A quick guide to ISO 26262

FEABHAS

What is ISO 26262?

ISO 26262 is an international standard focussing on the safety of automotive electrical / electronic systems.

That is, ISO 26262 is designed to ensure that embedded systems developed for road vehicles – cars, lorries motorbikes, etc – are designed with an *appropriate level of rigour* for their intended application.

What does 'appropriate level of rigour' mean?

ISO 26262 actually covers two aspects of a system's development: *Safety* and, to a lesser extent, *Intrinsic Quality*.

Safety focuses on ensuring that failures in the system software do not lead to (external) conditions that could cause harm to people.

Intrinsic Quality emphasises 'good' design – simplicity, robustness, maintainability, testability, etc.

Clearly, there is a close relationship between these factors: a design of high Intrinsic Quality is more likely to perform safely, and be more readily demonstrable to be safe.

How do I measure safety?

Safety refers to harm to people. A safe system is one which does not cause harm to people. Of course, no system can be made *completely* safe, so safe systems attempt to reduce the potential for harm to an acceptable level (In UK Health and Safety law this concept is called *ALARP* – As Low As Reasonably Practicable).

ISO 26262 takes a risk-based approach to managing potential harm (often referred to as *residual risk*), based on three factors:

- Severity - the potential harm
- Exposure – the probability of occurrence
- Controllability - the ability of the system to avoid the specified harm

ISO 26262 organises the risks into four *Automotive Safety Integrity Levels*, or ASILs. ASIL Level A is the lowest level; Level D is the highest.

Is ISO 26262 just about software development?

ISO 26262 covers the entire product development lifecycle of electrical / electronic automotive products. The standard is composed of 10 parts, as shown below:

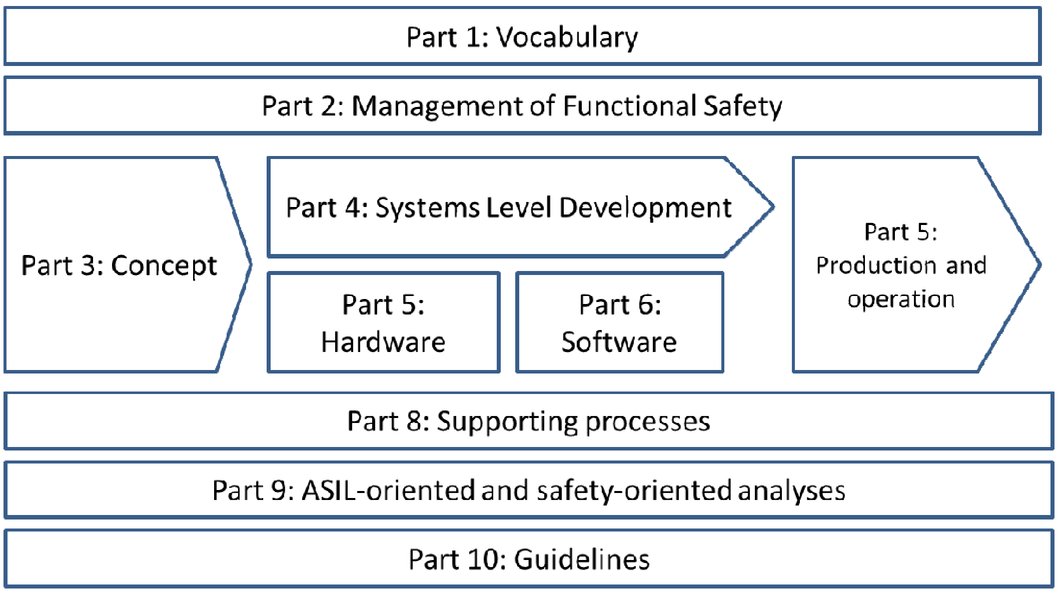
Part 1: Vocabulary
Part 2: Management of Functional Safety
Part 3: Concept
Part 4: Systems Level Development
Part 5: Hardware
Part 6: Software
Part 5: Production and operation
Part 8: Supporting processes
Part 9: ASIL-oriented and safety-oriented analyses
Part 10: Guidelines

Figure 1 - The parts of ISO 26262

Part 1 defines the language of ISO 26262 – terms, abbreviations, acronyms, etc.

Part 2 is an over-arching guide focusing on the management of safety requirements, both from a project and organisational point of view.

Part 3 focuses on what ISO 26262 calls the ‘concept phase’. This phase is concerned with initial project definition, establishing the safety requirements and criteria for the project and initiating the safety lifecycle.

Part 4 is concerned with systems level development – that is, detailed requirements analysis, system synthesis, functional and logical allocation, and system evaluation, validation and verification.

Part 5 covers the hardware aspects of system design and implementation.

Part 6 focuses on the software aspects of system design and implementation.

Part 7 details requirements for system production, operation, installation, servicing, decommission, etc.

Part 8 defines requirements for processes that support the development effort, including change management, documentation standards, tool qualification, verification and validation, etc.

Part 9 gives requirements and guidance with respect to safety analyses; in particular, all aspects related to ASIL-oriented requirements.

Part 10 gives guidance on applying ISO 26262. *This document is currently under development.*

So ISO 26262 is a process?

No. Although many people believe it does describe a process.

ISO 26262 makes the assumption you are already working to a defined development process. The standard applies additional constraints to your process, focussed on the system safety aspects.

ISO 26262 uses a classic 'V-model' framework to organise its requirements. The V-model is a standard way of describing the relationship between your development artefacts.

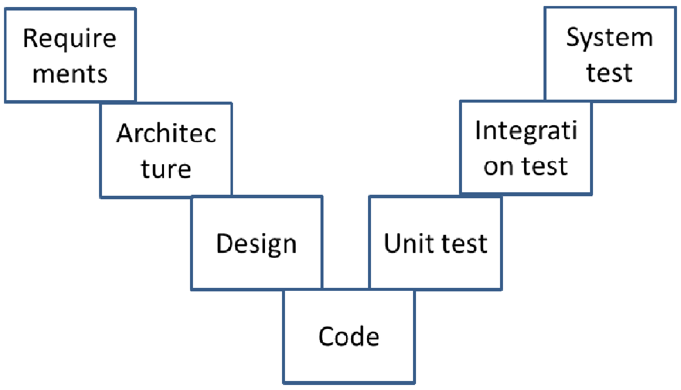

Figure 2 - A classic V-model

Every development process should contain – in some form – the elements shown on a V-model; but perhaps not exactly as required by ISO 26262. Complying with ISO 26262 will normally mean mapping existing design artefacts onto ISO 26262 Work Products.

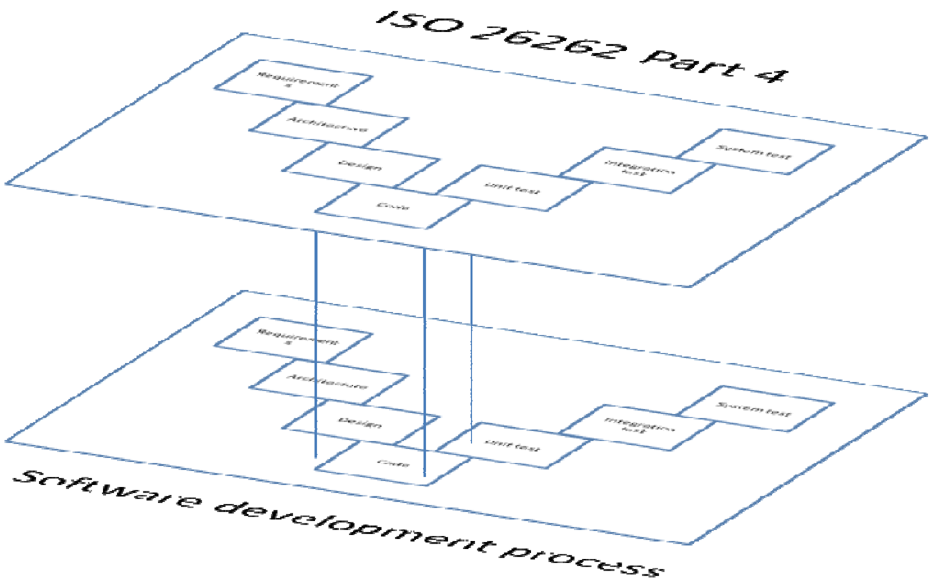

Figure 3 - ISO 26262 adds additional constraints on top of your development process

The confusion arises because many engineers mistakenly believe the V-model describes a process, with workflows and activities. In fact there are many ways to traverse the V-model – Waterfall processes, Agile processes, etc.

What additional constraints does ISO 26262 impose?

ISO 26262 is focussed on the safety aspects of the system under development.   The additional rigour (effort) that ISO 26262 imposes is focussed on ensuring the system meets its safety requirements. For any particular process step ISO 26262 (in general) requires the following aspects be considered:

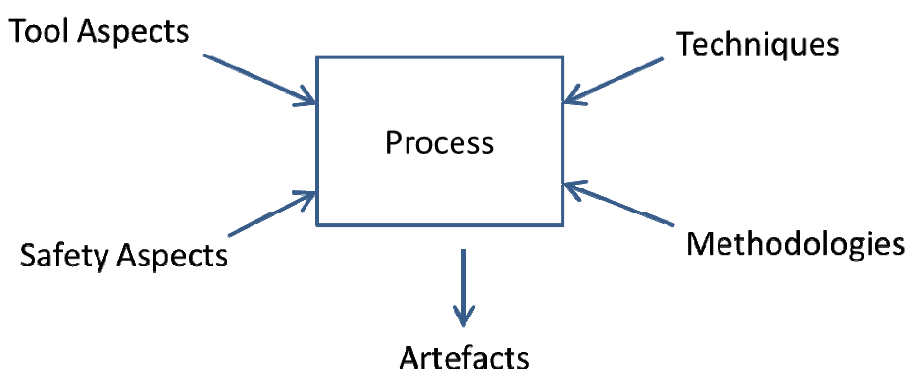

Figure 4 - Additional process aspects required by ISO 26262

Tool Aspects

The developer must consider what tools will be used at each stage; the applicability of the tools; and how they will be used

Techniques

For each process stage (what ISO 26262 calls a '*sub-phase*') ISO 26262 specifies a set of recommended techniques (for example, code inspection). Higher levels of integrity have more recommended techniques; and techniques move from being 'suggested' to 'required'.

Methodologies

Although ISO 26262 specifies a comprehensive set of techniques it does not give any guidance on how to apply them. That is, ISO 26262 does not specify any methodologies – for example, use of OO design. The development organisation must, however, specify and document methodologies / best practices / guidelines for each sub-phase of the development.

Artefacts

Artefacts capture the output of the design process. ISO 26262 calls these *Work Products,* and is somewhat prescriptive in the Work Products that must be produced at each stage.

Safety Aspects

Safety aspects are those inputs or methodologies directly related to system safety; as opposed to system intrinsic quality.

What's the impact of ASILs?

ASILs are used to configure the ISO 26262 requirements. In general, the higher the ASIL the more rigorous the development must be.

To be compliant with ISO 26262 the development organisation must perform all the activities, and produce all the work products specified by the standard. The difference is the amount of work (the rigour) that must be done at each stage.

ASILs are used to select appropriate techniques at each sub-phase. Techniques are specified, and can be interpreted, as followed:

"Highly Recommended" (++)

These techniques are, for all practical purposes, mandatory. Highly Recommended techniques must be applied. If the technique is not to be applied, there must be a very compelling justification for why it is being omitted.

"Recommended" (+)

Recommended techniques are good practice; although not strictly demanded by the standard. You should be doing Recommended practices as part of your normal development anyway. If the development organisation is not going to perform a Recommended practice it should justify the reasons.

"No Recommendation" (o)

The standard has no opinion on the technique. This *could* be interpreted as optional (if you are not already performing the technique)

What's the financial impact of higher ASIL?

Although there is no fixed calculation it is generally accepted that each increase is ASIL level causes a ten-fold increase in cost, due to the extra levels of rigour and effort required.

That is, an ASIL B system will be ten times more expensive that an ASIL A; an ASIL C one hundred times; and an ASIL D one thousand times more expensive.

It remains to be seen whether the automotive embedded systems industry can withstand such an increase in development costs; or what the consequence for safety-critical systems in automotive applications will be.

How does Part 6 (Software) fit in with the rest of ISO 26262?

ISO 26262 makes a clear distinction between Systems Engineering, and Software and Hardware development.

Systems Engineering is focussed on requirements capture, system design and evaluation and allocation of system behaviour (and appropriate qualities of service) onto hardware, software (and mechanical components). The outputs from the Systems Process are software and hardware design specifications

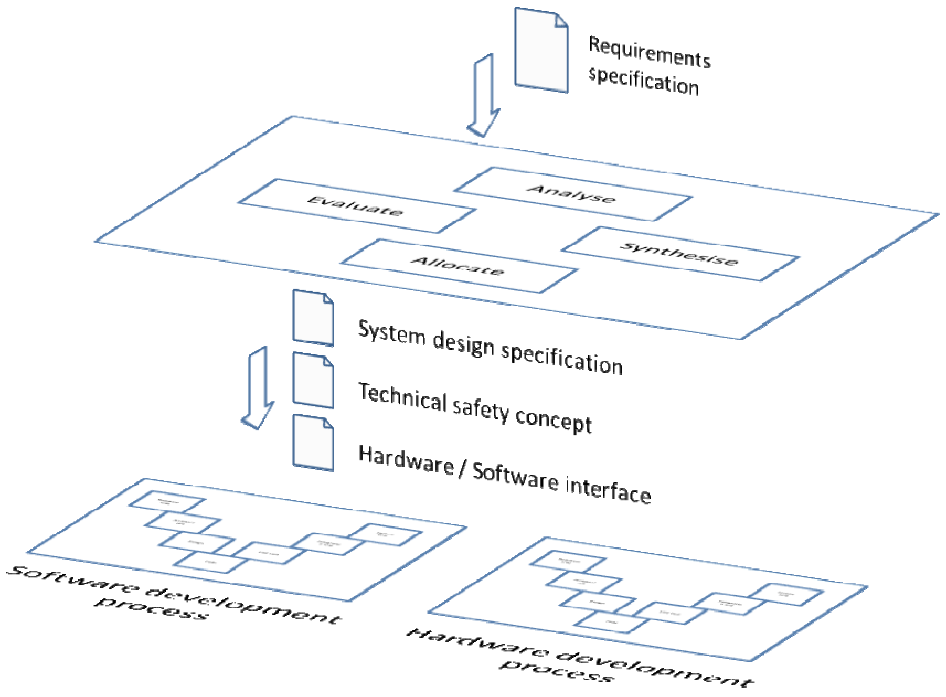
Requirements Specification
Analyse
Evaluate
Synthesise
Allocate
System design specification
Technical safety concept
Hardware / Software interface
Software development process
Hardware development process

Figure 5 - The relationship between Systems, Software and Hardware in ISO 26262

The software design process (Part 6) focuses on the development of software on one (or possibly more, depending on the systems design) microcontroller.

There is a potential overlap (and weakness in the process) between software allocation design and systems engineering. This overlap will need clarifying by the development organisation.

What's the difference between ISO 26262 and IEC 61508?

ISO 26262 is a variant of IEC 61508, designed specifically for the automotive industry.

IEC 61508 was originally intended for use with large, industrial safety-critical systems – chemical plants, nuclear reactors, etc. With such systems, installation is a major part of the safety process. Commonly, automotive embedded systems are sold as OEM products; and the installation process is not critical in the safety of the device. ISO 26262 drops most of the installation standards from IEC 61508 .

ISO 26262, being a newer standard, acknowledges many of the common practices used in automotive embedded systems development – particularly model-based development, using tools like Matlab or LabView.

Finally, IEC 61508 gives *guidelines* regarding documentation requirements. ISO 26262 is rather more prescriptive about the documents you are required to produce.

Example Systems

Below are typical examples from each of the ASIL levels: an ASIL level A system, ASIL level B, etc. These systems should be taken as representative only – that is, not all cruise control systems will be level A.

In these examples the ISO 26262 techniques for each development phase will be listed. It is assumed that where a technique is listed as Highly Recommended you *must* apply it; Recommended techniques *should* be performed; and techniques with no recommendation are considered optional and can be omitted. The aim is to suggest the minimal set of required techniques for each ASIL level.

Remember also that the standard does not contain any methodologies for achieving the required rigour levels. Where a technique is specified, the standard does not give any guidance for how, when or where the technique must be applied.

ASIL Level D System – Electric Steering

The electric steering system has the potential to cause significant harm – for example, providing the wrong level of assistance or feedback, or even completely incorrect output (locking the steering against an end-stop).

Initiation of software development.

The focus of this sub-phase is the planning of the software development activity, with particular emphasis on ensuring appropriate tools, methodologies, guidelines, etc. are in place.

With regard to modelling guidelines the following techniques should be covered:

Table 1 - Coding and modelling guidelines

Technique	Highly Recommended	Recommended
Enforcement of low complexity	●	
Use of language subsets	●	
Enforcement of strong typing	●	
Use of defensive implementation techniques	●	
Use of established design principles	●	
Use of unambiguous graphical representation	●	
Use of style guides	●	
Use of naming conventions	●	

Specification of software safety requirements

The emphasis of this sub-phase is ensuring the approach taken to ensure software safety is consistent with the system safety philosophy.

ISO 26262 provides recommendations for this sub-phase but does not specify any recommended techniques.

Software architectural design

In this sub-phase the software architecture is synthesised, evaluated and verified. ISO 26262 defines 'architectural design' as the design of the software down to the level of an implementable component. There is no guidance given on what this constitutes, but it can be assumed to be an individual module, package or class.

ISO 26262 defines techniques for design notations for the software architecture.

Table 2 - Notations for Architectural design

Technique	Highly Recommended	Recommended
Informational notations		●
Semi-formal notations	●	
Formal notations		●

The standard also encourages high intrinsic quality at the architectural level in order to avoid (for example) failures due to over-complicated designs. The following principles are specified. Once again, the standard gives no guidance on *how* these principles should be achieved or how they should be evaluated.

Table 3 - Principles for Architectural design

Technique	Highly Recommended	Recommended
Hierarchical structure of software components	●	
Restricted size of software components	●	
Restricted size of interfaces		●
High cohesion within each software component	●	
Restricted coupling between software components	●	
Appropriate scheduling properties	●	
Restricted use of interrupts	●	

One of the key safety mechanisms for software is error detection and handling. ISO 26262 specifies the use of the following techniques for error detection:

Table 4 – Mechanisms for error detection at architectural level

Technique	Highly Recommended	Recommended
Range checks of input and output data	●	
Plausibility checks	●	
Detection of data errors (e.g. parity checking)		●
External monitoring facility	●	
Control flow monitoring	●	
Diverse software design	●	

Correspondingly, the standard defines acceptable error handling strategies:

Table 5 – Error handling mechanisms at architectural level

Technique	Highly Recommended	Recommended
Static recovery mechanisms		●
Graceful degradation	●	
Independent parallel redundancy	●	
Correcting codes for data		●

Although no evaluation or validation techniques are specified the following techniques are defined for verification of the software architecture:

Table 6 – Methods for verification of software architecture

Technique	Highly Recommended	Recommended
Walk-through		
Inspection of the design	●	
Simulation of the dynamic parts of the design	●	
Prototype generation	●	
Formal verification		●
Control flow analysis	●	
Data flow analysis	●	

Software unit design and implementation

Software unit design is focused on code-level design, implementation and verification.

When specifying software unit designs ISO 26262 recommends the following techniques:

Table 7 - Notations for software unit design

Technique	Highly Recommended	Recommended
Natural language	●	
Informal notations		●
Semi-formal notations	●	
Formal notations		●

The above is assumed to mean prefer semi-formal notation over natural language.

(ISO 26262 does not give any guidance as to what constitutes an informal, semi-formal or formal notation)

As with architectural design, ISO 26262 gives recommended principles that must be complied with. As noted in the standard, many of the principles are covered (for C) by MISRA-C.

Table 8 – Design principles for software unit design and implementation

Technique	Highly Recommended	Recommended
One entry and one exit point in functions	●	
No dynamic objects or variables, or else online test during their creation	●	
Initialization of variables	●	
No multiple use of variable names	●	
Avoid global variables or else justify their usage	●	
Limited use of pointers	●	
No implicit type conversions	●	
No hidden data flow or control flow	●	
No unconditional jumps	●	
No recursions	●	

ISO 26262 makes a distinction (albeit not particularly explicitly) between dynamic verification and static verification. The following techniques are specified for *static* verification of the software unit design. Many of these techniques require third-party software tools.

Table 9 – Methods for verification of software unit design

Technique	Highly Recommended	Recommended
Walk-through		
Inspection	●	
Semi-formal verification	●	
Formal verification		●
Control flow analysis	●	
Data flow analysis	●	
Static code analysis	●	
Semantic code analysis		●

Software unit testing

Software unit testing is focused on demonstrating the software units fulfil their specification (if not necessarily the requirements of the system)

In this case unit testing is taken to mean *dynamic* testing – that is, testing by executing the software in a simulated environment.

The following types of dynamic testing are specified by ISO 26262

Table 10 – Methods for software unit testing

Technique	Highly Recommended	Recommended
Requirements-based test	●	
Interface test	●	
Fault injection test	●	
Resource usage test	●	
Back-to-back comparison tests between code and model (if applicable)	●	

The design of test cases is specified by using the following methods:

Table 11 – Methods for deriving test cases for unit tests

Technique	Highly Recommended	Recommended
Analysis of requirements	●	
Generation and analysis of equivalence classes	●	
Analysis of boundary values	●	
Error guessing		●

To evaluate the completeness of test cases and to demonstrate that there is no unintended functionality, ISO 26262 recommends structural coverage (also known as *white box*) testing. Structural coverage testing generates metrics to establish the completeness of testing. ISO 26262 specifies the following coverage metrics

Table 12 – Structural coverage metrics

Technique	Highly Recommended	Recommended
Statement coverage		●
Branch coverage	●	
Modified Condition / Decision Coverage (MC/DC)	●	

Software integration and testing

This sub-phase is focused on the construction of the software architecture from its component parts and ensuring the resulting system performs as expected.

In general software integration is done in controlled stages / iterations; these stages being planned in conjunction with the hardware and systems teams.

As previously, ISO 26262 gives no guidance on software integration methodologies and does not enforce any particular strategy. The standard does , however, define techniques for which types of integration test should be performed.

Table 13 – Methods for integration testing

Technique	Highly Recommended	Recommended
Requirements-based test	●	
Interface test	●	
Fault injection test	●	
Resource usage test	●	
Back-to-back comparison tests between code and model (if applicable)	●	

Note – this is the same set of test types as for unit testing; this time performed at a higher level of abstraction (composition)

Similarly, methods for identifying integration test cases are the same as for unit tests:

Table 14 – Methods for deriving integration test cases

Technique	Highly Recommended	Recommended
Analysis of requirements	●	
Generation and analysis of equivalence classes	●	
Analysis of boundary values	●	
Error guessing		●

Since coverage metrics for unit testing become increasing unwieldy as the integrated system grows, ISO 26262 identifies a more appropriate set of coverage metrics for integration testing.

Table 15 – Structural coverage metrics for integration tests

Technique	Highly Recommended	Recommended
Function coverage	●	
Call coverage	●	

(The standard does not really make clear the difference between the percentage of *executed* functions and the percentage of function *calls*)

Verification of software safety requirements

The emphasis of this phase is demonstrating (verifying) that the software fulfils its safety requirements.

Software testing may largely be performed in hosted or simulated environments. Such testing cannot guarantee that the software requirements have been met. ISO 26262 establishes additional

test environments for performing tests to increase confidence that the software is operating correctly.

Table 16 – Test environments for software safety verification

Technique	Highly Recommended	Recommended
Hardware-in-the-loop	●	
Electronic control unit network environments	●	
Vehicles	●	

ASIL Level C System – Passenger Airbag

The passenger airbag system has the potential to cause significant harm – for example, by deploying too early, too late, or not at all. As such, the requirements for ASIL level C systems are almost as strict as for Level D.

Initiation of software development.

The focus of this sub-phase is the planning of the software development activity, with particular emphasis on ensuring appropriate tools, methodologies, guidelines, etc. are in place.

With regard to modelling guidelines the following techniques should be covered:

Table 17 - Coding and modelling guidelines

Technique	Highly Recommended	Recommended
Enforcement of low complexity	●	
Use of language subsets	●	
Enforcement of strong typing	●	
Use of defensive implementation techniques	●	
Use of established design principles		●
Use of unambiguous graphical representation	●	
Use of style guides	●	
Use of naming conventions	●	

Specification of software safety requirements

The emphasis of this sub-phase is ensuring the approach taken to ensure software safety is consistent with the system safety philosophy.

ISO 26262 provides recommendations for this sub-phase but does not specify any recommended techniques.

Software architectural design

In this sub-phase the software architecture is synthesised, evaluated and verified. ISO 26262 defines 'architectural design' as the design of the software down to the level of an implementable component. There is no guidance given on what this constitutes, but it can be assumed to be an individual module, package or class.

ISO 26262 defines techniques for design notations for the software architecture.

Table 18 - Notations for Architectural design

Technique	Highly Recommended	Recommended
Informational notations		●
Semi-formal notations	●	
Formal notations		●

The standard also encourages high intrinsic quality at the architectural level in order to avoid (for example) failures due to over-complicated designs. The following principles are specified. Once again, the standard gives no guidance on *how* these principles should be achieved or how they should be evaluated.

Table 19 - Principles for Architectural design

Technique	Highly Recommended	Recommended
Hierarchical structure of software components	●	
Restricted size of software components	●	
Restricted size of interfaces		●
High cohesion within each software component	●	
Restricted coupling between software components	●	
Appropriate scheduling properties	●	
Restricted use of interrupts		●

One of the key safety mechanisms for software is error detection and handling. ISO 26262 specifies the use of the following techniques for error detection:

Table 20 – Mechanisms for error detection at architectural level

Technique	Highly Recommended	Recommended
Range checks of input and output data	●	
Plausibility checks		●
Detection of data errors (e.g. parity checking)		●
External monitoring facility		●
Control flow monitoring	●	
Diverse software design		●

Correspondingly, the standard defines acceptable error handling strategies:

Table 21 – Error handling mechanisms at architectural level

Technique	Highly Recommended	Recommended
Static recovery mechanisms		●
Graceful degradation	●	
Independent parallel redundancy		●
Correcting codes for data		●

Although no evaluation or validation techniques are specified the following techniques are defined for verification of the software architecture:

Table 22 – Methods for verification of software architecture

Technique	Highly Recommended	Recommended
Walk-through		
Inspection of the design	●	
Simulation of the dynamic parts of the design		●
Prototype generation		●
Formal verification		●
Control flow analysis	●	
Data flow analysis	●	

Software unit design and implementation

Software unit design is focused on code-level design, implementation and verification.

When specifying software unit designs ISO 26262 recommends the following techniques:

Table 23 - Notations for software unit design

Technique	Highly Recommended	Recommended
Natural language	●	
Informal notations		●
Semi-formal notations	●	
Formal notations		●

The above is assumed to mean prefer semi-formal notation over natural language.

(ISO 26262 does not give any guidance as to what constitutes an informal, semi-formal or formal notation)

As with architectural design, ISO 26262 gives recommended principles that must be complied with. As noted in the standard, many of the principles are covered (for C) by MISRA-C.

Table 24 – Design principles for software unit design and implementation

Technique	Highly Recommended	Recommended
One entry and one exit point in functions	●	
No dynamic objects or variables, or else online test during their creation	●	
Initialization of variables	●	
No multiple use of variable names	●	
Avoid global variables or else justify their usage	●	
Limited use of pointers		●
No implicit type conversions	●	
No hidden data flow or control flow	●	
No unconditional jumps	●	
No recursions	●	

ISO 26262 makes a distinction (albeit not particularly explicitly) between dynamic verification and static verification. The following techniques are specified for *static* verification of the software unit design. Many of these techniques require third-party software tools.

Table 25 – Methods for verification of software unit design

Technique	Highly Recommended	Recommended
Walk-through		
Inspection	●	
Semi-formal verification	●	
Formal verification		●
Control flow analysis	●	
Data flow analysis	●	
Static code analysis	●	
Semantic code analysis		●

Software unit testing

Software unit testing is focused on demonstrating the software units fulfil their specification (if not necessarily the requirements of the system)

In this case unit testing is taken to mean *dynamic* testing – that is, testing by executing the software in a simulated environment.

The following types of dynamic testing are specified by ISO 26262

Table 26 – Methods for software unit testing

Technique	Highly Recommended	Recommended
Requirements-based test	●	
Interface test	●	
Fault injection test		●
Resource usage test		●
Back-to-back comparison tests between code and model (if applicable)	●	

The design of test cases is specified by using the following methods:

Table 27 – Methods for deriving test cases for unit tests

Technique	Highly Recommended	Recommended
Analysis of requirements	●	
Generation and analysis of equivalence classes	●	
Analysis of boundary values	●	
Error guessing		●

To evaluate the completeness of test cases and to demonstrate that there is no unintended functionality, ISO 26262 recommends structural coverage (also known as *white box*) testing. Structural coverage testing generates metrics to establish the completeness of testing. ISO 26262 specifies the following coverage metrics

Table 28 – Structural coverage metrics

Technique	Highly Recommended	Recommended
Statement coverage		●
Branch coverage	●	
Modified Condition / Decision Coverage (MC/DC)		●

Software integration and testing

This sub-phase is focused on the construction of the software architecture from its component parts and ensuring the resulting system performs as expected.

In general software integration is done in controlled stages / iterations; these stages being planned in conjunction with the hardware and systems teams.

As previously, ISO 26262 gives no guidance on software integration methodologies and does not enforce any particular strategy. The standard does , however, define techniques for which types of integration test should be performed.

Table 29 – Methods for integration testing

Technique	Highly Recommended	Recommended
Requirements-based test	●	
Interface test	●	
Fault injection test	●	
Resource usage test		●
Back-to-back comparison tests between code and model (if applicable)	●	

Note – this is the same set of test types as for unit testing; this time performed at a higher level of abstraction (composition)

Similarly, methods for identifying integration test cases are the same as for unit tests:

Table 30 – Methods for deriving integration test cases

Technique	Highly Recommended	Recommended
Analysis of requirements	●	
Generation and analysis of equivalence classes	●	
Analysis of boundary values	●	
Error guessing		●

Since coverage metrics for unit testing become increasing unwieldy as the integrated system grows, ISO 26262 identifies a more appropriate set of coverage metrics for integration testing.

Table 31 – Structural coverage metrics for integration tests

Technique	Highly Recommended	Recommended
Function coverage	●	
Call coverage	●	

(The standard does not really make clear the difference between the percentage of *executed* functions and the percentage of function *calls*)

Verification of software safety requirements

The emphasis of this phase is demonstrating (verifying) that the software fulfils its safety requirements.

Software testing may largely be performed in hosted or simulated environments. Such testing cannot guarantee that the software requirements have been met. ISO 26262 establishes additional

test environments for performing tests to increase confidence that the software is operating correctly.

Table 32 – Test environments for software safety verification

Technique	Highly Recommended	Recommended
Hardware-in-the-loop	●	
Electronic control unit network environments	●	
Vehicles	●	

ASIL Level B System – Follow-to-Stop

The Follow-to-Stop system has less potential to cause significant hazard; although failure may cause it to (for example) decelerate outside its specified limits, causing injury to passengers. The Follow-to-Stop system is therefore in this example classified as a Level B system.

Initiation of software development.

The focus of this sub-phase is the planning of the software development activity, with particular emphasis on ensuring appropriate tools, methodologies, guidelines, etc. are in place.

With regard to modelling guidelines the following techniques should be covered:

Table 33 - Coding and modelling guidelines

Technique	Highly Recommended	Recommended
Enforcement of low complexity	●	
Use of language subsets	●	
Enforcement of strong typing	●	
Use of defensive implementation techniques		●
Use of established design principles		●
Use of unambiguous graphical representation	●	
Use of style guides	●	
Use of naming conventions	●	

Specification of software safety requirements

The emphasis of this sub-phase is ensuring the approach taken to ensure software safety is consistent with the system safety philosophy.

ISO 26262 provides recommendations for this sub-phase but does not specify any recommended techniques.

Software architectural design

In this sub-phase the software architecture is synthesised, evaluated and verified. ISO 26262 defines 'architectural design' as the design of the software down to the level of an implementable component. There is no guidance given on what this constitutes, but it can be assumed to be an individual module, package or class.

ISO 26262 defines techniques for design notations for the software architecture.

Table 34 - Notations for Architectural design

Technique	Highly Recommended	Recommended
Informational notations	●	
Semi-formal notations	●	
Formal notations		●

The standard also encourages high intrinsic quality at the architectural level in order to avoid (for example) failures due to over-complicated designs. The following principles are specified. Once again, the standard gives no guidance on *how* these principles should be achieved or how they should be evaluated.

Table 35 - Principles for Architectural design

Technique	Highly Recommended	Recommended
Hierarchical structure of software components	●	
Restricted size of software components	●	
Restricted size of interfaces		●
High cohesion within each software component	●	
Restricted coupling between software components	●	
Appropriate scheduling properties	●	
Restricted use of interrupts		●

One of the key safety mechanisms for software is error detection and handling. ISO 26262 specifies the use of the following techniques for error detection:

Table 36 – Mechanisms for error detection at architectural level

Technique	Highly Recommended	Recommended
Range checks of input and output data	●	
Plausibility checks		●
Detection of data errors (e.g. parity checking)		●
External monitoring facility		●
Control flow monitoring		●

Correspondingly, the standard defines acceptable error handling strategies:

Table 37 – Error handling mechanisms at architectural level

Technique	Highly Recommended	Recommended
Static recovery mechanisms		●
Graceful degradation		●
Independent parallel redundancy		
Correcting codes for data		●

Although no evaluation or validation techniques are specified the following techniques are defined for verification of the software architecture:

Table 38 – Methods for verification of software architecture

Technique	Highly Recommended	Recommended
Walk-through of the design		●
Inspection of the design	●	
Simulation of the dynamic parts of the design		●
Prototype generation		
Formal verification		
Control flow analysis	●	
Data flow analysis	●	

Software unit design and implementation

Software unit design is focused on code-level design, implementation and verification.

When specifying software unit designs ISO 26262 recommends the following techniques:

Table 39 - Notations for software unit design

Technique	Highly Recommended	Recommended
Natural language	●	
Informal notations	●	
Semi-formal notations	●	
Formal notations		●

The above is assumed to mean prefer semi-formal notation over natural language.

(ISO 26262 does not give any guidance as to what constitutes an informal, semi-formal or formal notation)

As with architectural design, ISO 26262 gives recommended principles that must be complied with. As noted in the standard, many of the principles are covered (for C) by MISRA-C.

Table 40 – Design principles for software unit design and implementation

Technique	Highly Recommended	Recommended
One entry and one exit point in functions	●	
No dynamic objects or variables, or else online test during their creation	●	
Initialization of variables	●	
No multiple use of variable names	●	
Avoid global variables or else justify their usage		●
Limited use of pointers		●
No implicit type conversions	●	
No hidden data flow or control flow	●	
No unconditional jumps	●	
No recursions		●

ISO 26262 makes a distinction (albeit not particularly explicitly) between dynamic verification and static verification. The following techniques are specified for *static* verification of the software unit design. Many of these techniques require third-party software tools.

Table 41 – Methods for verification of software unit design

Technique	Highly Recommended	Recommended
Walk-through		●
Inspection	●	
Semi-formal verification		●
Formal verification		
Control flow analysis		●
Data flow analysis		●
Static code analysis	●	
Semantic code analysis		●

Software unit testing

Software unit testing is focused on demonstrating the software units fulfil their specification (if not necessarily the requirements of the system)

In this case unit testing is taken to mean *dynamic* testing – that is, testing by executing the software in a simulated environment.

The following types of dynamic testing are specified by ISO 26262

Table 42 – Methods for software unit testing

Technique	Highly Recommended	Recommended
Requirements-based test	●	
Interface test	●	
Fault injection test		●
Resource usage test		●
Back-to-back comparison tests between code and model (if applicable)		●

The design of test cases is specified by using the following methods:

Table 43 – Methods for deriving test cases for unit tests

Technique	Highly Recommended	Recommended
Analysis of requirements	●	
Generation and analysis of equivalence classes	●	
Analysis of boundary values	●	
Error guessing		●

To evaluate the completeness of test cases and to demonstrate that there is no unintended functionality, ISO 26262 recommends structural coverage (also known as *white box*) testing. Structural coverage testing generates metrics to establish the completeness of testing. ISO 26262 specifies the following coverage metrics

Table 44 – Structural coverage metrics

Technique	Highly Recommended	Recommended
Statement coverage	●	
Branch coverage	●	
Modified Condition / Decision Coverage (MC/DC)		●

Software integration and testing

This sub-phase is focused on the construction of the software architecture from its component parts and ensuring the resulting system performs as expected.

In general software integration is done in controlled stages / iterations; these stages being planned in conjunction with the hardware and systems teams.

As previously, ISO 26262 gives no guidance on software integration methodologies and does not enforce any particular strategy. The standard does , however, define techniques for which types of integration test should be performed.

Table 45 – Methods for integration testing

Technique	Highly Recommended	Recommended
Requirements-based test	●	
Interface test	●	
Fault injection test		●
Resource usage test		●
Back-to-back comparison tests between code and model (if applicable)		●

Note – this is the same set of test types as for unit testing; this time performed at a higher level of abstraction (composition)

Similarly, methods for identifying integration test cases are the same as for unit tests:

Table 46 – Methods for deriving integration test cases

Technique	Highly Recommended	Recommended
Analysis of requirements	●	
Generation and analysis of equivalence classes	●	
Analysis of boundary values	●	
Error guessing		●

Since coverage metrics for unit testing become increasing unwieldy as the integrated system grows, ISO 26262 identifies a more appropriate set of coverage metrics for integration testing.

Table 47 – Structural coverage metrics for integration tests

Technique	Highly Recommended	Recommended
Function coverage		●
Call coverage		●

(The standard does not really make clear the difference between the percentage of *executed* functions and the percentage of function *calls*)

Verification of software safety requirements

The emphasis of this phase is demonstrating (verifying) that the software fulfils its safety requirements.

Software testing may largely be performed in hosted or simulated environments. Such testing cannot guarantee that the software requirements have been met. ISO 26262 establishes additional

test environments for performing tests to increase confidence that the software is operating correctly.

Table 48 – Test environments for software safety verification

Technique	Highly Recommended	Recommended
Hardware-in-the-loop		●
Electronic control unit network environments	●	
Vehicles	●	

ASIL Level A System – Cruise Control

Failure of the Cruise Control system may cause inconvenience or minor injury to the passengers – for example, if it fails to maintain its set speed. The severity of failure is low enough to classify this system as Level A.

Level A is roughly equivalent to good commercial embedded systems development practice.

Initiation of software development.

The focus of this sub-phase is the planning of the software development activity, with particular emphasis on ensuring appropriate tools, methodologies, guidelines, etc. are in place.

With regard to modelling guidelines the following techniques should be covered:

Table 49 - Coding and modelling guidelines

Technique	Highly Recommended	Recommended
Enforcement of low complexity	●	
Use of language subsets	●	
Enforcement of strong typing	●	
Use of defensive implementation techniques		
Use of established design principles		●
Use of unambiguous graphical representation		●
Use of style guides		●
Use of naming conventions	●	

Specification of software safety requirements

The emphasis of this sub-phase is ensuring the approach taken to ensure software safety is consistent with the system safety philosophy.

ISO 26262 provides recommendations for this sub-phase but does not specify any recommended techniques.

Software architectural design

In this sub-phase the software architecture is synthesised, evaluated and verified. ISO 26262 defines 'architectural design' as the design of the software down the level of an implementable component. There is no guidance given on what this constitutes, but it can be assumed to be an individual module, package or class.

ISO 26262 defines techniques for design notations for the software architecture.

Table 50 - Notations for Architectural design

Technique	Highly Recommended	Recommended
Informational notations	●	
Semi-formal notations		●
Formal notations		●

The standard also encourages high intrinsic quality at the architectural level in order to avoid (for example) failures due to over-complicated designs. The following principles are specified. Once again, the standard gives no guidance on *how* these principles should be achieved or how they should be evaluated.

Table 51 - Principles for Architectural design

Technique	Highly Recommended	Recommended
Hierarchical structure of software components	●	
Restricted size of software components	●	
Restricted size of interfaces		●
High cohesion within each software component		●
Restricted coupling between software components		●
Appropriate scheduling properties	●	
Restricted use of interrupts		●

One of the key safety mechanisms for software is error detection and handling. ISO 26262 specifies the use of the following techniques for error detection:

Table 52 – Mechanisms for error detection at architectural level

Technique	Highly Recommended	Recommended
Range checks of input and output data	●	
Plausibility checks		●
Detection of data errors (e.g. parity checking)		●
External monitoring facility		
Control flow monitoring		
Diverse software design		

Correspondingly, the standard defines acceptable error handling strategies:

Table 53 – Error handling mechanisms at architectural level

Technique	Highly Recommended	Recommended
Static recovery mechanisms		●
Graceful degradation		●
Independent parallel redundancy		
Correcting codes for data		●

Although no evaluation or validation techniques are specified the following techniques are defined for verification of the software architecture:

Table 54 – Methods for verification of software architecture

Technique	Highly Recommended	Recommended
Walk-through of the design	●	
Inspection of the design		●
Simulation of the dynamic parts of the design		●
Prototype generation		
Formal verification		
Control flow analysis		●
Data flow analysis		●

Software unit design and implementation

Software unit design is focused on code-level design, implementation and verification.

When specifying software unit designs ISO 26262 recommends the following techniques:

Table 55 - Notations for software unit design

Technique	Highly Recommended	Recommended
Natural language	●	
Informal notations	●	
Semi-formal notations		●
Formal notations		●

The above is assumed to mean prefer semi-formal notation over natural language.

(ISO 26262 does not give any guidance as to what constitutes an informal, semi-formal or formal notation)

As with architectural design, ISO 26262 gives recommended principles that must be complied with. As noted in the standard, many of the principles are covered (for C) by MISRA-C.

Table 56 – Design principles for software unit design and implementation

Technique	Highly Recommended	Recommended
One entry and one exit point in functions	●	
No dynamic objects or variables, or else online test during their creation		●
Initialization of variables	●	
No multiple use of variable names		●
Avoid global variables or else justify their usage		●
Limited use of pointers		
No implicit type conversions		●
No hidden data flow or control flow		●
No unconditional jumps	●	
No recursions		●

ISO 26262 makes a distinction (albeit not particularly explicitly) between dynamic verification and static verification. The following techniques are specified for *static* verification of the software unit design. Many of these techniques require third-party software tools.

Table 57 – Methods for verification of software unit design

Technique	Highly Recommended	Recommended
Walk-through	●	
Inspection		●
Semi-formal verification		●
Formal verification		
Control flow analysis		●
Data flow analysis		●
Static code analysis		●
Semantic code analysis		●

Software unit testing

Software unit testing is focused on demonstrating the software units fulfil their specification (if not necessarily the requirements of the system)

In this case unit testing is taken to mean *dynamic* testing – that is, testing by executing the software in a simulated environment.

The following types of dynamic testing are specified by ISO 26262

Table 58 – Methods for software unit testing

Technique	Highly Recommended	Recommended
Requirements-based test	●	
Interface test	●	
Fault injection test		●
Resource usage test		●
Back-to-back comparison tests between code and model (if applicable)		●

The design of test cases is specified by using the following methods:

Table 59 – Methods for deriving test cases for unit tests

Technique	Highly Recommended	Recommended
Analysis of requirements	●	
Generation and analysis of equivalence classes		●
Analysis of boundary values		●
Error guessing		●

To evaluate the completeness of test cases and to demonstrate that there is no unintended functionality, ISO 26262 recommends structural coverage (also known as *white box*) testing. Structural coverage testing generates metrics to establish the completeness of testing. ISO 26262 specifies the following coverage metrics

Table 60 – Structural coverage metrics

Technique	Highly Recommended	Recommended
Statement coverage	●	
Branch coverage		●
Modified Condition / Decision Coverage (MC/DC)		●

Software integration and testing

This sub-phase is focused on the construction of the software architecture from its component parts and ensuring the resulting system performs as expected.

In general software integration is done in controlled stages / iterations; these stages being planned in conjunction with the hardware and systems teams.

As previously, ISO 26262 gives no guidance on software integration methodologies and does not enforce any particular strategy. The standard does , however, define techniques for which types of integration test should be performed.

Table 61 – Methods for integration testing

Technique	Highly Recommended	Recommended
Requirements-based test	●	
Interface test	●	
Fault injection test		●
Resource usage test		●
Back-to-back comparison tests between code and model (if applicable)		●

Note – this is the same set of test types as for unit testing; this time performed at a higher level of abstraction (composition)

Similarly, methods for identifying integration test cases are the same as for unit tests:

Table 62 – Methods for deriving integration test cases

Technique	Highly Recommended	Recommended
Analysis of requirements	●	
Generation and analysis of equivalence classes		●
Analysis of boundary values		●
Error guessing		●

Since coverage metrics for unit testing become increasing unwieldy as the integrated system grows, ISO 26262 identifies a more appropriate set of coverage metrics for integration testing.

Table 63 – Structural coverage metrics for integration tests

Technique	Highly Recommended	Recommended
Function coverage		●
Call coverage		●

(The standard does not really make clear the difference between the percentage of *executed* functions and the percentage of function *calls*)

Verification of software safety requirements

The emphasis of this phase is demonstrating (verifying) that the software fulfils its safety requirements.

Software testing may largely be performed in hosted or simulated environments. Such testing cannot guarantee that the software requirements have been met. ISO 26262 establishes additional

test environments for performing tests to increase confidence that the software is operating correctly.

Table 64 – Test environments for software safety verification

Technique	Highly Recommended	Recommended
Hardware-in-the-loop		●
Electronic control unit network environments	●	
Vehicles	●	

Feabhas Ltd
5 Lowesden Works
Lambourn Woodlands
Hungerford
RG17 7RY
T: +44 1488 73050
W :www.feabhas.com
E : info@feabhas.com