SAN19-505 Functional Safety: An Introduction

Victoria Janicki, Engineering Director LITE

Linaro connect
San Diego 2019

Functional Software Safety: In the News

FILE PHOTO: Grounded Boeing 737 MAX aircraft are seen parked in an aerial photo at Boeing Field in Seattle, Washington, U.S. July 1, 2019. REUTERS/Lindsey Wasson/File Photo

5 Lessons Learned (So Far) from Jacob Beningo

Pressure to Ship

Single Point of Failure

Rely on Users

Absence of Defects

Assume Failure

Functional Safety: Definitions

Safety is

"Freedom from unacceptable **risk** of physical injury or of damage to the health of people, either directly, or indirectly as a result of damage to property or to the environment."

Risk is

"Combination of probability of occurrence of harm and the severity of that harm"

Functional safety is the part of the overall safety that depends on a system or equipment **operating correctly in response to its inputs**.

Functional safety is the detection of a potentially dangerous condition resulting in the **activation of a protective or corrective device or mechanism** to prevent hazardous events arising or providing mitigation to reduce the consequence of the hazardous event.

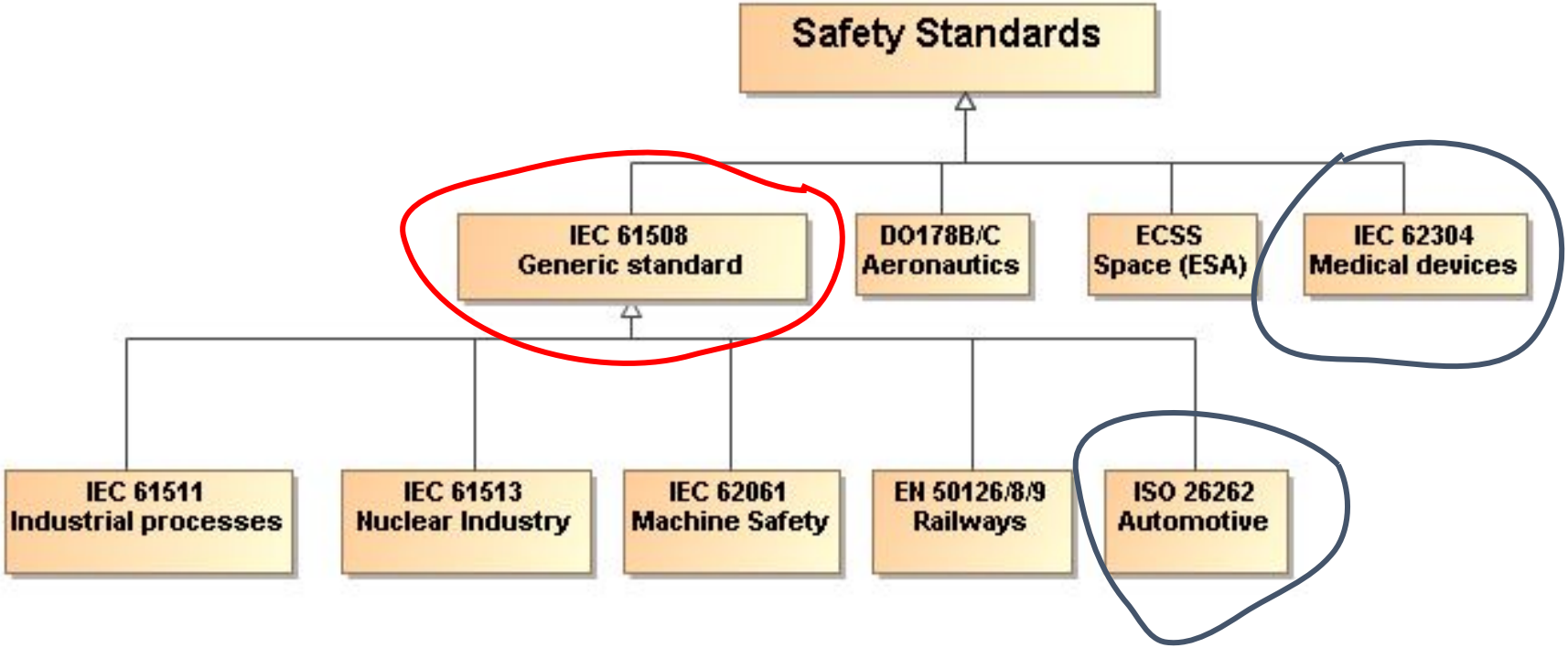
Functional Safety considers Quality an existing pre-condition.

Safety Integrity Level (SIL) is a relative level of risk reduction provided by a safety function.

SIL ratings correlate to frequency and severity of hazards. They determine the performance required to maintain and achieve safety — and the probability of failure.

There are four SILs — SIL 1, SIL 2, SIL 3, and SIL 4. The higher the SIL, the greater the risk of failure. And the greater the risk of failure, the stricter the safety requirements.

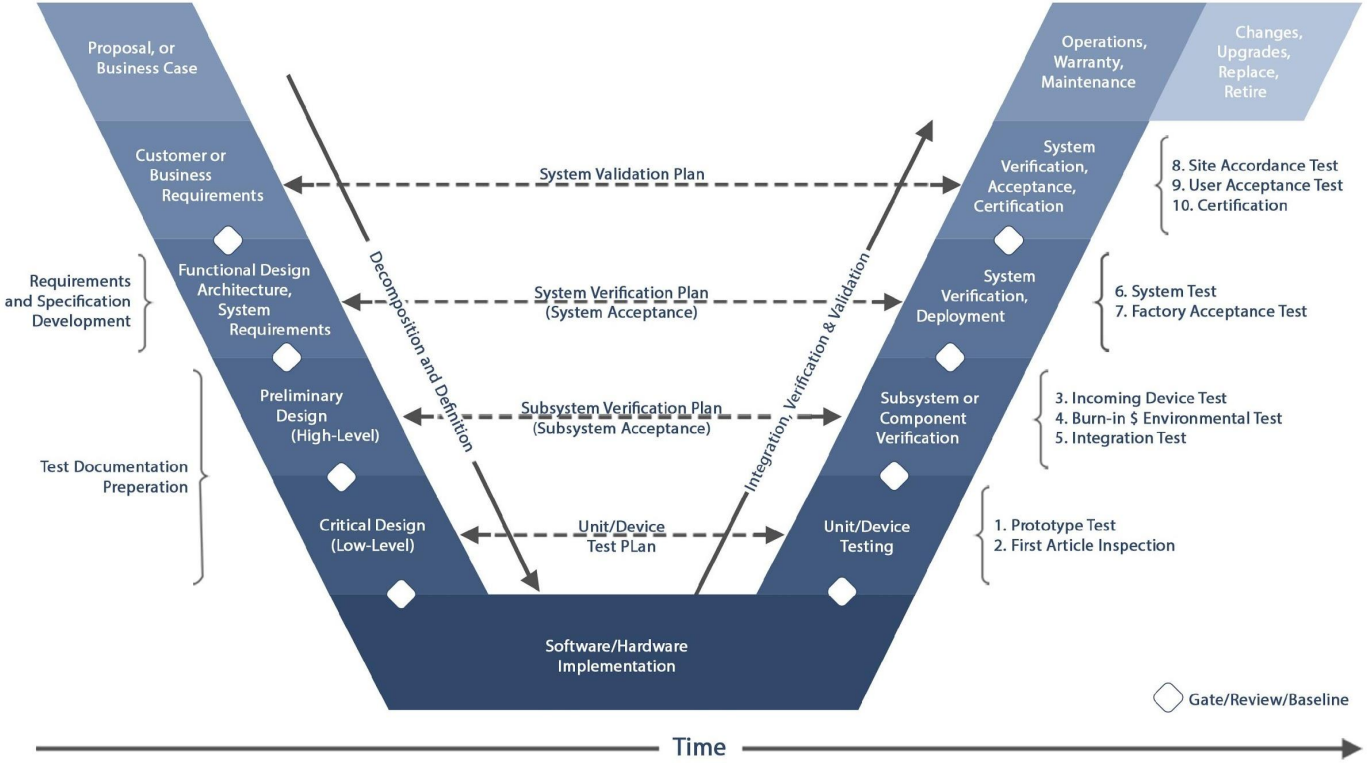
IEC 61508: 2010



How to ensure Functional Safety in Software?

- Identify and analyse risks => input to steps below
- Establish Requirements Traceability throughout the software lifecycle. (Documentation and Testing) to manage that risk
- Architect and design for safety
  - Separate safety critical components from non-critical
- Apply and verify coding standards for safety critical systems (MISRA is an example)
  - Examples:
    - if...else if statement must be terminated with an else
    - Switch statements have a default case
    - Runtime checks for NULL pointers
  - C Language not robust re: safety as other languages such as RUST
  - Enforce with static analysis, preferably automated
  - Human code reviews (for out of box thinking)
- Verify and Validate through testing

V Model Software Development



Zephyr Project Roadmap to Functional Safety

1. Limit the Scope

- Limit to officially supported and maintained code
- Start of the lowest layers and go up the stack

2. Robustness and operational safety

- Review and mitigate risk

3. Enhance and Increase Test Coverage

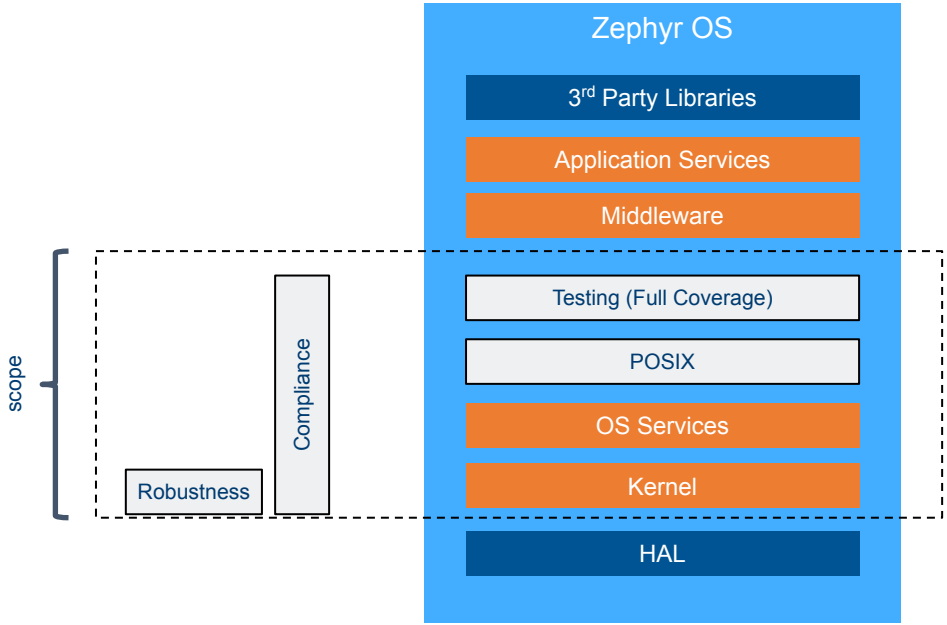
4. Compliance with coding and style guidelines, development process

- MISRA-C Compliance (MISRA-C:2012)

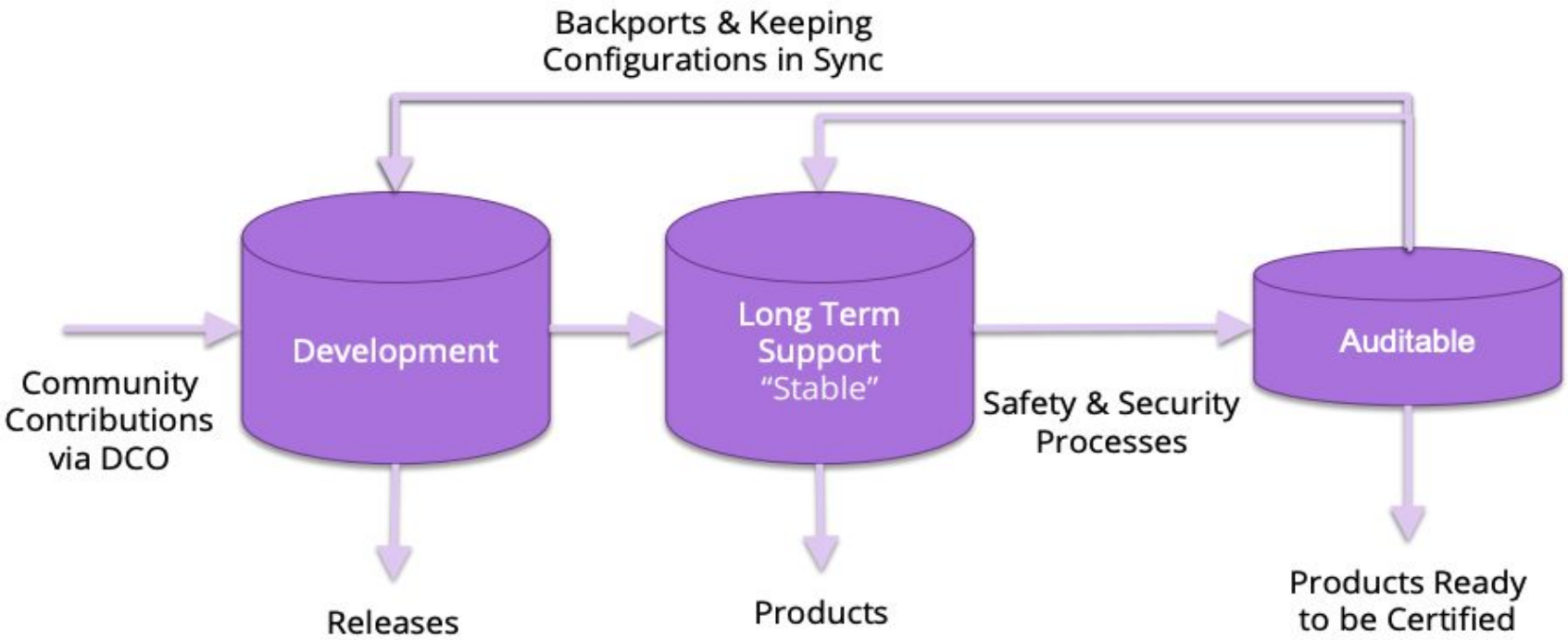
5. Well defined and Stable APIs

6. Portability

- Support POSIX APIs (PSE52, long term PSE54)

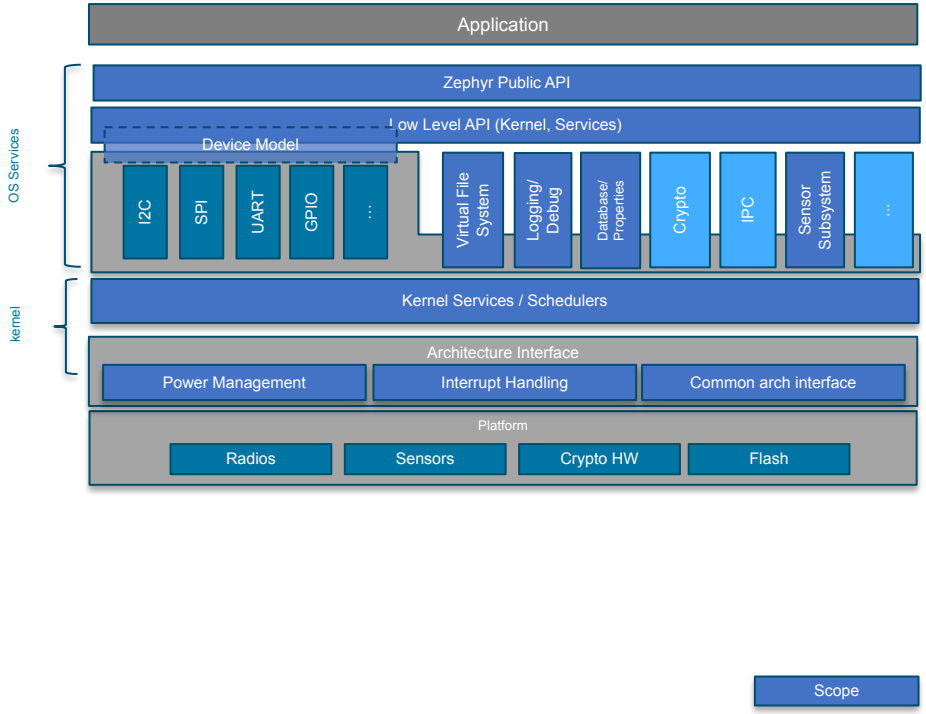


Zephyr Project Strategy: Repos



Zephyr Project: Start of the lowest layers and go up the stack

- Not in scope:
  - Platform drivers or BSPs
  - No platform specific power management implementation, only device and kernel part of PM.
  - No Filesystem or driver implementations, only interface and infrastructure to support those on top of existing APIs



Zephyr Project Coding Guidelines

- Establish and maintain a coding guideline for the project
- Goal is to be MISRA-C compliant
  - MISRA-C with deviations should be the baseline coding guideline
- Add guidelines from other standards
  - CERT-C
  - JPL
  - Barr Group
- Stick to a C standard
  - C99 vs C11 (MISRA-C now only supports C99)
- Enforce guidelines (Tools needed)

Example: JPL Coding Guidelines

1. Avoid complex flow constructs, such as goto and recursion.
2. All loops must have fixed bounds. This prevents runaway code.
3. Avoid heap memory allocation.
4. Restrict functions to a single printed page.
5. Use a minimum of two runtime assertions per function.
6. Restrict the scope of data to the smallest possible.
7. Check the return value of all non-void functions, or cast to void to indicate the return value is useless.
8. Use the preprocessor sparingly.
9. Limit pointer use to a single dereference, and do not use function pointers.
10. Compile with all possible warnings active; all warnings should then be addressed before release of the software.

References and Resources

- https://www.reuters.com/article/us-ethiopia-airplane-boeing-logistics-fo/crisis-hit-boeing-readies-huge-effort-to-return-737-max-to-the-skies-idUSKCN1VY1NV
- https://www.beningo.com/5-lessons-to-learn-from-the-boeing-737-max-fiasco/
- https://www.iec.ch/functionalsafety/explained/
- https://spectrum.ieee.org/aerospace/aviation/how-the-boeing-737-max-disaster-looks-to-a-software-developer
- https://www.perforce.com/blog/qac/what-iec-61508-plus-safety-integrity-level-basics
- https://www.perforce.com/blog/kw/NASA-rules-for-developing-safety-critical-code
- https://ez.analog.com/b/engineerzone-spotlight/posts/functional-safety-for-software
- https://ez.analog.com/b/engineerzone-spotlight/posts/verification-and-validation
- https://static1.squarespace.com/static/5a60ec649f8dce866f011db6/t/5ad016871ae6cf72ec208cb8/1523586697234/The+Challenge+of+Using+C+in+Safety-Critical+Applications.pdf
- https://wiki.sei.cmu.edu/confluence/display/c/Introduction

Thank you

Join Linaro to accelerate deployment of your Arm-based solutions through collaboration

contactus@linaro.org

Linaro connect
San Diego 2019