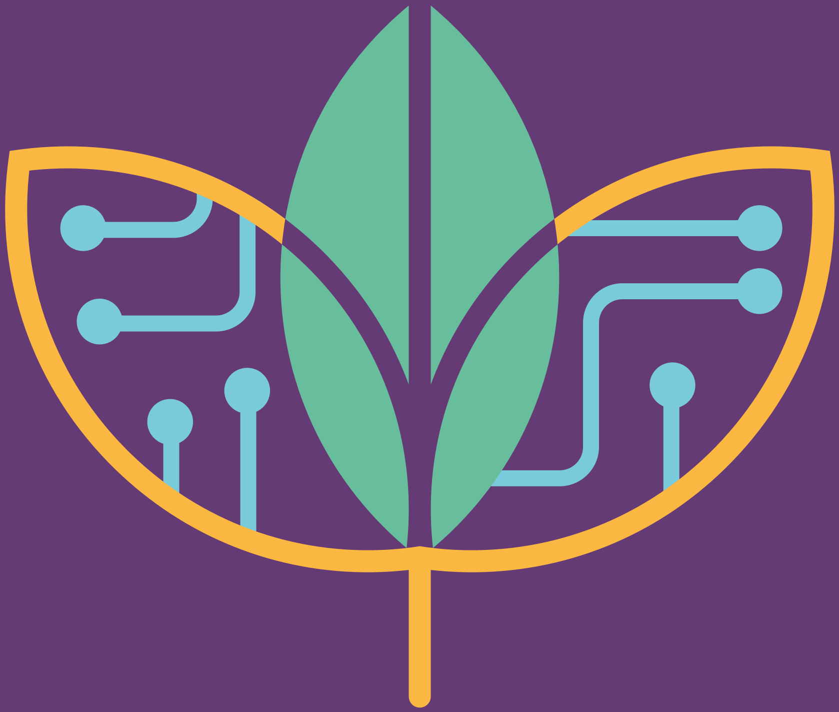
Bluefruit® Software



The essential IEC 61508 checklist for embedded software compliance

Contents

Chapter 1: 3

Introduction 3

What is IEC 61508? 3

Is compliance always non-negotiable? 3

Your functional safety companion guide 4

Chapter 2: 5

What is functional safety? 5

How are functional safety measures determined? 6

How does IEC 61508 grade and define SILs? 7

Understanding advised coding standards 9

Chapter 3: 10

Recognising barriers to functional safety 10

Functional safety is an opportunity, not a burden 11

The seven parts of IEC 61508 13

Chapter 4: 14

Your essential IEC 61508 checklist 14

5.2 Have you recorded all evidence of functional safety? 15

6.2 Have you assigned functional safety managers? 17

7.1 Have you created a safety lifecycle model? 19

7.4 Have you identified and analysed all likely hazards and risks? 22

7.5 7.6 Have you defined what safety functions your device must perform, and how they will be implemented? 24

8.2 Have you independently assessed that functional safety has been achieved 26

Specifying software safety requirements 28

What about clauses 7.1 and 7.2? 29

7.4 Have you really designed and developed the software to be functionally safe? 30

Tackle the software re-use conundrum 32

7.5 Have you tested if your software is compatible with real hardware? 33

7.7 7.9 Have you verified and validated your software against its safety goals? 35

Your IEC 61508 maturity assessment 37

About Bluefruit Software 38

1

Chapter 1: Introduction

What is IEC 61508?

IEC 61508 provides the definitive framework for building functional safety into your electrical, electronic, and programmable electronic (E/E/PE) safety-related systems. It's a dense and highly technical document defined across more than 500 pages, but it also serves a clear purpose: to prevent harm.

The standard applies to any safety-related system where an electrical, mechanical, or functional failure could pose an unacceptable risk to a person's health, from ABS brake controls and platform screen doors (PSDs) to train signalling and emergency shutdown (ESD) systems.

This guide focuses on PE systems and their components that run embedded software, such as a domestic gas boiler that relies on a microcontroller to continuously monitor inputs and execute safety-critical logic. For clarity and simplicity, we'll refer to PE systems as embedded devices throughout.

So, if your teams develop and engineer embedded software that manages or interacts with these systems, you're in the right place.

Is compliance always non-negotiable?

While compliance with IEC 61508 isn't a legal requirement, you'll need it to access and distribute safety-related systems in many global markets.

In fact, its principles show up in countless global safety regulations, either through derived standards or direct references to IEC 61508. For example, working to its provisions can make it much easier to demonstrate compliance with EU frameworks such as IEC 62061 for machinery and ISO 26262 for automotive.

Compliance also indicates that your products can be trusted to reliably perform their intended safety functions, such as in cases of environmental interference or human error. This arguably makes functional safety both a moral and competitive necessity.

Your functional safety companion guide

But what does IEC 61508 mean for your embedded software development teams—and how can you reasonably expect to comply with its many clauses? In this guide, you'll find a curated checklist to help you identify if and where compliance gaps exist in your safety-related embedded software projects and best practices to help you effectively manage each clause's requirements. You'll also learn how to:

- **Demonstrate** the operational benefits of proactive compliance.
- **Recognise** common barriers to building a compliant culture.
- **Integrate** functional safety principles into every project action.

But first, let's define functional safety and its intuitive measurement metrics.

2 Chapter 2: What is functional safety?

To comply with IEC 61508, your safety-related systems—including embedded devices and software within those systems—must be shown to meet the standard's functional safety requirements.

As defined by the IEC, functional safety aims to bring risk down to a tolerable level and to reduce its negative impact. It also ensures that safety-related functions in devices or systems work correctly in response to the commands they receive. However, there is no way to guarantee a complete elimination of risk.

For example, automatic doors on a gondola ski lift may adhere to functional safety standards by using sensors and control software to detect obstructions, stopping, or retracting the doors before an injury can occur.

How are functional safety measures determined?

Functional safety is defined using **safety integrity levels (SILs)**, which reflect how dependable your safety-related system and its functions must be to avoid dangerous failures. SILs are graded on a scale of one to four: the higher the SIL, the lower the accepted probability of failure.

A hazard and risk assessment will help you determine and allocate a SIL to each safety function, which is often managed by embedded devices and software. You may find that simpler systems, like HVAC controllers, only require a single safety function to comply with IEC 61508, so they only need a single SIL assignment. More complex embedded devices often manage multiple safety functions, each requiring its own SIL.

For example, an automotive airbag control unit performs multiple safety functions, so it has different SIL requirements. In the case of a crash, the unit must deploy the airbag immediately, so this would be a SIL 3-allocated function. The software function used to detect and report airbag faults to the driver before they diminish its reliability would be SIL 2.

SILs significantly impact a project's scope, performance targets, and hardware and software requirements—including design and testing practices. Identifying each safety function's SIL early in the project lifecycle can also help align software developers with its requirements to avoid late-stage disruption. But more on that later.

Why do we define software SILs by consequence, not probability?

IEC 61508 measures SILs using probability metrics like probability of dangerous failure on demand (PFD) and probability of dangerous failure per hour (PFH). These metrics work well for hardware where failures are random and statistical, but not software.

Software failures are systematic and caused by design or implementation errors, so statistical failure rates don't meaningfully apply. Instead, software SILs are assigned based on consequence at the system level, with compliance shown through rigorous development and verification practices in IEC 61508-3, tailored for software teams.

How does IEC 61508 grade and define SILs?

Here's how SILs are graded as defined in IEC 61508-1, including our recommendations for compliant development and verification and validation (V&V):

- **SIL 1: Basic risk reduction**

Typical outcome of failure: Inconvenience, degraded user experiences, or mild operational disruption.

Software V&V requirements: Basic static analysis, such as linting and simple unit testing, and requirements-to-test traceability is recommended.

Development practices: Code can be written in a general-purpose language using only advisory coding standards, supported by informal peer reviews.

Example applications: HVAC controllers in commercial buildings, access gate interlock systems, and water treatment plant monitors.

- **SIL 2: Moderate risk reduction**

Typical outcome of failure: Moderate operational hazards that could lead to injury or costly short-term downtime.

Software V&V requirements: Structured unit testing—including functional testing at the integration and system level—systematic static analysis, periodic code reviews, and end-to-end traceability of all test outcomes.

Development practices: Clear and consistent documentation, with version control and configuration management processes in place.

Example applications: Industrial access control systems, automated train doors, and non-life-critical medical devices such as syringe pumps.

- **SIL 3: High risk reduction**

Typical outcome of failure: Significant risk of serious injury or environmental damage.

Software V&V requirements: Modified Condition/Decision Coverage (MC/DC), independent verification, qualified toolchains, and rigorous traceability across the full lifecycle.

Development practices: Fault containment is design-critical and must be supported by a formalised testing strategy and comprehensive reviews.

Example applications: Gas detection systems, safety-critical surgical robots, automotive emergency steering systems, and flight control software.

- **SIL 4: Maximum Risk Reduction**

Typical outcome of failure: Catastrophic consequences, including potential loss of life or large-scale system collapse.

Software V&V requirements: Software must be checked thoroughly by an independent validation team, and include a safety redundancy system in case of failure.

Development practices: Safety-critical code must be strictly isolated and coded with the highest level of care, and validated using mathematical methods.

Example applications: Nuclear reactor shutdown systems, railway signalling interlocks, and launch vehicle abort controllers.

For now, sit tight; you'll find everything needed to execute the assessment later in the guide.

Understanding advised coding standards

To achieve functional safety certification, it's unlikely that you'll need to adopt a new coding language. However, programming languages used for developing all safety-related software must adhere to a suitable coding standard.
Here's what that may look like across each of the four SIL grades:

SIL 1
The coding standard can be lightweight or advisory, and focused on clarity and maintainability rather than strict enforcement. This includes informal peer reviews, consistent naming conventions, and basic traceability of each requirement test.

SIL 2
Coding standards should be systematic and documented, with clearly defined rules for repeat reference. Using a restricted C/C++ subset with static analysis is recommended, but avoid recursion, dynamic memory allocation, and unsafe language features.

SIL 3
At this level, coding standards must be strictly enforced, supported by mandatory static analysis, formal reviews, documented deviation handling, and rule checks. Code can be written in MISRA C, MISRA C++, and AUTOSAR C++ 14.

SIL 4
Coding standards must be enforced with the highest strictness. This means using a restricted subset like MISRA C, MISRA C++, and AUTOSAR C++ 14. Coding rules must be fully documented, including any exceptions and justifications. Always avoid unsafe programming features unless justified with a watertight rationale.

3 Chapter 3: Recognising barriers to functional safety

Before diving into the precautionary world of IEC 61508, it's crucial that you can identify the operational barriers and disruptive influences that may discourage your teams from integrating functional safety principles into every project decision.

To begin with, question whether any of the following challenges apply to your safety-related software projects, either past or present:

- **Balancing effectiveness and risk mitigation**

Difficulties arise in creating a development environment where innovation thrives while operational risks and functional safety failures are effectively mitigated.

- **Managing the compliant use of developer tools**

Limited control over the qualification of developer tools makes it difficult to prove that compilers, static analysis tools, or model-based design environments are fit for purpose.

- **Ensuring every team decision leaves a trace**

Disconnected teams use conflicting processes, preventing effective collaboration that's traceable, documented, and auditable.

- **Avoiding costly and disruptive late-stage changes**

Inconsistent regulatory awareness and stakeholder misalignment cause costly late-stage issues, like extensive patching, product redesigns, and delivery delays.

- **Clarifying responsibilities and task ownership**

Ambiguous requirements and insufficient knowledge prevent effective role delegation, undermining developers' ability to deliver functional safety.

If any of these challenges resonate with your projects, you're ready to move on to the IEC 61508 checklist. There, you'll find clear, actionable advice to help you more readily assess safety-related software projects against the standard and meaningful changes to streamline the development lifecycle.

Functional safety is an opportunity, not a burden

At first glance, achieving functional safety can feel like just another box-ticking exercise designed to distract and slow teams down. However, when approached strategically, its operational benefits are significant and far-reaching.

To get it right, early buy-in from developers and engineers is essential. Assigning an IEC 61508 awareness lead to advocate for its business value can help software teams more easily understand and embrace the standard.

While functional safety can be managed through linear Waterfall methods, IEC 61508 provisions often align more smoothly with Agile working methods. For example, applying Agile principles can help you unlock the following headline benefits:

Fewer costly late-stage changes

Building functional safety requirements into every software iteration—such as each sprint—means risks are surfaced and addressed early, not buried until validation.

IEC 61508 requires ongoing hazard assessment, traceability, and regular reviews—all of which align with Agile's fast feedback loops. This prevents disruptive rework and gives teams confidence that what they deliver at each increment is both safe and compliant.

Clearer scope and responsibilities

Breaking safety goals into smaller, well-defined tasks helps managers allocate project roles more effectively. Agile practices—like sprint planning and daily standups—can reinforce this shared understanding, so everyone knows what they must do to stay on budget and schedule, and most importantly, achieve compliance.

This clarifies targets, makes activities more manageable, and enables smoother cross-functional collaboration, strengthening team relationships and minimising friction.

Smoother certification and market access

Concise safety requirements help teams know exactly what iterative documentation to capture for their sprint commitments. This makes it easier to steadily build the evidence that auditors need while demonstrating consistent progress.

This approach prevents a last-minute scramble at the end of the project, reduces certification bottlenecks, and reassures customers and regulators that safety is part of the process—not an afterthought.

Accountable change management

Agile working encourages teams to continuously review, update, and prioritise tasks rather than locking them down at the start of a project. And it happens to fit neatly with IEC 61508's demand for structured change control.

In practice, every modification to safety-critical code is logged, justified, and tested before integration. Coupled with impact analysis and re-verification, this keeps safety-critical updates visible and accountable while allowing for timely delivery.

Improved cross-disciplinary coordination

IEC 61508 enforces alignment between software, hardware, and system-level safety activities. Agile's collaborative principles support this by encouraging open, regular communication across teams and disciplines.

Regular alignment sessions can help teams catch integration problems early, reducing friction and avoiding costly surprises when software-hardware integrations can have a significant influence on safety outcomes.

Sub Chapter - IEC 61508 NAVIGATION
The seven parts of IEC 61508

The IEC 61508 standard has seven parts, each containing multiple clauses and sub-clauses. This guide's functional safety checklist focuses primarily on **parts 1 and 3**, which provide critical requirements for compliant software development projects.

Part 1 General requirements

Establishes the standard's overall framework, including the principles and general overarching requirements for achieving functional safety (such as the safety lifecycle).

Part 2 Requirements for E/E/PE safety-related systems

Details the requirements for designing and manufacturing E/E/PE safety-related systems, except software.

Part 3 Software requirements systems

Defines software requirements across the entire development lifecycle to ensure software is robust, traceable, and compliant with its SIL target.

Part 4 Definitions and abbreviations

Provides a comprehensive glossary of terms and abbreviations used throughout the standard to promote clear communication across teams and stakeholders.

Part 5 Examples of methods for the determination of safety integrity levels

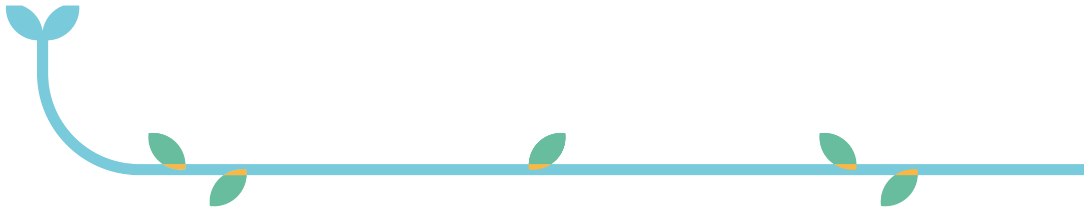
Offers guidance and illustrative examples to help determine appropriate SILs for safety functions through risk assessment and hazard analysis.

Part 6 Guidelines on the application of Parts 2 and 3

Includes practical guidance for implementing the technical requirements outlined in parts 2 (hardware) and 3 (software).

Part 7 Overview of techniques and measures

Summarises the hardware and software techniques that can be used to support compliance and achieve functional safety (by SIL level).



4 Chapter 4: Your essential IEC 61508 checklist

Remember to be honest when assessing your systems and team activities against these clauses. While it can be tempting to fast-track promising projects, proactive adherence to functional safety will help you avoid any nasty—and seriously costly—surprises when it comes time for certification.

IEC 61508 provides goal-based requirements to help you achieve a compliant level of functional safety. You can choose how to implement these, but you must ensure everyone on the team understands the evidence they need to generate, and that you have a clear plan to measure and demonstrate how your safety-related system meets the standard's targets.

An important note about this checklist

This checklist follows the order of clauses as they appear in the official IEC 61508 documentation. Please be aware that clauses not included here may still be critical to achieving full compliance.

This guide is intended to highlight the risks and opportunities associated with the foundational clauses most relevant to safety-related software projects, and not as a replacement for reading the full standard.

Documentation essentials

Part 1: Clause 5.2 Have you recorded all evidence of functional safety?

What is this?

A fundamental clause that earmarks the information you must document across the project lifecycle to prove you've met all functional safety requirements.

Documentation must be consistent and traceable, forming an audit-ready trail of every plan, test, result, and decision made to ensure functional safety. Your storage structure must also support version control, reviews, approvals, and on-demand user search.

This is also necessary to complete the functional safety assessment and perform the functional safety management and verification tasks later in the lifecycle.

The risk of non-compliance

Safety-related activities can become difficult to trace, review, or justify without proper documentation. Also, providing records with inconsistent or missing information raises serious concerns about whether critical safety requirements were met.

However inconsequential a decision or test result may feel at the time, auditors will notice its absence. Failing to capture this information also makes it difficult for teams to check historical results, identify root causes, and deliver better—and safer—software.

From a business standpoint, non-compliance can:

- Undermine your project's credibility and integrity.
- Delay certification and limit market access.
- Prompt costly rework, revisions, and penalties.

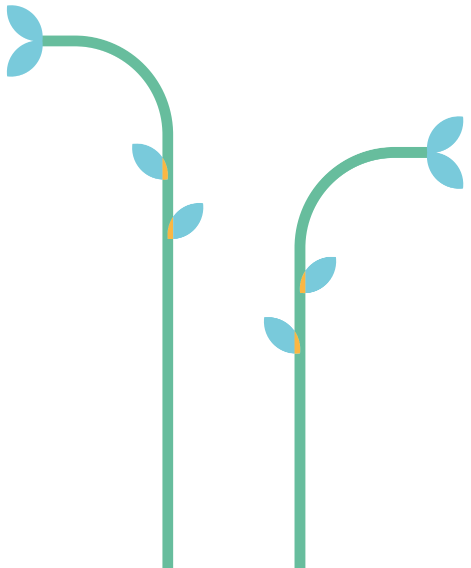
Part 1: Clause 5.2 Have you recorded all evidence of functional safety?

How to get it right...

The documentation required will vary based on the size and complexity of your safety-related systems and embedded devices. While IEC 61508 doesn't mandate a specific format, the internal or external assessor will expect documentation to be clearly organised and accessible.

Living documentation practices such as Behaviour-Driven Development (BDD) or Specification by Example let you manage a single source of truth that's easy to access and reference in Agile feedback loops. For example, BDD practices like Gherkin use basic, human-readable language to define how the system should behave.

You may already use a distributed version control system (DVCS) such as Git for source code, but these are ideal for documentation too. DVCS can turn documentation into a living and auditable resource as each change is time-stamped, attributable, and stored in an immutable log.



PRO TIP

You'll find practical workflow examples to help you meet these requirements in Annex A of IEC 61508-1. These are flexible enough to allow Agile teams to structure and deliver documentation iteratively, so you can capture just enough information during a sprint to meet traceability and assessment needs without disrupting effectiveness.

Part 1: Clause 6.2 Have you assigned functional safety managers?

What is this?

Functional safety management ensures that all responsibilities in the safety lifecycle are clearly defined, effectively managed, and performed by a qualified owner.

Transparent communication is critical, as all persons, departments, and organisations assigned to an activity must understand precisely what is expected of them—including how and when to perform their responsibilities.

This maintains accountability, improves project efficiency, and maximises the use of available skills and individual strengths.

The risk of non-compliance

Safety-related tasks may be missed, duplicated, or performed incorrectly when roles and responsibilities are unclear or assigned to unqualified owners.

For example, if no one knows who's responsible for verifying critical test results, major software issues may slip through unnoticed until the final project stretch, or even into the final build. And when even minor coding errors can have lethal consequences, you can't afford to let internal conflict and choice paralysis thrive—accountability matters.

Part 1: Clause 6.2 Have you assigned functional safety managers?

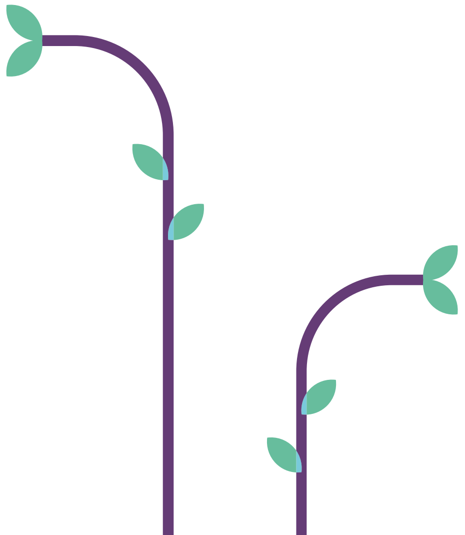
How to get it right…

To support functional safety management, designate individuals with the knowledge and authority to coordinate team responsibilities and sequence tasks in compliance with IEC 61508 requirements. Think of them as functional safety guardians, as they will:

- Define and communicate responsibilities to everyone involved in safety-critical projects, including software developers and engineers.
- Oversee individual competencies through ongoing training and periodic reviews to identify and close skill gaps and optimise role placements.
- Implement and monitor safety-related procedures such as configuration control, incident handling, audits, and software changes.

PRO TIP

Build a shared Functional Safety Management (FSM) plan to define all relevant processes, responsibilities, competence, tools, and verification activities needed to meet the required SIL.



How to balance empowerment and control

You may have experience using Responsibility Assignment Matrix (RACI) charts to assign project responsibilities. This hierarchical approach suits traditional Waterfall projects, but Paul M. **Konnersman's Decision Process Specification model** is a more dynamic and Agile-friendly alternative.

In an IEC 61508 context, the model strikes a balance between empowerment and control by adapting role responsibilities to the context of each decision. This allows teams to assign the right contributors at the right time, while ensuring that all safety decisions—such as SIL allocation and tool qualification—are justified, documented, and auditable.

Overall safety lifecycle requirements

Part 1: Clause 7.1 Have you created a safety lifecycle model?

What is this?

A practical safety lifecycle model gives your teams a comprehensive, structured project plan that defines:

- What activities must happen when.
- Who is responsible for each activity.
- How safety decisions are documented.

This ensures that each part of the overall safety lifecycle is addressed and sequenced systematically to achieve functional safety. In IEC 61508, the safety lifecycle consists of 16 steps divided into three major activity sets:

- **Analysis** focuses on identifying risks and defining safety requirements.
- **Realisation** encompasses planning, development, and validation*.
- **Operation** covers system use, maintenance, and end-of-life processes.

Aligning teams around your roadmap keeps certification efforts on track, as everyone understands what must happen, when, and why. It can also help make handovers between activities more seamless and provide a predictable path to compliance.

*You may find that some planning steps can be completed in parallel with development.

The risk of non-compliance

Failing to follow this clause consistently can lead to missed regulatory steps, unclear responsibilities, and non-compliant documentation.

Skipping or merging critical activities without justification—such as by treating integration and safety validation as the same task—can compromise your ability to show functional safety. And when outputs such as validation evidence are poorly defined or missing, it often creates needless rework for your teams, further delaying certification.

Without an agreed-upon safety lifecycle plan, developers and engineers may resort to unauthorised workarounds, introducing risk where you can least afford it.

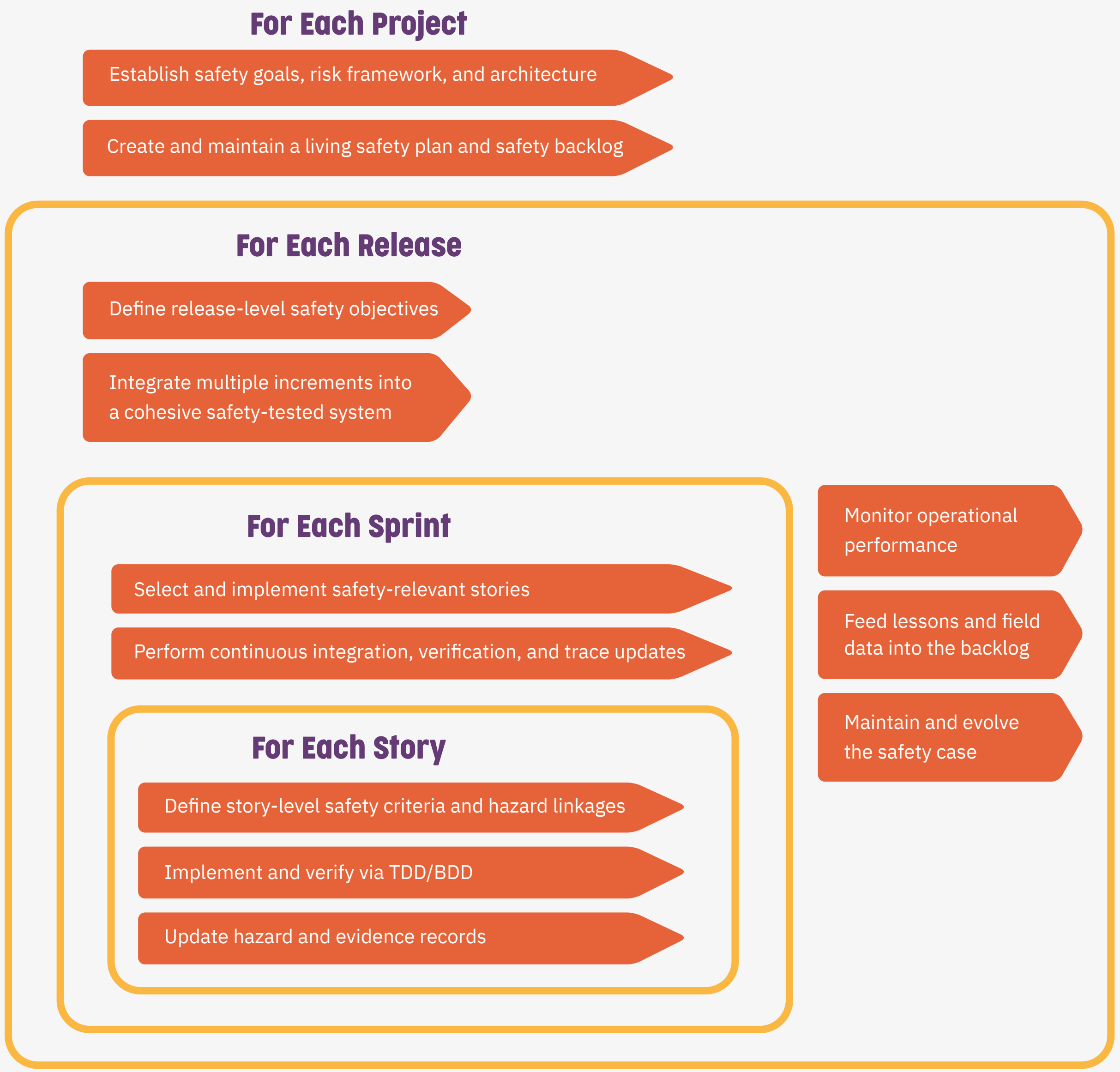
Part 1: Clause 7.1

Have you created a safety lifecycle model?

How to get it right…

Each set of lifecycle activities must be clearly defined and planned for, providing a shared, end-to-end view of project success. Unless you can justify a valid reason, your organisation should adopt the standard lifecycle model shown below.

In an Agile context, activities can be expressed as user stories delivered in sprints and combined into releases. These should have a defined scope, clear inputs, and measurable outputs to support consistency, traceability, and effective handovers.


For Each Project
Establish safety goals, risk framework, and architecture
Create and maintain a living safety plan and safety backlog
For Each Release
Define release-level safety objectives
Integrate multiple increments into a cohesive safety-tested system
For Each Sprint
Select and implement safety-relevant stories
Perform continuous integration, verification, and trace updates
For Each Story
Define story-level safety criteria and hazard linkages
Implement and verify via TDD/BDD
Update hazard and evidence records
Monitor operational performance
Feed lessons and field data into the backlog
Maintain and evolve the safety case

PRO TIP

While the overall safety lifecycle is shown in IEC 61508-1 as a traditional phased journey, nothing prevents you from treating each stage as an Agile-aligned story. You can even layer them as recommended in the Technical Information Report (TIR) 45 to support iterative, cross-functional delivery.*

* TIR45 provides official guidance for applying Agile practices in medical device software development without breaching compliance with safety standards like IEC 61508 and IEC 62304.

Part 1: Clause 7.4 Have you identified and analysed all likely hazards and risks?

What is this?

This clause requires you to determine the hazards, hazardous events and situations that relate to your equipment under control (EUC). For example, a faulty valve in a chemical reactor may result in a rupture, releasing toxic chemicals. A safety control system—your embedded software—must be able to detect and prevent this or prompt intervention.

You must consider hazards for all reasonably foreseeable circumstances, including faults, misuse, and security threats. You'll also need to determine how each hazardous event may unfold, its severity, what contributed to it, and how likely it is to occur.

The risk of non-compliance

Skipping or rushing this analysis can create blind spots in your product's safety profile. If you miss potential hazards, especially in edge cases like misuse and abnormal operation, you may underestimate the need for critical safety functions.

This will compromise your ability to assign and justify risk reduction measures and may have profound consequences later in the project. In a worst-case scenario, your design, feature, or testing decisions are based on missing data or flawed assumptions. And if an incident does occur, the legal, financial, and reputational fallout will likely be dire.

Part 1: Clause 7.4 Have you identified and analysed all likely hazards and risks?

How to get it right...

Hazard and risk analysis helps focus your team where it matters, so all the proper safety functions make it into the final build. In an Agile setup, this works best when treated as an ongoing, iterative activity.

For example, whenever the design changes, loop back and check if it has introduced any new risks. Building this into sprint reviews and planning sessions keeps everyone aligned and makes it easier to spot and resolve issues early.

You may use methods like Hazard and Operability Study (HAZOP), Failure Modes and Effects Analysis (FMEA), or a bespoke risk matrix to assess hazards and classify their likelihood and severity. These IEC-recognised methods make it simpler to assign the right SILs and build documentation that's ready for audit.

PRO TIP

You'll find a hazardous event severity matrix in Annex G of IEC 61508-5 to help you get started. Results must be documented and maintained for audit readiness until product decommissioning.

Part 1: Clauses 7.5 and 7.6

Have you defined what safety functions your device must perform, and how they will be implemented?

What is this?

Clauses 7.5 and 7.6 guide you through turning hazard and risk analysis results into system requirements for functional safety. This may be a little confusing as the two clauses are closely linked but serve different purposes, so let's clarify:

Clause 7.5 has you define each of the required overall safety functions based on the hazardous events you identified when following clause 7.4. For example, if you found that a motor poses a hazard above 2,000 rpm, you'll need a safety function to prevent that. You'll also need to assign a safety integrity requirement to each function, as represented by a SIL, based on how reliable it must be to achieve tolerable risk.

Clause 7.6 has you allocate each function and its SIL to specific components in your system. So, you'll likely assign different SIL responsibilities to different embedded devices, such as microcontrollers and software modules. You must also consider independence, failure modes, and whether the system can achieve its SIL goals.

The risk of non-compliance

Consider this: software developers may quickly lose momentum and focus if safety functions are vague or missing. They may also be forced to make false assumptions to meet their deadlines, resulting in software designs that either overdeliver—creating unnecessary points of failure—or cannot reduce risk at any meaningful level.

For example, if a heating element's temperature sensor and safety function both rely on the same analogue-to-digital converter (ADC) channel or firmware task, a single failure could leave the heater running uncontrolled. By splitting the sensor and shutdown logic across separate hardware or software, you can provide the necessary independence for the assigned SIL.

Part 1: Clauses 7.5 and 7.6

Have you defined what safety functions your device must perform, and how they will be implemented?

How to get it right...

Firstly, you'll need to define a safety function for each risk and hazard identified in the analysis by applying engineering judgment or domain-specific practices.

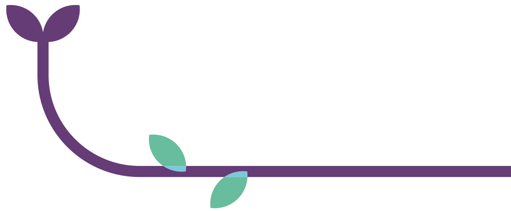
Next, you'll determine the required SIL for each function using a structured method like a risk graph, matrix, or Layer of Protection Analysis (LOPA)—as shown in Annex F of IEC 61508-5. LOPA is particularly useful for calculating risk reduction measures for high SIL systems, such as those with multiple layers of protection. However, you may find that its complexity can impede the effectiveness of your Agile workflows.

When allocating functions, consider their technical feasibility and your team's ability to meet the required SIL across development and maintenance. And if two functions share code and independence isn't proven, be mindful to assign a higher SIL—it's much safer to overestimate its requirements than risk a tragedy.

As always, document all assumptions, rationales, and dependencies. This keeps your audit trail disciplined and simplifies design changes.

PRO TIP

Involve hardware and software teams when defining safety functions and allocating SILs. This shared understanding supports realistic decisions and helps teams iteratively build and deliver a safe system.



Executing functional safety assessments

Part 1: Clause 8.2 Have you independently assessed that functional safety has been achieved

What is this?

Clause 8.2 outlines the requirements for a Functional Safety Assessment (FSA) of all parts of the overall safety-related system your team is responsible for. An FSA is an independent review that judges whether adequate functional safety has been achieved, based on compliance with IEC 61508 and supporting evidence.

Think of it as a recurring checkpoint: a chance to review progress so far, spot gaps early, and confirm whether your system is on track, defensible, and technically sound.

In Agile contexts, FSAs can be completed after each iteration or release, helping you catch new risks or vulnerabilities as they emerge—and before they cause havoc.

While you may see FSAs as a regulatory burden, optimal planning and regular reviews can boost stakeholder confidence in your embedded devices, reduce uncertainty, and accelerate certification. Plus, it validates your team's skills at delivering safe software.

Part 1: Clause 8.2 Have you independently assessed that functional safety has been achieved

The risk of non-compliance

Failing to complete an FSA—or treating it as a box-ticking exercise—can weaken your functional safety case. Assessors and regulators may delay or deny certification until you can confirm a watertight, independent review of safety-related functions. Other risks to a successful FSA include:

- Careless assignment of assessors, such as by overlooking their proximity to the design team, exceeding the minimum accepted levels of independence.
- Administrative gaps, such as over-assigning responsibilities, leading to lost or poorly maintained records that breach compliance.
- Apathetic or poorly aligned teams may withhold critical information, such as FSA acceptance rules, from downstream developers or integrators.

Timing matters, as delaying the first FSA until after the system is in operation is a breach of IEC 61508.

How to get it right…

Treat FSAs as a routine part of the project lifecycle, rather than an obligation that you can delay until the final hour. Iterative safety reviews can evolve alongside the system, giving you actionable insights when they're easiest to act on.
Here's how you can use them effectively in Agile development:

- Plan FSAs as part of your sprint or release rhythm so they always reflect your latest software iteration and provide timely insight.
- Define scope, roles, and competencies up front, but take care to adjust them as the project evolves.*
- Capture conclusions and recommendations in each cycle, even if provisional, so your safety case grows organically with each iteration.

*Independence and assessor expertise remain non-negotiable.

When managed this way, you'll find that FSAs strengthen your safety case, surface issues before they escalate, and enable agility rather than hinder it.

PRO TIP

Select assessors with the right competence and independence level using the tables found in Part 1, Clause 8.2.18 of the official standard. For high SILs or high-consequence scenarios, consider using external reviewers to build credibility with certifiers and customers.

Specifying software safety requirements

Part three of IEC 61508 outlines how to develop and assess safety-related software within the wider functional safety lifecycle. Think of it as your rulebook to ensure software meets its required SIL, covering all aspects of development.

What about clauses 7.1 and 7.2?

Before we dive into the fundamentals of IEC 61508-3, let's briefly examine two early clauses that act as a springboard for all subsequent development activities.

Clause 7.1 requires defining a software safety lifecycle model, such as bespoke V-models for specific project needs. These can support staged development and be adapted for incremental delivery in an Agile environment.

When approaching this clause, do:

- **Define** clear responsibilities for each activity, including architecture, design, V&V, and modification.
- **Plan** how you'll automate routine regression testing, such as at the end of each sprint, to make spotting safety gaps easier as development progresses.
- **Take care** to justify the merging of any steps, such as designing a software system in parallel with its architecture.

Clause 7.2 shifts the focus from lifecycle planning to defining the software's safety requirements. This means spelling out what each safety function requires for correct and reliable implementation at its assigned SIL. These sit in parallel with other functional requirements, but are subject to stricter rules for traceability, verification, and validation.

When approaching this clause, do:

- **Define** each software safety requirement as behaviour-focused acceptance criteria, describing triggers, expected responses, and key constraints.*
- **Use** a shared requirements tool to align each function to its SIL and specify response time, accuracy, and fault behaviour.
- **Document** how software and hardware interact—especially concerning timing, testability, and monitoring.

You'll find recommended techniques and measures to help you comply with clause 7.2 requirements in Annex A and Annex B of IEC 61508-3.

*When using a BDD practice like Gherkin, this means writing requirements in terms of what the system should do when something happens. For example: "When the temperature exceeds 90 degrees, the fan will stop within 500 milliseconds and raise an alarm."

Part 3: Clause 7.4 Have you really designed and developed the software to be functionally safe?

What is this?

Welcome to what may be the most expansive clause you'll cover yet: the end-to-end software design and development journey. From high-level architecture and coding to integration testing, this is where most of the technical implementation happens.

Divided across six major objectives, the clause asks that you:

1. Create a software architecture that fulfils its assigned SIL requirements.
2. Assess how software handles and interacts with hardware requirements.
3. Select and qualify tools for use across the software safety lifecycle.
4. Design and implement code that can be safely tested, verified and modified.
5. Verify that safety-related software requirements have been achieved.
6. Ensure that embedded devices using configuration data are safe and reliable.

We strongly recommend reading this clause in its entirety while paying close attention to its many nuances. For example, it's easy to miss that if you reuse pre-existing code—such as third-party middleware—the supplier will need to provide a safety manual for the integrator that precisely explains its behaviour, assumptions, and limitations.

The risk of non-compliance

At this point in development, negligence can lead to erratic software behaviour, costly integration issues, and unreliable code that's likely to fail under edge conditions.

And while your software may pass internal tests, it won't satisfy any auditors.

You may also encounter:

- Poor separation of safety and non-safety code, making it unclear which parts must meet higher SIL rules.
- Repeated safety logic in multiple places without explanation, which increases the chance of inconsistent or hidden features.
- Auditors rejecting reused or third-party software if no safety manual exists, or if reduced testing isn't properly justified.

And without proper controls or thorough code assessment, bugs may slip by unnoticed until field deployment, breaching your claimed SIL and leading to corrective costs.

Part 3: Clause 7.4 Have you really designed and developed the software to be functionally safe?

How to get it right...

The smartest approach to functional safety is to design your software with modularity, testability, and fault tolerance in mind from day one. And as subclause 7.4.2.2 states, your chosen software design method must:

- Include features that allow software modification, including encapsulation.
- Be holistically treated as safety-related, even if it has non-safety functions.
- Treat all software functions as the highest SIL unless independence is shown.*

Clause 7.4.4 likewise asks that you select and justify development tools based on how much the system relies on them for safety.

IEC 61508 provides tool classification tiers for analysing their potential impact on safety-related systems, these are:

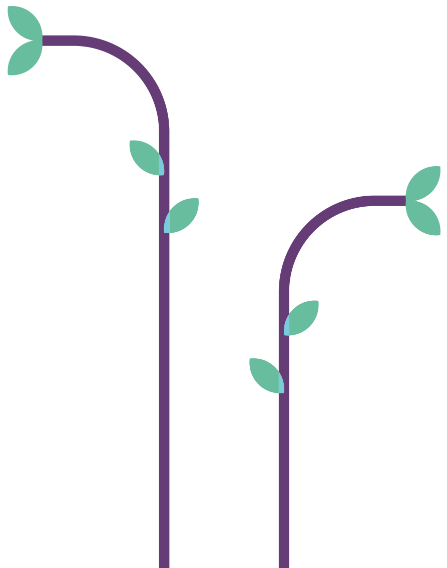
- **T1 tools** can't introduce software faults and aren't relied upon to detect faults.
- **T2 tools** can introduce software faults, but aren't relied upon to detect faults.
- **T3 tools** are relied upon to detect faults, but could fail to do so.

T1 tools such as text editors and spreadsheet apps require no special justification for use. T2 and T3 tools, such as code compilers and static analysers, must be assessed.

*You can find techniques for achieving independence—and preventing interference— between different software elements in Annex F, IEC 61508-3.
**You'll find helpful design and verification measures in Annexes A and B of IEC 61508-3.

PRO TIP

Frame early design and tool decisions as safety decisions; they'll influence how easily you can demonstrate compliance later.



Tackle the software re-use conundrum

If using third-party software or recycling legacy code, clause 7.4 offers three possible routes to compliance:

Route 1S is taken if the software was originally developed following IEC 61508 standards and remains fully documented and compliant.

- Ideal for: In-house components or vendor software built explicitly for functional safety applications.

Route 2S 2S is taken if it wasn't developed to the IEC 61508 standard, but you have proven operational maturity.

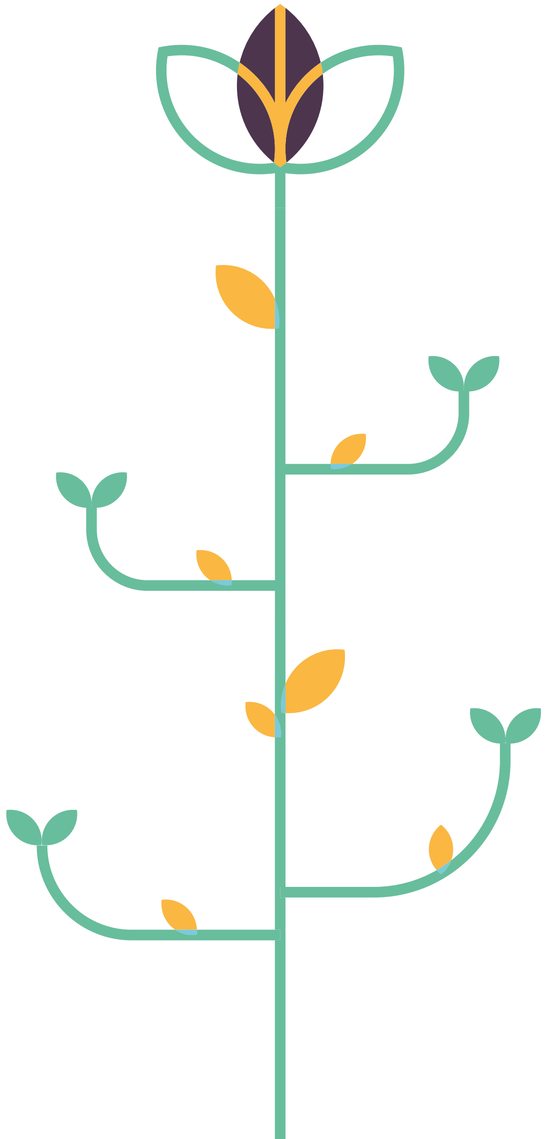
- Ideal for: Commercial off-the-shelf software that's been used in similar embedded devices.

Route 3S is only taken when 1S or 2S aren't possible, given that you can fully justify and support it with extra V&V and the creation of a safety manual.

- Ideal for: Legacy code or third-party software without a demonstrable history of use in safety applications.

Remember, no matter which route you take, you must supply a safety manual for the team or individual responsible for integrating pre-existing software into a safety-related system. This ensures they understand how the component can be safely used, what assumptions and constraints must be respected, and how to avoid introducing systematic faults during integration.

In the case of Route 3S, you may need to author this yourself.



Part 3: Clause 7.5 Have you tested if your software is compatible with real hardware?

Are your software and hardware inseparable?
If your embedded software and hardware can't be separated during testing, you must follow this clause in parallel with the software integration testing required by clause 7.4.8.

What is this?

Clause 7.5 ensures that your safety-related software correctly integrates with and runs on actual hardware. This means verifying the software's performance and integrity by checking behaviour, timing, and compatibility with the embedded device's hardware, plus any connected input/output (I/O) like sensors or actuators.

To keep things focused, create a formal integration plan that defines test levels, test cases, tools, environments, and acceptance criteria. In an Agile environment, this plan can be exercised iteratively throughout each sprint. And if anything changes, like an embedded board revision, be sure to reassess what it does and reverify as needed.

Done right, you'll prove that your software and hardware can function safely together under operation. It also helps reveal discreet bugs such as timing mismatches before the system is commissioned.

The risk of non-compliance

If resources are limited, it may be tempting to veto or downplay the importance of integration testing. However, this poses a serious risk to the stability of your final code, as potentially ruinous bugs may slip through unnoticed.

Software code that runs fine in simulation may behave differently when running on real hardware, revealing performance quirks and bugs that nobody could've predicted. Not only is this an instant SIL breach, but it's also an all-too-common source of delay that can be easily avoided through frequent and proactive integration and testing.

Failing to document tests, test environments, or the outcome also weakens your safety case, and rework becomes even harder if no baseline exists. Another common pitfall? Pushing ahead with final testing or certification without properly capturing integration issues that arose earlier.

Part 3: Clause 7.5 Have you tested if your software is compatible with real hardware?

How to get it right...

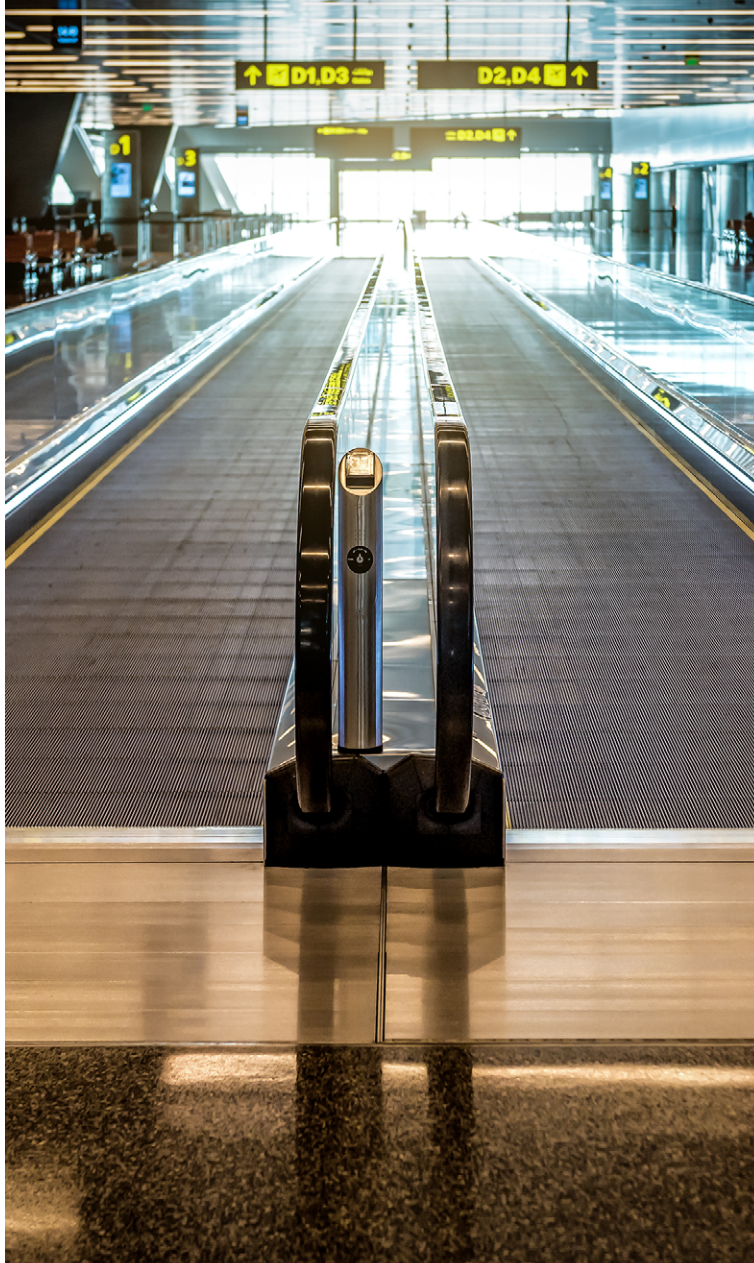
Integrate software and hardware as early and often as your resources allow, rather than treating it as a task for after development has wrapped up. Catching hardware/software mismatches now is far cheaper than chasing down bugs when your code is out in the field.

Use target hardware and real I/O where possible, logging tool versions, configuration states, and expected results, so any failures can be easily traced and resolved. If using the recommended techniques found in Annex A and B or IEC 61508-3, automate routine tests to quickly verify changes or try hardware-in-the-loop testing to catch regressions.

After each integration cycle, run an impact analysis to determine whether changes require re-verification. And while it should've become instinct by now, ensure results are auditable—especially for higher SIL claims. This stage is your last line of defence before the system enters real-world environments.

PRO TIP

Treat integration like a sprint deliverable: aim to end each sprint with software that's been tested on real hardware, even if only for a subset of functions. This keeps integration risks visible and prevents nasty surprises from piling up at the end.



Part 3: Clause 7.7 and 7.9 Have you verified and validated your software against its safety goals?

What is this?

Let's look at clauses 7.7 and 7.9 together, as they unpack verification and validation, respectively. Together, they require that your software does what it should, and in alignment with its target SIL.

Clause 7.7 has you check that the fully integrated system, including all software and embedded devices, behaves as expected. Software tools must also be verified so they meet the design requirements in clause 7.4.4. But take note: typically, software can't be validated separately from its underlying hardware and system environment.

Clause 7.9 governs software verification, outlining the testing and evaluation of outputs across each development activity. This shows whether the software does what it should under expected conditions, including those that may cause faults, and that each stage of development was logically performed.

The risk of non-compliance

By treating V&V as a last-minute concern, you may find specification errors, overlooked edge cases, or even architectural flaws that are too far gone to fix before release, or may require an overwhelming amount of revisionism and investment to correct.

Negligent V&V practices also leave teams struggling to explain safety performance to auditors, even if it works at face value, resulting in delayed time-to-market, regulatory pushback, and excessive capital losses.

Part 3: Clause 7.7 and 7.9 Have you verified and validated your software against its safety goals?

How to get it right...

Build your V&V plan early—in fact, clause 7.3 calls for verification processes to be defined alongside development. For a watertight approach, use traceable test plans that simulate both normal and fault conditions, and make sure test environments are defined and repeatable.

You can also use static techniques like code reviews and data inspections to help build confidence in the test results. And for validation, document both the outcomes and the rationale behind pass and fail decisions, especially when discrepancies arise.

If software is reused or modified, revisit earlier lifecycle phases as required by clause 7.8. For ease of compliance, create templates for test specifications and validation summaries, and automate report generation where possible. These steps help make assessments smoother and reduce the burden of future audits.

Your IEC 61508 maturity assessment

In functional safety, your compliance journey is unlikely to follow a binary "yes or no" roadmap. Instead, it matures over time as you apply and continuously build upon the requirements covered in this guide.

To help you track your progress, we've created a bespoke maturity assessment for IEC 61508-aligned activities, based on the widely used Capability Maturity Model (CMM). It scores your current status for each activity using the following levels:

- **Level 0 (Ad-Hoc):** The activity is incomplete or performed inconsistently, with little structure, traceability, or evidence.
- **Level 1 (Defined):** The activity's process and requirements are documented and planned, but implementation has not yet made any meaningful progress.
- **Level 2 (Realised):** The activity is implemented, traceable, and carried out consistently across design, software, and hardware.
- **Level 3 (Verified):** The activity has been tested, reviewed, and independently assessed, supported by audit-ready evidence.

How to use the maturity assessment table

For each clause-aligned activity in the table, choose the highest level (0-3) that accurately reflects your current practices and record it in the adjacent score box.

Upon completion, calculate the total of your ten scores using the maturity banding guide on page 39 to determine your overall maturity level and inform your next steps.

Activity and Standard Reference	Level 1: Defined	Level 2: Realised	Level 3: Verified	Score (0-3)
Documentation and Evidence (Part 1, Clause 5.2)	Documentation approach defined	Evidence maintained with version control and traceability	Documentation independently reviewed and safety-case ready	
Functional Safety Management (Part 1, Clause 6.2)	Roles and responsibilities defined	FSM plan executed across the lifecycle	FSM independently assessed with competence evidence	
Safety Lifecycle Model (Part 1, Clause 7.1)	Lifecycle phases defined	Activities executed with clear inputs and measured outputs	Lifecycle model reviewed internally/externally with continuous improvements	
Hazard and Risk Analysis (Part 1, Clause 7.4)	Hazard analysis method selected (e.g. HAZOP)	Hazard analysis results evaluated iteratively and recorded	Analysis independently reviewed and linked to architecture	
Safety Functions and SIL Allocation (Part 1, Clauses 7.5–7.6)	Safety functions and SILs defined	SILs allocated in architecture with independence considered	Allocation independently reviewed, assumptions evidenced	
Functional Safety Assessment (FSA) (Part 1, Clause 8.2)	FSA scope and criteria defined	FSAs performed at lifecycle stages with findings logged	Independent assessors confirm adequacy of evidence	
Software Safety Requirements (Part 3, Clauses 7.1–7.2)	Responsibilities and requirements defined, the latter as behaviour-led acceptance criteria	Traceable to hazards, SILs, architecture, and V&V	Independently reviewed, traceability consistently maintained	
Software Design and Development (Part 3, Clause 7.4)	Architecture, coding rules, and tools are established and active	Software releases conform to the architecture, with justified reliance on tools and reuse	Independent verification validates that the release meets SIL claims with traceable evidence	
Hardware/Software Integration (Part 3, Clause 7.5)	Integration strategy and acceptance criteria defined	Testing executed on target hardware	Results independently reviewed, behaviour under fault conditions demonstrated	
Verification and Validation (V&V) (Part 3, Clauses 7.7 and 7.9)	V&V plans define methods and coverage	V&V performed with traceable results	V&V independently reviewed with full requirements-to-test traceability	

Calculate your overall maturity

Add up each of your activity scores from the maturity assessment table to learn how mature your project is in its compliance with IEC 61508 requirements.

0–7: Ad-hoc

High risk due to significant structural and evidence gaps. You are not ready for certification or continued development without immediate and dramatic changes.

8–15: Defined

Functional safety foundations are in place, but execution is inconsistent. Substantial work is required before the project is audit ready.

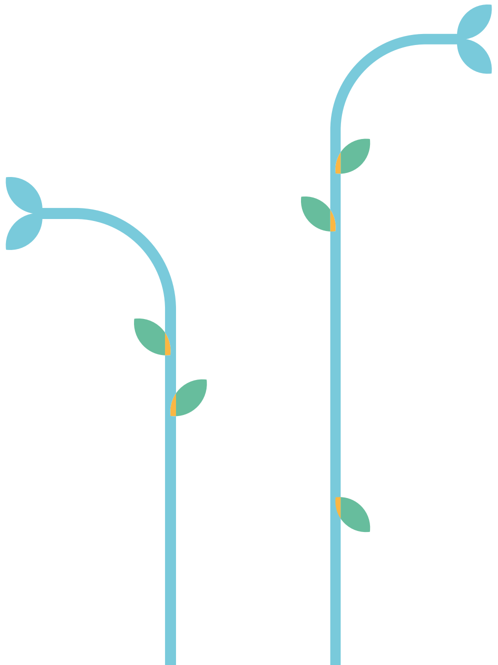
16–23: Realised

Strong implementation of functional safety. The embedded system is largely compliant with IEC 61508, but evidence or independence gaps remain.

24–30: Verified

Audit ready. Activities are clearly defined, executed, evidenced, and independently confirmed.

Total Score (0–30): __________



About Bluefruit Software

For more than twenty-five years, Bluefruit has supported safety-critical organisations with world-class software and regulatory expertise. And we can help ready your safety-critical products for audit, compliance, and successful global market launch.

Our core services include:

- Embedded software development (including firmware and UX)
- Embedded software testing (including Verification and Validation)
- Embedded software consultation
- Software compliance and regulatory support
- Hardware development and 3D prototyping
- Edge AI / AI for embedded systems
- Lean-Agile training and workshops

Plus, we believe in a collaborative approach. That's why our software teams will work alongside your in-house team members and existing outsourcers to help you achieve the results you need.

Bluefruit Software

Bluefruit Software
Gateway Business Centre
Barncoose Gateway Park
Redruth
Cornwall
United Kingdom TR15 3RQ

+44 (0) 808 18 000 55 (FREEPHONE)
+44 (0) 333 577 7111

www.bluefruit.co.uk
hello@bluefruit.co.uk