POLITECNICO DI TORINO
Repository ISTITUZIONALE

Software test libraries and real-time operating systems: A system integrator perspective

Original
Software test libraries and real-time operating systems: A system integrator perspective / Angione, F., Bernardi, P., Cantoro, R.. - In: JOURNAL OF SYSTEMS ARCHITECTURE. - ISSN 1383-7621. - 177:(2026). [10.1016/j.sysarc.2026.103854]

Availability:
This version is available at: 11583/3011528 since: 2026-05-28T13:47:16Z

Publisher:
Elsevier

Published
DOI:10.1016/j.sysarc.2026.103854

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright

(Article begins on next page)

07 November 2026

Journal of Systems Architecture 177 (2026) 103854

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

Software test libraries and real-time operating systems: A system integrator perspective

Francesco Angione *, Paolo Bernardi, Riccardo Cantoro

Politecnico di Torino, Corso Duca degli Abruzzi, 24, Turin, 10129, Italy

ARTICLE INFO

Keywords:
Software test library
Online testing
Real-time operating system
Software integration
FuSA
Software engineering
Real-time software requirements
Software dependability

ABSTRACT

In safety-critical systems, functional safety (FuSA) standards require online testing capabilities, for example Software Test Libraries (STLs), which are flexible functional self-test procedures executed to verify the SoC functional behavior. However, the growing complexity of scenarios requires effective hardware management and orchestration achieved by Real-Time Operating Systems (RTOSs). The rising popularity of STLs as online testing techniques is flowing into the coexistence of RTOSs and STLs, raising concerns about how to effectively integrate those two software modules.

This work proposes methodologies to correctly and effectively integrate STLs and RTOSs already developed from a system integrator perspective. It describes a portable library interface, called *PTLIX*, for correctly and effectively integrate STLs across different RTOSs. In addition, it describes analytical execution time and memory footprints requirements to effectively integrate the two software modules. In order to support the analytical requirements, different case studies are presented for STLs based on two different Instruction Set Architectures (RISC-V and PowerPC) integrated in different RTOSs, such as FreeRTOS, Micrium-C OS III and Zephyr.

1. Introduction

Both hardware and software are subject to real-time constraints in safety-critical Electrical/Electronic (E/E) systems, i.e., the entire system must react to an event in a predefined time frame. These requirements are typical of Real-Time Operating Systems (RTOSs), which manage and orchestrate effectively task execution and serve user-based requests during the mission mode [1,2].

In safety-critical systems, Functional Safety (FuSA) standards (e.g., *ISO 26262* [3] for automotive application, *IEC 61508* [4] or *RTCA/DO 178C* [5] for aerospace and industrial application, and *IEC 62304* [6] for biomedical application) require implementing software or hardware safety mechanism (SM) capable of performing periodic and online testing, system hardening and/or continuous fault detection and reporting.

In CPU-based systems, a common software SM technique is the introduction of Software Test Libraries (STLs), self-test procedures exploiting CPU-based instructions to verify the System-on-Chip (SoC) functional behavior [7–12].

The extensive use of RTOSs for orchestrating and managing hardware resources, combined with the deployment of STLs as online testing mechanisms, raises concerns about how to integrate these two most effectively. STLs verify hardware functionalities in an invasive manner; that is, they alter the CPU state and can momentarily affect system functionality.

The delicate process of integrating STLs in RTOSs has been partially analyzed in prior works. In [13], authors present various software architectures with their pros and cons; in [14], authors provide analysis on different nested interrupts scenarios (external or not) during the execution of STLs inside RTOSs, and solutions to overcome such problems by exploiting watchdogs. Moreover, companies involved in the development of STLs are providing high-level guidelines for STL integrators [15]. In literature, methodologies for migrating STLs to multicore have been presented [16,17]. These methodologies mainly consist of analyzing their single-core execution, in which critical sections are identified, and then scheduling tests to maximize the concurrency among different tests while maintaining fault detection capabilities. Although techniques for correctly scheduling by minimizing test execution time and maximizing test concurrency of STLs have been proposed in [16,18–21], integration methodologies within RTOSs are not provided.

Minimizing power consumption in modern embedded systems is crucial; however, the optimal approach often depends on the specific system requirements. For instance, in safety-critical domains, the primary objective is to ensure system safety, with power consumption being a secondary consideration. Hardware-oriented safety mechanisms,

* Corresponding author.
E-mail addresses: francesco.angione@polito.it (F. Angione), paolo.bernardi@polito.it (P. Bernardi), riccardo.cantoro@polito.it (R. Cantoro).

https://doi.org/10.1016/j.sysarc.2026.103854
Received 15 December 2025; Received in revised form 5 May 2026; Accepted 21 May 2026
Available online 27 May 2026
1383-7621/© 2026 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

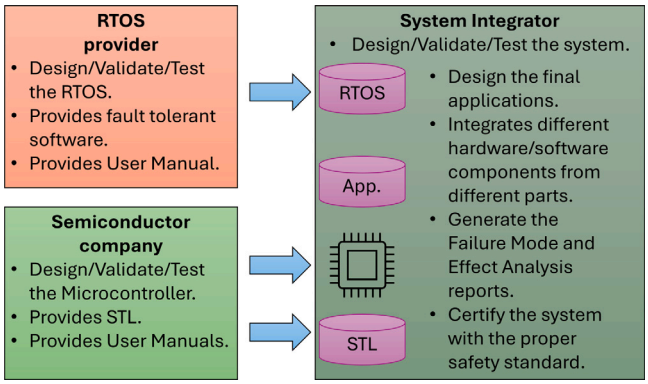


Fig. 1. Development-Integration process of STL, applications and RTOSs.

such as lockstep and Error Correction Code (ECC), typically consume less power but introduce additional area overhead. However, there are scenarios where hardware-based solutions are not available, and software-based safety mechanisms, such as STL, must be employed to meet the required safety standards. In addition, techniques for scheduling tests within STLs to minimize power consumption have also been proposed [22].

To the best of our knowledge, a collection of guidelines for correctly and effectively integrating STLs into RTOSs does not exist in the literature. The current landscape is fragmented, to some extent, in [23], authors solely analyzes the impact on the CPU utilization factor after the introduction of STLs; there is no standardized method for integrating STLs into RTOSs. Instead, system integrators are often faced with proprietary, vendor-specific interfaces that are opaque "black boxes" protected by strict Non-Disclosure Agreements. This lack of standardization prevents a systematic quantitative comparison of existing solutions and, more critically, forces developers to adapt STLs on a case-by-case basis. This manual adaptation significantly slows down the development flow and increases the complexity of porting applications across different RTOSs and hardware architectures.

The STLs are produced by STL providers in such a way that their fault-handling capacity is independent from other software properties, such as compilation strategies, code relocation, and system performance. Regarding fault-tolerant mechanisms (specifically, how to harden STL-related software modules relocated in memory against memory errors by using additional hardware safety mechanisms) they are presented at the system level [13,24]. Fig. 1 represents the development-integration process in software engineering where STLs are combined with applications and the RTOS.

The scope of this work is to fill the gap during the integration, from a system integrator perspective, of their developed applications, already developed STLs – usually, developed and graded on the gate-level design by test engineers – developed according state-of-the-art methodologies [14–16,18–21], and well-established RTOSs developed by RTOS providers according to common accepted methodologies [1, 25,26].

Eventually, RTOS providers can develop fault tolerant software implemented mechanism such as software redundancy, Cycly Redundancy checks (CRCs), canary, etc.

To bridge the gap between fragmented STL implementations and rigid RTOS requirements, this work provides for the integration step:

- A standardized software interface (PTLIX): it proposed a unified abstraction layer that decouples STLs from RTOS-specific constraints, enabling seamless integration across different RTOSs.
- Architectural Trade-off Analysis: it proposes a comprehensive evaluation of standard software architectures, identifying the optimal balance between performance overheads.
- Analytical methodologies for Feasibility: analytical methodologies are provided for calculating Execution Time and Memory Footprint requirements, allowing system integrators to verify integration constraints compliance and resource allocation during early design phases.

The focus of this work is on the integration of STLs into RTOSs, primarily from a system integrator perspective during the early phases of the integration process; i.e., software architecture, execution time and memory footprint aspects. As a result, safety metrics, risk mitigation strategies, and, more generally, safety analysis are outside the scope of this work; these elements require specific environmental and mission-mode assumptions that cannot be easily generalized, as they are highly application-specific. These aspects are typically addressed once the system is complete, as they depend on the environment, technology, and final customer requirements, including the targeted safety integrity level. Moreover, they also depend on overall system parameters that integrators must consider when performing Failure Mode, Effects and Diagnostic Analysis (FMEDA) or Failure Mode and Effect Analysis (FMEA).

The theoretical foundation of this work bridges RTOS scheduling theory with FuSA timing formulas [1,3,27,28]. The objective is to provide an analytical framework capable of pre-calculating the temporal constraints imposed by integrating safety mechanisms, specifically STLs, into the application and RTOS, and viceversa. By focusing on how these mechanisms impact scheduler feasibility and overall execution timing, the proposed approach allows system integrators to assess system-level viability before proceeding with full-scale development and any violation of the Worst-Case Execution Time (WCET) can be verified in advance.

Furthermore, this methodology is application-agnostic, as tasks scheduled by the RTOS are treated as "black boxes" defined primarily by their execution time (CPU capacity). By isolating these requirements, integrators can determine the remaining CPU overhead and verify if the system still meets real-time requirements after the STL is introduced. Ultimately, this formal modeling empowers system integrators to define and validate the integration of risk mitigation strategies during the early design phases of the final product.

The experimental evaluation reports on STLs compliant with PTLIX for different Instruction Set Architectures (ISAs), such as RISC-V and PowerPC, and on three different open-source RTOSs: FreeRTOS [29], Micrium-C OS III [30], and Zephyr [31]. The use of PTLIX has reduced the porting time from one RTOS to another in the case of RISC-V-based STLs. Experimental results showed the impact, in terms of memory footprint, of introducing STLs as flexible SM. Moreover, it emphasizes the criticality of carefully choosing and optimizing an STL to successfully integrate the SM into an existing RTOS concerning scheduling feasibility and CPU utilization factors.

The paper first establishes the fundamental background, in Section 2, of RTOSs, their classifications, and their software architectures. Afterward, in Section 3, it provides FuSA basic concepts and formulas used in the proposed analytical methodologies, and it describes STLs as a commonly used SM in safety-critical systems. Sections 2 and 3 illustrate the concepts of system availability from the RTOS perspective and meeting safety requirements from a FuSA timing perspective, providing the basis for the analytical integration methodologies proposed in this paper.

Sections 4 and 5 address the integration of STLs into RTOSs, proposing methodologies from both software engineering and analytical perspectives. Section 6 presents an experimental evaluation across various case studies, while Section 8 summarizes the findings and offers concluding remarks.

2. Real-time operating systems

A real-time operating system (RTOS) is an operating system (OS) for real-time applications for processing data and events that have critically defined deadlines [32].

All processing occurs within specified time deadlines, requiring a complete understanding of safety and application requirements. RTOSs are event-driven and preemptive, monitoring relevant events and the priority of the task set, switching between tasks based on their priorities.

Real-time tasks, and RTOSs, are usually classified depending on the consequence of a missed deadline [1]:

- *Hard-real time*, they must respect unconditional deadlines and guarantee timely responses in all possible scenarios. It is unacceptable in highly dynamic environments because it would lead to resource waste and a cost increase.
- *Soft-real time*, in case the produced results arrive after their deadlines, it still has some utility from a system perspective, even if it causes performance degradation.
- *Firm-real time*, when the produced results arrive after the deadlines, it is useless but does not cause any harm.

2.1. RTOSs software architecture

RTOSs and applications code and data can be tied together in different software architectures [33,34], with advantages and disadvantages, as Fig. 2 summarizes.

- *Monolithic architecture in a single memory space*, in Fig. 2(a), where there is no strict separation between the application and the RTOS, they are in a single memory space without any protection. Applications can freely call any RTOS components. Therefore, it provides most of the functionalities in the least possible space. On the other hand, errors due to faults propagation (in red) can freely propagate, corrupting the whole system.
- *Monolithic Architecture with processes separation*, in Fig. 2(b), where there is a distinct separation between the RTOS and application in different address spaces. It is a simplified software architecture used, for example, by AUTOSAR [35]. The application can call services delivered by a single monolithic OS component. Regarding faults propagating in the applications, they cannot propagate into the OS. In case some recovery mechanisms are provided, RTOS can restore application code and data and resume normal operations, certainly with some performance overheads. Meanwhile, faults propagation in the OS can corrupt the whole system.
- *Microkernel Architecture*, in Fig. 2(c), where each application or RTOS manager operates in a dedicated address space. It is reliable, secure, and easier to extend and port to new architectures. On the other hand, it presents performance overheads due to the communication between the application and a given RTOS manager and the passage through the system call interface. Errors in the application or an RTOS cannot corrupt the whole system.

3. Functional safety

Functional safety (FuSA) standards, such as ISO 26262 [3], IEC 61508 [4] or RTCADO 178C [5], aim at achieving the absence of risk due to hazards caused by malfunctioning behavior of (E/E) systems [36]. This implies that a malfunction of an E/E component could cause an unintended situation (Hazard), which can cause harm or damages (Risk). Therefore, FuSA requirements demands a risk reduction mechanisms [36,37]. Hereinafter, the paper provides some basic terminology on *ISO 26262* nomenclature.

A malfunction of an E/E component can be classified into two different failures:

- Systematic failures, deterministic failures that arise during development, manufacturing, or maintenance.
- Random failures, appearing during the useful lifetime. They can be further classified as permanent faults (e.g., stuck-at faults) and transient faults (e.g., Single-Event-Upsets).

Safety mechanisms (SMs) are technical solutions designed to detect faults or control failures, ensuring a safe state during mission mode. They can be categorized into hardware-based and software-based mechanisms [16]. Hardware-based safety mechanisms include:

- Fault detection and correction: examples are Triple Modular Redundancy (TMR), Lockstep computing, and End-to-End Error Correction Code for memories.
- In-field testing: Logic and Memory Built-in Self-Test (LBIST and MBIST) provide high fault coverage when correctly implemented, typically during Power-on Self-Test (POST) as periodic testing. FuSA standards also require online testing, tests executed concurrently with the mission applications.

Software-based mechanisms, for example, STLs, are better suited for online testing or for devices without hardware-based safety mechanisms to guarantee the needed safety level.

Depending on the fault detection capabilities, an E/E system reaches an *Automotive Safety Integrity Level* (ASIL) [36,38,39]. The behavior of an E/E component lacking Safety Mechanisms (SMs) is illustrated in Fig. 3; this represents the unsafe baseline where a fault can lead to a malfunctioning behavior that violates a safety goal. To mitigate potential random hardware faults during normal operation, FuSA standards require the introduction of a safety mechanism, as shown in Fig. 4. This SM operates within a defined latency to notify the system of a fault before a hazardous event occurs.

The **Fault Tolerant Time Interval** (FTTI), in *ISO2626*, is "*The minimum time-span from the occurrence of a fault in an item to a possible occurrence of a hazardous event, if the safety mechanisms are not activated it can become a system level hazard*", as Fig. 3 represents. MBMT stands for **Malfunctioning Behavior Manifestation Time**, and it is "*The minimum time span from the occurrence of the fault to the manifestation of the Malfunctioning Behavior at the system level*" [3]. While HMT is for **Hazard Manifestation Time**, and it is "*The minimum time span from the onset of the Malfunctioning Behavior to the violation of the Safety Goal*" [3].

On the other hand, if SMs are introduced in the system, the behavior is represented in Fig. 4 [40].

Where FHTI stands for **Fault Handling Time Interval**, and it represents "*The sum of fault detection time interval and the fault reaction time interval*" [3]. FDTI is the **Fault Detection Time Interval**, which is "*The time-span from the occurrence of a fault to its detection*" [3], and FRTI is the **Fault Reaction Time Interval** that represents "*Time-span from the detection of a fault to reaching a safe state or to reaching emergency operation*" [3].

The relationship between FTTI, MBMT, HMT, FHTI, FDTI, and FRTI can be expressed using the equations in [40], reported in Eq. (1), and (2). Eq. (2) can be used during the development a safety critical system, and it must never be violated to guarantee safety over failures for a given SM.

$$FTTI = MBMT + HMT \tag{1}$$

$$FHTI = FDTI + FRTI \leq FTTI \tag{2}$$

3.1. Software test libraries

There are various hardware-oriented SMs, such as watchdog timers (which detect and recover from system hangs through resets), error-correcting codes (which maintain data integrity), lockstep execution

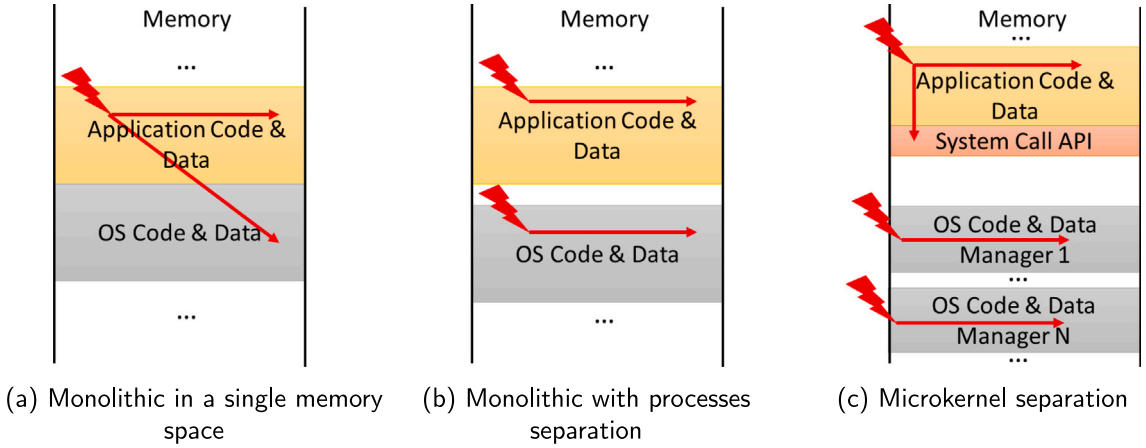


Fig. 2. Different software architecture for RTOSs and application(s), and propagation of hardware or malicious faults (errors) in the software modules.

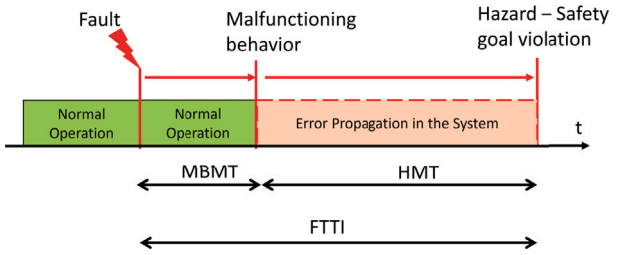


Fig. 3. Example behavior of an E/E components without Safety mechanisms.

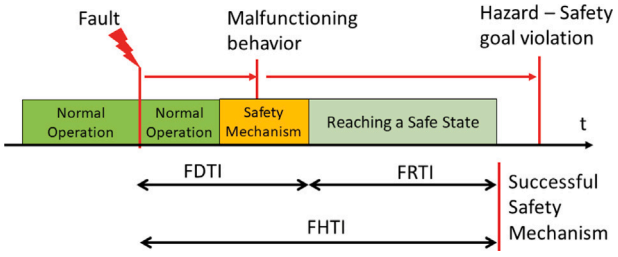


Fig. 4. Example behavior of an E/E components with Safety mechanisms.

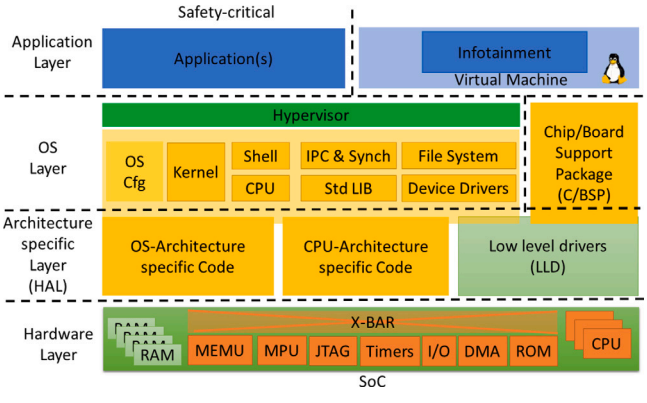


Fig. 5. Generic HW-SW stack for a safety-critical SoC.

(which duplicates operations for real-time fault detection), and BIST-based approaches (which allow systems to self-diagnose). These hardware mechanisms typically require additional area overhead or can reduce system availability (as with BIST-based methods [24]), and they demand careful development and hardware integration with the entire system. These mechanisms are often combined [14,24], including Software Test Libraries (STLs), in safety-critical systems to maximize reliability and ensure continuous operation even in the presence of faults.

Software Test Libraries (STLs) are more flexible compared to hardware-based SMs [41,42]. For example, they require minimal silicon area since they are stored in memory, unlike hardware SMs such as lockstep execution [14]. STLs provide a flexible online testing option for safety-critical domains and have a reduced development time compared to hardware-oriented SMs. In systems without any hardware SMs, STLs are often the only affordable solution for meeting minimal safety standards. Moreover, software can be updated if flaws are discovered or easily modified to meet the target ASIL requirements.

STLs consist of a collection of Software-Based Self-Tests (SBSTs) carefully developed explicitly in assembly to verify the functional behavior of a given unit [10,14,43–46]. STLs developers have limited awareness of the final RTOS, they develop the STLs according to the mission requirements and constraints.

The effectiveness of an STL is measured in terms of *Diagnostic Coverage* in *ISO 26262*, i.e., the capability of exciting and propagating a fault to an observable point. *Diagnostic Coverage* is the fault coverage detected by SMs without application dependent safe-faults, i.e., faults not causing hazards in the given application. By assuming the absence of safe faults, the *Diagnostic Coverage* is traceable to the fault coverage, and the development of the STL is agnostic from the application. STLs can be used as additional tests during manufacturing test flow for capturing faults escaped from other test strategies [47].

An important aspect during the development of STLs is that SBSTs can be divided into two macro type [7,14]:

- Intrusive or destructive: SBSTs for testing critical CPU parts. In order to do that, they uses special registers and memory locations without restoring the previous contents. Therefore, they are executed before the boot of the RTOS (Boot-time SBSTs), also referred to as periodic testing.
- Non-Intrusive (NI) or non-destructive: SBSTs for testing modules concurrently (also referred to as online testing) with the mission application, and they do not alter or change the system's current state (RunTime SBSTs). They usually cover computational modules such as arithmetic modules.

The macro division is already imposing constraints and defining the scheduled time of SBSTs. For example, intrusive SBSTs are scheduled at the system boot and can expansively alter the system's state without

compromising the in-field behavior. On the other hand, especially for developing Non-Intrusive SBSTs, they must tackle heavy constraints since they must be scheduled jointly with the user applications.

Authors in [7] provide guidelines for the proper development of non-intrusive self-tests:

- Special-Purpose Registers (SPRs) must not be altered. Therefore, its management is entrusted to the RTOS.
- Shared RAM preservation (with RTOS and/or the application) after the SBST.
- Bounded Maximum execution time, since by definition, RTOS has to provide real-time capabilities to the application. Therefore, a time slot must be singled out during the application execution.
- The SBST must be preemptable by other high-priority tasks or interrupt requests.
- No use of peripherals during non-intrusive tests since they alter the system's behavior.
- SBST must not generate interrupts nor exceptions.
- User Mode execution as the application does not have special privileges for accessing all the system resources.
- Disabled caches and not dynamic memory allocation for forcing deterministic execution.

To prevent hangs or other potential issues during the execution of SBSTs when interrupts occur, a dedicated test setup phase is performed at the beginning of each SBST. During this phase, a hardware watchdog is configured to safeguard against problems that may arise from the preemption of non-intrusive self-tests, thereby maintaining the necessary safety guarantees throughout execution [14].

Pondering the aforementioned considerations, integrating STLs into an RTOS requires devising a task to handle the runtime execution of SBSTs and strategies for the intrusive SBSTs.

4. Software architectures and standardized interface for STL-RTOS integration

RTOSs and applications are growing in complexity by adding more and more software layers on top of each other (see Fig. 5), by exploiting an hypervisor in order to separate the virtual machines from the safety-critical applications [2]. At the same time, RTOS are providing more complex services, not only the basic tasks scheduling but shell capabilities, Inter Process Communication (IPC) and synchronization, file system and device driver managers [35]; relying on a complex hardware structure.

Integrating STLs in such complex software and hardware environment is challenging. The integration is purely left to the capability and experience of integration/system engineers.

In the following subsections, guidelines to correctly and effectively integrate STLs and RTOS from a software engineering perspectives are proposed:

(A) Define a standard software interface between STLs and RTOSs called *Portable Test Library Interface* (PTLIX) and its Application Programming Interface (API).
(B) Illustrate safety and security pros and cons of different software architectures when integrating STLs in RTOSs.

4.1. Portable test library interface (PTLIX)

One of the key advantages of PTLIX is its flexibility; it allows for seamless integration across different platforms and RTOSs, which is crucial in a landscape where there is no standardized method for providing STLs. Most STL interfaces are proprietary and dependent on individual companies, leading to inconsistencies and these solutions are protected by strict NDAs and intellectual property rights, preventing a transparent, side-by-side analysis of their internal mechanisms or performance.

A systematic quantitative evaluation of these different STL interfaces is hindered by their proprietary nature, as such an analysis would require navigating complex NDAs and exposing sensitive vendor software architectures. Moreover, a critical limitation of these proprietary STLs is that they must be adapted on a case-by-case basis for different RTOSs and scenarios. This fragmentation forces developers to spend excessive time on manual adaptation, creating a bottleneck in the development flow that PTLIX aims to resolve.

PTLIX addresses this issue by offering a more universal solution, enabling system integrators to work more efficiently and effectively, regardless of the underlying RTOSs and STLs. *PTLIX*, and its documentation, are available on GitHub [48].

During the integration phase in the software development flow of an E/E system the application, the RTOSs, and STLs are tied together [49, 50]. The absence of standardized interfaces could potentially create software bugs, security and safety flaws.

Fig. 7 shows the proposed logical organization of the an STL and its subcomponent for reducing the integration, and portability, efforts among different RTOSs.

Every software subcomponent has a specific functionality:

- Configuration & Setup Info: it includes a set of defines and variables used to configure the underlying software in the STL. For example, by editing those defines, integration engineers can add, or remove, boot-time SBSTs, fine select the SBSTs to be executed, modify the linker sections in which the library is placed during the compilation and linking processes, enable/disable test setup for SBSTs such as the watchdog configuration and the MPU. In addition, it may contain the release version and other information.
- Error Management (EM): it is the software module responsible for handling information about the expected and actual results of SBSTs, which are uniquely identified.
- Scheduler: it is the software module in charge of handling the static scheduling (i.e., known at compile time) of SBSTs. For boot-time SBSTs, they are executed prior the RTOSs bootstrap since they are destructive, as shown in Fig. 6. On the other hand, the execution of RunTime SBSTs is slightly different, they can be executed in one row or splitted in subgroups for every time the RTOS invokes the scheduler function. As a wrapper module, it is capable of running tests in a bare-metal environment or integrating into an RTOS task for routine test scheduling.
- Data Types, common data types used across the STL.
- Common utilities, it contains the definition of keywords used within the SBSTs, general compiler dependent utils such as assembly macros for SBSTs.
- Portability layer, Test Setup Support Package (TSSP) together with the OS-related functions, this subcomponent is in charge of abstracting the CPU/SoC/OS specific function. They are implemented by integration engineers, and it is the only component that must be edited when changing RTOS.
- SBST separation in boot-time (intrusive) and Runtime (non intrusive) tests, and by targeted units (such as the CPU, glue logic, peripherals, etc.). The clear separation between SBSTs in the STL minimizes mistakes from the integration/system engineers.

Moreover, PTLIX can interface with proprietary bare-metal STL (provided in binary form and obfuscated to protect intellectual property) by directly integrating them into the CPU-dependent module (which is ISA and vendor-specific, as shown in Fig. 7), and by implementing the necessary software to interface with PTLIX's internal modules. PTLIX and its related STL can be integrated into the overall application and operating system by directly compiling all components together. Alternatively, they can be compiled out of the box and inserted into the OS as an external library, with the necessary calls made to their API.

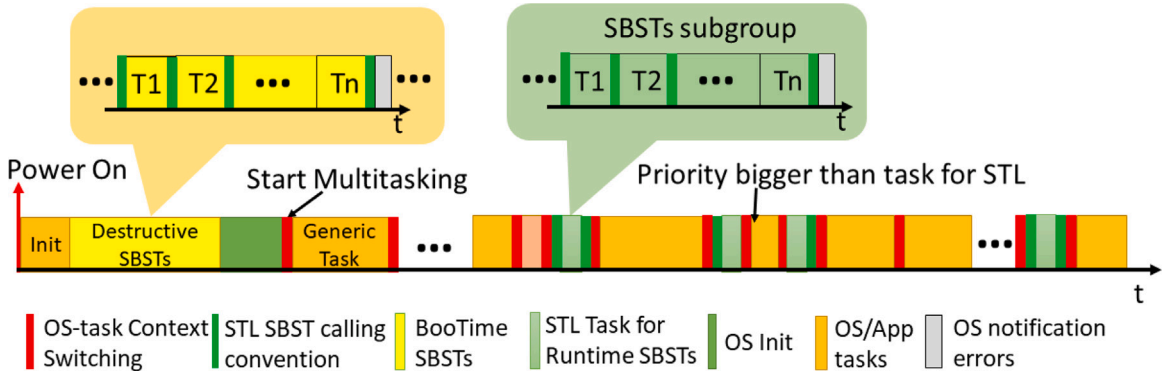


Fig. 6. Example of SBSTs in STL scheduling along with RTOS tasks and application.

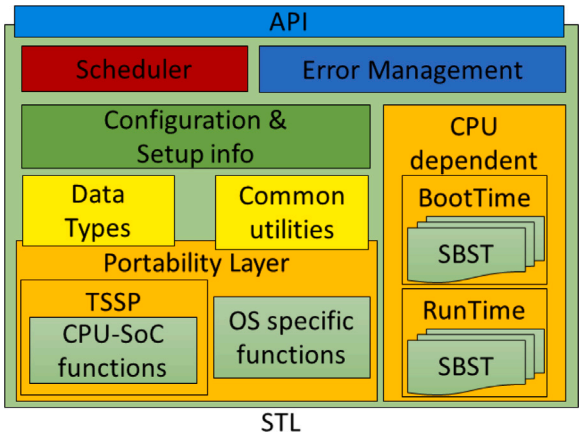


Fig. 7. PTLIX Logical organization.

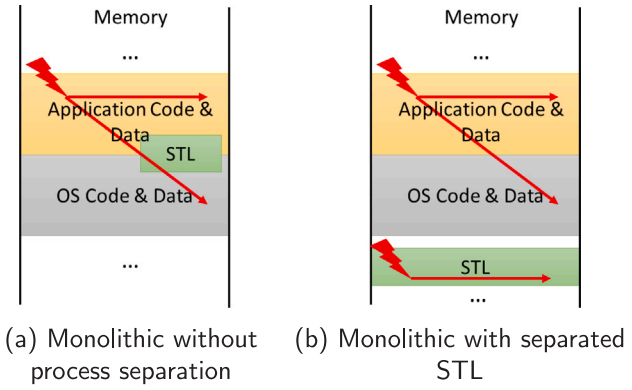


Fig. 8. Different Software Architecture for RTOS and application(s), and STL.

Regarding the API to/from RTOS, it is of a more delicate manner since it is the access point from and to the external software modules that integrates the STL. The goal of the proposed API [48] is to expose library function to the programmers without deep implementation details on the underlying software. As a consequence, APIs allow to speed up the development, or integration, process of more complex software modules, without exposing details of STLs, which could be encrypted for proprietary reasons.

4.2. Software architecture

An important aspect to tackle during the integration of STLs in RTOSs is the software architecture in which the two software modules, together with the applications, are going to coexist. It is important to mention that the software architectures hereinafter described are placed in the RAM memory. Same software architecture concepts can be also applied to the non-volatile memories (with reduced performances).

Furthermore, when STL is referenced in this section, it refers to runtime SBSTs and data needed for their correct execution. Boot-time SBSTs are not taken into account since they are executed before the bootstrap of the RTOS, and naturally they do not coexist in the RAM memory.

The following subsections discuss the advantages and disadvantages of the three main software architectures for OS in which the STL can be integrated.

4.2.1. Monolithic architecture without processes separation

In a monolithic architecture without processes separation, when considering both the RTOS and application perspectives, there are two primary ways to integrate the STL, as outlined in Fig. 8.

In the monolithic architecture without processes separation, the STL is tied together with the application code, and the RTOS.

Conversely, STL can be introduced as an independent module capable of communicating with the RTOS. This approach introduces a level of separation between the STL and the other components of the system, providing increased modularity and isolation.

As illustrated in Fig. 8, fault can propagate (indicated in red) within the software modules possible leading to system failures. In Fig. 8(b), the STL separation contributes to a reduction in system-level malfunctions. In this configuration, the STL can be reloaded or relocated in memory, and any potential fault propagation within the module is contained.

Moreover, in a monolithic architecture without processes separation, the granularity of Memory Protection Unit (MPU) configuration is significantly reduced since nearly everything can be accessed by every task. This can lead to security problems due to the broad access permissions granted to tasks within the system. Memory protection can be configure to some extend for the STL and the monolithic RTOS and application.

From a manual effort perspective, a monolithic architecture is generally simpler, as it does not require additional management of address spaces or process separation. In contrast, a separated STL demands extra effort: developers must carefully select and configure the appropriate address space, set up the MPU, handle code relocation, and ensure that both code and data are properly isolated from the rest of the application. This added complexity is necessary to achieve the required separation and safety integrity, but it does increase the manual workload during development and integration.

4.2.2. Monolithic architecture with processes separation

In a monolithic architecture with processes separation, both from the perspective of the RTOS and the application, there are three ways to

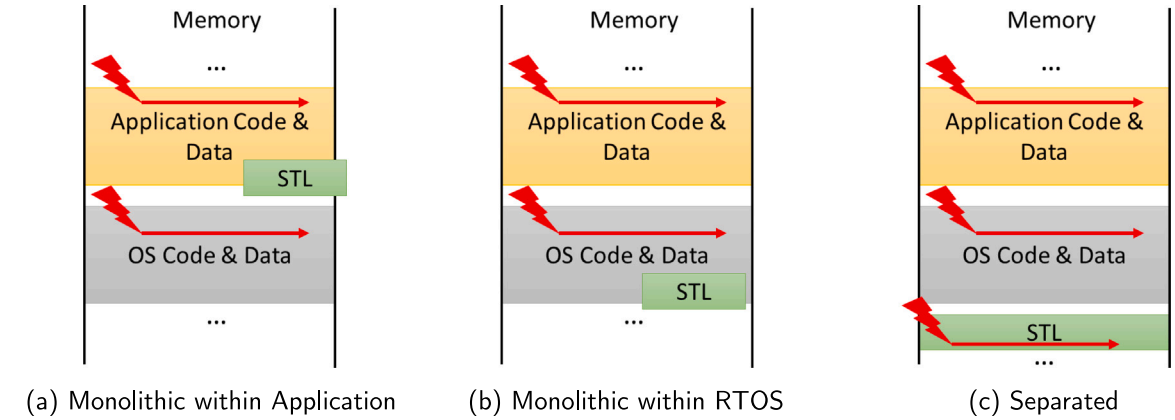


Fig. 9. Different Software Architecture for monolithic architecture with processes separation RTOS and application(s), and STL.

integrate the STL, as illustrated in Fig. 9. These integration approaches include incorporating the STL directly within the RTOS/application space, or alternatively, as a separate module.

This architectural approach distinctly separates the application and the RTOS, with a key focus on fault propagation, highlighted in red. The propagation of faults can lead to failure in the system.

For instance, as depicted in Fig. 9(a), a fault may propagate within the user space, potentially corrupting the entire application code. In this scenario, the RTOS can restore normal behavior if it possesses mechanisms to recover the current state of the application. Conversely, this does not apply to Fig. 9(b), where a fault propagating within the RTOS may generate an undefined behavior.

The STL can also be loaded as a separated module, as shown in Fig. 9(c), thereby confining fault propagation to a single module. This software architecture exploits MPUs to exercise a fine control over memory accesses from a task-oriented perspective.

From a manual effort perspective, a monolithic architecture (within the application or the RTOS) is generally simpler, as it does not require additional management of address spaces or process separation. In contrast, a separated STL demands extra effort: developers must carefully select and configure the appropriate address space, set up the MPU, handle code relocation, and ensure that both code and data are properly isolated from the rest of the application and RTOS. This added complexity is necessary to achieve the required separation and safety integrity, but it does increase the manual workload during development and integration.

4.2.3. *Microkernel architecture*

In a Microkernel architecture, when considering both the RTOS and application perspectives, there are three distinct approaches for integrating the STL, as summarized in Fig. 10. These integration options involve incorporating the STL directly within the RTOS module, within the application, or as a self-contained module.

Specifically, the STL can be integrated seamlessly with the application and the RTOS module, as depicted in Figs. 10(a) and 10(b), respectively.

In these described software architectures, faults freely propagate within the STL (highlighted in red), the application (Fig. 10(a)) or the RTOS module (Fig. 10(b)). However, the significance of fault propagation becomes less critical compared to Monolithic architectures if the RTOS offers mechanisms to restore the application or a specific RTOS module to a normal operational state.

Conversely, the STL can always be introduced as an isolated separated module, where any potential fault propagation remain confined within the module itself, as illustrated in Fig. 10(c).

From a manual effort standpoint, a microkernel architecture requires significant attention to address space selection, configuration, and process separation. Each service running in user space must be carefully isolated, and the MPU or similar mechanisms must be configured to enforce these boundaries. Additionally, developers need to handle IPC and code/data separation for each service, which adds complexity compared to a monolithic approach. On one hand, introducing the STL as separated module requires the same effort mentioned above for every microkernel module increasing the manual efforts during development and integration. The manual effort required can be slightly reduced if a monolithic architecture for the STL within the RTOS or application is chosen.

4.2.4. *Selecting the appropriate software architecture*

Selecting the appropriate software architecture is a critical task during both the integration and development phases, especially in safety-critical domains. The architecture chosen directly impacts the system's ability to meet stringent safety, security, and performance requirements. Fig. 11 presents possible decision making process for selecting a software architecture for the final system, based on a range of characterization factors relevant to E/E (Electrical/Electronic) systems. The factors are selected from the common criteria used in system architecture and requirements definition when a critical E/E system needs to be selected for development. These can be categorized into safety, security, and reliability [51,52]; performance [53]; physical constraints [54]; and economic and lifecycle factors.

Key factors influencing this selection include the definition of the safe domain—where safety-critical tasks are scheduled by the RTOS, and the secure domain, which is dedicated to security-oriented tasks. Additional considerations involve constraints on the overall memory footprint (RAM usage), the availability and configurability of a MPU, and the presence of tightly coupled memory (TCM) for code relocation, ensuring the CPU can access critical code without delay. Furthermore, the real-time requirements of the E/E system – whether hard or soft real-time – must be addressed, as these directly affect system performance and the ability to meet deadlines for critical tasks. Power consumption is also a crucial factor, particularly in applications where energy efficiency is paramount. By systematically evaluating these factors, engineers can select a software architecture that not only satisfies functional and non-functional requirements but also supports modularity, maintainability, and compliance with industry safety standards.

Moreover, it is crucial to highlight that the guidelines presented may not be exhaustive for all possible safety-critical scenarios. Instead, they should be viewed as a starting point for selecting a suitable software architecture, which can then be adapted or expanded based on the specific domain, user needs, and environmental requirements. The table does not account for the manual effort required for software integration; however, these efforts must be considered from a project management perspective to ensure realistic planning and resource allocation.

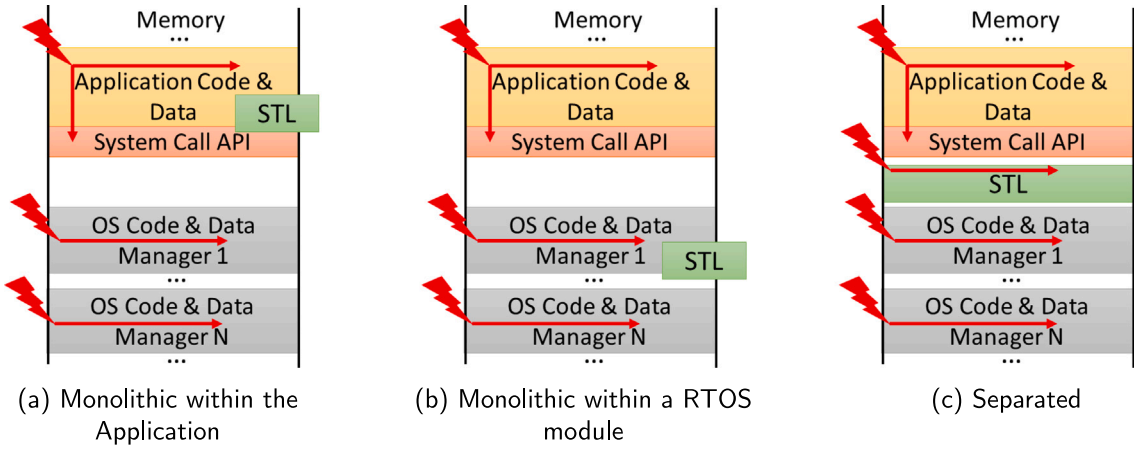


Fig. 10. Different Software Architecture for Microkernel RTOS and application(s), and STL.

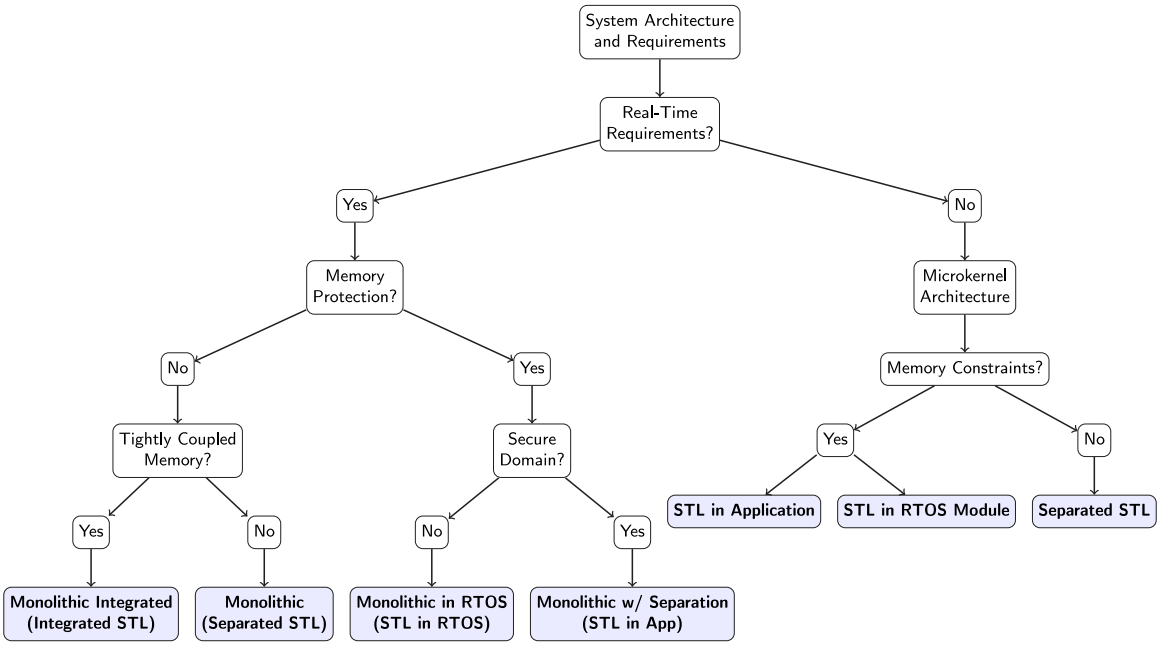


Fig. 11. Decising-making process for selecting a software architecture in safety-critical domains.

Additionally, Fig. 11 can be used not only as a guide for selecting a software architecture but also as a tool for verifying whether a device's hardware architecture is capable of supporting the identified software requirements. This bidirectional approach helps ensure that both software and hardware architectures are aligned to meet the system's safety, and performance needs.

5. Analytical methodologies for the integration

In the following, analytical methodologies are analyzed in order to impose constrains on the STL and/or RTOS and applications:

(A) Provide Execution Time requirements imposed by the applications and RTOS on the scheduling of SBSTs within the STLs.
(B) Provide Memory footprint requirements by tackling the requirements considering the system memories and the existing software of RTOSs and applications.
(C) Provide FuSa timing scheduler feasibility analysis.
(D) Provide error management consideration at RTOS level.

An important aspect to consider is the rising popularity of multi-core systems, which impose additional constraints and challenges on scheduling algorithms [25]. Although they introduce additional concerns, for multi-core systems, the STL responsible for testing the CPU must rely on a different task for each core. This is achieved through task affinity, and bypassing any load-balancing mechanisms if present. These tasks must be bound to specific cores, and their resources must be shared among the different tasks scheduled on that core. For the SBSTs responsible for testing other pieces of logic (outside the core), proper synchronization at the operating system level is required (this aspect is beyond the scope of this paper).

In the following, we present the runtime execution requirements for representative classes of scheduling algorithms for periodic tasks [1, 25], such as Rate Monotonic (RM) and Earliest Deadline First (EDF). In these algorithms, the STL is treated as a hard-deadline periodic task (as FuSa mandates), executed periodically alongside the RTOS and application tasks in a homogeneous preemptive system. The resource allocation for these tasks is assumed to have been defined offline, for example, using approaches such as those described in [55]. These considerations can be easily extended to other scheduling algorithms.

Execution time requirements are analyzed based on the single-core CPU utilization factor, as the STL is primarily bound to a specific core. From a FuSA timing perspective, there is no loss of generality in using

the utilization factor of a single-core CPU rather than conducting a scheduler feasibility analysis for a multi-core environment. The goal is to provide a FuSA timing analysis per core rather than per system.

5.1. Execution time requirements

The execution time of the overall system has to be deterministic and statically computed before the deployment of an E/E system in a safety-critical environment [1].

As for the development of SBSTs in STLs, the division in macro groups (intrusive and non-intrusive) can also be extended to the execution time requirements. For the sake of this chapter, all the above equations represent the execution time in terms of *Clock Cycles (cc)*, in order to provide a more consistent metric and hardware agnostic for comparison.

Intrusive SBSTs are executed in the field before the bootstrap of the RTOS and the scheduling of any other applications. Therefore, the execution time requirements for the Intrusive SBSTs in the STLs are represented in the following Requirement.

Theorem 1 (*Boot Time Execution Time*). ***Conditions:***

- $\mathcal{T}\{RTOS_{BOOT}\}$, *it is the RTOS-related parameter; it represents the bootstrap time of the RTOS and the configuration time of the peripherals.*
- $\sum_{i=0}^{n} C_i^{STL} + \epsilon$, *they are the STL related parameters, they represent the execution time of SBTSs in the STL (aka Capacity [1]) and the error management notification in case of a faulty unit, respectively.*
- Γ, *it is the mission parameter. It represents the maximum time in which the system has to start or to be halted in case a fault is detected.*

$$\begin{cases} \sum_{i=0}^{n} C_i^{STL} + \epsilon > 0 \\ \Gamma > 0 \\ \mathcal{T}\{RTOS_{BOOT}\} > 0 \end{cases} \tag{3}$$

Requirement:

$$\mathcal{T}\{RTOS_{BOOT}\} + \sum_{i=0}^{n} C_i^{STL} + \epsilon \leq \Gamma \tag{4}$$

On the other hand, the execution time for non-intrusive routines must be carefully defined, as dictated from FuSA standards, SMs based on STLs must be executed as a periodic task in order to verify the correctness of the underlaying hardware.

Theorem 2 (*Runtime Execution Requirements*). ***Conditions*** *Let us assume that τ is the task set for non-intrusive SBSTs in the STL, Π is the task set for RTOS tasks and applications, and k is the bearing for utilization factor in order to take into account interrupts.*

*By considering as safety mechanism an SBST in an STL, the execution behavior is represented in Fig. 12, where two subintervals of **FDTI** can be defined.*

*This means that the $SBST_i$ can detect the specific ith fault in the subunit it was designed to test. The $SMTI^i$ is the time interval for which the SM, in this case, an SBST for a given ith fault, is executed. Where the **FDTI** can be defined as the sum of **FSMTI** and **SMTI**.*

- $FSMTI^i$, *it is the Fault to Safety mechanism Time Interval, and it represents the time from an occurrence of the fault ith to the start of a Safety mechanism.*
- $SMTI^i$, *it is the Safety Mechanism Time Interval, it is the execution time of the Safety mechanism (an SBST in an STL for the module to test i-th), including the notification to the RTOS in case of a malfunction behavior is detected, and, if needed, the lock/unlock procedure for the shared resource in case of multi-core systems.*

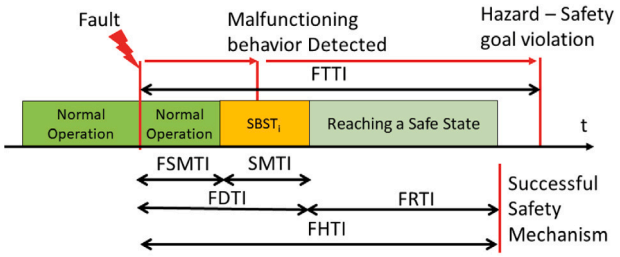


Fig. 12. Example behavior of an E/E component with SBST as Safety mechanism.

Let us assume the reaction time ($FRTI$) is constant among all the SMs. Let us assume the fault-tolerant time interval ($FTTI$) is constant among all the faults.

Moreover the $\forall i \in \tau$, means that An EDF scheduling policy must hold the following equations systems to integrate the STL into an RTOS effectively, i.e., the RTOS and the application impose the following constraint to the STL.

Requirement for EDF scheduling:

$$\sum_{i=0}^{n} \frac{SMTI^i}{FTTI^i} \leq ((1-k) - U(\Pi)) \tag{5}$$

Regarding the RM scheduling policy, it must hold the following equation systems in order to integrate the STL into an RTOS effectively.

Requirement for RM scheduling:

$$\prod_{i=0}^{n} \left(\frac{SMTI^i}{FTTI^i} + 1 \right) \leq 2 \cdot \frac{(1-k)}{(U(\Pi)+1)} \tag{6}$$

Proof. Firstly, the CPU utilization factor has to be defined [56] for periodic tasks in a task set Ω.

$$U(\Omega) = \sum_{i=0}^{n} U_i = \sum_{i=0}^{n} \frac{C_i}{T_i} \tag{7}$$

U_i is the CPU utilization factor for a given task, C_i is the time the processor executes the task without interruption, and T_i is the task period. In other words, given a set of n periodic tasks, the processor utilization factor (U) is the fraction of processor time spent executing the tasks.

Generally speaking, a scheduling algorithm feasibility (i.e., no deadlines are missed) for RTOS is computed according to Eq. (8), given a task set Ω and a scheduling policy.

$$Sched(\Omega, Policy) \leq SchedBound(Policy) \cdot (1-k) \tag{8}$$

The $SchedBound$ function represents the upper bound for which the scheduling is feasible, the $Sched$ function computes the CPU utilization factor for the given task set, and scheduling policy. In a real case scenario, there are also aperiodic tasks and interrupts; therefore, $(1-k)$ is a reduction factor for taking into account aperiodic tasks and interrupts.

Sched and SchedBound are scheduling algorithm dependent functions [1] as Eqs. (9) and (10) show for most common scheduling algorithms such as Rate Monotonic (RM) and Earliest Deadline First (EDF).

$$SchedBound(Policy) = \begin{cases} EDF & 1 \\ RM & 2 \end{cases} \tag{9}$$

$$Sched(\Omega, Policy) = \begin{cases} EDF & \sum_{i=0}^{n} \frac{C_i}{T_i} \\ RM & \prod_{i=1}^{n} (U_i + 1) \end{cases} \tag{10}$$

Let us divide the task set Ω into two distinct sets, τ for non-intrusive SBSTs in the STL and Π for RTOS tasks and application tasks. Therefore, from Eq. (8), the scheduling policy can be substituted according

to Eq. (10), and the task set division.

$$Sched(\tau \cup \Pi, Policy) \leq Sched\,Bound(Policy) \cdot (1-k) \tag{11}$$

Focusing on the EDF scheduling algorithm.

$$U(\Omega) \leq (1-k) \tag{12}$$

Moreover, the STL and RTOS-application task sets are separated.

$$U(\Pi) + U(\tau) = \sum_{i=0}^{n} \frac{C_i}{T_i} + \sum_{i=0}^{n} \frac{C_i^{STL}}{T_i^{STL}} \leq (1-k) \tag{13}$$

Where execution time requirements on non-intrusive SBSTs in STL can be computed as:

$$\sum_{i=0}^{n} \frac{C_i^{STL}}{T_i^{STL}} \leq (1-k) - U(\Pi) \tag{14}$$

Now, Let us recall Eq. (2) from FuSA Standard, which must hold for each hazardous event i in the task set τ.

$$FHTI = FDTI^i + FRTI^i \leq FTTI^i \quad \forall i \in \tau \tag{15}$$

By considering a safety mechanism, an SBST in an STL, the behavior is represented in Fig. 12, where two subintervals of **FDTI** can be defined. The **FDTI** interval can be defined as the sum of **FSMTI** and **SMTI**.

The C_i^{STL} is the time for the processor to execute the SBST-ith, detect a fault, and notify the RTOS. It is precisely the definition of **SMTI**. On the other hand, T_i^{STL} is the periodic deadline for which the task devoted to executing the SBSTs must verify the CPU behavior. In the worst-case scenario, T_i^{STL} is the tolerance time for which a SBST can safely detect a fault and reach a safe state. Therefore, for each T_i^{STL}, it can be safely assumed that it is equal to the **FTTI**^{i}, i.e., the fault ith captured by the SBST ith in the STL has a T_i^{STL} equal to **FTTI**^{i}. Therefore, Eq. (14) can be rewritten as:

$$\sum_{i=0}^{n} \frac{SMTI^i}{FTTI^i} \leq (1-k) - U(\Pi) \tag{16}$$

Eq. (14) imposes constraints on the runtime execution time of SBSTs in the STL from the RTOS and application task set for a CPU in the system. Regarding the Rate monotonic scheduling algorithms, the final equation system can be derived by taking into account all the aforementioned considerations.QED

5.2. Memory footprint requirements

Due to the size limitations for embedded memories, application, RTOS, STL code, and data section must coexist in the nonvolatile memories but, most importantly, in the RAMs.

The integration/system engineers are in charge of devising the correct memory layout to allow the coexistence of RTOS, application, and STL. It is expected that the STLs do not occupy a considerable memory footprint of the executable. Application and RTOS should imposes constraints on the memory footprint of the STL.

As mentioned before, STLs are developed and grouped into two groups: the intrusive SBTSs (or destructive) and the non-intrusive (or non-destructive). Two macro groups coexist in the non-volatile memory and, eventually, in the RAMs.

As for developing SBSTs in STLs, the division in macro groups can also be extended to the memory footprint requirements. For the sake of this work, the cache memories are not considered since they introduce indeterminism in the execution. In addition, all memory sizes are considered in terms of the number of bytes.

The executable must be stored in the nonvolatile memory, i.e., the ROM.

$$Size\{Executable\} \leq Size\{ROM\} \tag{17}$$

The size of an executable is divided into code and data section size (both read only and read write), additionally it contains executable info which is trascurable in terms of size.

$$\begin{aligned} &Size\{Sec(Code)\} + Size\{Sec(ROData)\} + \\ &+Size\{Sec(RW\,Data)\} \leq Size\{ROM\} \end{aligned} \tag{18}$$

It can be observed that every section of Eq. (21) is composed of RTOS, application, and STL parts.

Therefore, let us define the following:

$$\begin{cases} \Theta = Size\{Sec(Code^{STL})\} + Size\{Sec(ROData^{STL})\} + \\ \quad +Size\{Sec(RW\,Data^{STL})\} \\ \Lambda = Size\{Sec(Code^{RTOS+App})\} + \\ \quad +Size\{Sec(ROData^{RTOS+App})\} + \\ \quad +Size\{Sec(RW\,Data^{RTOS+App})\} \end{cases} \tag{19}$$

Where Θ represents the overall memory footprint in the ROM of STL, while Λ represents the joint memory footprint of RTOS and application.

Therefore, Eq. (19) can be substitute in Eq. (21).

$$\Theta + \Lambda \leq Size\{ROM\} \tag{20}$$

Therefore, ROM memory footprint requirements for STL can be written as following:

Theorem 3 (*Non-volatile Memory footprint*).

$$\Theta \leq Size\{ROM\} - \Lambda \tag{21}$$

Following the same philosophy of memory footprint requirements in the ROM, requirements for RAMs can be found. The assumption is that STLs are relocated as close as possible to the CPUs, along with the RTOS, application code, and data. Let us define the following:

$$\begin{cases} \theta = Size\{Sec(Code^{STL})\} + Size\{Sec(ROData^{STL})\} + \\ \quad +Size\{Sec(RW\,Data^{STL})\} \\ \lambda = Size\{Sec(Code^{RTOS+App})\} + \\ \quad +Size\{Sec(ROData^{RTOS+App})\} + \\ \quad +Size\{Sec(RW\,Data^{RTOS+App})\} \end{cases} \tag{22}$$

The lower case Greek letter represents the memory footprint in RAMs, compared to the capital letter in Eq. (19) for the ROM.

Generally speaking, if all the sections of code and data must be relocated in a single RAM (von Neumann Architecture), it holds the following equation:

$$\theta + \lambda + Size\{Stack\} + Size\{Heap\} \leq Size\{RAM\} \tag{23}$$

Where the stack size is deterministically computed by the linker and the application engineer. Meanwhile, the heap can be removed because there is no dynamic allocation in embedded systems for avoiding undeterminism [1], especially in safety-critical domains.

On the other hand, there could be separated memories for instruction and data (Harvard Architecture). Therefore, Eq. (23) become:

$$\begin{cases} Size\{Sec(Code^{STL})\} + Size\{Sec(ROData^{STL})\} + \\ \quad +Size\{Sec(ROData^{RTOS+App})\} + \\ \quad +Size\{Sec(Code^{RTOS+App})\} \leq Size\{IRAM\} \\ Size\{Sec(RW\,Data^{RTOS+App})\} + \\ \quad +Size\{Sec(RW\,Data^{STL})\} + \\ \quad +Size\{Stack\} \leq Size\{DRAM\} \end{cases} \tag{24}$$

Let define:

$$\begin{cases} \theta^{RO} = Size\{Sec(Code^{STL})\} + Size\{Sec(ROData^{STL})\} \\ \lambda^{RO} = \{Sec(ROData^{RTOS+App})\} + \\ \quad +Size\{Sec(Code^{RTOS+App})\} \end{cases} \tag{25}$$

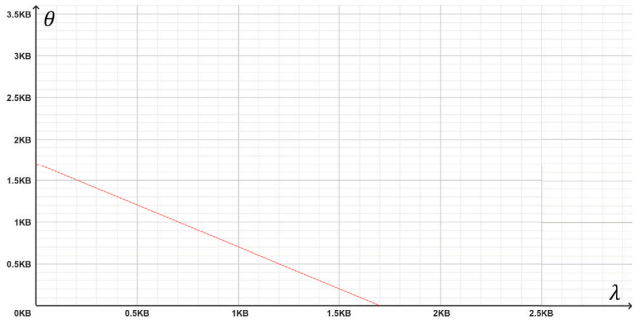


Fig. 13. STL memory footprint in terms of IRAM size, Application, and RTOS.

Therefore, let substitute Eq. (25) in Eq. (24), and write the STL requirements in the function of all the other parameters:

$$\begin{cases} \theta^{RO} \leq Size\{IRAM\} - \lambda^{RO} \\ Size\{Sec(RW\,Data^{STL})\} \leq Size\{DRAM\} - \\ \quad -Size\{Sec(RW\,Data^{RTOS+App})\} - Size\{Stack\} \end{cases} \tag{26}$$

Eq. (26) represents the inequality that the STL memory footprint must satisfy in order to be included in the RTOS and the application. An important aspect to consider is that terms in the equations above can disappear if they are located in other memories.

It can be observed that the first equation in (26) can be expressed in the cartesian space as θ in the function of a parametric IRAM size, and λ as x, as Fig. 13 shows.

The same can be applied to the second equation in (26), by grouping the DRAM and stack sizes, as well as ROM memories in Eq. (21).

As a consequence, it can be noticed that in Fig. 13, a triangle is the outcome graph, assuming all inputs are strictly greater than zero. All possible values under the read line (included) are feasible for the coexistence of RTOS, application and STL in memory.

Therefore, by exploiting some trigonometric properties, it can be defined the *STL shrinkage factor* as follows, visually in Fig. 13 the angle between x axis and the red line:

Theorem 4 (*Volatile memory footprint*)**.**

$$STL_shrinkage = \beta = atan\left(\frac{\lambda}{Size\{IRAM\}}\right) \tag{27}$$

Moreover, it must always hold strictly:

$$\lambda < Size\{IRAM\} \tag{28}$$

It represents how much the RTOS and Application weigh on the overall memory size and affects the STL's weight on the memory size (valid for RAM and ROM).

By supposing a λ equal to the $Size\{IRAM\}$, the argument of atan function becomes 1, which means an angle of 45°. However, in case λ is equal to the $Size\{IRAM\}$, it implies that the STL cannot coexist within the same memory.

The memory footprint in terms of data and code is strongly limited by the application, the RTOS, and certainly the memory sizes in the devices. Therefore, SBSTs, or more in general STL must have a low memory footprint on the overall code. Therefore, as a rule of thumb, the STL memory footprint should be less than 50% of the overall memory size (with a shrinkage factor of 26.57°) (See Eq. (29) given in Box I).

5.3. Executing the STL at RTOS-level

The execution of STL at the RTOS level requires, as for every other application, to devise a task in charge of handling the runtime execution of non-intrusive SBSTs concurrently with the application (the task is a hard real-time tasks with disabled caches and not dynamic memory allocation to have a deterministic behavior, missing the deadline can cause catastrophic outcomes due to the wrong behavior of the system). Depending on the

Most RTOSs [1] require two characteristics for a task:

- Stack Size, RTOSs for safety critical systems require to know the stack size at compile since dynamic stack sizes could create undeterminism.
- Priority, most of the schedulers in RTOSs are based on executing the task depending on their priority; consequently, the integration/system engineers must calculate the STL task priority by experience or experimental evaluations.

The following formula can be used to compute the stack size for the task devoted to execute the STL in the RTOS.

$$\begin{aligned} STL\,Task\,Stk\,Size = MinStkSize + \\ + \max_{\forall SBST_{NI} \in STL} \{StkSize(SBST_{NI})\} \end{aligned} \tag{30}$$

Eq. (30) is used for computing the maximum stack size for the task devoted to executing the STL, *MinStkSize* is the minimum required stack size for context switching of the RTOS (from RTOS documentation) plus the maximum value required by the SBSTs in the STL (from STL documentation).

On the other hand, the task's priority is computed according to the current information of the system, RTOS data, the number of non-intrusive SBSTs in STL, and the environment.

Firstly, let define the *Hyperperiod* of a task set [1].

Theorem 5 (*Hyperperiod*)**.**

$$H := lcm(T_1, \ldots, T_n) \tag{31}$$

It is the minimum interval of time after which the schedule repeats itself. If H is the length of such an interval, then the schedule in [0,H] is the same as that in [kK, (k + 1)K] for any integer $k > 0$. For a set of periodic tasks synchronously activated at time $t = 0$; the Hyperperiod is given by the least common multiple of the task periods (T_i) in the tasks set.

Therefore, the task's priority can be computed using Eq. 4.

Where the $Type_{RTOS}$ is a function that considers how the RTOS behaves in case of a missing deadline, i.e., it represents the slack for the STL-Hyperperiod. The value is computed accordingly to the RTOS classification [1] as Eq. (32) shows.

$$Type_{RTOS}(x) = \begin{cases} Hard & 1 \\ Soft & 0.5 \\ Firm & 0.8 \end{cases} \tag{32}$$

In Eq. 4, *MaxPrioOSTick* represents the priority associated with the RTOS tick, most of the time the task with the highest priority. Meanwhile, the inner part of the equation extracts the integer value of a *Criticality measure* of the task in charge of executing the non-intrusive SBSTs from the STL. The H^{STL}, the hyperperiod of STL, is defined as following:

$$H^{STL} := lcm(FTTI^1, \ldots., FTTI^n) \tag{33}$$

The criticality measure is composed of the ratio between the H^{STL} and the maximum value of the Worst-Case Execution Time (WCET) for non-intrusive routines. It is multiplied by the total number of non-intrusive routines in the STL divided by the clock frequency, for considering the rate at which every instruction of SBSTs is executed.

The final absolute value considers that the priority, from an implementation point of view, can be represented by a high integer number for low-priority tasks or by a low integer number for high-priority tasks.

$$Task\ Priority = \left| MaxPrioOSTick - \left\lfloor \frac{H^{STL}}{\max_{\forall SBST_{NI} \in STL}\{WCET(SBST_{NI})\}} \cdot \frac{\#SBST_{NI}}{f_{clk}} \cdot 1 + Type_{RTOS}(x) \right\rfloor \right| \quad (29)$$

Box I.

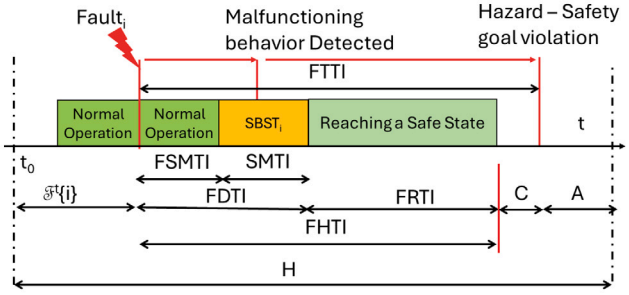


Fig. 14. Example behavior of an E/E component with SBST as Safety mechanism.

5.4. Scheduler feasibility - functional safety timing analysis at RTOS-level

The task devoted to the STL execution must satisfy a given CPU utilization constraint depending on the scheduler policy.

However, the time in which the fault manifests is not known at priori, which could lead to a hazardous event if the behavior is not analyzed analytically. For example, the SBST can miss the detection of a fault within the $FTTI$ constraints if the fault manifests after the related SBST execution.

In order to have a feasible scheduler for critical situations, from a FuSa perspective, the following requirements must be satisfied (with the assumption of a single fault in the SoC, and the presence of hardware-based SMs that can potentially detect multiple concurrent faults and notify the system). In addition, it is important to highlight that $FTTI_i$ is never greater than the Hyperperiod for every fault ith (by definition of Hyperperiod).

Theorem 6 (*FuSA scheduler feasibility - Single SBST Hyperperiod execution*). ***Conditions:*** *Let be $\mathcal{F}^t\{i\}$ the interval from zero to the occurrence of $Fault_i$. H is the Hyperperiod of RTOS, Application, and STL defined as follows:*

$$H = lcm(T_0, T_1, \dots, T_n, FTTI_0, FTTI_1, \dots, FTTI_m) \quad (34)$$

where T_i represents the deadline of RTOS and Application tasks, meanwhile $FTTI_i$ represents the deadline for a given $SBST_i$.

Moreover, Let us assume that:

$$\frac{H}{FTTI_i} = 1 \quad \forall i \in \tau \quad (35)$$

Eq. (35) states how many times a $SBST_i$ is executed in the overall Hyperperiod.

In order to have a scheduler feasibility from a FuSa perspective for capturing the fault i it must hold the following.

Requirement:

$$0 \le \mathcal{F}^t\{i\} < H - FTTI_i \quad (36)$$

Proof. Firstly, let us assume the situation represented by Fig. 14.

Moreover, an $SBST_i$ is executed one time in the Hyperperiod H, modeled by the following equation:

$$\frac{H}{FTTI_i} = 1 \quad \forall i \in \tau \quad (37)$$

Let us assume that the fault i can arrive after t_0:

$$\mathcal{F}^t\{i\} \ge 0 \quad (38)$$

The fault i is handled within the Hyperperiod:

$$\mathcal{F}^t\{i\} + FTTI_i < H \quad (39)$$

Let us recall from FuSa equations that:

$$FSMTI_i + SMTI_i + FRTI \le FTTI_i \quad \forall i \in \tau \quad (40)$$

Eq. (39) and (40) must be satisfied, where Eq. (39) can be rewritten in function of $FTTI_i$.

$$\begin{cases} FTTI_i \le H - \mathcal{F}^t\{i\} \\ FSMTI_i + SMTI_i + FRTI \le FTTI_i \quad \forall i \in \tau \end{cases} \quad (41)$$

Therefore, by assuming the fault i, the equations system becomes:

$$FSMTI_i + SMTI_i + FRTI \le FTTI_i < H - \mathcal{F}^t\{i\} \quad (42)$$

From Fig. 14, the Hyperperiod can be computed as the sum of intervals:

$$H = \mathcal{F}^t\{i\} + FSMTI_i + SMTI_i + FRTI + C + A \quad (43)$$

Where $FSMTI_i$ can be extracted:

$$FSMTI_i = H - \mathcal{F}^t\{i\} - SMTI_i - FRTI - C - A \quad (44)$$

And substituted in Eq. (42):

$$H - \mathcal{F}^t\{i\} - C - A \le FTTI_i < H - \mathcal{F}^t\{i\} \quad (45)$$

By removing $FTTI_i$ and summing $\mathcal{F}^t\{i\}$ to all terms:

$$H - C - A - FTTI_i \le \mathcal{F}^t\{i\} < H - FTTI_i \quad (46)$$

From Fig. 14, $\mathcal{F}^t\{i\}$ is also equal to:

$$H - C - A = \mathcal{F}^t\{i\} \quad (47)$$

Therefore, Eq. (46) becomes:

$$\mathcal{F}^t\{i\} = H - C - A - FTTI_i \le \mathcal{F}^t\{i\} < H - FTTI_i \quad (48)$$

With the initial assumption in Eq. (38), $FTTI_i$ is never greater than the Hyperperiod, and time slots C and A are never executed, it becomes:

$$0 \le \mathcal{F}^t\{i\} < H - FTTI_i QED \quad (49)$$

A fault i can arrive after the associated SBST has been executed. Consequently, the scheduler must ensure that the malfunctioning behavior is detected even in this worst-case scenario and the system is brought into a safe state.

The fault must arrive in a given time interval. Otherwise, there is a safety goal violation, as the following Requirement states.

Theorem 7 (*FuSA scheduler feasibility - Single SBST hyperperiod execution Worst case*). ***Conditions:*** *Let be $F^t\{i\}$ the interval from zero to the occurrence of $Fault_i$. H is the Hyperperiod of RTOS, Application, and STL defined as before. Moreover, let us assume that:*

$$\frac{H}{FTTI_i} = 1 \quad \forall i \in \tau \quad (50)$$

Eq. (50) states how many times a $SBST_i$ is executed in the overall Hyperperiod.

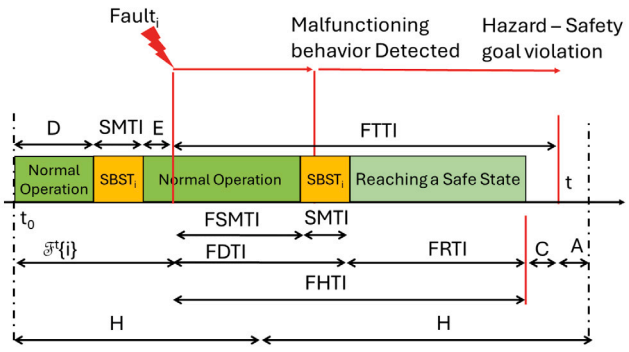


Fig. 15. Example behavior of an E/E component with SBST as a Safety mechanism in the worst case scenario, the fault i arrives after the $SBST_i$ has been executed.

In order to have a scheduler feasibility from a FuSa perspective for capturing the fault i, in case it arrives after the related SBST is executed, it must hold the following:

$$\begin{cases} E \leq 2H - FTTI_i - D - SMTI_i \\ H - FTTI_i \leq \mathscr{F}^t\{i\} < 2H - FTTI_i \end{cases} \tag{51}$$

Where D is the interval from the beginning of the Hyperperiod to the execution of the SBST, and E is the time interval after the execution of the SBST to the arrival of the fault i. In addition, it is crucial to notice that the previous requirement is valid when $\mathscr{F}^t\{i\} < D$; in this case, $\mathscr{F}^t\{i\} > D$.

In case the fault i arrives in the following interval.

Requirement:

$$SMTI_i + D \leq \mathscr{F}^t\{i\} < H - FTTI_i \tag{52}$$

The scheduler is not feasible from a FuSa perspective, i.e., a safety goal is violated.

Proof. Let us assume that the fault i arrives after the related SM (or SBST) is executed, as represented in Fig. 15.

Moreover, an $SBST_i$ is executed one time in the Hyperperiod H, modeled by the following equation:

$$\frac{H}{FTTI_i} = 1 \quad \forall i \in \tau \tag{53}$$

Let us assume that the fault i can arrive after t0:

$$\mathscr{F}^t\{i\} \geq D + SMTI_i \tag{54}$$

The fault i must be handled within the Hyperperiod:

$$\mathscr{F}^t\{i\} + FTTI_i < H \tag{55}$$

Visually, from Fig. 15, it means that the fault arrival time should lay between Eqs. (54) and (55).

$$D + SMTI_i \leq \mathscr{F}^t\{i\} < H - FTTI_i \tag{56}$$

This is impossible due to the assumption that a SBST is executed only once in the Hyperperiod, and by definition $FTTI_i$ is contained in the Hyperperiod (from Eq. (53)); consequently, the resulting scheduler violates a safety goal.

Let us see the case when the fault can handled outside the Hyperperiod, but it must be handled within the second execution of the Hyperperiod:

$$H \leq \mathscr{F}^t\{i\} + FTTI_i \leq 2H \tag{57}$$

Let us recall from FuSa equations that:

$$FSMTI_i + SMTI_i + FRTI \leq FTTI_i \quad \forall i \in \tau \tag{58}$$

Eq. (57) and (58) must be satisfied, where Eq. (57) can be rewritten in function of $\mathscr{F}^t\{i\}$.

$$\begin{cases} H - FTTI_i \leq \mathscr{F}^t\{i\} \leq 2H - FTTI_i \\ FSMTI_i + SMTI_i + FRTI \leq FTTI_i \quad \forall i \in \tau \end{cases} \tag{59}$$

From Fig. 15, two times the Hyperperiod H is:

$$2H = \mathscr{F}^t\{i\} + FSMTI_i + SMTI_i + FRTI + C + A \tag{60}$$

Where A can be eliminated from the equation since the system will never reach that execution time. In addition, Eq. (63) can be written in function of $FSMTI_i$, and substituted in Eq. (58).

$$2H - \mathscr{F}^t\{i\} - C \leq FTTI_i \tag{61}$$

By rewriting in function of $\mathscr{F}^t\{i\}$:

$$\mathscr{F}^t\{i\} \leq 2H - FTTI_i - C \tag{62}$$

Moreover, from Fig. 15, $\mathscr{F}^t\{i\} = D + E + SMTI_i$. Therefore, by substituting the equivalence in Eq. (62), the only unknown is E.

$$E \leq 2H - FTTI_i - D - SMTI_i QED \tag{63}$$

Another case is important to consider: SBTSs can be scheduled several times in the Hyperperiod, and their behavior has to be correctly defined from a scheduler perspective with the following requirement.

Theorem 8 (*FuSa scheduler feasibility - N-th SBST execution in the Hyperperiod*)**.** ***Conditions:*** *Let be $\mathscr{F}^t\{i\}$ the interval from zero to the occurrence of $Fault_i$. H is the Hyperperiod of RTOS, Application, and STL defined as before. The H^{STL} is defined as following:*

$$H^{STL} = lcm(FTTI_0, FTTI_1, \dots, FTTI_m) \quad \text{such that } \frac{H}{H^{STL}} = m \quad m > 1 \tag{64}$$

Moreover, let us assume that:

$$\frac{H}{FTTI_i} = n \quad \forall i \in \tau \tag{65}$$

In other words, Eq. (65) states how many times a $SBST_i$ is executed in the overall Hyperperiod.

In order to have a scheduler feasibility from a FuSa perspective for capturing the fault i, in case it arrives after the related SBST is executed, it must hold the following.

Requirement:

$$2H^{STL} - SMTI_i - FTTI_i < \mathscr{F}^t\{i\} \leq H - FTTI_i \tag{66}$$

In addition, it is important to notice that the previous theorem is valid when $\mathscr{F}^t\{i\} > H^{STL}$; in case $\mathscr{F}^t\{i\} < H^{STL}$, it is the case of Requirement - Single SBST Hyperperiod execution.

Proof. Let us assume that the fault i arrives after the related SM (SBST) is executed, as represented in Fig. 16.

Moreover, an $SBST_i$ is executed more than one time in the Hyperperiod H, modeled by the following equation:

$$\frac{H}{FTTI_i} = n \quad \forall i \in \tau, n \geq 1 \tag{67}$$

And the Hyperperiod of STL is defined as follows:

$$H^{STL} = lcm(FTTI_0, \dots, FTTI_n) | \frac{H}{H^{STL}} = m, m \geq 1 \tag{68}$$

Let us assume that the fault i can arrive after the SBST:

$$\mathscr{F}^t\{i\} \geq H^{STL} \tag{69}$$

The fault i must be handled within the Hyperperiod:

$$\mathscr{F}^t\{i\} + FTTI_i < H \tag{70}$$

Let us recall the following:

$$FSMTI_i + SMTI_i + FRTI \leq FTTI_i \quad \forall i \in \tau \tag{71}$$

Table 1
Case studies characteristics.

ISA	Device	RTOS			STL mem. footprint [KB]/([%)]		STL execution time [cc]	
		Name	Bootstrap time [cc]	Mem. footprint [KB]	intrusive	non-intrusive	intrusive	non-intrusive
PowerPC VLE 2.06	SPC58 family	Micrium C-OS III	1,357,050	364	131 (4.3)	16 (12.6)	271,530	15,267
RISC-V	pulpino SoC[1]	FreeRTOS	548,883	9	58 (1.9)	67 (52.0)	360,643	830,962
		Micrium C-OS III	452,666	127	58 (1.9)	67 (52.0)	360,643	830,962
		Zephyr	1,216,482	419	58 (1.9)	67 (52.0)	360,643	830,962

[1] Achieved by Logic Simulation.

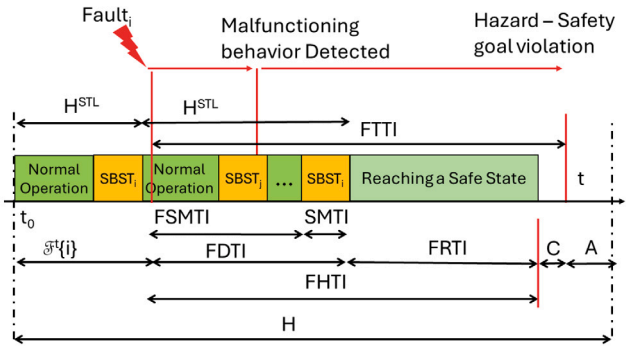


Fig. 16. Example behavior of an E/E component with SBST as Safety mechanism executed n times in the Hyperperiod.

From Fig. 15, the Hyperperiod can be computed as following, using Eq. (68):

$$2H^{STL} = \mathcal{F}^t\{i\} + FSMTI_i + SMTI_i \tag{72}$$

Where A can be removed since it is an allocated time frame in which the CPU does not execute any program because it reaches a safe state before. By rewriting Eq. (75) in function of $FSMTI_i$ and substituting into Eq. (71).

$$2H^{STL} - \mathcal{F}^t\{i\} - SMTI_i \leq FTTI_i \tag{73}$$

In addition, Eq. (70) imposes the upper bound for $FTTI_i$. Therefore:

$$2H^{STL} - \mathcal{F}^t\{i\} - SMTI_i \leq FTTI_i \leq H - \mathcal{F}^t\{i\} \tag{74}$$

By exploiting the equivalence in Eq. (68), adding $\mathcal{F}^t\{i\}$ and subtracting $FTTI_i$ to all members:

$$2H^{STL} - FTTI_i - SMTI_i \leq \mathcal{F}^t\{i\} \leq mH^{STL} - FTTI_i QED \tag{75}$$

From the previous requirement, the following can be deduced.

Theorem 9 (*N-th SBST execution in the Hyperperiod*). ***Conditions:*** *The fault i detected by its related SBST can arrive during the Hyperperiod but after the execution of the test routine.*

Requirement:

$$AH^{STL} - SMTI_i - FTTI_i \leq \mathcal{F}^t\{i\} \leq mH^{STL} - FTTI_i \tag{76}$$

Where $A \in \mathbb{N}$ and $A < m$, in order to have a feasible scheduler from a FuSa perspective, i.e., the SBST captures the faulty behavior before the FTTI.

Furthermore, aperiodic interrupts may arrive at the CPU. Therefore, the Hyperperiod can be defined as following.

$$H^* = H \cdot (1 + k) \quad 0 \leq k < 1 \tag{77}$$

Where H^* represents the Hyperperiod with aperiodic interrupts or tasks, H is the Hyperperiod of RTOS, application, and STL tasks. The factor $(1 + k)$ is an increasing factor depending on the value of k in the range $[0, 1)$. It represents the percentage of expansion of the original Hyperperiod for considering potential aperiodic interrupts or tasks, and it is application-dependent.

5.5. Error management at RTOS-level

The notification to the RTOS, and accordingly, the reach of a safe state in the system must be constant among all types of captured faults. This is mainly dictated by the propagation of information between tasks in the RTOS (Inter-Process Communication) to the activation of hardware safety mechanisms regarding a safe state (dictated by the SoC architecture).

From Eq. (2), the following equation can be derived, considering the interval divisions in the aforementioned requirements (valid for every fault ith in τ).

$$FSMTI^i \leq FTTI^i - SMTI^i - FRTI \quad \forall i \in \tau \tag{78}$$

The time in which a safe state is reached is the **FRTI**, which can be moved in the first member of the equation.

As a consequence, the following requirement can be defined.

Theorem 10 (*Reaction Time to Safe State Interval Time*).

$$FRTI \leq FTTI^i - SMTI^i - FSMTI^i \quad \forall i \in \tau \tag{79}$$

The above requirement imposes constraints on the RTOS to have a constant time interval for which a safe state must be reached; as mentioned before, it depends on the SoC architecture and the RTOS. It must be constant among all the faults that the SBTSs could capture in the task set of STL (τ). $FSMT_i$ can be extracted from system simulation, taking into account the previous requirements.

6. Experimental evaluation

6.1. Case studies

The case studies used in the experimental results are:

- A PowerPC VLE 2.06 based STL previously published [7,9,13,14,24] for a microcontroller of the SPC58 family. It has been evaluated on:
  - *Micrium-C OS III* [30] with a RM scheduler strategy.
- A RISC-V based STL developed (not optimized) for an open-source SoC (PULPINO) [57,58]. It has been evaluated on three different RTOSs:
  - *FreeRTOS* [29] with a RM scheduler strategy.
  - *Micrium-C OS III* [30] with a RM scheduler strategy.
  - *Zephyr* [31], it can be configured to use a scheduler strategy based on EDF or RM.

The porting time for the *PTLIX* library to each SoC was approximately 16 h for a single senior developer; the development of STL is not considered. However, after the initial porting, particularly in the case of RISC-V and different RTOSs, the integration time was reduced to around 4 h, primarily for validation purposes. Initially, developers may experience an increase in porting time due to the learning curve associated with PTLIX and its documentation. Since PTLIX is a newly proposed solution and not yet widely adopted, large-scale experimental

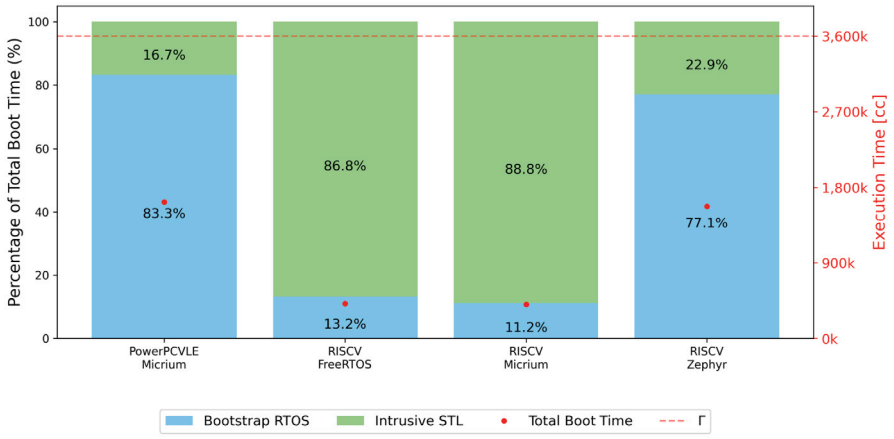


Fig. 17. Boot time comparison and STL weight for different case studies.

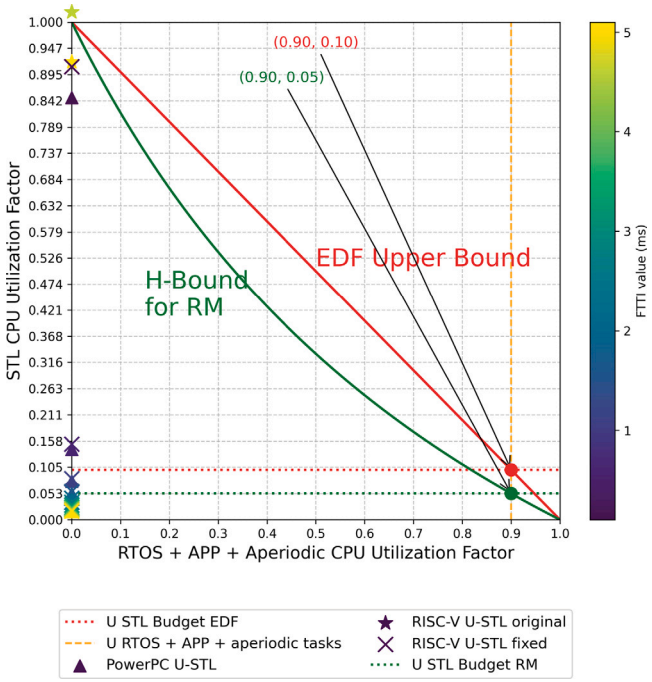


Fig. 18. Utilization space for STL, RTOS and applications.

data is currently unavailable. However, we estimate that once the interface is understood, the time required to port different STLs to different RTOSs is reduced, regardless of the developer's background.

PTLIX introduces a negligible computational overhead (less than 1 s) when executing SBSTs within the STL via the RTOS. However, it requires a memory footprint of approximately 32 KB to accommodate data structures containing test function pointers, golden and current signatures, and error-tracking information.

In all case studies, the STLs are implemented as standalone modules. This approach introduces additional overhead due to the need to configure the MPU unit and manage its relocation during execution. Table 1 summarizes the characteristics of the case studies in terms of ISAs, devices, RTOSs, and STLs. For the RTOSs, the bootstrap time and memory footprint (in ROM and RAM) are reported, and they depend on the initialized peripheral, as the same RTOS (Micrium) have different bootstrap time for different ISA. For the STLs, the memory footprint and execution time are provided for both intrusive (relocated in ROM) and non-intrusive (relocated in RAM) SBSTs.

Finally, the STL memory footprint is approximately 4.3% (ROM size of 3 MB) for PowerPC VLE ISA and 1.9% for RISCV ISA, which satisfies all memory footprint requirements. Regarding the memory requirements in the volatile memory (RAM size of 128KB), for PowerPC VLE ISA, the footprint is 12.6% that satisfies the requirement; meanwhile for the RISC ISA, the footprint is 52.0%, and it does not satisfy the requirement, thus, the STL should be further optimized.

The STLs, designed to concurrently test the device alongside the applications and RTOS, are introduced in the case studies as periodic hard real-time constraints, as dictated by FuSA standards. A priority is assigned according to Eq. 4, with a value of 5 (very critical), and an average stack size of 576 bytes, depending on the ABI and RTOS (computed based on Eq. (30)).

The RTOSs and STLs are integrated in such a way that no other critical application tasks conflict with them, even under high-load conditions, ensuring that the STL execution maintains a very high priority, as FuSa standards require.

6.2. Requirement: Boot time execution time

Fig. 17 represents of boot times for various platforms, RTOSs and STLs. On the x-axis, the platforms are labeled, including PowerPCVLE with Micrium, RISC-V with FreeRTOS, RISC-V with Micrium, and RISC-V with Zephyr. The left y-axis indicates the percentage of total boot time, while the right y-axis displays execution time in clock cycles (for providing a more consistent metric and hardware agnostic for comparison), labeled in red.

The primary feature of the plot is the stacked bar chart, where each bar is divided into two segments, in percentage. The sky blue segment represents the boot time for the RTOS, while the light green segment represents the boot time for the STL. This stacking allows for a clear comparison of how much each component contributes to the total boot time for each platform.

In addition to the stacked bars, red dots are plotted to represent the total boot time for each platform. These dots provide a quick visual reference for the overall performance of each platform. A dashed red line indicates the upper bound for which the execution is feasible (Γ) at 3,600,000 clock cycles.

Fig. 17 shows the impact of introducing the STL in simple RTOS implementations, such as FreeRTOS and Micrium-C OS III for RISC-V, compared to more complex bootstrap flows. Additionally, in all cases, there is a time budget for ensuring a feasible bootstrap process from a FuSA perspective; this budget is primarily dictated by the Γ value (environment and safety level driven).

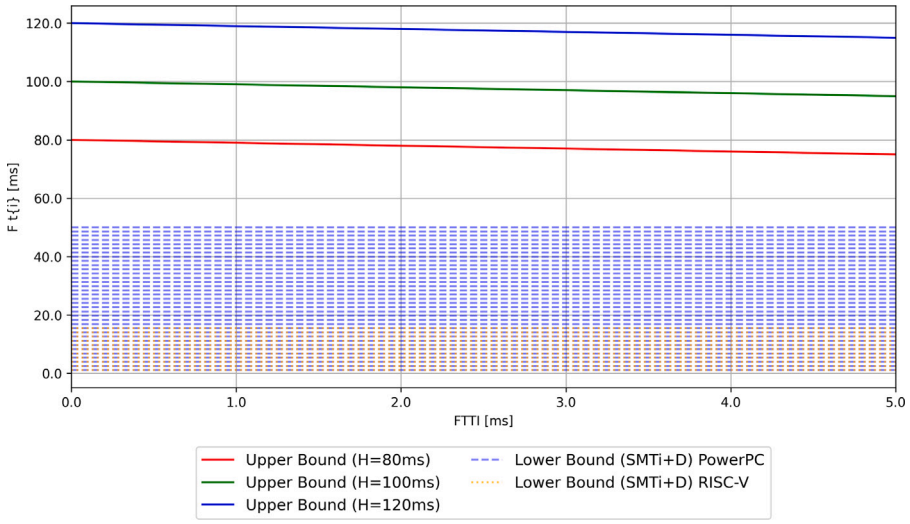


Fig. 19. Feasible arrival time cases for fault-th (y-axis) depending on the hyperperiod, different SMTI, and FTTI for Single SBST hyperperiod execution (Worst case).

6.3. Requirement: Runtime execution time

Fig. 18 illustrates the utilization space [1], showing the CPU utilization of different task sets, with the STL on the y-axis and the RTOS, applications, and potential aperiodic tasks on the x-axis.

The red curve indicates the upper bound for CPU utilization under the EDF scheduling algorithm, while the green curve represents the utilization bound for RM scheduling.

Various markers denote specific STL versions: stars for RISC-V U-STL original, crosses for RISC-V U-STL fixed (the number of runtime SBSTs is half of the original total number), and triangles for PowerPC U-STL. The colors of these markers correspond to FTTI values, which are indicated on the right side of the plot. SMTI for STLs has been computed using an FTTI starting from a value of 0.1 ms up to 5 ms. The assumption is that the utilization factor of the application and the RTOS is at the maximum (0.8), leaving 0.1 as additional slack to accommodate eventual critical interrupts.

Dashed lines represent different budget constraints, with the orange line indicating the U STL Budget EDF and the green dotted line showing the U STL Budget RM. Specific points on the plot are annotated with their coordinates, highlighting utilization factors at certain thresholds.

Overall, the scope of Fig. 18 is to provide a visual representation of the upper bound for which the scheduler remains feasible by introducing the STLs, based on the requirement for the runtime execution. The requirements introduce an upper bound for which the introduction of the STL keeps the scheduler feasible, with the red dotted line for the EDF scheduler and the green dotted line for the RM scheduler. STLs with a CPU utilization factor lower than these boundaries ensure that the system meets all deadlines. The green star outside the plot represents the version of STL for RISC-V that has an unfeasible utilization factor, even with a very high FTTI (by reducing the FTTI, the utilization factor of the STL increases). Therefore, RM schedulers (*FreeRTOS*, *Micrium-C OS III*, and *Zephyr*) can achieve a feasible schedule by introducing the STL (utilization factor below 0.05) with a medium-high FTTI for the RISC-V ISA; they are also feasible for EDF schedulers. PowerPC STL and RISC-V STL fixed with a very low FTTI can only be scheduled by the EDF scheduler (*Zephyr*), with a utilization factor below 0.10.

6.4. Requirement: FuSA scheduler feasibility

Fig. 19 presents the evaluation for the requirement on scheduler feasibility from a FuSa perspective when the SBST detecting a fault is executed only once in the hyperperiod.

Depending on the hyperperiod, the FTTI, and the SMTI for an SBST, the upper and lower bounds are defined. These bounds indicate the intervals within which the fault must not arrive; otherwise, the scheduler is not feasible from a FuSa perspective. This means that the SM (the STL in this case) responsible for capturing the fault will not detect the faulty behavior before the FTTI expires, leading to dangerous situations. The upper bound depends on the value of the FTTI, as indicated by the slope shown in Fig. 19. The same applies to the lower bound, but it primarily depends on the value of the SMTI. As seen in Fig. 19, different SBSTs in different STL may restrict the interval between the upper and lower bounds in the case of the PowerPC (with the dashed blue lines) or increase it in the case of the RISC-V (with the orange dotted lines).

The SBST can be executed more than once in the hyperperiod, and the requirements change, as Fig. 20 shows.

Although the aforementioned considerations are still valid, it is important to highlight that the lower bound depends on the hyperperiod of the STL (H^{STL}), which is contained within the total hyperperiod (H). An increasing value of H^{STL} raises the lower bound higher in the plot. In Fig. 20, there are no evident differences between RISC-V and PowerPC STL, since the H^{STL} is the same, and the SMTI has a slight influence on the lower bound in this case.

6.5. Discussion

The boot-time analysis reveals that the introduction of STL is highly sensitive to the underlying software abstraction; while simpler RTOS environments like FreeRTOS on RISC-V provide more slack before the required time to boot the system, they rapidly saturate to the maximum value as STL complexity increases.

Regarding runtime tests, the utilization space analysis proves that standard RM scheduling is often insufficient for unoptimized STLs, necessitating either a shift toward EDF schemes or an optimization of the STL must be done.

By analyzing the performance metrics on the FuSA feasibily intervals, the experimental evaluation shows that a scheduling in the classic sense (from the RTOS theory) does not necessarily guarantee a scheduling from a safety perspective. A system may meet its task deadlines but fail its safety goals if the fault arrival occurs within the detection gap defined by the $SMTI$ and $FTTI$. By increasing the frequency of SBSTs—represented by the H^{STL} within the total hyperperiod, the architecture can effectively reduce the detection gap. Consequently, the design of a resilient safety-critical system is not merely about

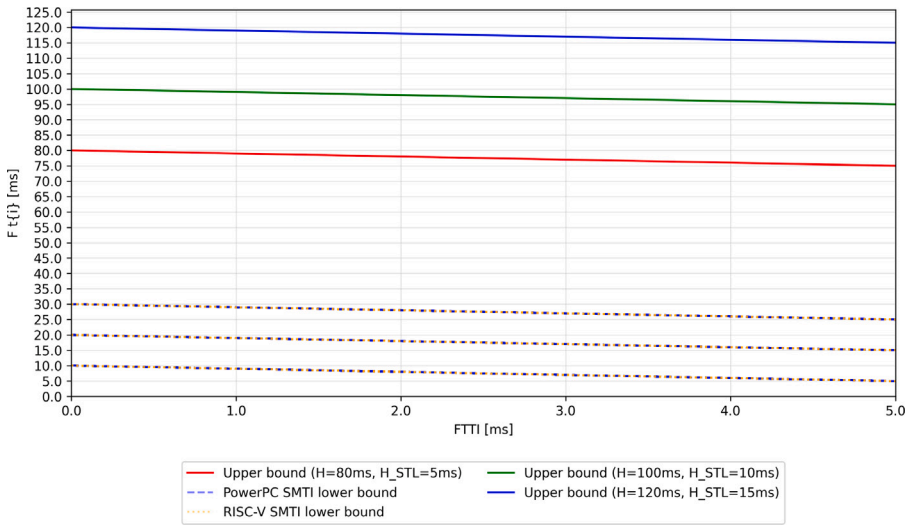


Fig. 20. Feasible arrival time cases for fault-th (y-axis) depending on the hyperperiod, different SMTI, H^{STL}, and FTTI for N-th SBST execution in the hyperperiod.

minimizing execution time, but about strategically balancing the H^{STL} frequency against the CPU utilization bound. This ensures that the system remains both computationally deterministic and temporally safe across the entire operational lifecycle, from the first millisecond of the bootstrap to the continuous monitoring in the field.

7. Related works

The introduction of *PTLIX* provides a standardized interface for STLs that complements outer middleware and existing frameworks like AUTOSAR. By acting as a unified interface for CPU-dependent tests and RTOS-oriented APIs, *PTLIX* eliminates the need for manual adaptation during each integration, potentially existing framework and RTOSs can already include abstract function calls to *PTLIX*. Its primary scope is to encapsulate SBSTs, allowing them to be seamlessly utilized by the RTOS, middleware, or other existing frameworks.

Regarding related work, a comprehensive collection of guidelines for the correct and effective integration of STLs into RTOSs—from both a software and API perspective—is currently absent from the literature. While the authors in [23] analyze the impact of STL introduction on the CPU utilization factor, they do not provide a standardized method for integration, nor do they address the impact on RTOS scheduling feasibility when introducing SMs.

The proposed work allow to improve the integration process of STLs and RTOSs by introducing software interface that eliminates "black box" proprietary hurdles, reducing manual adaptation effort; and by providing analytical methods to derive safety-critical scheduling constraints from FuSA requirements, ensuring RTOS tasks meet their deadlines post-STL integration.

8. Conclusion

Due to the shift to RTOS-based systems, STLs must be shifted and adapted. In the literature, an attempt does not exist to analyze and effectively guide the labile border of integrating STLs into RTOSs as a system integrator. As carried out from this work, the integration of STLs in RTOSs is not as smooth as it sounds; during the integration, some criticalities may arise.

To conclude, this work analyzes the integration of STLs in RTOSs from a system integrator's perspective. It covers diverse aspects ranging from scheduler feasibility (considering FuSA timing analysis per core rather than per system) and error management, to the memory and execution requirements of running STLs at the RTOS level. Furthermore, it addresses software engineering challenges by proposing a standardized interface. This interface bridges low-level SBSTs developed by silicon vendors with high-level integration, easing the adaptation of different STLs to various RTOSs and applications. Specifically, *PTLIX* is a standardized interface that aims to connect different STLs (for different ISAs and architectures) to different RTOSs.

KEY TAKEAWAY: System Integration process

Analytically assessing the overall system feasibility before the development is cost-effective.

This proposed methodologies allow system integrators to assess the overall system feasibility in advance before proceeding with development. This early validation can lead to significant cost savings, not only in terms of time but also regarding investment in the infrastructure for system validation and testing. The proposed work allow to improve the integration process in the following directions:

- Interoperability: The introduction of a software interface that eliminates "black box" proprietary hurdles, reducing manual adaptation effort.
- Timing and Resource Predictability: Analytical methods to derive safety-critical scheduling constraints from FuSA requirements, ensuring RTOS tasks meet their deadlines post-STL integration.

KEY TAKEAWAY: Interoperability & Portability

The introduction of a software interface for STLs reduces manual adaptation effort across different RTOSs and it eliminates proprietary hurdles.

KEY TAKEAWAY: Scheduling & Feasibility for FuSA

Classic RTOS scheduling theory does not inherently guarantee functional safety, i.e., the system may meets its task deadlines but fails its safety goals. System integrators must balance the SBSTs frequency in the STL (H^{STL}) against the CPU utilization bound.

Analytical expressions have been evaluated on STLs compliant with PTLIX for different ISAs (RISC-V and PowerPC) and on three different open-source RTOSs, such as FreeRTOS [29], Micrium-C OS III [30], and Zephyr [31]. The use of PTLIX has reduced the porting time from one RTOS to another in the case of RISC-V-based STLs. Experimental results showed the impact, in terms of memory footprint, of introducing STLs as flexible SM. Moreover, it emphasizes the criticality of carefully choosing and optimizing an STL to successfully integrate the SM into an existing RTOS concerning scheduling feasibility and CPU utilization factors.

In addition, there are rising concerns about the software security of embedded devices [59–61], STLs are an additional software module introduced into the system, and their security, in terms of timing and signatures correctness, should be evaluated by incorporating security concepts into the proposed analysis.

CRediT authorship contribution statement

Francesco Angione: Writing – original draft, Software, Methodology, Investigation, Conceptualization. **Paolo Bernardi:** Writing – review & editing, Methodology. **Riccardo Cantoro:** Writing – review & editing, Methodology.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

All the data are available in the experimental results.

References

[1] G.C. Buttazzo, Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications, third ed., Springer Publishing Company, Incorporated, 2011.

[2] P. Hambarde, et al., The survey of real time operating system: RTOS, in: IEEE ICESC, 2014, pp. 34–39, http://dx.doi.org/10.1109/ICESC.2014.15.

[3] ISO 26262-[1-10], Road vehicles – Functional safety, 2011.

[4] International standard - IEC 61508 - Functional safety of electrical/electronic/programmable electronic safety-related systems, Int. Electrotech. Comm. (2010).

[5] DO-178c, software considerations in airborne systems and equipment certification, 2012, RTCA SC-205, EUROCAE WG-12.

[6] International standard - IEC 62304 - Medical device software – Software life cycle processes, Int. Electrotech. Comm. (2012).

[7] P. Bernardi, et al., Non-intrusive self-test library for automotive critical applications: Constraints and solutions, in: IEEE DATE, 2019, http://dx.doi.org/10.23919/DATE.2019.8714780.

[8] A. Paschalis, et al., Effective software-based self-test strategies for on-line periodic testing of embedded processors, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. (2005) http://dx.doi.org/10.1109/TCAD.2004.839486.

[9] D. Piumatti, et al., An efficient strategy for the development of software test libraries for an automotive microcontroller family, Microelectron. Reliab. Vol. 115 115 (2020) http://dx.doi.org/10.1016/j.microrel.2020.113962.

[10] A.o. Riefert, On the automatic generation of SBST test programs for in-field test, in: IEEE DATE, 2015, pp. 1186–1191, http://dx.doi.org/10.7873/DATE.2015.0271.

[11] Cypress, AN204377 - FM3 and FM4 family, IEC61508 SIL2 self-test library, 2023, http://www.cypress.com/file/249196/download. (Accessed: 23 September 2023).

[12] MicroChip, 16-Bit CPU self-test library user's guide, 2023, http://ww1.microchip.com/downloads/en/DeviceDoc/52076a.pdf. (Accessed: 23 September 2023).

[13] F. Angione, et al., On the integration and hardening of software test libraries in real-time operating systems, in: LATS, 2023.

[14] P. Bernardi, et al., Development flow for on-line core self-test of automotive microcontrollers, IEEE Trans. Comput. 65 (3) (2016) 744–754, http://dx.doi.org/10.1109/TC.2015.2498546.

[15] Arm, State of the art software test libraries (STL) and ASIL b: Truths, myths, and guidance, 2023, https://www.arm.com/resources/white-paper/software-test-libraries. (Accessed: 13 December 2023).

[16] A. Floridia, et al., A decentralized scheduler for on-line self-test routines in multi-core automotive system-on-chips, in: IEEE ITC, 2019, pp. 1–10, http://dx.doi.org/10.1109/ITC44170.2019.9000129.

[17] A. Floridia, D. Piumatti, E. Sanchez, S. De Luca, A. Sansonetti, Parallel software-based self-test suite for multi-core system-on-chip: Migration from single-core to multi-core automotive microcontrollers, in: 2018 13th International Conference on Design & Technology of Integrated Systems in Nanoscale Era, DTIS, 2018, pp. 1–6, http://dx.doi.org/10.1109/DTIS.2018.8368558.

[18] M.A. Skitsas, et al., DaemonGuard: Enabling O/S-orchestrated fine-grained software-based selective-testing in multi-/many-core microprocessors, IEEE Trans. Comput. (2016) 1453–1466, http://dx.doi.org/10.1109/TC.2015.2449840.

[19] N. Bartzoudis, et al., Dynamic scheduling of test routines for efficient online self-testing of embedded microprocessors, in: 2008 14th IEEE International on-Line Testing Symposium, 2008, pp. 185–187, http://dx.doi.org/10.1109/IOLTS.2008.55.

[20] M.A. Skitsas, et al., Toward efficient check-pointing and rollback under on-demand SBST in chip multi-processors, in: IEEE IOLTS, 2015, http://dx.doi.org/10.1109/IOLTS.2015.7229842.

[21] Y. Li, et al., Operating system scheduling for efficient online self-test in robust systems, in: IEEE/ACM International Conference on Computer-Aided Design, 2009, pp. 201–208.

[22] M. Nourani, J. Chin, Power-time tradeoff in test scheduling for SoCs, in: Proceedings 21st International Conference on Computer Design, 2003, pp. 548–553, http://dx.doi.org/10.1109/ICCD.2003.1240954.

[23] D. Gizopoulos, Online periodic self-test scheduling for real-time processor-based systems dependability enhancement, IEEE Trans. Dependable Secur. Comput. 6 (2) (2009) 152–158, http://dx.doi.org/10.1109/TDSC.2009.12.

[24] F. Angione, et al., Online scheduling of concurrent memory BISTs execution at real-time operating-system level, in: IEEE DFT, IEEE Computer Society, 2022, pp. 1–6, http://dx.doi.org/10.1109/DFT56152.2022.9962338, URL https://doi.ieeecomputersociety.org/10.1109/DFT56152.2022.9962338.

[25] R.I. Davis, A. Burns, A survey of hard real-time scheduling for multiprocessor systems, ACM Comput. Surv. 43 (4) (2011) http://dx.doi.org/10.1145/1978802.1978814.

[26] K. Lakshmanan, et al., Scheduling parallel real-time tasks on multi-core processors, in: 2010 31st IEEE Real-Time Systems Symposium, 2010, pp. 259–268, http://dx.doi.org/10.1109/RTSS.2010.42.

[27] S. Vestal, Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance, in: 28th IEEE International Real-Time Systems Symposium, RTSS 2007, 2007, pp. 239–243, http://dx.doi.org/10.1109/RTSS.2007.47.

[28] R.M. Pathan, Real-time scheduling algorithm for safety-critical systems on faulty multicore environments, Real-Time Syst. 53 (1) (2017) 45–81, http://dx.doi.org/10.1007/s11241-016-9258-z.

[29] FreeRTOS, Real-time operating system for microcontrollers and small microprocessors, 2025, https://www.freertos.org/. (Accessed: 22 January 2025).

[30] J.J. Labrosse, UC/OS-III, The Real-Time Kernel, or a High Performance, Scalable, ROMable, Preemptive, Multitasking Kernel for Microprocessors, Microcontrollers & DSPs, Micrium Press, Weston, FL, USA, 2009.

[31] A.L.F.P. Zephyr, Zephyr project overview, 2024, https://zephyrproject.org/wp-content/uploads/sites/38/2024/01/Zephyr-Overview-20240110.pdf. (Accessed: 16 January 2024).

[32] G. Buttazzo, Research trends in real-time computing for embedded systems, SIGBED Rev. 3 (3) (2006) 1–10, http://dx.doi.org/10.1145/1164050.1164052.

[33] F. Mehnert, et al., Cost and benefit of separate address spaces in real-time operating systems, in: IEEE RTSS, 2002, pp. 124–133, http://dx.doi.org/10.1109/REAL.2002.1181568.

[34] A.S. Tanenbaum, Modern Operating System, second ed., in: GOAL Series, Prentice Hall, 2001, URL https://www.libgen.li/file.php?md5=d4e41de0b388c36f8e2415d5a7ca57a4.

[35] AUTOSAR — The Worldwide Automotive Standard for E/E Systems, Springer Fachmedien Wiesbaden, 2013.

[36] A. Nardi, A. Armato, Functional safety methodologies for automotive applications, in: ICCAD, 2017, pp. 970–975, http://dx.doi.org/10.1109/ICCAD.2017.8203886.

[37] J.M. Waters, L.A. Peterson, P.W. Brazis, Implementation of an electrical safety authority at a global testing, inspection and certification (TIC) organization, in: 2025 IEEE IAS Electrical Safety Workshop, ESW, 2025, pp. 1–6, http://dx.doi.org/10.1109/ESW58401.2025.11159086.

[38] Y. Fu, H. Sun, B. Yin, B. Li, H. Wu, B. Wang, Research on functional safety development and testing methods of PCU system for HEV, in: 2025 International Conference on Applied Electrical Engineering and Technology, AEET, 2025, pp. 173–178, http://dx.doi.org/10.1109/AEET66561.2025.11307117.

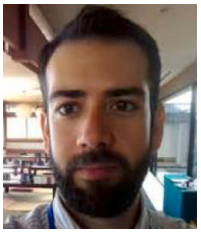
[39] Z. Xu, A.J. Köhler, T.C. Traub, M. Dazer, Enhancing safety of power supply systems in automotive applications: Integrating functional safety (FuSa) and safety of the intended functionality (SOTIF), in: 2024 8th International Conference on System Reliability and Safety, ICSRS, 2024, pp. 16–28, http://dx.doi.org/10.1109/ICSRS63046.2024.10927044.

[40] D. Denomme, et al., A fault tolerant time interval process for functional safety development, in: WCX SAE World Congress Experience, SAE International, 2019, http://dx.doi.org/10.4271/2019-01-0110.

[41] L. Chen, et al., Software-based self-testing methodology for processor cores, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. (2001).
[42] M. Psarakis, et al., Microprocessor software-based self-testing, IEEE Des. Test Comput. 27 (3) (2010) 4–19, http://dx.doi.org/10.1109/MDT.2010.5.
[43] A. Ruospo, et al., A suitability analysis of software based testing strategies for the on-line testing of artificial neural networks applications in embedded devices, in: IOLTS, 2021, http://dx.doi.org/10.1109/IOLTS52814.2021.9486704.
[44] A. Ruospo, et al., Image test libraries for the on-line self-test of functional units in GPUs running CNNs, in: IEEE ETS, 2023, http://dx.doi.org/10.1109/ETS56758.2023.10174176.
[45] P. Bernardi, et al., On the functional test of the register forwarding and pipeline interlocking unit in pipelined processors, in: MTV, 2013, http://dx.doi.org/10.1109/MTV.2013.10.
[46] K. Christou, et al., A novel SBST generation technique for path-delay faults in microprocessors exploiting gate- and RT-level descriptions, in: IEEE VTS, 2008, pp. 389–394, http://dx.doi.org/10.1109/VTS.2008.37.
[47] P. Bernardi, et al., Applicative system level test introduction to increase confidence on screening quality, in: IEEE DDECS, 2020, pp. 1–6, http://dx.doi.org/10.1109/DDECS50862.2020.9095569.
[48] F. Angione, et al., PTLIX: A Portable Test Library Interface, CAD & Reliability Group, 2025, https://github.com/franout/PTLIX.
[49] A. Söderberg, B. Vedder, Composable safety-critical systems based on pre-certified software components, in: IEEE ISSREW, 2012, http://dx.doi.org/10.1109/ISSREW.2012.83.
[50] W. Xi, W. Guo, R. Shi, Research on software architecture design based on functional safety, in: 2025 8th International Conference on Computer Information Science and Application Technology, CISAT, 2025, pp. 925–929, http://dx.doi.org/10.1109/CISAT66811.2025.11181942.
[51] K. Beckers, M. Heisel, T. Frese, D. Hatebur, A structured and model-based hazard analysis and risk assessment method for automotive systems, in: 2013 IEEE 24th International Symposium on Software Reliability Engineering, ISSRE, 2013, pp. 238–247, http://dx.doi.org/10.1109/ISSRE.2013.6698923.
[52] R. Pietrantuono, S. Russo, Introduction to safety critical systems, in: D. Cotroneo (Ed.), Innovative Technologies for Dependable OTS-Based Critical Systems: Challenges and Achievements of the CRITICAL STEP Project, Springer Milan, Milano, 2013, pp. 17–27, http://dx.doi.org/10.1007/978-88-470-2772-5_2.
[53] E. Lisova, R. Broux, J. Denil, A. Bucaioni, S. Mubeen, Communication patterns for evaluating vehicular e/e architectures, in: 2022 International Conference on Electrical, Computer, Communications and Mechatronics Engineering, ICECCME, 2022, pp. 1–8, http://dx.doi.org/10.1109/ICECCME55909.2022.9987747.
[54] H. Askaripoor, M. Hashemi Farzaneh, A. Knoll, E/E architecture synthesis: Challenges and technologies, Electronics 11 (4) (2022) http://dx.doi.org/10.3390/electronics11040518, URL https://www.mdpi.com/2079-9292/11/4/518.
[55] A. Floridia, et al., Parallel software-based self-test suite for multi-core system-on-chip: Migration from single-core to multi-core automotive microcontrollers, in: IEEE DTIS, 2018, pp. 1–6, http://dx.doi.org/10.1109/DTIS.2018.8368558.
[56] C.L. Liu, J.W. Layland, Scheduling algorithms for multiprogramming in a hard-real-time environment, J. ACM 20 (1) (1973) http://dx.doi.org/10.1145/321738.321743.
[57] CAD & Reliability Group, Politecnico di Torino, Stuck-At STLs for pulpino-ri5cy, 2023, https://github.com/cad-polito-it/pulpino_ri5cy_stls.
[58] T. Faller, et al., Special session: Software-based self-test generation for RISC-V – Stuck-at generation, functional cell-aware untestability, and FPGA demonstration –, in: 2024 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems, DFT, 2024, pp. 1–6, http://dx.doi.org/10.1109/DFT63277.2024.10753557.
[59] J. Viega, H. Thompson, The state of embedded-device security (Spoiler alert: It's bad), IEEE Secur. Priv. 10 (5) (2012) http://dx.doi.org/10.1109/MSP.2012.134.
[60] M. Khatun, A. Armato, S. Fischer, Co-analysis of functional safety and cybersecurity with failure and threat modes and effect analysis, in: 2025 9th International Conference on System Reliability and Safety, ICSRS, 2025, pp. 471–476, http://dx.doi.org/10.1109/ICSRS68021.2025.11422158.
[61] A. Serino, L. Cheng, Real-time operating systems for cyber-physical systems: Current status and future research, in: IEEE IThings and GreenCom and CPSCom and SmartData and Cybermatics, 2020, http://dx.doi.org/10.1109/iThings-GreenCom-CPSCom-SmartData-Cybermatics50389.2020.00080.

Francesco Angione is a Computer Engineer with an M.Sc. in Embedded Systems track, obtained from Politecnico di Torino in 2020. He holds a Ph.D. on "System-Level-Test techniques for Automotive SoCs". He is currently a Post-doctoral researcher at Politecnico di Torino in the CAD & Reliability group. His main interests are real-time operating systems, computer architectures, and their dependability.

Paolo Bernardi (MS'02 and PhD'06 in Computer Science) is an Associate Professor of Politecnico di Torino University, where he works in the Electronic CAD and Reliability research group. His current interests include System-on-Chip tests and reliability, especially in the direction of high-quality automotive devices. Prof. Bernardi is the Program Chair of the Automotive Reliability, Test and Safety (ARTS) Workshop held in conjunction with the International Test Conference and General Chair of the IEEE European Test Symposium 2023. Paolo Bernardi is an IEEE senior member.



Riccardo Cantoro received the M.S. and Ph.D. degrees in computer engineering from the Politecnico di Torino, Turin, Italy, in 2013 and 2017, respectively.,He is currently a Researcher with the Department of Computer Engineering, Politecnico di Torino. His research interests include functional testing of SoCs and memories, data analysis, and machine learning applied to test and diagnosis.,Dr. Cantoro was involved in the program committees and organizing committees of several IEEE conferences and workshops and is currently the Program Co-Chair of the Test Technology Educational Program of the Test Technology Technical Council. He is a member of the IEEE Computer Society.