Zephyr: An Ideal Platform for Introducing the Software Development Processes of Safety-Critical Embedded Systems

Prof. Roberto Bagnara[1,2,*,†], Ph.D., Ayoub Bourjilat[3,†], Ph.D., Luca Ciucci[1,†], B.Sc., Roy Jamil[3,†], Ph.D. and Nicola Vetrini[1,†], B.Sc.

[1]*BUGSENG, Italy, https://bugseng.com*
[2]*Department of Mathematical, Physical and Computer Sciences, University of Parma, Italy, https://www.unipr.it/en*
[3]*AC6, France, https://ac6.fr/en/*

Abstract

Zephyr has grown into a reference RTOS for resource-constrained devices, now advertising 800+ supported boards across architectures (Arm, RISC-V, x86, ARC, RX, Tensilica, and more), backed by an active, expanding community (1,100 unique contributors in 2024). In 2024 the project achieved written concept approval toward IEC 61508 certification of the Zephyr kernel, and the *Safety Committee* continues work toward SIL 3/SC 3 under the IEC 61508 *Route* 3_S pathway, with a *Safety-Element-out-of-Context* (SEooC) scope. This combination of transparent governance and a maturing safety program makes Zephyr a compelling, pedagogically effective vehicle for teaching the disciplined software-engineering processes used in safety-critical embedded systems.

This paper argues that Zephyr is not just "safety-capable" but purpose-built for instruction. Based on the lifecycle demanded by IEC 61508 and ISO 26262 (planning, requirements, design controls, verification/validation, and release), we focus on four hands-on pillars: (1) requirements and traceability; (2) software architectural constraints; (3) coding policy enforcement; (4) testing.

We outline how Zephyr's open processes and tooling operationalize each pillar in ways that transfer directly to Zephyr-based products. The approach is demonstrated on a minimal application suitable for both curricula and in-house training, with all artifacts released in an accompanying repository.

Keywords

Zephyr, RTOS, functional safety, IEC 61508, ISO 26262, requirements, traceability, architecture, development processes, verification, static analysis, MISRA, testing, education, training

1. Introduction

Safety-critical embedded systems demand more than correct code: they require disciplined processes that make intent explicit and evidence traceable. In practice, newcomers often learn an RTOS API or a driver stack without also learning how to plan, specify, review, verify, release, and maintain software so that it remains dependable under change. This paper takes the position that Zephyr, thanks to its open governance, breadth of supported targets, and ongoing IEC 61508 safety program, is a uniquely effective vehicle for teaching those processes end-to-end. We build directly on the project's public artifacts and workflows and show how to adapt them in courses and on-the-job training for beginner-to-intermediate audiences.

Concretely, we map the safety lifecycle expected by IEC 61508 and ISO 26262 (planning, requirements, design controls, verification/validation, and release) onto reproducible steps that use Zephyr's standard

ZiSE 2025: 1st International Conference on Zephyr in Science and Education, September 23–24, Jena, Germany
*Corresponding author.
†These authors contributed equally.
roberto.bagnara@bugseng.com (R. Bagnara); ayoub.bourjilat@ac6.fr (A. Bourjilat); luca.ciucci@bugseng.com (L. Ciucci); roy.jamil@ac6.fr (R. Jamil); nicola.vetrini@bugseng.com (N. Vetrini)
https://linkedin.com/in/robertobagnara/ (R. Bagnara); https://linkedin.com/in/ayoub-bourjilat-a55b58165/ (A. Bourjilat); https://linkedin.com/in/luca-ciucci-3b22991a0/ (L. Ciucci); https://linkedin.com/in/royjamil/ (R. Jamil); https://linkedin.com/in/nicola-vetrini-a42471253 (N. Vetrini)
0000-0002-6163-6278 (R. Bagnara); 0000-0002-0734-6130 (A. Bourjilat); 0009-0000-9693-9415 (L. Ciucci); 0009-0007-6861-9925 (R. Jamil); 0009-0007-1199-8215 (N. Vetrini)

toolchain and documentation. The approach is organized around four hands-on pillars:

1. requirements and traceability;
2. architectural constraints;
3. coding policy enforcement;
4. testing.

Each pillar is demonstrated on a minimal application and the accompanying repository comes with ready-to-use material and templates that favor transparency, automation, and auditability.

Why Zephyr for education and training? First, the project produces open, reviewable artifacts (requirements documents, coding-policy guidance, testing infrastructure) that can be inspected and discussed in class without licensing barriers. Second, the diversity of supported boards and SoCs enables low-cost lab setups with realistic peripherals and concurrency scenarios. Third, the safety program, pursuing *Route* 3_S for kernel certification within a *Safety-Element-out-of-Context* scope, exposes students to the vocabulary and expectations of modern functional safety. Finally, the community's continuous-integration and documentation practices model habits we want learners to acquire: small, reviewable changes; automated checks; and traceable decisions.

This is not a certification guide and does not claim or imply certification for the demonstration materials. Instead, it is a pragmatic methodology for learning and teaching. We emphasize repeatability and proportionate rigor: the same mechanics that help a student understand requirement decomposition and traceability also help a team prototype a safety-style workflow for a Zephyr-based product.

1.1. Motivation and Audience

Our primary audience comprises (i) university instructors designing courses or modules on dependable embedded software, and (ii) industry trainers onboarding engineers to safety-style development. We assume basic familiarity with C, build systems, and microcontroller development, but not prior experience with safety standards. The goals are to:

- demystify the safety lifecycle by tying each activity to concrete Zephyr artifacts and machinery;
- foster good engineering habits (such as configuration control, code review, automated checks, and evidence capture) from the outset.

1.2. Contributions

This paper makes the following contributions:

- **A reusable methodology** that maps IEC 61508/ISO 26262 activities to Zephyr workflows across four pillars: requirements/traceability, software architectural constraints, coding policy enforcement, and testing.
- **A minimal, open demonstration application** with requirements, specifications, tests, and build scripts that illustrate each pillar; the artifacts are suitable for direct inclusion in curricula and in-house training.

2. Background

2.1. Functional-Safety Lifecycles

IEC 61508 defines a risk-based framework for the development of Electric/Electronic/Programmable Electronic (E/E/PE) safety-related systems across the entire lifecycle, from concept and hazard/risk analysis through requirements, architecture, implementation, verification/validation (V&V), and release,

with explicit evidence capture at each step [1]. The standard emphasizes control of *systematic* faults (process-driven) alongside *random hardware* faults and introduces *Safety Integrity Levels* (SIL 1–4) as target failure measures for safety functions.

ISO 26262 adapts these principles to road vehicles and frames process activities and work products around the *Automotive Safety Integrity Level* (ASIL) taxonomy, but remains conceptually aligned with IEC 61508 in its lifecycle view and artifact orientation [2].

In parallel, coding standards such as MISRA C [3, 4] operationalize defensiveness at the source-code level (rules, directives, deviations, and compliance reporting) and are widely adopted in all industry sectors, wherever disciplined C/C++ development is required [5].

In educational and training settings, the practical challenge is not only to *explain* the lifecycle, but to provide a reproducible workflow and an open set of artifacts that illustrate: (i) requirement decomposition and traceability; (ii) software architectural constraints; (iii) coding policy enforcement; and (iv) testing and coverage. The remainder of this paper shows how these are realized with Zephyr in a way that students and trainees can reproduce on low-cost hardware and common development hosts.

2.2. An Overview of the Zephyr Safety Program

The *Zephyr Safety Program* is the umbrella effort run by the *Safety Working Group/Committee* (processes, plans, requirements work, qualification path, etc.).[1][2][3] The *Zephyr Safety Overview* is a specific documentation page that summarizes the program and links to its artifacts. In particular, it contains public safety documentation that states the *general scope* of pursuing IEC 61508 qualification for the kernel with an objective of *SIL 3 / Systematic Capability (SC) 3* using the *Route* 3_S pathway for pre-existing sources.[4]

Requirements are authored and published openly using *StrictDoc*[5] in the dedicated `reqmgmt` repository, which exposes navigable system and software requirement hierarchies as HTML and CSV.[6] For example, the semaphore requirements tree illustrates how high-level system intents (`ZEP-SYRS-*`) are refined into software requirements (`ZEP-SRS-*`) with parent/child links that are machine-processed into traceability matrices.[7]

To connect requirements with implementation and verification artifacts, the official documentation prescribes Doxygen-based requirement references (e.g., `@req`, `@satisfy`, `@verify`) and cross-references from tests/samples back to APIs and requirements. It is planned to harvest these requirement references into a traceability matrix;[8] at the moment of writing, requirement UIDs have not been added to the main zephyr source repository, and only a proof of concept has been created.[9]

Testing[10] is orchestrated via the *Twister* runner,[11] which discovers tests and samples, builds them across boards and configurations, integrates with Ztest[12] and `pytest`,[13] and emits reports in various formats (JSON, HTML, XML); Twister can also generate code coverage reports.[14]

For coding policy enforcement and broader static verification, Zephyr offers a documented *Static Code Analysis* (SCA) integration in *CMake* and specific guides for multiple tools so that MISRA-style checks and other analyses can be executed in a standard way.[15]

---

[1] All links provided in this paper were last checked on November 3rd, 2025.
[2] https://docs.zephyrproject.org/latest/safety/index.html
[3] https://github.com/zephyrproject-rtos/zephyr/wiki/Safety-Committee
[4] https://docs.zephyrproject.org/latest/safety/safety_overview.html
[5] https://github.com/strictdoc-project/strictdoc
[6] https://zephyrproject-rtos.github.io/reqmgmt/
[7] https://zephyrproject-rtos.github.io/reqmgmt/reqmgmt/docs/software_requirements/semaphore.html
[8] https://docs.zephyrproject.org/latest/project/documentation.html (section "Reference to Requirements")
[9] https://zephyrproject-rtos.github.io/safety-doc/traceability/index.html
[10] https://docs.zephyrproject.org/latest/develop/test/index.html
[11] https://docs.zephyrproject.org/latest/develop/test/twister.html
[12] https://docs.zephyrproject.org/latest/develop/test/ztest.html
[13] https://docs.zephyrproject.org/latest/develop/test/pytest.html
[14] https://docs.zephyrproject.org/latest/develop/test/coverage.html
[15] https://docs.zephyrproject.org/latest/develop/sca/index.html

Work towards the MISRA compliance of Zephyr with respect to MISRA C:2012 Revision 1 with Amendment 2 [3, 6] uses the *ECLAIR Software Verification Platform* (referred to as *ECLAIR* in the sequel).[16] In 2022, a BUGSENG team funded by the Linux Foundation submitted numerous pull requests for the correct handling of MISRA C violations, which were then discussed and applied upstream into the `zephyr-v2.7-auditable` branch on GitHub. Over the course of 5 months, an initial 500k violations of the selected MISRA C guidelines were brought down to less than 10k.

2.3. RTOS Qualification vs. Product Qualification

The Zephyr kernel is being qualified as a *Safety Element out of Context* (SEooC) under IEC 61508 using *Route 3S* for pre-existing sources. This provides supplier evidence *about the element itself* (scope, assumptions of use, requirements, analyses, tests). In contrast, a software-enabled product that *uses* Zephyr must build its own item/system safety case: perform hazard analysis, allocate safety requirements, configure and integrate Zephyr within declared assumptions, and produce product-specific verification evidence. Supplier qualification *reduces* integrator effort, significantly, but never replaces the product's own obligations. Said that, for users and product makers Zephyr qualification is an extremely important result: they will have a much easier task in obtaining the required certification for their products. The split of responsibilities between the supplier and the integrator is summarized in Table 1.

One crucial activity in the hands of the integrator is assessing which parts of Zephyr the product is using, hoping that such parts are in the Zephyr qualification scope or, if that is not true, which parts need to be qualified by the integrator. Doing this reliably requires qualifiable tools like *ECLAIR Code Scout*,[17] which can quickly and automatically determine which pieces of code are actually used in the configuration being analyzed. Being a tool that is qualifiable for use in safety-related development, soundness is paramount: namely code that is deemed unused or unreachable is indeed so. *Code Scout* correctly handles the full C and C++ languages; for the latter it does the right thing even in presence of templates, overloading and inheritance [7].

Pillar I — Requirements & traceability. From the supplier side, Zephyr publishes requirement hierarchies and upstream traceability that make the kernel's intended behavior explicit. For a product, the flow inverts: start from the item's hazards and safety goals, derive product software requirements, and *allocate* some of those to Zephyr services (threads, timers, device drivers) and some to the application. The integrator must (i) keep the allocation explicit (which requirement is satisfied by which configured Zephyr capability vs. which application component), (ii) preserve end-to-end traceability across versions and configurations, and (iii) perform impact analysis when updating Zephyr or altering *Kconfig*/*devicetree*. Supplier traceability is reusable evidence, but product traceability remains the integrator's responsibility.

Pillar II — Software architectural constraints (SACs). In this paper, SACs are *precise, verifiable restrictions on allowed interactions among software elements* (e.g., who may call whom, which headers may be included, which modules may write specific data, and which ones may perform MMIO). Zephyr's mechanisms (module boundaries, Kconfig, devicetree, priorities/work queues, and optional MPU support) are useful *means* to realize SACs, but the constraints themselves must be defined and evidenced at the product level. Accordingly, the integrator: (a) declares an explicit component model for the application, its Zephyr-facing wrappers, and any local drivers; (b) states and enforces SACs such as layering/no-bypass, criticality segregation (freedom-from-interference), data ownership, HAL-only MMIO, and header/API visibility rules; (c) checks those rules automatically over the full code base (possibly via static analysis in CI), treating violations as build-blocking; and (d) freezes Kconfig/devicetree per variant and shows that supplier assumptions are upheld at the interfaces. Timing and resource budgets (scheduling policies, bounded blocking, measurements on the target) are complementary

[16] https://bugseng.com/eclair/
[17] https://bugseng.com/eclair-code-scout/

Table 1
Four pillars: what Zephyr qualification offers vs. what a Zephyr-based product must still provide.

Pillar	Zephyr (SEooC / supplier side)	Product using Zephyr (integrator side)
Requirements & traceability	Public, versioned requirement sets for kernel behavior; upstream traceability across requirement → design/code → tests; stated assumptions/limits for the qualified scope.	Derive *product* safety requirements; map allocated software requirements to the Zephyr features actually used and their configuration; maintain bi-directional traceability from hazards → product software (safety) requirements → Zephyr requirements/artifacts and to application code/tests; perform change/impact analysis when versions/configs change.
Architectural constraints	Documented kernel architecture and interfaces; explicit assumptions of use; mechanisms that *support* enforcement of product SACs (module boundaries, *Kconfig*, *devicetree*, priorities/work queues, optional MPU) and guidance on recommended patterns/prohibited usage.	Define an explicit component model (application, Zephyr-facing wrappers, local drivers); state *software architectural constraints* as verifiable interaction rules (e.g., layering/no-bypass; criticality segregation/FFI; data ownership; HAL-only MMIO; header/API visibility); *automatically check* these rules over the full code base (static analysis/CI gating); freeze Kconfig/devicetree per variant and demonstrate that supplier assumptions hold; separately substantiate timing/resource budgets on target hardware.
Coding policy enforcement	Supplier policy for kernel coding (e.g., MISRA-based C language subset) and evidence from static analysis and reviews within the qualified scope.	Treat Zephyr as *adopted code* per MISRA Compliance:2020; plan full compliance for *native* application/wrapper code; record deviations and usage constraints. Use supplier MISRA evidence to justify a simplified route for adopted code while still enforcing policy on native code and any local drivers.
Testing (including coverage)	Supplier test strategy and results for the qualified kernel scope; regression suites; configuration guidance.	Build a product-level test portfolio (unit/integration/HIL) tied to product requirements; collect structural coverage on *native* code (statements/branches and, where mandated, MC/DC); provide adequate interface/integration testing for *adopted* kernel code under declared assumptions; verify configuration-specific behavior and timing on the target.

evidence but remain separate claims from SAC enforcement. Supplier architectural documentation accelerates review; it never replaces product-specific SAC definitions and the associated evidence.

Pillar III — Coding policy enforcement (native vs. adopted code). In MISRA terminology, *native code* is written/owned by the product team; *adopted code* is incorporated from elsewhere (e.g., Zephyr, vendor HALs). MISRA Compliance:2020 permits a *simplified compliance route* for adopted code: the product may rely on supplier evidence and need not re-enforce every rule internally, *provided* the usage is justified, constrained, and any residual risks are addressed [5]. This is where Zephyr's pursuit of a MISRA-based policy materially helps: supplier static-analysis reports, rule interpretation and deviations shrink the integrator's due-diligence burden. That said, two obligations remain squarely with the product: (i) *full policy enforcement* on native application and wrapper code (including glue around

Zephyr APIs and any local drivers), with deviations logged and referenced in code; and (ii) *usage-based justification* for adopted code (document which Zephyr subsystems and APIs are used, which rules are relevant at the interfaces, and any compensating measures).

Pillar IV — Testing and coverage. Supplier qualification evidence demonstrates that the kernel behaves correctly within its stated scope; it cannot demonstrate that *your* configured product meets its item-level requirements on *your* hardware. Thus, product teams must define a test portfolio tied to product requirements, with: (1) unit tests for application logic, (2) integration tests for timing budgets and error handling under the chosen configuration, and (3) HIL tests where hardware effects matter. Structural coverage obligations (statements/branches and, if required by context, MC/DC) apply to *native* product code. For adopted code (the Zephyr kernel), standards typically allow interface-focused and configuration-focused evidence instead of re-measuring kernel-wide structural coverage, provided supplier evidence is accepted and assumptions are met.

Summarizing, the qualification of Zephyr provides credible, reusable building blocks: requirements, architecture documents, coding-standard conformance evidence, and kernel test artifacts. Product qualification assembles those blocks into a system-specific safety case that is configuration- and usage-aware, adds evidence for native code and integration behavior, and maintains traceability across releases. Framed by the four pillars, this split clarifies what can be reused as supplier evidence and what must be produced anew for each Zephyr-based product.

3. Case Study: A Minimal Zephyr Application

Here we present the minimal Zephyr application we developed as a case study for this paper.

3.1. Purpose and Scope

`temp_alert` is a small Zephyr application that monitors ambient temperature via a BME280 sensor and: (i) displays the current temperature in *Celsius* degrees as `XX.YY` on a TM1637 4-digit 7-segment display; (ii) issues an alert when the temperature crosses a configurable threshold using both a *temporal-3* fire-alarm buzzer pattern and a four-LED chase animation. The application is intentionally compact to enable reproduction on low-cost boards in a lab session.

3.2. Informal Functional Behavior

The `temp_alert` app periodically samples the BME280 via the Zephyr sensor subsystem over I^2C, converts raw readings to centi-degrees Celsius, updates the display without blocking sampling, and drives alert outputs when the measured temperature exceeds a threshold. Output features (display, buzzer, LEDs) are independently configurable in `prj.conf`,[18] enabling "feature-matrix" builds for teaching configurability and compliance.

3.3. Architecture Overview

The architecture of `temp_alert` follows a dataflow style in which information propagates from the sensor to the actuators without reverse dependencies. The application layer interacts exclusively with Zephyr subsystems (sensor, GPIO, and timing services), avoiding direct hardware manipulation and thus preserving portability across boards. A central control thread periodically retrieves temperature samples and updates the shared state, while dedicated worker threads handle display refresh, buzzer signaling, and LED animation, ensuring that actuation never blocks sampling. Feature toggles in `prj.conf`

[18] The `prj.conf` file is a standard component of Zephyr's build system: it contains a Kconfig fragment that specifies application-specific values for one or more Kconfig options. See https://docs.zephyrproject.org/latest/develop/application/index.html for more information.

determine which subsystems are compiled in, effectively shaping the runtime architecture into different minimal variants. This separation of concerns emphasizes predictability, testability, and didactic clarity.

In the next four sections, for each of the considered functional safety pillars, we:

- briefly describe what the pillar is about;
- summarize what IEC 61508 [1] and ISO 26262 [2] say on the subject;
- describe the corresponding Zephyr project processes;
- describe the processes we drafted for our didactic case study.

4. Pillar I: Requirements and Traceability

A popular quote in the software verification and validation community is the following:

> "Without a specification, a system cannot be right or wrong, it can only be surprising!"
>
> — Paraphrased from [8]

Needless to say, safety-critical software must not be surprising at all. Indeed, as all functional-safety standards, both IEC 61508 and ISO 26262 make *explicit, testable specification* the cornerstone of all downstream activities. IEC 61508 Part 3 requires a dedicated *software safety requirements specification* (SSRS) and ties all downstream activities to it [9, §7.2; §7.3 for the validation plan]. ISO 26262 makes the same pivot at the software level in Part 6, *Specification of software safety requirements* [10, §6], complemented by requirements management in Part 8 [11, §6].

IEC 61508 prescribes suitable quality attributes of requirements: the SSRS shall be clear, precise/unambiguous, complete, feasible and *verifiable*, with content sufficient for design, implementation and assessment [9, §7.2.1–§7.2.2]. ISO 26262 calls for similarly well-formed software safety requirements and alignment with the hardware-software interface [10, §6.1–§6.4], and governs attributes and management under Part 8 [11, §6].

As for bidirectional traceability, traceability must run from hazards and system goals through software requirements to design/code and verification artifacts, and back again for change/impact analysis. ISO 26262 treats this as a supporting process obligation [11, §6], and it is reinforced across Part 6 work products and verifications. As for IEC 61508, in Part 3 it anchors design and V&V to the SSRS and, together with its change-management clauses, effectively demands both forward (requirements $\rightarrow$ design/implementation/tests) and backward (implementation/tests $\rightarrow$ requirements) traceability.

For both standards, verification evidence is judged against the stated requirements: IEC 61508 Part 3 prescribes software module and integration testing within the lifecycle [9, §7.4.7–§7.4.8], while ISO 26262 Part 6 requires unit verification, integration verification, and verification of the software safety requirements themselves [10, §9, §10, §11]. Coverage metrics *complement* but never replace requirement-based testing.

In this pillar we therefore insist on short, atomic, verifiable requirements with stable identifiers; explicit allocation to Zephyr services versus application logic; and end-to-end traceability (requirement $\leftrightarrow$ design/code $\leftrightarrow$ tests).

4.1. Requirements and Traceability in the Zephyr Project

In this section, we describe how requirements and their traceability are handled in the Zephyr project itself, as depicted in Figure 1.

Zephyr's Safety Committee maintains a public requirements catalog managed with StrictDoc[19] and published as browsable HTML. The Safety Committee's "Safety Requirements" page[20] explains scope

[19] https://github.com/strictdoc-project/strictdoc/
[20] https://docs.zephyrproject.org/latest/safety/safety_requirements.html

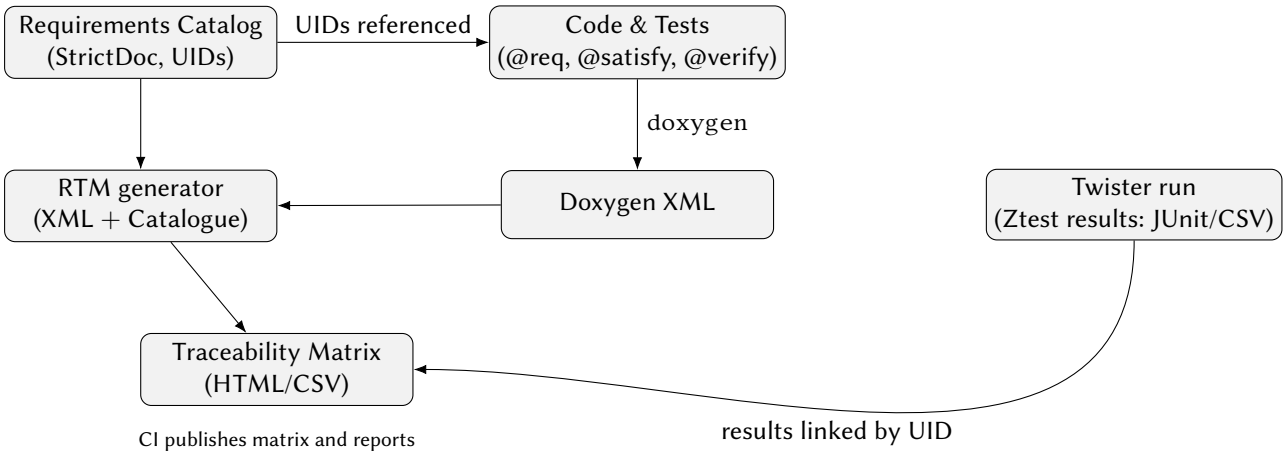

Figure 1: Zephyr traceability data flow. Requirement UIDs are authored in StrictDoc, referenced from code/tests via Doxygen tags, exported to XML, and joined by a generator into an HTML/CSV traceability matrix. Twister provides structured test execution/results that are cross-referenced in the matrix.

(kernel), levels (system vs. software requirements), characteristics of good requirements (including recommended styles), and describes how unique identifiers (UIDs) are assigned and validated in CI; the same page links to the dedicated requirements repository.[21]

StrictDoc is an open-source tool for technical documentation and requirements management. It employs a structured plain-text format with a rigorously defined syntax, ensuring that requirements are structured uniformly, auditable, and easy to qualify under safety standards. Each requirement has a stable unique identifier and explicit relations to others, while the textual format remains human-readable and version-control friendly. StrictDoc can export requirements to formats such as HTML and CSV, which can then be processed further or linked into existing traceability pipelines. StrictDoc also has support for exporting to and importing from the *ReqIF* format,[22] which helps for interoperability with other requirements management tools.

Within the *Safety Overview*, "Requirements and requirements tracing" is called out explicitly as a quality precondition for an auditable code base, alongside coding guidelines and coverage. This frames traceability as a standing expectation for the kernel's safety scope rather than an afterthought.[23]

Zephyr uses Doxygen as the central "hub" for traceability: APIs and tests will be annotated with custom tags that reference requirement UIDs kept in the catalog. In particular, tests would use `@verify{@req{...}}` to state what a test verifies, and APIs/implementation points would use `@satisfy{@req{...}}` to show where a requirement is implemented. The proof-of-concept documentation build produces Doxygen XML; a project script can then parse the XML to generate a traceability matrix (requirements ↔ APIs/code ↔ tests).[24]

Functional tests are written with the Ztest framework and orchestrated by Twister, which discovers, builds, and runs tests across boards and emulators. While traceability itself will be produced from the Doxygen/XML pipeline summarized above, Twister provides the structured test execution and result aggregation that can be cross-referenced from the matrix.

[21] https://zephyrproject-rtos.github.io/reqmgmt/
[22] https://omg.org/spec/ReqIF
[23] https://docs.zephyrproject.org/latest/safety/safety_overview.html
[24] https://docs.zephyrproject.org/latest/project/documentation.html (see "Reference to Requirements" and "Test Documentation").

4.2. Requirements and Traceability in `temp_alert`

The `temp_alert` sample application uses, just like Zephyr, StrictDoc to capture requirements in a file that is part of the repository containing the application code, so that changes are tracked by revision control. The following convention is used to partition requirements: high-level requirements (HLR-xx), low-level requirements (LLR-xx), and software specifications (SRS-xx). High-level requirements are linked to low-level requirements and low-level requirements are linked to software specifications.

The current code base shipped in the accompanying GitHub repository is structured as follows: Zephyr subsystems (*adopted code*) provide threads, logging, sensors API, GPIO, and devicetree bindings; `temp_alert` (*native*) code implements the behaviors and interfaces referenced by the SRSs:

- **Display:** `display_init()`, `display_write()` in `src/display.c`; TM1637 driver wrapper in `src/tm1637.c`; public headers under `inc/display.h`, `inc/tm1637.h`.
- **Buzzer:** `buzzer_fire_pattern()` and `buzzer_thread_fn()` in `src/buzzer.c`; interface in `inc/buzzer.h`.
- **LEDs:** `leds_init()` and `leds_thread_fn()` in `src/leds.c`; interface `inc/leds.h`.
- **Temperature acquisition thread:** `temp_thread_fn()` defined in `src/main.c`.

4.2.1. Requirements Traceability for the Implementation and Tests

Traceability anchors for SRSs are added as Doxygen tags near the implementation of a functionality via the `@implements` tag followed by the requirement UID:

```
/**
 * @implements SRS-02
 */
int display_write(const uint8_t segs[4])
{
    return tm1637_write_segments(&disp, segs);
}
```

Tests are annotated with Doxygen comments using the `@tests` tag attached to the ZTEST macro, since the Ztest framework is used for testing, as explained in Section 7.3:

```
/**
 * @tests SRS-02
 */
ZTEST(display_tests, test_display_write_zeroes)
{
  /* ... */
}
```

To provide the required traceability evidence we leverage the ECLAIR support for MISRA C Directive 3.1 (*All code shall be traceable to documented requirements*), which is part of the coding policy explained in Section 6. StrictDoc requirements are automatically translated into a suitable ECLAIR configuration, so that the tool does all the rest. This is carried out during the static analysis pass, therefore any configuration options given during the analyzed build are automatically reflected into the requirements anchors that are extracted from the source code. The checker takes as input the specifications that are expected to be satisfied, where in the code it is expected to find anchors referring to them (i.e., an SRS for the display should be referred by display implementation code and its test code) and to which entities they are expected to be bound to (i.e., functions, macros and possibly other program entities). Whenever something does not meet the expectations, a *violation report* is emitted, and this check can block a change from being merged. Examples of the most common issues are:

- a function implements some functionality, but it is not linked to any SRS;
- a test tests some functionality, but it is not linked to any SRS;
- a non-existing SRS is referred to;
- an SRS is not referred to anywhere, or it is referred to only from unrelated constructs and files;
- an SRS is referred to by the implementation with an `@implements` tag, but no test refers to it (either because the test is not present, or it does not have an anchor to that SRS).

Conversely, for any SRS that is correctly referenced by the implementation and/or tests, an *information report* is emitted by ECLAIR, thereby allowing bidirectional traceability of all SRSs, which satisfies the objectives of the functional safety standards.

4.2.2. On the Proper Handling of Configurability

The `temp_alert` application is modular and configured via Kconfig, using configuration variables `APP_DISPLAY`, `APP_BUZZER` and `APP_LEDS`. As a result, when the application is built it may or may not contain the implementation of certain requirements from the software specification; the same considerations apply for testing.

Configurability is an essential aspect of many applications and, to a maximum degree, of Zephyr itself. To account for this, the requirements associated to excluded configurations should be also excluded from traceability checks in order to avoid reporting false positives. Therefore, the StrictDoc format for requirements is extended using a *custom requirement grammar* in order to partition them into sets that correspond to the Kconfig toggle that is associated to the implemented feature. For example:

```
[REQUIREMENT]
UID: SRS-01
TITLE: Display Initialization
COMPONENT: APP_DISPLAY
STATEMENT: >>>
...
<<<
RATIONALE: >>>
...
<<<
RELATIONS:
- TYPE: Parent
  VALUE: LLR-09
```

The picture is complete by considering that the ECLAIR service for MISRA C Directive 3.1 is based on static analysis of the code that is actually compiled. This implies that any feature excluded from the build by means of Kconfig (and, for that matter, any other mechanisms) will not produce any spurious traceability violation reports.

4.2.3. Quality Gates and Consistency Checks for Requirements

For `temp_alert`, we use simple, automatable checks that keep the requirements and traceability healthy:

- **UID uniqueness:** no duplicate UIDs
  $\Longrightarrow$ enforced by StrictDoc for the specification; enforced by ECLAIR for the code/test anchors.

- **Orphan/Leaf detection:** only high-level requirements may be orphans (no parents); only software specifications (SRS) may be leaf nodes (no children)
  $\Longrightarrow$ enforced by a custom script ran on every change to the `.sdoc` file.
- **Cycle detection:** requirements relations shall not form cycles
  $\Longrightarrow$ enforced by a custom script ran on every change to the `.sdoc` file.
- **Hierarchy validation:** requirements hierarchy follows proper decomposition (HLR $\rightarrow$ LLR $\rightarrow$ SRS).
  $\Longrightarrow$ enforced by a custom script ran on every change to the `.sdoc` file.
- **Traceability gaps:** requirements without code/tests, code not associated to requirements, test without requirement
  $\Longrightarrow$ enforced by the ECLAIR checker.
- **Dead references:** anchors pointing to non-existent UIDs
  $\Longrightarrow$ enforced by the ECLAIR checker.
- **Configurability support:** for each configuration, only applicable requirements must be checked, others are N/A
  $\Longrightarrow$ enforced by a custom filtering script ran on every analysis before checking the traceability to the code and tests with ECLAIR.

These checks are run in CI on every change or locally via a `check_requirements.py` script.

5. Pillar II: Software Architectural Constraints

In this paper, by *software architectural constraints* (SACs) we mean *precise, verifiable restrictions on interactions among software elements*. SACs constrain: (i) *dynamic dependencies* such as function calls, data passing/return, and read/write access to globals or memory-mapped registers; and (ii) *static dependencies* such as header inclusion and macro expansion. The objective is to enforce partitioning (e.g., layering) and independence (*freedom from interference*) across components of potentially different criticalities, and to make these properties *checkable on the code base* by static analysis.

Typical SACs include: (a) *Layering:* higher layers must not bypass intermediate layers; (b) *Criticality segregation:* higher SIL/ASIL code must not call lower SIL/ASIL code, as the latter might have been developed with a lower level of rigor; (c) *Hardware access isolation:* only HAL modules may perform direct MMIO; application modules must not include low-level hardware headers; (d) *Data ownership:* only designated modules may write specific globals or fields; others may read them at most; (e) *Interface visibility:* APIs exported by component A are not visible to component C unless mediated by component B (header discipline).

5.1. Software Architectural Constraints in IEC 61508 and ISO 26262

Both IEC 61508 and ISO 26262 require an explicit *software architectural design* whose purpose aligns with what we call software architectural constraints (SACs): partitioning, well-defined interfaces, controlled dependencies, resource/timing discipline, and measures to prevent unintended interactions among software elements.

IEC 61508 Part 3 establishes the software safety lifecycle and makes software architecture a first-class activity [9]. Clause 7.4.3 (*Requirements for software architecture design*) requires a documented architectural design consistent with the software safety requirements, covering decomposition into elements, interfaces, control and data flows, and constraints needed to control complexity. The standard also grades *techniques and measures* for architecture selection by target systematic capability via Annex A (*Software architecture design*, Table A.2) and gives additional guidance in Annex B (*Detailed tables*, e.g., Table B.9 on *Modular approach*) and Annex C (properties contributing to systematic capability). Of direct relevance to SACs, Annex F (*Techniques for achieving non-interference between software elements on a*

single computer) provides specific techniques to achieve *non-interference* between software elements (e.g., partitioning, scheduling isolation, memory protection), linking architectural rules to concrete enforcement means.

ISO 26262 Part 6 organizes software development and dedicates Clause 7 to *Software architectural design* [10]. The architecture work products describe both static and dynamic aspects of software elements (hierarchies, interfaces, dependencies, data/control flows), and provide the basis for downstream unit design, verification, and integration. ISO 26262 introduces and emphasizes *freedom from interference* (FFI): the absence of cascading failures between elements that could violate a safety requirement. Demonstrating FFI is expected when elements of different criticalities share resources; Annex D (*Freedom from interference between software elements*, informative) enumerates typical mechanisms (e.g., memory and time partitioning, controlled communication, monitoring) while Clause 7 references the architectural design objectives and work products. Annex E (*Application of safety analyses and analyses of dependent failures at the software architectural level*, informative) further explains safety analyses at the software architectural level, complementing system and hardware analyses.

Summarizing, under both IEC 61508 and ISO 26262, SACs are not optional "style" choices; they are *normative architectural obligations* backed by graded techniques and by evidence-bearing work products (an architectural design specification and results of architectural-level analyses). In IEC 61508, non-interference and modularity are made explicit through Clause 7.4.3 plus Annex A/B/F; in ISO 26262, the same intent appears through Clause 7 and the FFI concept with concrete architectural mechanisms and analyses.

5.2. Zephyr Mechanisms that Support SACs

While SACs are defined at the *interaction* level, Zephyr provides some practical means to realize and support them:

Module boundaries and headers: Subsystems (sensor, GPIO, PWM, display, logging) already expose narrow interfaces. Header discipline (public vs. private headers; include paths) can make forbidden edges syntactically impossible.

Devicetree: Encodes which devices exist and how they are bound. This is used to *instantiate* dependencies (e.g., which GPIO a TM1637 wrapper uses) without leaking low-level headers to the application.

Kconfig and build selection: Ensures only the components for the selected variant are compiled and linked; this reduces the visible API surface and simplifies the SAC model.

Compile-time checks: `BUILD_ASSERT` (and friends) can be used to enforce interface invariants based on constants, Kconfig, and devicetree. SACs of the form "no HAL headers outside the `tm1637` module" can be enforced via header fencing (e.g., `#error` guard or keeping private HAL headers off the include path) so violations fail at build time.

These mechanisms are just supporting tools. The SACs themselves remain the *explicit rules about allowed interactions* that we verify in the code. Their proper enforcement can be achieved through an automated static analysis approach, as the one described in [12], which we use for `temp_alert`, but could be used exactly in the same way for Zephyr itself.

5.3. Proper Enforcement of Software Architectural Constraints for `temp_alert`

Following the approach of Bagnara et al. [12], in our work on `temp_alert` SACs are verified by a static analysis that tracks call edges, data accesses, header inclusions and macro expansions, and checks them against an explicit, formal project organization model. This yields objective evidence that the stated decomposition and interaction rules hold across the entire code base. For `temp_alert`, we adopt the

following (didactic) checklist, where all items are automatically checkable via the *ECLAIR Independence Checker* [7]:[25]

Layering (no bypass): Application → Display, LEDs and Sensor wrappers → HAL; application must not call HAL directly.

Hardware access isolation: Only HAL files may expand MMIO register macros; application and wrapper files may not.

Data ownership: Only the acquisition thread may *write* the shared temperature value; buzzer/LED threads may *read* but not write the alarm state.

Interface visibility: Application-visible headers export only wrapper APIs; HAL headers are not transitively included from app code.

The *ECLAIR Independence Checker* requires the project to be divided into *components*, which can only interact as prescribed by explicit *constraints*. The configuration used for `temp_alert` is as follows.

Components. The `temp_alert` code base is partitioned into components APPLICATION, DISPLAY, BUZZER, LEDS, and UTILS. In addition, Zephyr functionality used by the application is grouped into the following components: ZEP/ATOMIC, ZEP/MMIO (memory-mapped I/O access), ZEP/TIMING, ZEP/DEVICE_TREE, ZEP/THREADING, ZEP/LOGGING, and ZEP/KERNEL, the latter comprising exposed kernel facilities that are not part of ZEP/THREADING or ZEP/TIMING.

Constraints. Given the above partitioning, an illustrative minimal set of rules that enforce the above SACs is as follows:

Layering: APPLICATION may call DISPLAY, BUZZER, and LEDS, and of course may include their headers. APPLICATION may also include and call ZEP/SENSOR to read the temperature.

Hardware access isolation: only BUZZER, DISPLAY and LEDS may use MMIO (e.g., expand GPIO registers macros) and use ZEP/DEVICE_TREE.

Data ownership: only APPLICATION (`temp_thread_fn`) may write the shared temperature and alarm flag; BUZZER and LEDS may only read.

Interface visibility: APPLICATION, BUZZER, DISPLAY and LEDS may use ZEP/THREADING as well as ZEP/TIMING and ZEP/LOGGING APIs; APPLICATION, BUZZER and LEDS may use UTILS and ZEP/ATOMIC; APPLICATION may include ZEP/KERNEL but not call its APIs or expand its macros.

These rules are encoded in the *ECLAIR Independence Checker* configuration, which maps project files and entities to components and enforces constraints on file inclusion, macro expansion, and function calls. For example, any accidental inclusion of hardware headers in the application is reported as a violation. In the CI infrastructure we developed for `temp_alert`, a job runs the architectural check on the full tree: if case of any violations, the pull request is rejected and a report about the detected violations is published. Exactly the same technology could be used for Zephyr itself.

6. Pillar III: Coding Policy Enforcement

By *language subsetting* we mean constraining a general-purpose language like C to a safer, verifiable subset that bans or tightly controls hazardous constructs (e.g., unrestricted pointer arithmetic, recursion, hidden conversions, dynamic allocation without policy). MISRA C is the de-facto standard for such

[25] https://bugseng.com/checking-of-architectural-constraints/

subsetting in C, particularly for critical embedded systems; it provides a taxonomy of *rules* and *directives* with clear rationales and compliance process (required vs. advisory, decidable vs. undecidable, etc.). In parallel, *stylistic* guides (naming, headers, commenting/Doxygen, layout, file structure) are not about safety per se, but they reduce cognitive load and defects, and they make conformance to the safety subset *auditable*. In practice, high-integrity teams adopt: (i) a *coding policy* (language subset + style + banned idioms), (ii) automated checking (static analysis, possibly in CI), and (iii) a controlled *deviation* workflow (documented justification, local mitigations, and linkage to requirements/tests).

6.1. Coding Policy in IEC 61508 and ISO 26262

Both IEC 61508 and ISO 26262 make coding policy a *normative* part of the software lifecycle. In IEC 61508 Part 3, the software development clauses [9, §7.4] require the use of *suitable programming languages* and *coding standards* consistent with the target integrity and with the software safety requirements. The techniques-and-measures annexes [9, Annexes A and B] grade practices such as *language subsetting*, *strong typing*, *defensive programming*, and *static analysis* with increasing recommendation at higher SILs, and emphasize the avoidance or strict control of error-prone features. On the other hand, in ISO 26262 Part 6, the software unit design/implementation clause requires a *language subset* and *coding guidelines* suitable for the ASIL, alongside prohibited/restricted features and tool-based enforcement. ISO 26262 provides method tables where "use of a language subset," "naming conventions," "use of static analysis," and "complexity limits" are recommended/required according to ASIL, and it mandates a documented deviation process [10, §8],

Two practical implications follow. First, *coding policy is not optional*: it is a required work product that must be enforced and evidenced. Second, *policy scope matters*: the policy must cover all code in the safety argument, distinguishing between *native* code (developed in-house) and *adopted* code (third-party or supplier code), because the standards allow different compliance strategies for these categories (see below and Table 1).

6.2. Coding Policy Practices in the Zephyr project

On the supplier side, Zephyr maintains public *coding guidelines* and enforces a project style (formatting, naming, header structure, documentation) through review and CI (formatter/lint, doc build), while the *safety subset* for the kernel's qualified scope is aligned to MISRA C and checked with static-analysis tooling. From the product/integrator perspective, two points are particularly relevant:

(1) Language subsetting and MISRA. Within the safety scope (kernel/selected subsystems), the *Zephyr Safety Program* aligns to a language subset[26] based on MISRA C:2012 Revision 1 with Amendment 2 [3, 6], and runs an automated checker to detect violations and regressions; historical work has shown large-scale reduction of violations and a clear triage path (fix, justify/deviate, or suppress false positives). For *products*, treat Zephyr as *adopted code* per MISRA Compliance:2020 [5] and apply a *full policy* to the *native* application/wrapper code. Supplier evidence (subset definition, checker configuration, deviation records) can be leveraged to justify a simplified route for adopted code in the MISRA compliance report, while ensuring that interfaces to native code honor the subset.

(2) Stylistic guidelines. Zephyr's project style codifies naming/layout, file organization (public vs. private headers), Doxygen usage, and documentation structure. CI checks (format and doc build) keep the repository consistent and reduce reviewer burden. In this case, the choices made by the Zephyr project need not constrain the product teams: treating Zephyr as *adopted code* implies no product team member needs to read or understand Zephyr code, but only its documentation.

[26] https://docs.zephyrproject.org/latest/contribute/coding_guidelines/index.html

6.3. Coding Policy in `temp_alert`

For the `temp_alert` application, we decided to apply the latest edition of MISRA C, namely, MISRA C:2025 [4], by just omitting a handful of advisory guidelines.[27]. For stylistic guidelines, we opted for the stylistic rules of BARR-C:2018 [13, 14]. The resulting coding policy is enforced using the ECLAIR, both in GitHub pipelines and for the developers' PCs, so that they can check locally before submitting a pull request.

6.3.1. Deviation Records

One of the hard-to-die misconceptions about the MISRA coding standards is that they require blind adherence. To the contrary, MISRA is adamant in saying that code quality *always comes first*. Indeed, the deviation process is an integral part of MISRA compliance: whenever compliance would decrease code quality, we *ought to deviate* [5]. More precisely, a MISRA deviation can be approved if it can be justified on the basis of one or more of: (1) code quality; (2) access to hardware; (3) adopted code integration; (4) non-compliant adopted code.

The last two points are quite important for the development of Zephyr-based safety-critical applications. While the *Zephyr Safety Program* will be highly beneficial in this respect, this does not imply: (a) that it will reach 100% compliance with all guidelines; (b) that the project has to target the same version of MISRA C as Zephyr;[28] (c) that the combination of two pieces of code not violating any MISRA guidelines is free from MISRA violations: a function or macro can be compliant with the guidelines per se, but a use of that function or macro might be non-compliant.

So, when the combination of Zephyr code and application code is found to be safe despite violating one or more MISRA guidelines, a deviation will have to be raised using an appropriate SCA tool configuration. Naturally the flexibility and scalability of this approach crucially depends on the mechanisms offered by the chosen SCA tool [15].

Below is an example, taken from the `temp_alert` configuration for ECLAIR, where the required justification has been shortened for presentation purposes.

```
-doc_begin="Integration with macros defined in adopted code."
-config=MC4.R5.10,reports+={adopted,
↪  "any_area(any_loc(any_exp(macro(name(LOG_MODULE_DECLARE||Z_LOG2)))))"}
```

When expanding the Zephyr logging macros `LOG_MODULE_DECLARE` and `Z_LOG2` (an internal macro) in application code, certain reserved identifiers are declared, which constitutes a violation of MISRA C Rule 5.10 (*A reserved identifier or reserved macro name shall not be declared*).

```
#define LOG_MODULE_DECLARE(...)                                    \
        /* ... */                                                  \
        static const uint32_t __log_level __unused =               \
                                        _LOG_LEVEL_RESOLVE(__VA_ARGS__)
```

Here, `__log_level` is a reserved identifier because it starts with two underscores. Since Zephyr strives to be MISRA-compliant, this identifier (and many others, including `__unused` in that same excerpt) will either be replaced or suitably justified. In the latter case, the justification can be leveraged (possibly in modified form) by downstream Zephyr users.

Another deviation is required due to the fact that Zephyr's macro `__ASSERT(test, fmt, ...)` is called without specifying any variadic argument. This possibility has been standardized only in C23 [16], but it is supported as a language extension by the toolchain used by Zephyr and `temp_alert`. In order to comply with MISRA C Rule 1.1 (*The program shall contain no violations of the standard C syntax and constraints, and shall not exceed the implementation's translation limits*) and Directive 1.2

[27] That is, advisory Directives 4.5 and 4.9, and Rules 12.1, 19.2, 20.10 and 15.5, the latter being *disapplied* by default in MISRA C:2025

[28] Zephyr is currently targeting MISRA C:2012 Revision 1 with Amendment 2 [3, 6], but a new Zephyr-based project would have good reasons to start with MISRA C:2025.

(*The use of language extensions should be minimized*), a deviation is introduced, documenting that this behavior is a supported, documented compiler extension, as follows:[29]

```
-file_tag+={ZEP_CC,"^/opt/zephyr-sdk-0.17.2/arm-zephyr-eabi/bin/arm-zephyr-eabi-gc
↪  c$"}
-config=STD.diag,behavior={c99, ZEP_CC, "name(ext_c_missing_varargs_arg)"}
```

6.3.2. Coding Policy Gating

The easiest and most cost-effective way of ensuring that code stays MISRA compliant is to never allow new violations of MISRA guidelines to enter the code base. In `temp_alert` the emergence of new violations for guidelines that were previously "clean" (i.e., free from violations) is used as a gating block. For guidelines that are not clean, CI runs produce differential reports detailing the number of resolved violations, the number of violations being introduced, and the total number of unresolved violations.

7. Pillar IV: Testing

Testing provides objective evidence that the implemented software behaves as specified, catches regressions as the code evolves, and supplies the artifacts needed for safety arguments. In practice we combine *unit* tests (exercising individual modules in isolation), *integration* tests (verifying interactions across interfaces), and *system/acceptance* tests (against end-to-end requirements). For embedded software, unit testing deserves particular emphasis: it enables fast, deterministic checks of small units with stubs/mocks and emulation, supports requirements-based test design, and produces repeatable results suitable for continuous integration.

7.1. Testing in IEC 61508 and ISO 26262

Both IEC 61508 and ISO 26262 treat testing as a primary, evidence-producing activity of the software lifecycle: it must be *planned*, *requirements-based*, and *commensurate with criticality*, spanning unit/module, integration, and system levels, with technique recommendations that strengthen as SIL/ASIL increases.

The IEC 61508 standard defines, in Part 3, software lifecycle activities including module (unit) and integration testing [9, §7]: (i) *software module testing* requirements are in §7.4.7; (ii) *software integration testing* requirements are in §7.4.8; (iii) *software aspects of system safety validation* are addressed in §7.7. Annex A provides "techniques and measures" tables that recommend methods per SIL, notably Table A.5 (software module testing and integration), Table A.6 (hardware/software integration), and Table A.7 (software safety validation). These tables rate methods such as requirements-based functional testing, interface testing, dynamic analysis, and traceability as *recommended* or *highly recommended* as SIL increases.

ISO 26262 Part 6 structures product development at the software level and is quite explicit about testing: (i) Clause 9 is *Software unit verification*, calling for requirements-based tests, verification of interfaces and resource usage, and analysis of requirements/structural coverage for each unit; see, e.g., [10, §9.4]; (ii) Clause 10 covers *Software integration and verification*; (iii) Clause 11 covers *Testing of embedded software*. These clauses also reference tool qualification in Part 8 [11, §11] where test/coverage tools are used to generate evidence.

7.2. Testing in the Zephyr Project and the Safety Program

Within the public *Zephyr Safety Overview*, testing and coverage appear as explicit quality pillars alongside coding guidelines and requirements tracing, and the process description places test/analysis artifact among the inputs to an auditable safety release. This positions testing and coverage as standing expectations for the kernel's safety scope and as reusable practices for downstream products.[30]

[29] `ZEP_CC` identifies the compiler by means of the compiler executable's path.
[30] https://docs.zephyrproject.org/latest/safety/safety_overview.html

Zephyr provides the in-tree *Ztest* framework (assertions, fixtures, suites) for both unit and integration scenarios.[31] A browsable API view is also available.[32] *Ztest* integrates tightly with Zephyr's kernel and device model, allowing test code to exercise the same APIs, concurrency primitives, and drivers used in production.

Zephyr's test runner *Twister* discovers tests, builds them across boards and emulators (e.g., `native_sim`), executes when possible, and emits machine-readable reports (JSON/XUnit) suitable for CI dashboards and gating.[33] The `native_sim` host board enables fast, deterministic execution on a developer/CI machine.[34]

Twister can generate code-coverage reports directly with `--coverage`; the documentation details usage and examples (e.g., `twister --coverage -p native_sim -T <path>`).[35]

At the time of writing this paper, Zephyr ships with an extensive Ztest/Twister-driven test suite and published coverage tooling; the *Safety Program* is actively building auditable, requirements-based unit/integration tests and aiming for very high structural coverage within the defined safety scope, but overall project-wide coverage remains heterogeneous.[36]

Together, `Ztest` and Twister enable systematic unit verification in Zephyr projects in alignment with the IEC 61508 and ISO 26262 standard, which emphasize unit testing as a foundation for safety assurance. The deterministic execution model of the `native_sim` board,[37] along with traceability from requirements to test cases, and reproducibility of coverage evidence are essential pillars of safety-critical compliance, all of which are facilitated by Zephyr's testing infrastructure.

7.3. Testing in `temp_alert`

The `temp_alert` application comes with a suite of unit tests to assess the functionality of each of its components. Tests should be run after every change to the code that is meant to be integrated into the main development branch, to ensure that there are no regressions. Tests are also run after integrating the changes in the CI environment.

7.3.1. Native Simulation Environment

As part of the verification flow, we employed Zephyr's `native_sim` target. This back end compiles the application as a Linux process, mapping Zephyr kernel abstractions onto POSIX primitives (threads, timers and synchronization mechanisms), to provide application developers with an execution environment on the host (x86_64 in this case) in which the same source code runs without modification, while hardware interactions are stubbed or redirected.

This environment was used to accelerate testing by removing the flash-and-reset cycle of physical boards, as well as enabling a faster integration into automated test pipelines that spares the need to test on real hardware.

While `native_sim` cannot verify low-level device behavior such as I^2C timings or GPIO toggling, it serves as a solid foundation to ensure that the program logic satisfies the expectations given by requirements; for more extensive testing a hardware-in-the-loop (HIL) setup is required.

7.3.2. Unit testing with `Ztest`

Tests in `temp_alert` use the *Ztest* framework to verify that the software specification requirements are implemented correctly by the application.

[31] https://docs.zephyrproject.org/latest/develop/test/ztest.html
[32] https://docs.zephyrproject.org/apidoc/latest/group__ztest.html
[33] https://docs.zephyrproject.org/latest/develop/test/twister.html
[34] https://docs.zephyrproject.org/latest/boards/native/native_sim/doc/index.html
[35] https://docs.zephyrproject.org/latest/develop/test/coverage.html
[36] https://docs.zephyrproject.org/api-coverage/latest
[37] https://docs.zephyrproject.org/latest/boards/native/native_sim/doc/index.html

To accomplish this, a test application, built using the same sources and some additional Kconfig options in its `prj.conf`, has been devised. The additional configurations are:

`CONFIG_ZTEST=y` Enables the *Ztest* framework.

`CONFIG_LOG_MODE_MINIMAL=y` Chooses an implementation of the logging API with very low footprint.

`CONFIG_EMUL=y` Enables emulation drivers (e.g., for GPIO and I^2C) to be configured.

`CONFIG_BME280=y` Enables the driver for the BME280 I^2C temperature and pressure sensor.

`CONFIG_GPIO=y` Enables GPIO drivers.

`CONFIG_GPIO_EMUL=y` Enables the emulated GPIO driver; useful for testing with `native_sim`.

`CONFIG_I2C_EMUL=y` Enables the I^2C emulator driver.

Test cases are grouped into suites and registered automatically through macros. Assertions are provided by `zassert_*` macros (e.g., `zassert_equal`, `zassert_true`), which verify expected outcomes and produce uniform reports consumable by Twister and continuous integration systems. Each suite can be run on any supported board, but at the time of writing only the `native_sim` board is used.

`temp_alert` has tests to validate the functionality of its display, buzzer, and LED output modules, as well as application-level threshold logic according to the SRS. Both normal and exceptional control-flow branches are tested, ensuring that initialization, boundary conditions, and error handling are covered. For example, the following test case verifies that the display driver correctly writes valid digits to the TM1637 display interface:

```
/**
 * @tests SRS-02
 */
ZTEST(display_tests, test_display_write_valid_digits)
{
    extern const uint8_t tm1637_segment_map[16];
    uint8_t digits[4] = {
        tm1637_segment_map[1], /* digit '1' */
        tm1637_segment_map[2], /* digit '2' */
        tm1637_segment_map[3], /* digit '3' */
        tm1637_segment_map[4]  /* digit '4' */
    };
    int ret = display_write(digits);
    zassert_equal(ret, 0, "display_write() failed with valid digits, got %d", ret);
}
ZTEST_SUITE(display_tests, NULL, NULL, NULL, NULL, NULL);
```

In `Ztest`, each test case is declared with the `ZTEST` macro, and suites are registered with `ZTEST_SUITE`. This setup ensures that all test cases are automatically registered. When paired with a `testcase.yaml` file, Twister can then discover those tests without requiring additional boilerplate code.

When executed under Twister on `native_sim`, such tests provide deterministic verification of functional requirements (`SRS-02` in the example above) and contribute to coverage metrics collected by `gcov` via Twister, as detailed in Section 7.3.4. Coverage data is then automatically post-processed by Twister with tools like `gcovr` to produce human-readable HTML reports alongside Twister's native machine-readable outputs (`twister.json`, `twister.xml`). These outputs enable visualization and the enforcement of coverage thresholds in continuous integration pipelines.

7.3.3. Structural Code Coverage

Structural code coverage metrics give objective evidence that tests based on requirements exercise the implementation to a degree that functional safety standards regard as "sufficient" depending on the criticality of the unit under test. We use such metrics as a *complement* to requirement-based testing evidences, not as a replacement. Various kinds of code coverage metrics may be defined, depending on the criticality of the software and the quality objectives of the development team. Below is a list of the coverage metrics calculated on `temp_alert`.

Function: Every function is called at least once: insufficient as it does not give confidence about the adequacy of the tests being performed on the function.

Statement: Every executable statement is executed at least once: useful for finding untested blocks; insufficient for decisions with multiple outcomes.

Branch: Every decision (e.g., `if`/`else`, `switch` case) took each possible outcome at least once (true/false, each case): stronger than statement coverage; still misses interactions among conditions inside compound decisions.

Other structural code coverage methodologies exist (such as MC/DC coverage, which gives stronger guarantees than branch coverage). Regarding the relationship between statement and branch coverage, below is an example taken from the `temp_alert` application:

```
err = gpio_pin_configure_dt(&leds[i], GPIO_OUTPUT_INACTIVE);
if (err != 0) {
  LOG_ERR("Failed to configure LED%u (%d)", (int)i, err);
  return err;
}
```

In order to reach statement coverage, it is sufficient to write a test that triggers the error condition, thereby executing all statements of this snippet. Branch coverage needs at least one test where the condition `err != 0` is true and another one where the condition is false.

7.3.4. Collecting Coverage for `temp_alert`

The toolchain used by the Zephyr SDK provides support for *function*, *statement* and *branch* coverage collection by executing the instrumented application and assembling code coverage reports using Zephyr tooling with `Twister`.[38]

The following command is used in `temp_alert` to build and run tests with twister on the `native_sim` platform; code coverage results are collected and exported in HTML and JSON:

```
west twister --coverage --coverage-basedir . -T tests --platform native_sim
```

Note the use of the `--coverage-basedir` flag, which instructs Zephyr to report code coverage relative to the application directory, rather than Zephyr, as code coverage for the zephyr kernel is assessed using Zephyr's own test suites and infrastructure; integrators may refer to these upstream reports to assess the adequacy of testing for the RTOS itself.

When code coverage decreases between subsequent changes, it is mainly due to two factors: (1) new features are added without adding tests or (2) refactoring existing features changes the control flow of the application, making existing tests inadequate. Both issues usually do not cause any new test failures and are therefore not caught by testing on its own, but they are nonetheless scenarios that should be avoided to decrease the risk of introducing errors. To address this problem, a blocking check run in CI workflows, also known as *gating*, is used in the CI infrastructure for `temp_alert` to make the testing workflow fail if the branch coverage percentage is below a certain threshold for any file in application code; the coverage information is extracted by processing the JSON coverage output summary generated by `Twister`.

[38] https://docs.zephyrproject.org/latest/develop/test/coverage.html

8. Availability of Materials

A companion repository is available at https://github.com/BUGSENG/ZiSE25. It contains the Zephyr-based `temp_alert` application, used throughout this paper as a demonstrator. The repository provides the full application source code, configuration files, build scripts, and test suites. A copy of the `temp_alert` application code can be obtained by cloning the repository:

```
git clone https://github.com/BUGSENG/ZiSE25.git
```

In the project root directory, the `README.md` file provides all the information required to reproduce the experiments summarized in this paper. The project can also be forked in order to obtain a starting point for adaptation to completely different projects.

8.1. Local Setup

To reproduce the experiments and explore the project in a local environment, the following setup is recommended:

- Operating system: Ubuntu 22.04 or later;
- Zephyr SDK[39] version 0.17.2 or later;
- (Optional) Visual Studio Code with the Zephyr Workbench extension for an integrated development environment;[40]
- (Optional) ECLAIR for static analysis.[41]

Detailed instructions for installing the Zephyr SDK and toolchain, as well as building and testing the application, are provided in `README.md`. Note that the Zephyr RTOS itself (version 4.2.0 at the time of writing) is automatically downloaded and used as part of the setup process.

Static analysis with ECLAIR can be performed using the scripts in the `ECLAIR` directory or via custom `west` commands, both documented in `README.md`.

8.2. CI Workflows

The repository includes GitHub Actions workflows that automatically run tests and static analyses on pull requests and pushes to the main branch. These are defined in the `.github/workflows` directory. They assume an Ubuntu 22.04 (or later) runner with ECLAIR installed, while the Zephyr SDK and other dependencies are set up as part of the workflow.

9. Conclusion

This paper presented a pragmatic method for teaching the disciplined software-development processes used in safety-critical embedded systems using Zephyr as the instructional substrate. Anchored in four pillars — requirements and traceability (Section 4), software architectural constraints (Section 5), coding policy enforcement (Section 6), and testing (Section 7) — the approach emphasizes transparent artifact, high automation, and objective assessment.

The companion application `temp_alert` demonstrates how a small, realistic code base can support end-to-end traceability, configuration-aware compliance, static analysis with deviations, and testing

[39] https://docs.zephyrproject.org/latest/develop/toolchains/zephyr_sdk.html
[40] https://zephyr-workbench.com/docs/documentation/installation/
[41] See https://bugseng.com/eclair-static-analysis-tool/. Free-of-charge, fully functional trial versions can be requested from https://bugseng.com/eclair-request-trial/. Educational licenses are also available for instructors and students: https://bugseng.com/educational-program/.

with coverage guarantees. The material is suitable for the production of teaching modules to be used for both academia and industry training.

Note that we deliberately avoid certification claims: the goal is pedagogy and process fluency, not qualification. Nevertheless, the artifact and habits cultivated here map well to the kinds of work products expected by functional-safety standards and can seed process improvements in product teams.

Indeed, it is very important to understand that the four pillars we presented are *necessary enablers* for disciplined software development, but they are not the entirety of what IEC 61508 or ISO 26262 require for functional-safety compliance. Full conformity also relies on life-cycle governance and system-level activities outside our scope: safety management and planning, confirmation measures and independence, tool and environment qualification, supplier management and contractual assumptions, and the construction of a safety case that assembles all evidence.[42]

Declaration on Generative AI

During the preparation of this work, the authors used **ChatGPT** in order to perform: **Sentence polishing** and **Rephrasing**. After using this tool/service, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] IEC, IEC 61508-1:2010: Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems, IEC, Geneva, Switzerland, 2010.

[2] ISO, ISO 26262:2018: Road Vehicles — Functional Safety, ISO, Geneva, Switzerland, 2018.

[3] MISRA, MISRA C:2012 — Guidelines for the use of the C language critical systems, HORIBA MIRA Limited, Nuneaton, Warwickshire CV10 0TU, UK, 2019. Third edition, first revision.

[4] MISRA, MISRA C:2025 — Guidelines for the use of the C language in critical systems, The MISRA Consortium Limited, Norwich, Norfolk, NR3 1RU, UK, 2025.

[5] MISRA, MISRA Compliance:2020 — Achieving compliance with MISRA Coding Guidelines, HORIBA MIRA Limited, Nuneaton, Warwickshire CV10 0TU, UK, 2020.

[6] MISRA, MISRA C:2012 Amendment 2 — Updates for ISO/IEC 9899:2011 Core functionality, HORIBA MIRA Limited, Nuneaton, Warwickshire CV10 0TU, UK, 2020.

[7] R. Bagnara, A. Bagnara, P. M. Hill, N. Vetrini, Software Verification Done Right: Introduction to Static Analysis, 1st ed., BUGSENG, Parma, Italy, 2025. URL: https://doi.org/10.979.12210/84894. doi:10.979.12210/84894.

[8] W. D. Young, W. E. Boebert, R. Y. Kain, Proving a computer system secure, The Scientific Honeyweller 6 (1985) 18–27.

[9] IEC, IEC 61508-3:2010: Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems — Part 3: Software Requirements, IEC, Geneva, Switzerland, 2010.

[10] ISO, ISO 26262:2018: Road Vehicles — Functional Safety — Part 6: Product development at the software level, ISO, Geneva, Switzerland, 2018.

[42] The following is an indicative, not exhaustive, list of key activities we left out. At the *concept/system* level: hazard analysis and risk assessment (e.g., HARA), determination of SIL/ASIL and safety goals, the functional safety concept, allocation of safety requirements across HW/SW, hardware-software interface specifications, and system architecture measures for freedom from interference; both failure-based methods (FMEA/FTA/FMEdA) and system-theoretic methods such as STPA [17] for deriving safety constraints and identifying unsafe control actions are applicable here. At the *hardware* level: quantitative hardware metrics (e.g., random hardware failure targets), diagnostic coverage analyses (FMEA/FTA/FMEdA), and HW/SW integration evidence. For *supporting processes*: configuration/change/problem resolution management, documentation control, assessments/audits with required independence, and qualification or confidence arguments for the tools used (requirements management, static analysis, compilers/linkers, test and coverage tools). Finally, *post-development* topics include production, operation, maintenance, and field monitoring, and (where relevant) coordination with adjacent standards such as cybersecurity and SOTIF. Again, our aim is to complement, not replace these activities by providing teaching material focused on the software practices most amenable to hands-on instruction.

[11] ISO, ISO 26262:2018: Road Vehicles — Functional Safety — Part 8: Supporting processes, ISO, Geneva, Switzerland, 2018.

[12] R. Bagnara, A. Bagnara, P. M. Hill, Formal verification of software architectural constraints, in: DESIGN&ELEKTRONIK (Ed.), embedded world Conference 2023 — Proceedings, WEKA FACHMEDIEN, Richard-Reitzner-Allee 2, 85540 Haar, Germany, Nuremberg, Germany, 2023, pp. 271–279.

[13] M. Barr, BARR-C:2018 — Embedded C Coding Standard, Barr Group, 2018. URL: https://barrgroup.com/embedded-systems/books.

[14] R. Bagnara, M. Barr, P. M. Hill, BARR-C:2018 and MISRA C:2012 (with Amendment 2): Synergy between the two most widely used C coding standards, in: DESIGN&ELEKTRONIK (Ed.), embedded world Conference 2021 DIGITAL — Proceedings, WEKA FACHMEDIEN, Richard-Reitzner-Allee 2, 85540 Haar, Germany, Nuremberg, Germany, 2021, pp. 378–391.

[15] R. Bagnara, S. Stabellini, N. Vetrini, A. Bagnara, S. Ballarin, P. M. Hill, F. Serafini, Bringing existing code into MISRA compliance: Challenges and solutions, in: DESIGN&ELEKTRONIK (Ed.), embedded world Conference 2024 — Proceedings, WEKA FACHMEDIEN, Richard-Reitzner-Allee 2, 85540 Haar, Germany, Nuremberg, Germany, 2024, pp. 327–338.

[16] ISO/IEC, ISO/IEC 9899:2024: Programming Languages — C, ISO/IEC, Geneva, Switzerland, 2024.

[17] N. G. Leveson, J. P. Thomas, STPA Handbook, 2018. Available at https://psas.scripts.mit.edu.