrected recoverable errors, and corrected errors.

All cores are interconnected with buses, crossbars, meshes and typical routed communication structures. To have a coherent system, interconnect accesses require arbitration accesses from the other cores due to the utilized architecture memory hierarchy defined as (L1, L2 and L3) caches per each core. Furthermore, additional core communication is required, since the L1 cache data of a core may be old as this data is renewed either in the L1 cache of another core, or in the memory controller.

A shared resource access causes variants interconnect traffic challenges that appear on the processor interconnection to process a single instruction. This traffic includes data traffic, cohesion traffic and eviction traffic. The first interconnect traffic challenge is a cacheable read access issued by one core. If there is a cache hit to another core, the cacheable read memory access produces a silent communication. While, if there is a cache miss to another core, it initiates a read request. Finally, it initiates a prime write access to evict the modified data from the cache.

Meanwhile, the second interconnect traffic challenge is a write access to a cacheable memory area issued by one core. With the same methodology, if there is a cache hit to another core, the cacheable write access memory causes no traffic. While, if a cache hit occurs to update directories of other cores, it produces a coherency traffic. Moreover, if there is a cache miss, it initiates a read access. Finally, it initiates a prime write access to evict the modified data from the cache.

Lockstep mode is a hardware safety mechanism represented in many microcontrollers (i.e. Aurix Tri-core and Renesas RH850). The lockstep mode includes two identical hardware cores that execute the same software code. A unique independent hardware comparator is placed to compare each core output. ISO 26262 assures the microcontroller goes to a safe state if the comparator result is false, without having an additional multicore software handling (i.e. no intention to increase computing power). As it eliminates all interferences within cores that execute the same set of instructions in parallel, it makes the processor behave like a single-core architecture. When all available safety and performance hardware mechanism are utilized, the resolved challenges of core interconnect make the resulted timing bounds reach an accurate WCET.

## 3. FREEDOM FROM REAL-TIME INTERFERENCE CHALLENGES IN MULTICORE ARCHITECTURES

In mixed safety critical systems, if a SWC experiences with the coexistence, where it includes mixed-ASIL sub-functions. The SWC is treated with the highest ASIL represented in its sub-functions if it interferes with other ASIL SWCs, as means of FFI are interpreted in Figure 3. While, from the FFI definition, where cascading failures absence among SWCs lead to a safety goal violation. Therefore, developing the whole SWCs with the highest ASIL assures FFI analysis by its definition, since there are no QM SWCs.

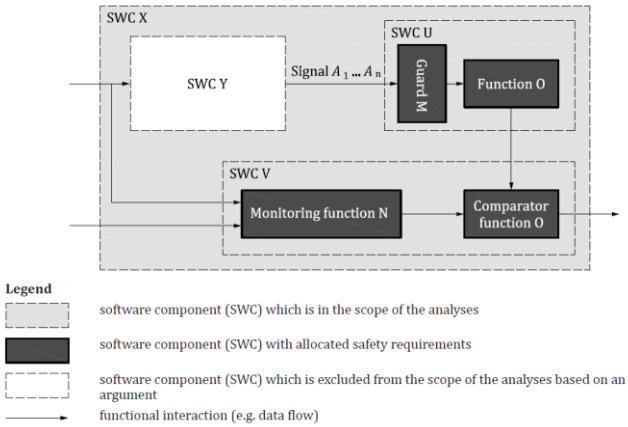


**Figure 3.** Example of FFI due to information exchange interference, memory interference and shared peripheral interference

In general, there is at least one critical path represents the data flow of a critical signal from input conditions to the output root-cause in a safety-related software architecture. It is represented in a software design critical path analysis that also includes different-ASIL SWCs interferences. In the critical path, it is sufficient to have SWCs that detect and react to means of software/ hardware faults. If all SWCs are developed according to the highest-ASIL ISO 26262 compliance matrix, there are redundant safety mechanisms that perform the same detection and reaction behavior. Thereby, the CPU load will exceed its limits enough to make the system not performing at all. As a result, this is a high-cost inefficient design choice [25].

On the other hand, if set of safety mechanisms are provided to the mixed critical system to contain QM SWCs failures on the ASIL SWCs. Hence, no safety efforts are needed in QM SWCs with the expense of a CPU overhead and an architecture optimization. Interferences to critical SWCs could affect its properties in multicore architectures with data faults, timing faults, OS faults, sequence faults and hardware faults.

ISO 26262 abides to analyze dependent failures, portrayed in Figure 3, to show independence between software components used to implement independence requirements coming from ASIL decomposition at system level [26]. Thus, neither cascading failures nor common cause failures shall propagate among SWCs whether they are successive or placed in different paths, accordingly. Although common cause failures result from a single specific event or a root cause that shall affect 2 or more internal sub-functions of a SWC or external SWCs, they may result from a defined hardware block.

This means that single point of failure metrics of the highest ASIL SWC before decomposition should be covered by an analysis method. To perform the dependent failure analysis, such ways of FFI methods among SWCs are used to implement the ASIL decomposition shall be progressed, even if they have the same ASIL level. In addition, FFI between each ASIL SWC that is used to implement ASIL decomposition and the shared component shall be analyzed.

Software data faults may corrupt either memory [27, 28] (i.e. RAM, Flash, EEPROM, registers, DMA) or initialization data or calibration data (in pre-compile, link-time, post-build). They may affect logical data processing and data transmission among SWCs (in inter/intra ECU communication). Means of exchange faults are:

(1) multiple message reception,
(2) message deletion/ loss,
(3) additional message insertion,
(4) message corruption,
(5) incorrect message flow,
(6) message delay/ timeout,
(7) invalid message destination address,
(8) message inconsistency due to faulty network status in communication nodes,
(9) blocking access to a communication channel, and (10) invalid message data range.

Timing faults are represented as:

(1) aliveness timing issues incomplete execution or no execution of a Supervised Entity (SE) within the OS periodicity due to unexpected termination), as shown in Figure 4(a); and

(2) deadline timing issues (i.e. non-terminating calculation or incorrect frequency/ timing execution or which means execution is either too slow/ fast or too early/ late), as shown in Figure 4(b).

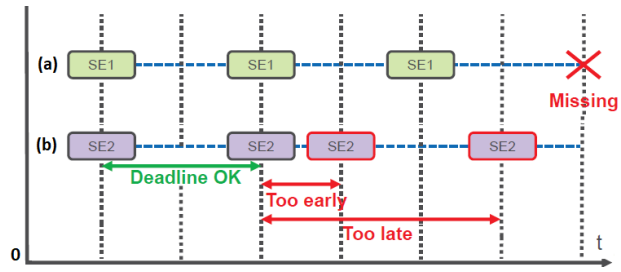


**Figure 4.** Timing diagram of safety-related impacted supervised entities (SE) (a) Aliveness timing issue (b) Deadline timing issue

SEs of a critical SWC include multiple checkpoints to represent important elements (i.e. ASIL task, runnable, and function) for timing measurements and control flow.

The SE has transitions to checkpoints with one or multiple beginning/ end checkpoints. Processing a fault instruction in the program flow or even missing to process a correct one from any beginning/ end checkpoint may lead to data corruption, data inconsistency, fail-silent violations, and process crashes in the control flow. These faults produce incorrect checkpoints control flow and timing faults that affect the program flow due to divergence that lead to sequence faults.

The real-time interference is represented by means of runtime faults as:

(1) Lower-ASIL non-preemptive tasks with a longer execution time (i.e. larger than the maximum allowed higher-ASIL task jitter) which delay execution of higher-ASIL tasks,
(2) Critical sections used by lower-ASIL tasks, with undefined WCET, longer than the maximum allowed jitter of higher-ASIL tasks with lower priority,

(3) Waiting hardware or external event loops without a timeout in lower-ASIL tasks/ interrupts may cause blocking of higher-ASIL tasks/ interrupts,
(4) Shared resources (peripheral, or non-reentrant code segment, or shared data structure) acquired by lower-ASIL SWCs longer than the maximum allowed jitter of a higher-ASIL SWCs shared the same resource, may block critical tasks,
(5) Interrupts WCET of lower-ASIL SWCs longer than the maximum latency of higher-ASIL tasks/ interrupts which may cause a violation of real-time constraints allocated to higher-ASIL SWCs,
(6) Improper choice of priority among higher-ASIL and lower-ASIL SWCs (i.e. higher-ASIL interrupts latency increases when lower-ASIL interrupts are assigned with higher priority than ASIL interrupts priority; or when 2 mixed-ASIL tasks are ready at the same time, but the lower-ASIL task with higher priority starts causing delaying the execution of higher-ASIL task with lower priority; or when a lower-ASIL task with higher priority become ready, and may interrupt the execution of the currently running preemptive higher-ASIL task with lower priority),
(7) Blocking of higher-ASIL tasks execution due to interrupt overloads in lower-ASIL interrupts,
(8) Execution of higher-ASIL tasks triggered by external events communicated by lower-ASIL tasks may be delayed or not activated which causes a violation of a safety goal,
(9) A higher-ASIL task calling synchronous services with a longer execution time from a lower-ASIL task enough to increase the WCET of the higher-ASIL task more than the maximum latency of critical output, and to delay/ block critical outputs.

In this section, safety mechanisms for failure detection and reaction are proposed to develop ISO 26262 methods of FFI efficiently in multicore architectures. For real-time interference: timing monitoring with temporal protection, runtime monitoring, and service protection mechanisms are proposed to resolve timing faults, sequence faults and OS faults, respectively.

## 4. PROPOSED SAFETY MECHANISMS FOR MULTICORE ARCHITECTURES

The proposed safety mechanisms presented in this section are carried out for Aurix Tri-core, Renesas RH850, and Freescale targets. They detect and react to timing faults, sequence faults and services faults that take place during real-time intercommunication of multicores among mixed-ASIL SWCs or even inside a single core at runtime.

Practically, the safe OS is developed by a supplier with the minimum required quality of ISO 26262 methods for a target ASIL to get this SWC accredited and certified. It implements additional safety requirements to guarantee a systematic behavior at all expected operation failures for different kinds of freedom from interferences. The OS supplier takes responsibility if the OS fails (due to internal systematic fault in the OS), given that the safe OS is well integrated as defined in the supplier integration manual.

Meanwhile, the QM OS is developed, as per the standard quality process, with no guarantee whether it is better or not than what ISO 26262 requirements cover. There are no additional mechanisms, added within it, to cover runtime errors (other than what stated by the OSEK standard). Thus, it is preferred to define additional safety mechanisms to cover possible OS failures, as stated in Section 3, identified by the performed safety analysis. Because the safe OS is costly in a way compared to the QM OS, the decision to begin a mixed-critical project with a specific OS should be made earlier.

There are major metrics must be ensured in choosing an OS:
(1) Freedom from real-time interferences where the use of an ASIL watchdog manager with a proper monitoring strategy and good software integration could be sufficient, while using the QM OS to schedule safety critical tasks,
(2) Freedom from memory interference where the choice between either the safe OS or the QM OS depends on:
   a. the memory protection safety mechanisms that shall be implemented,
   b. the used software architecture,
   c. implemented safety requirements in SWCs,
   d. the method and amounts information exchange in cross partitions including the critical shared variables, and
   e. in AUTOSAR, whether the Run-Time Environment (RTE) is used or not to communicate among SWCs and the basic software via the IPC.

Choices to utilize the QM OS are based on whether the implementation of software requirements is centralized in a few SWCs with having a few cross partitions communication, a few amounts of critical data, an efficient memory mapping where safety critical data are aligned together and with non-AUTOSAR architecture.

Developing the OS as a specific ASIL level ensures only that there are no real-time failures caused by the OS itself, during scheduling (i.e. causing wrong context switching, delaying certain ASIL tasks, or blocking certain tasks from execution). However, ASIL and QM activities entitled in a software architecture inherit observable real-time interferences, on the scheduling sequence of the OS itself, caused by the QM runnable/ interrupts. Consequently, an interference on the ASIL tasks might reveal (the QM tasks takes more time than expected by preventing the ASIL tasks from operation. Thus, the solution is to ensure an efficient design with using monitoring functionalities to satisfy the safety real-time constrains and to ensure the freedom from real-time interference.

### 4.1 Timing monitoring safety mechanisms

Timing monitoring safety mechanisms aim to let safety critical tasks meet their execution time budgets. On top of that, the mechanisms shall detect potential risks, in which whether the QM tasks monopolize the OS by requesting many interrupts or loading the CPU in a way to block the critical tasks. Timing faults are not limited to execution blockage, deadlocks, live-locks, erroneous allocation of execution time, and invalid synchronization among SWCs. This means either SEs, or unrelated QM tasks or Cat2 interrupts miss their deadline at runtime, and they become blocking. As a result, this fault propagates through the critical system until reaching a target ASIL SE that misses its deadline, which will be detected by the watchdog.

There are many reasons to consider QM or lower-ASIL interrupts configured as Cat2 over Cat1 in the real-time interferences. Cat2 interrupts are managed by the OS interrupt handler before the user's interrupt. Thus, they interact with OS and can make OS calls. They have a higher latency, if

requested by the hardware until the first instruction execution. Besides, they can be completely controlled by the OS. They can communicate with other tasks or Cat2 interrupt handlers with the help of the OS resource.

Unlike Cat2 interrupts, Cat1 interrupts are managed by the interrupt handler, which is called by the hardware interrupt vector. They are not supported by the OS and can just make a minor selection of OS calls to disable/ enable all interrupts. Manipulation of Cat1 interrupts depends on the target itself. There is no need to lock out interrupts as the shared critical regions are shared with low-priority tasks or interrupts. The hardware interrupts occurrences must be limited with an appropriate recovery strategy, in case of such failures.

In contrast, Cat1 interrupts must be configured as Trusted, as proposed in Table 1, since:

(1) blocking all interrupts eliminates the execution timer monitoring of such interruptions,
(2) not supported by spatial and temporal protection configured in the OS as they sup-port only Non-Trusted code to detect and prevent time or space overruns, and
(3) the usage of simple scheduler that disables interrupts.

Thus, in such critical systems, Cat1 interrupts usage and frequency should be tuned, if and only if:

(1) Cat2 interrupts latency are low,
(2) small amount of jitter is required from interrupts, and
(3) inter-arrival rate of an interrupt increases with extra overheads due to nested interrupts or interrupt wrappers effects.

**Table 1.** Proposed OS application and MPU configurations for ISR categories given that there are ASIL-D SWCs in a software architecture

| Entity | OS Application | CPU Mode | MPU Configuration Set |
|---|---|---|---|
| OS | Trusted | Supervisor | 0 |
| CAT1 ISR/ TRAP | Trusted | Supervisor | 0 |
| CAT2 ISR | Non-Trusted | User 0 or 1 | 1 |

To control timing faults in runtime for a multicore architecture, firstly, an interrupt/ task meets its deadline, if the fixed-priority preemptive OS is accurately configured with Scalability Class 2 (SC2) to have the OS timing protection safety mechanisms as:

(1) Monitoring the execution time budget upper bound for tasks/ Cat2 interrupts to detect when lower-ASIL tasks exceed the expected execution time, as represented in Figure 5(a). An exception shall be thrown when lower-ASIL task execution time exceeds the expected value specified during the task creation.
(2) Monitoring the uppers bound of resources/ peripherals blockage, locking budget and suspending all interrupts to prevent lower-ASIL SWCs from blocking higher-ASIL components execution due to excessive usage of the shared resources. Mutex, semaphore or spinlocks can be used by higher-ASIL SWCs to ensure mutual access to resources shared with lower-ASIL SWCs. An exception shall be thrown when lower-ASIL tasks continue using the shared resources for more than the maximum allowed interval specified during task creation.

Supervision of the lower bound among activated successive tasks (at running or at ready state for basic tasks and at waiting state for extended tasks) or Cat2 interrupts inter-arrival, as revealed in Figure 5(b). This means that interrupt overload protection monitors number of interrupts received on certain channel to be disabled temporarily once they exceed the expected limits (interrupt counter is reset). This is implemented inside an interrupt. After consuming the configured delay, the interrupt will be reenabled. If the interrupt overrun is detected again, then the interrupt will be disabled permanently until the next ignition cycle. Thus, the usage of interrupts that are based on external trigger signals shall be limited to the avoid interrupt overload. This is implemented inside man-ager function that is responsible of the interrupt. On other words, the interrupt overload mechanism protects higher-ASIL SWCs from being blocked/ delayed due to high CPU overload occurs because of the arrival of many interrupts.

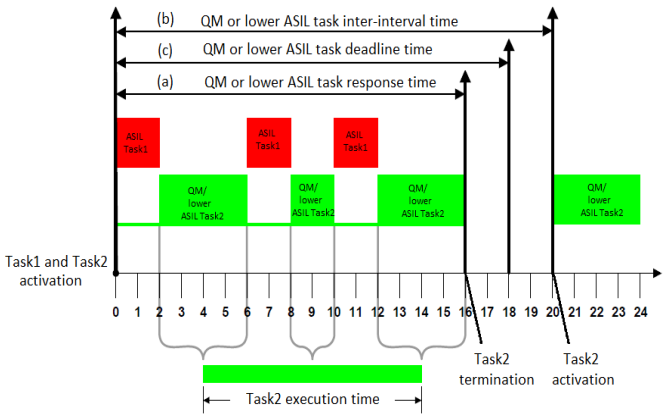


**Figure 5.** Timing diagram illustrated between ASIL task1 interfered with QM or lower-ASIL task2 (a) Practical execution timing (b) Inter-arrival timing (C) Deadline timing

Then, with the support of safety mechanisms built-in a hardware timer element, and with setting the relevant interrupt with a higher priority, the timing enforcement is promised. The mode of hardware watchdog shall be configured as Slow at initialization, as Fast at steady state, and as Off. Furthermore, interrupts latency time shall abide architecture real-time constraints.

### 4.2 Temporal protection safety mechanisms

Even though a safe behavior permits the system to detect and react on a failure during the FTTI, as illustrated in Figure 6, timing protection of AUTOSAR OS cannot individually assure exact timing protection in multicore architectures. Thereupon, it must be combined with temporal protection safety mechanisms to provide a fully timing protection to correctly identify tasks/ interrupts that cause timing faults.

In temporal protection, a non-safety code is forbidden to impact safety-related code timings. This is monitored by the qualified watchdog component as shown in Figure 7. The AUTOSAR watchdog manager SWC monitors SEs execution by triggering the watchdog hardware component. It periodically monitors the frequency (i.e. the configured occurrence number of cyclic checkpoints) during the OS periodicity range to feature the SE aliveness supervision.

On top of that, the watchdog manager monitors the time duration delay (not the exact timeout) of aperiodic consecutive checkpoints in a SE in case of irrelevant interrupts/ tasks are interfering with the SE execution. Hence, it features the deadline supervision, as interference delineated in Figure 5(c); to assure that the SE flow is meeting its deadline. The

watchdog manger shall check the timing before calling the next checkpoint, so as not to fail to detect non-occurrence of the second checkpoint. Thus, more checkpoints may be proposed to critical tasks or functions (at the expense of RAM consumption) as a runtime safety mechanism to make use of the watchdog supervision mechanisms.

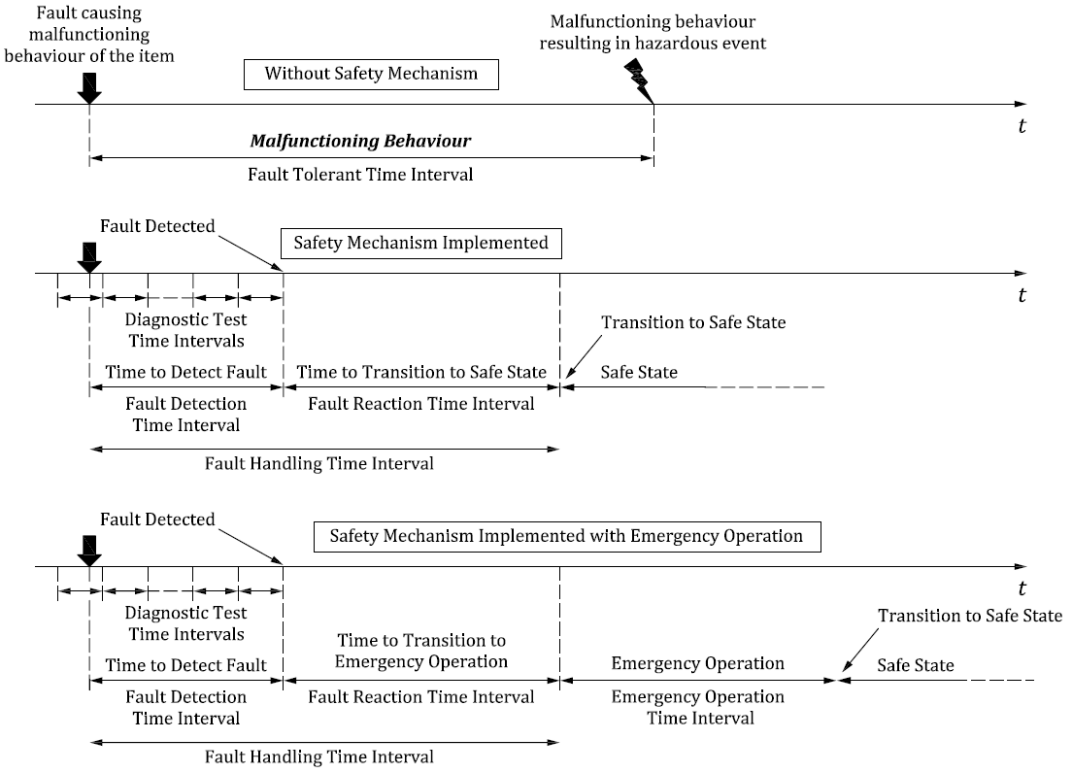


**Figure 6.** Achieving the safe state after applying a safety mechanism during the FTTI slot

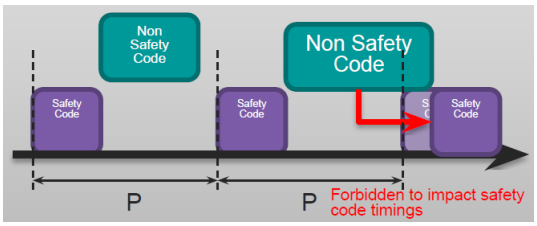


**Figure 7.** Temporal protection for the FFI performed by the watchdog component

Figure 8 shows a time span with 3 aliveness supervision cycles as a detection mechanism. In each cycle, checkpoints (CP1 and CP2) are hit once. Once the watchdog manager main function is called, the window for the next watchdog trigger is defined by WdgMTriggerWindowStart and WdgMTriggerConditionValue. Whereas Figure 9 and Figure 10 show the minimum and the maximum reaction time required by the watchdog manager because of the aliveness supervision, respectively. At first a checkpoint being hit first. Then, after the next checkpoint hit, the fault can be detected, which is due to the subsequent supervision cycle. Therefore, violation, detection, communication and system reset take place in the second call of the watchdog manager main function. In other words, the fault detection is placed at the end of the next supervision reference cycle for alive supervision.

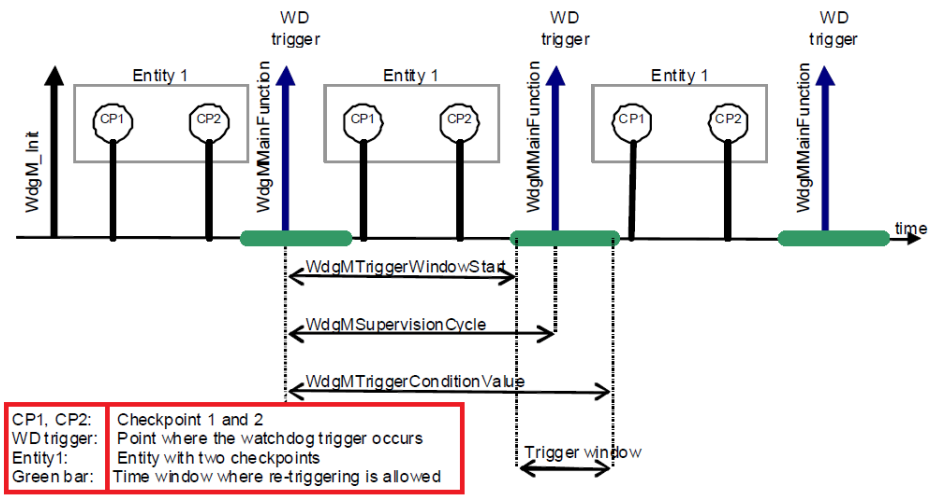


**Figure 8.** Aliveness supervision detection cycle of the watchdog manager

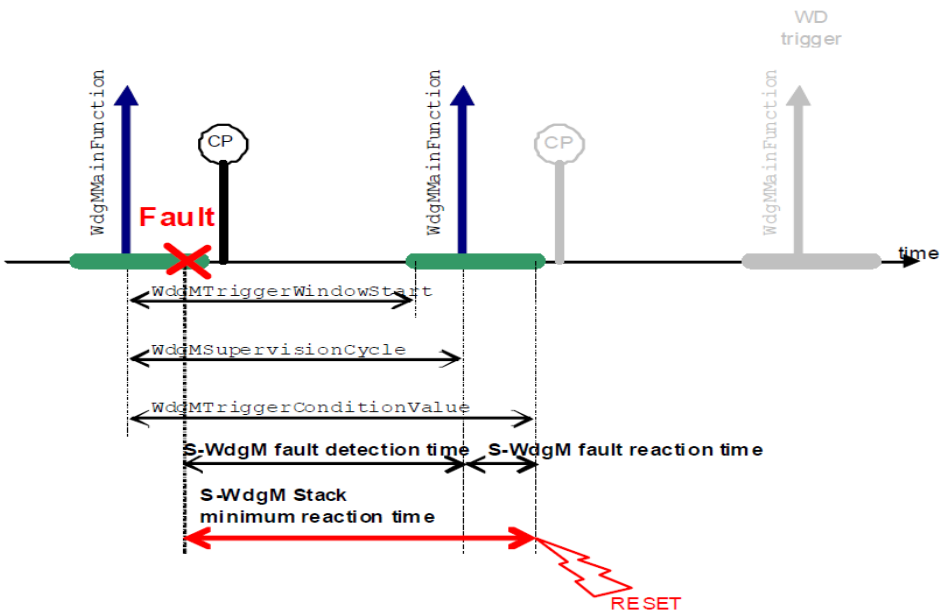


**Figure 9.** Timing diagram of the minimum reaction time required by the watchdog

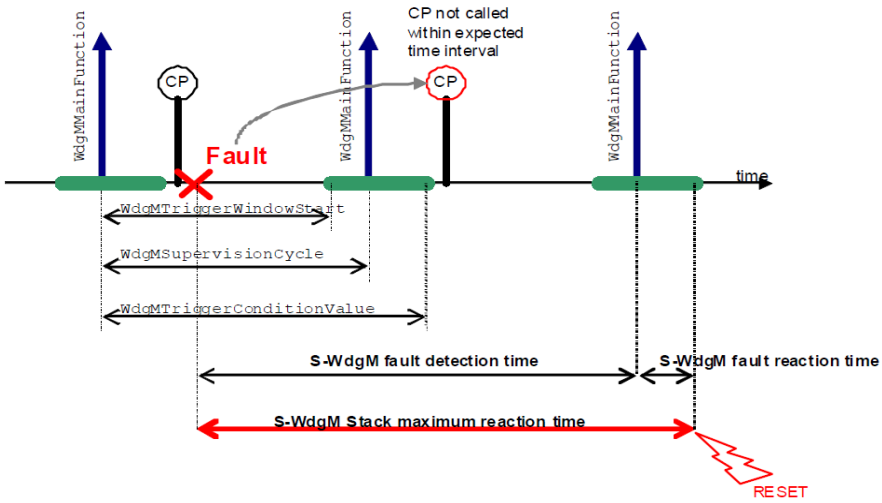


**Figure 10.** Timing diagram of the maximum reaction time allowed by the watchdog

### 4.3 Runtime monitoring safety mechanisms

In SC2, AUTOSAR OS experiences runtime monitoring, in which it verifies no QM task grants continued privilege to access interrupts hardware elements, or to operate with uncontrolled deadline. In addition to watchdog manager features illustrated in previous section, it performs the logical supervision, which monitors an accurate program flow order at runtime (i.e. the execution sequence of a SE that is represented in check-points transition directions according to its configured graph). Moreover, it will verify the checkpoints timings in the SE. However, the transition timing itself is verified by the deadline supervision featured by the watchdog manager SWC.

Figure 11 and Figure 12 represent the program flow monitoring mechanism of a multicore architecture. It is recommended be implemented for each core address the following challenges:

(1) no mutual checkpoints are involved in the SE of cores,
(2) checkpoint availability request placed in one core and called by the program flow monitoring placed in the other core,
(3) core interconnect synchronization to verify the interconnect acceptable jitter among the cores, and
(4) one-core program flow monitoring mechanism fatal failure that may need to activate its watchdog reaction mechanism synched accurately with reporting this status, to the other-core program flow monitoring mechanism.

In this case, the second program flow mechanism assesses the first-core failure reaction mechanism with the help of its watchdog manager, for the sake of activating its own failure reaction mechanism. Based on the system architecture constraints, different watchdog drivers may be interfaced to each core or a global watchdog may be utilized for all cores. The main purpose of this use-case is verifying the multicore initialization synchronization.

If a checkpoint is reached, SEs report to the watchdog manager SWC through function calls. An instance of the SE is created, for each core. Hence, concurrent SEs and overlapping checkpoints among SEs are limitations to that solution. However, it gathers and monitors all SEs logical sequence inside or among all cores that trigger the watchdog. In addition, in each core, the SEs run independently and can inform their status to the watchdog SWC over core boundaries.

Consequently, temporal protection and logical supervision of program flow sequences are utilized as safety measures of failure detection of either the hardware clock or the microcontroller unit.

A local SE failure status reveals once a confirmed failure reaches a SE where the detection mechanism occurs, while the global failure status of a microcontroller represents all gathered and combined local SEs failure status. The watchdog shall activate recovery mechanisms from such failures based on the global and local failure status.

Firstly, the watchdog manager SWC shall report such failures to the SE with the help of the RTE protection mechanism. Moreover, it stores a new SE failure with its relevant attributes that illustrate the faulty items with the help of the diagnostic manager. Thus, the SE shall react to recover from such failures based on those reporting mechanisms.

Secondly, if there is no watchdog hardware triggering performed by the watchdog stack (i.e. refreshments timeout due to checkpoints failures), a hardware reset to either the microcontroller or the whole Electronic Control Unit (ECU) is performed by the watchdog hardware element. After that, an initialization is progressed to free such hardware failures. The watchdog refreshment ensures the program execution monitoring as it covers all ECU and functional modes, their transitions and all exceptions. Lastly, a global SE failure affects the whole system mandates an immediate microcontroller reset may occur by the watchdog manager followed by an initialization to the watchdog element.

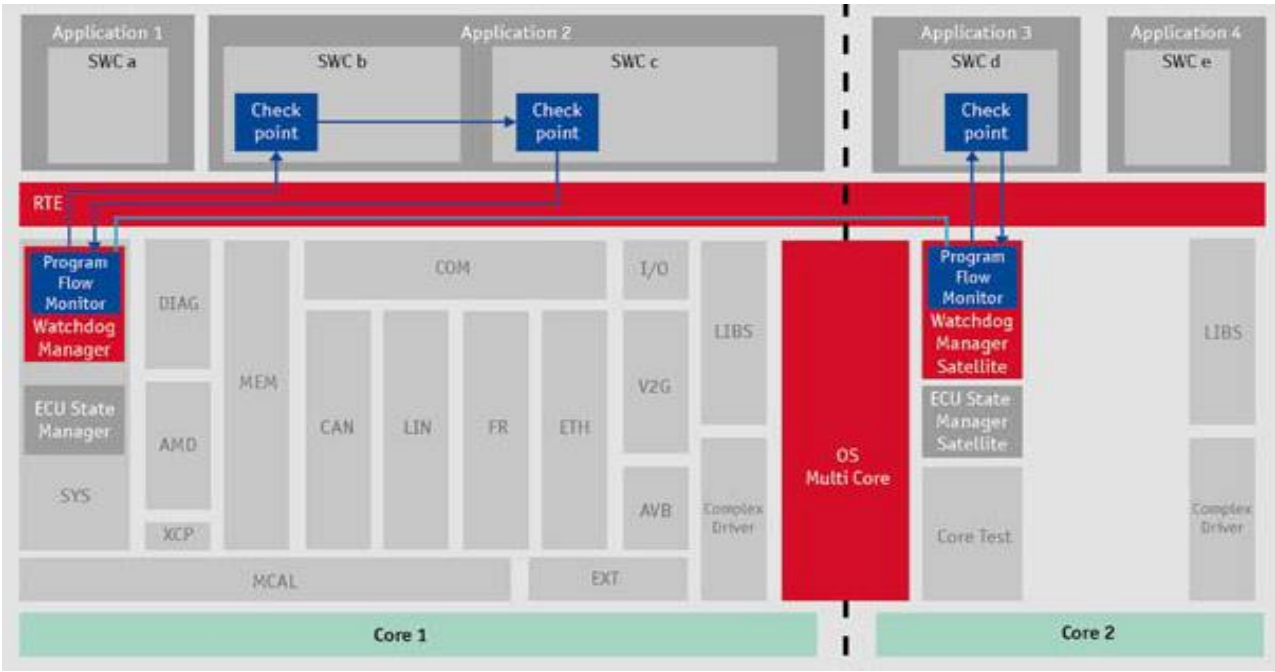


**Figure 11.** Watchdog flow monitoring safety mechanism in the AU-TOSAR layered architecture

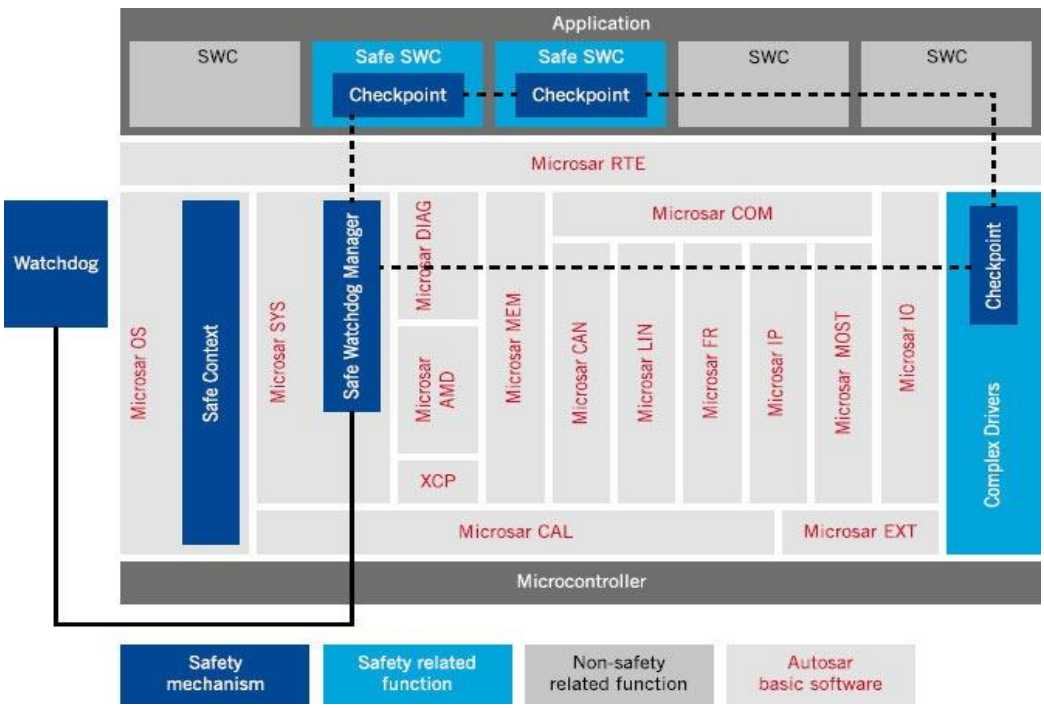


**Figure 12.** Watchdog flow monitoring safety mechanism in the Microsar layered architecture

### 4.4 Services protection safety mechanisms

During interaction between an ASIL OS application and QM or lower-ASIL OS services, the services calls (i.e. handled by StartupHook, PreTaskHook, PostTaskHook, Alarm Callback, Tasks, Cat1/ Cat2 interrupts, Shutdown Hook, Protection Hook and Error Hook) shall not corrupt the OS itself. Several service protection safety mechanisms are highly recommended to be configured in the OS if they are supported or to be developed in case of an inhouse developed OS.

Firstly, services in wrong context, which are not called from Cat1 interrupts (i.e. calling non-reentrant higher-ASIL services in reentrant context by lower-ASIL SWCs or calling out-of-order higher-ASIL services), shall not be processed as the OS is highly recommended to protect them against the Non-Trusted Cat2 interrupts by returning an invalid value or a call level error. These out-of-bound context services shall not produce any behavior once called. Besides, the whole OS services must be fully configured whether used or not for the OS objects related to an OS application. Meanwhile, the higher-ASIL SWC design shall assure the exclusive access to non-reentrant services. Reentrance shall be supported and checked if used.

Secondly, Non-Trusted QM or lower-ASIL OS applications may affect higher-ASIL OS applications indirectly though OS services that have a global context through means of service faults such as: non-safe service calls; or non-safe handling of either global data, or function input parameters, or function input/output parameters, or function return value, or wrong periodicity, or wrong function pointer (invalid pointer arithmetic, or memory corruption). As a result, the OS services calls context shall be restricted. Besides, they can perform trusted restricted actions (i.e. not shutting down the OS). Values of different tasks shall be selected to be unique and to keep large with proper hamming distance among different tag values to detect bit errors easily.

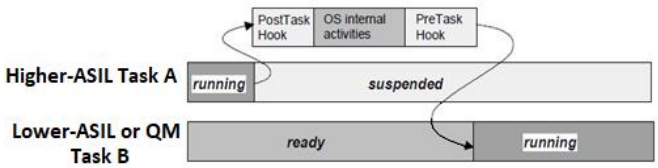


**Figure 13.** Protection hooks representation for between higher-ASIL Task A and lower-ASIL or QM Task B in the extended status OS

In general, restoring the wrong context for higher-ASIL tasks can be detected by using unique task context tag pushed into stack when task preempted and checked when task resumed. PreTaskHook is periodically called directly after a new task enters the running state, while PostTaskHook, is periodically called directly before the old task leaves the running state, as shown in Figure 13. Thus, GetTaskId does not return any issue, if the task is still/ already in the running state.

Thirdly, calls to undefined services are enough to make the OS behavior is undefined in an extended state and to be corrupted. The service protection shall describe all use-cases for such behaviors so as not to jeopardize the impacted OS application, the OS and the whole system. This shall be considered either for tasks that end without a termination, or for Cat2 interrupts that end with locked resources and interrupts, or out of order call (i.e. processing Post hook during shutdown call, processing interrupts without the corresponding disable or calling services during disabling interrupts). On top of that, Disable/ Enable interrupts shall support nested calls.

Fourthly, service calls with invalid objects not defined in the OSEK Implementation Language (OIL) or with out of range parameters (i.e. erroneous set of alarm cycle) shall not be processed and the OS shall return either an invalid identifier or an invalid value, respectively.

A configuration shall be done to permit Non-Trusted OS application with invoking Trusted OS services provided by the Trusted OS application with the help of the OS interrupt or trap. The OS shall verify the memory access rights allocated to the calling OS application against concurrent accesses, for proper memory protection, with such services to assert the memory left in the stack region. Every memory write access shall be conditioned by a writing request and a writing authorization, located in non-consecutive source code areas. Meanwhile, the maximum memory write duration shall be guaranteed.

Lastly, in multi OS applications, as in shown in Figure 1, controlling OS objects related to other Non-Trusted higher-ASIL OS applications by Non-Trusted QM or lower-ASIL OS application could provide an interference. Consequently, the QM OS application should not have such permissions to modify the ASIL OS objects. Moreover, the OS shall return an invalid identifier for that restricted access rights privileges. In case of an error detected during message reception, the reception buffer shall be reinitialized to ensure the erroneous previous message will not be used.

In case of failure, a safe reaction shall be performed in order to go to a safe state. Suitable error handling mechanisms shall be implemented in the OS with the intention of trapping such an erroneous state and even before an OS fault detection. This mechanism shall detect such protection errors, which are considered as software systematic faults generated in an OS application.

The protection errors are not limited to illegal service (i.e. unauthorized service call); memory access violations; timing faults (exceeding WCET); and hardware exceptions (i.e. illegal arithmetic instructions). An out-of-context occurrence of a protection error (i.e. during OS shutdown) leads to operating on an infinite loop, even before calling the reasonable mechanism. With the support of the watchdog, a microcontroller reset is activated due to this endless loop. A timeout limitation of the endless loop shall be verified to let the elapsed time measurement be performed before the process completion test.

Firstly, in SC3 and SC4, the application-specific startup hook mechanisms relevant to OS applications may be called by the OS after the OS startup call, to initiate other hook safety mechanisms. Secondly, in SC3 and SC4, the configured generalized error hook mechanism shall be activated before the application-specific error hook, which is activated if Cat2 interrupts/ tasks related to an OS application produce an error.

Thirdly, in SC3 and SC4, the configured generalized shutdown hook mechanism shall be activated after calling the application-specific shutdown hook, which is activated if the safety critical system begins to shutdown itself. It is preferably to have all application-specific shutdown hooks return parameters to the corresponding calling OS application so that the processing of the generalized shutdown hook is initiated.

Lastly, in SC2, SC3, and SC4, the protection hook

mechanism is called by the OS in a Trusted code to notify the means of protection errors take place at runtime. Depending on the return value of the protection error, the protection hook shall respond with either an OS shutdown, or a silent behavior, or an immediate termination of the current faulty Cat2 interrupt (while the newly requested/ waiting interrupts are invoked correctly), or an immediate termination of all interrupts (including newly requested/ waiting interrupts) and tasks related to a faulty OS application with/ without restarting the OS application.

There are limitations on choosing the OS scalability class as in SC2 there are timing protection, global time synchronization support, and protection hook features, while in SC3 there are MPU, OS application, other hook functions, service protection and Trusted functions features. Regardless all features can be configured in the SC4 OS, the SC4 OS is costly, and all SC4 proposed features might not be the aimed design choices for such a system architecture. As a result, this work is proposing multiple safety mechanisms to cover the gap of not being privileged with such OS features.

## 5. FUTURE WORK

The full scope of this work is to design a safety-compliant efficient multicore architecture that serve various autonomous driving applications to demonstrate the benefits of the proposed safety mechanisms to vehicle decisions (sensor fusion) of deep reinforcement learning. Assuring that higher-ASIL SWCs operate with no impact of the lower-ASIL or QM SWCs (i.e. guaranteeing the freedom from interference), regardless the used sophisticated architecture is, is mandatory to have a safe improved accuracy of vehicle decisions. The improvement in the safe vehicle operation obtained with fault injection verification asserts that there is still a lot of scope for improvement. The future work is summarized as follows:

(1) Experiencing, proposing safety mechanisms for possible ways of interferences (information exchange interference, shared peripheral interference) are the next step to have a fully compliant set of multicore architectures.
(2) Incorporating safe configuration of different set of multicore processor targets to tolerate means of hardware faults.
(3) Proposing and examining ISO 26262 compliant enhanced algorithms for sensor fusion for moving object detection, tracking, and calibration.

The author hopes that this study becomes a candidate to encourage for further deep research in exploring other real-time residual faults for perfect detection and reaction in cutting-edge nanoscale processors, or in web-based processors to validate the detection accuracy, or possible enhancements of the proposed safety mechanisms nature.

## 6. CONCLUSION

In this paper, safety-critical challenges of multicore architectures for autonomous driving applications have been explored, leveraged, analyzed and mitigated. These challenges represented for set of multi-cache multicore architectures in symmetric and asymmetric processors, critical timing, data coherency and synchronization predictability, core interconnects. Furthermore, various novel solutions, to each single challenge/ constraint, are proposed to present complex architectures designs to be compliant with the ISO 26262 methods and principals based on the examined system architecture ASIL. The proposed safety mechanisms target real-time faults detection and immediate reaction mechanisms, enough to let the system behave in the safe state before the defined FTTI.

Several proposed safety mechanisms to detect timing faults are combined as: timing monitoring, resource locking time protection, execution time protection and inter-arrival time protection with possible configurations improvements to Cat2 interrupts rather than Cat1 interrupts; as well as temporal protection are proposed: watchdog aliveness supervision, and watchdog deadline supervision. Whereas runtime flow monitoring and logical supervision detect the sequence faults with the support of the watchdog. Meanwhile, safe OS hooks configuration and safety mechanisms are proposed to detect all runtime services faults to higher ASIL OS applications, tasks and interrupts.

## ACKNOWLEDGMENT

The work presented here has been partially carried out for the framework of autonomous driving applications, which are supported by the TTTech Auto Iberia, Spain. The author would like to thank the expert staff of the Research and Development Center of Valeo.

## REFERENCES

[1] El-Bayoumi, A., Mostafa, H., Soliman, A.M. (2017). A novel MIM-capacitor-based 1-GS/s 14-bit variation-tolerant fully-differential voltage-to-time converter (VTC) circuit. Journal of Circuits, Systems and Computers, 26(5): 1750073. https://doi.org/10.1142/S0218126617500736

[2] El-Bayoumi, A., Salem, M.A., Khalil, A., El-Emam, E. (2015). A new Checkout-and-Testing-Equipment (CTE) for a satellite Telemetry using LabVIEW. In 2015 IEEE Aerospace Conference, pp. 1-9. https://doi.org/10.1109/AERO.2015.7119305

[3] Datta, A.K., Patel, R. (2014). CPU scheduling for power/energy management on multicore processors using cache miss and context switch data. IEEE Transactions on Parallel and Distributed Systems, 25(5): 1190-1199. https://doi.org/10.1109/TPDS.2013.148

[4] Schliecker, S., Negrean, M., Ernst, R. (2009). Response time analysis on multicore ECUs with shared resources. IEEE Transactions on Industrial Informatics, 5(4): 402-413. https://doi.org/10.1109/TII.2009.2032068

[5] Xie, G., Zeng, G., Li, R. (2020). Safety enhancement for real-time parallel applications in distributed automotive embedded systems: A stable stopping approach. IEEE Transactions on Parallel and Distributed Systems, 31(9): 2067-2080. https://doi.org/10.1109/TPDS.2020.2984719

[6] Kang, D., Kum, D. (2020). Camera and radar sensor fusion for robust vehicle localization via vehicle part localization. IEEE Access, 8(1): 75223-75236. https://doi.org/10.1109/ACCESS.2020.2985075

[7] Schalling, F., Ljungberg, S., Mohan, N. (2019). Benchmarking LiDAR sensors for development and evaluation of automotive perception. 2019 4th IEEE

International Conference and Workshops on Recent Advances and Innovations in Engineering (ICRAIE), pp. 1-6. https://doi.org/10.1109/ICRAIE47735.2019.9037761

[8] Iturbe, X., Venu, B., Jagst, J., Ozer, E., Harrod, P., Turner, C. (2018). Addressing functional safety challenges in autonomous vehicles with the arm TCL S architecture. IEEE Design & Test Magazine, 35(3): 7-14. https://doi.org/10.1109/MDAT.2018.2799799

[9] Martin, H., Winkler, B., Grubmüller, S., Watzenig, D. (2019). Identification of performance limitations of sensing technologies for automated driving. 2019 IEEE International Conference on Connected Vehicles and Expo (ICCVE), pp. 1-6. https://doi.org/10.1109/ICCVE45908.2019.8965181

[10] ISO 26262:2018- Road Vehicles - Functional Safety – Part 1–12. https://www.iso.org/standards.html, accessed on 2 Dec., 2018.

[11] Tobias, S. (2018). Safety analysis for highly automated driving. 2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), pp. 154-157. https://doi.org/10.1109/ISSREW.2018.000-7

[12] Sini, J., Violante, M., Dodde, V., Gnaniah, R., Pecorella L. (2019). A novel simulation-based approach for ISO 26262 hazard analysis and risk assessment. 2019 25th IEEE International Symposium on On-Line Testing and Robust System Design (IOLTS), pp. 253-254. https://doi.org/10.1109/IOLTS.2019.8854385

[13] Adedjouma, M., Pedroza, G., Bannour, B. (2018). Representative safety assessment of autonomous vehicle for public transportation. 2018 21st IEEE International Symposium on Real-Time Distributed Computing (ISORC), pp. 124-129. https://doi.org/10.1109/ISORC.2018.00025

[14] Nag, P., Ghanekar, U., Harmalkar, J. (2019). A novel multi-core approach for functional safety compliance of automotive electronic control unit according to ISO 26262. 2019 5th IEEE International Conference for Convergence in Technology (I2CT), pp. 1-5. https://doi.org/10.1109/I2CT45611.2019.9033841

[15] AUTOSAR Layered Architecture. http://www.autosar.org/standards/classic-platform/release-40/software-architecture/general/, accessed on 6 Jul. 2017.

[16] Gupta, P., Singh, N.P., Srinivasan, G. (2019). A framework for real-time automotive applications to multicore platform in perspective of AUTOSAR. 2019 4th IEEE International Conference on Recent Trends on Electronics, Information, Communication & Technology (RTEICT), pp. 706-709. https://doi.org/10.1109/RTEICT46194.2019.9016689

[17] Agirre, I., Cazorla, F.J., Abella, J., Hernandez, C., Mezzetti, E., Azkarate-askatsua, M. (2018). Fitting software execution-time exceedance into a residual random fault in ISO 26262. IEEE Transactions on Reliability, 67(3): 1314-1327. https://doi.org/10.1109/TR.2018.2828222

[18] Piper, T., Winter, S., Schwahn, O., Bidarahalli, S., Suri, N. (2015). Mitigating timing error propagation in mixed-criticality automotive systems. 2015 IEEE 18th International Symposium on Real-Time Distributed Computing, pp. 102-109. https://doi.org/10.1109/ISORC .2015.13

[19] Pan, X., Jonsson, B. (2014). Modeling cache coherence misses on multicores. Proceeding of 2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2014), pp. 96-105. https://doi.org/10.1109/ISPASS.2014.6844465

[20] Naderializadeh, N., Maddah-Ali, M.A., Avestimehr, A.S. (2017). Fundamental limits of cache-aided interference management. IEEE Transactions on Information Theory, 63(5): 3092-3107. https://doi.org/10.1109/TIT.2017.2669942

[21] Hachem, J., Niesen, U., Diggavi, S. (2016). A layered caching architecture for the interference channel. 2016 IEEE International Symposium on Information Theory (ISIT), pp. 415-419. https://doi.org/10.1109/ISIT.2016.7541332

[22] Piovano, E., Joudeh, H., Clerckx, B. (2020). Centralized and decentralized cache-aided interference management in heterogeneous parallel channels. IEEE Transactions on Communications, 68(3): 1881-1896. https://doi.org/10.1109/TCOMM.2019.2960503

[23] Tellabi, A., Ruland, C. (2019). Empirical study of real-time hypervisors for industrial systems. 2019 IEEE International Conference on Computational Science and Computational Intelligence (CSCI), pp. 208-213. https://doi.org/10.1109/CSCI49370.2019.00042

[24] Zuepke, A., Kaiser, R. (2019). Deterministic Futexes: Addressing WCET and Bounded Interference Concerns. 2019 IEEE Real-Time and Embedded Technology and Applications Symposium, pp. 65-76. https://doi.org/10.1109/RTAS.2019.00014

[25] Xie, G., Chen, Y., Liu, Y., Li, R., Li, K. (2018). Minimizing development cost with reliability goal for automotive functional safety during design phase. IEEE Transactions on Reliability, 67(1): 196-211. https://doi.org/10.1109/TR.2017.2778070

[26] Frigerio, A., Vermeulen, B., Goossens, K. (2019). Component-level ASIL decomposition for automotive architectures. 2019 49th IEEE Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W), pp. 62-69. https://doi.org/10.1109/DSN-W.2019.00021

[27] El-Bayoumi, A. (2020). An enhanced algorithm for memory systematic faults detection in multicore architectures suitable for mixed-critical automotive applications. International Journal of Safety and Security Engineering, 10(4): 467-474. https://doi.org/10.18280/ijsse.100405

[28] El-Bayoumi, A. (2021). New safe reliable design methodologies examined by fault injection testing and Monte Carlo simulation: Tolerating shared-memory interferences in multicore architectures. International Journal of Embedded Systems, 1-12.

## NOMENCLATURE

| | |
|---|---|
| ADAS | advanced driver assistance system |
| LiDAR | light detection and ranging |
| OS | operating system |
| FMECA | failure mode, effect and criticality analysis |
| ASIL | automotive safety integrity level |
| QM | quality management |
| TDMA | time division multiple access |
| FFI | freedom from interference |

| | |
|---|---|
| FTTI | fault time tolerant interval |
| SWC | software component |
| IC | integrated circuit |
| IPC | inter-partition communication |
| WCET | worst case execution time |
| WCRT | worst case response time |
| MPU | memory protection unit |
| LRU | least-recently-used |
| FIFO | first-in-first-out |
| PLRU | pseudo least-recently-used |
| DMA | direct memory access |
| ECC | error correcting code |
| RAM | random access memory |
| RTE | run-time environment |
| SC | scalability class |
| ECU | electronic control unit |
| OIL | osek implementation language |