**DRAFT INTERNATIONAL STANDARD** ISO/DIS 26262-6

ISO/TC **22**/SC **3** Secretariat: **DIN**

Voting begins on: **2009-07-08**

Voting terminates on: **2009-12-08**

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION • МЕЖДУНАРОДНАЯ ОРГАНИЗАЦИЯ ПО СТАНДАРТИЗАЦИИ • ORGANISATION INTERNATIONALE DE NORMALISATION

# Road vehicles — Functional safety —

## Part 6: **Product development: software level**

*Véhicules routiers — Sécurité fonctionnelle —*

*Partie 6: Développement du produit: niveau logiciel*

ICS 43.040.10

**In accordance with the provisions of Council Resolution 15/1993 this document is circulated in the English language only.**

**Conformément aux dispositions de la Résolution du Conseil 15/1993, ce document est distribué en version anglaise seulement.**

**To expedite distribution, this document is circulated as received from the committee secretariat. ISO Central Secretariat work of editing and text composition will be undertaken at publication stage.**

**Pour accélérer la distribution, le présent document est distribué tel qu'il est parvenu du secrétariat du comité. Le travail de rédaction et de composition de texte sera effectué au Secrétariat central de l'ISO au stade de publication.**

# Contents

Page





# Foreword

ISO (the International Organization for Standardization) is a worldwide federation of national standards bodies (ISO member bodies). The work of preparing International Standards is normally carried out through ISO technical committees. Each member body interested in a subject for which a technical committee has been established has the right to be represented on that committee. International organizations, governmental and non-governmental, in liaison with ISO, also take part in the work. ISO collaborates closely with the International Electrotechnical Commission (IEC) on all matters of electrotechnical standardization.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of technical committees is to prepare International Standards. Draft International Standards adopted by the technical committees are circulated to the member bodies for voting. Publication as an International Standard requires approval by at least 75 % of the member bodies casting a vote.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO shall not be held responsible for identifying any or all such patent rights.

ISO 26262-6 was prepared by Technical Committee ISO/TC 22, *Road vehicles*, Subcommittee SC 3, *Electrical and electronic equipment*.

ISO 26262 consists of the following parts, under the general title *Road vehicles — Functional safety*:

— *Part 1: Vocabulary*

— *Part 2: Management of functional safety*

— *Part 3: Concept phase*

— *Part 4: Product development: system level*

— *Part 5: Product development: hardware level*

— *Part 6: Product development: software level*

— *Part 7: Production and operation*

— *Part 8: Supporting processes*

— *Part 9: ASIL-oriented and safety-oriented analyses*

— *Part 10: Guideline on ISO 26262*

# Introduction

ISO 26262 is the adaptation of IEC 61508 to comply with needs specific to the application sector of E/E systems within road vehicles.

This adaptation applies to all activities during the safety lifecycle of safety-related systems comprised of electrical, electronic, and software elements that provide safety-related functions.

Safety is one of the key issues of future automobile development. New functionality not only in the area of driver assistance but also in vehicle dynamics control and active and passive safety systems increasingly touches the domain of safety engineering. Future development and integration of these functionalities will even strengthen the need of safe system development processes and the possibility to provide evidence that all reasonable safety objectives are satisfied.

With the trend of increasing complexity, software content and mechatronic implementation, there are increasing risks from systematic failures and random hardware failures. ISO 26262 includes guidance to avoid these risks by providing feasible requirements and processes.

System safety is achieved through a number of safety measures, which are implemented in a variety of technologies (for example: mechanical, hydraulic, pneumatic, electrical, electronic, programmable electronic etc). Although ISO 26262 is concerned with E/E systems, it provides a framework within which safety-related systems based on other technologies can be considered.

ISO 26262:

— provides an automotive safety lifecycle (management, development, production, operation, service, decommissioning) and supports tailoring the necessary activities during these lifecycle phases;

— provides an automotive specific risk-based approach for determining risk classes (Automotive Safety Integrity Levels, ASILs);

— uses ASILs for specifying the item's necessary safety requirements for achieving an acceptable residual risk; and

— provides requirements for validation and confirmation measures to ensure a sufficient and acceptable level of safety being achieved.

Functional safety is influenced by the development process (including such activities as requirements specification, design, implementation, integration, verification, validation and configuration), the production and service processes and by the management processes.

Safety issues are intertwined with common function-oriented and quality-oriented development activities and work products. ISO 26262 addresses the safety-related aspects of the development activities and work products.

Figure 1 shows the overall structure of ISO 26262. ISO 26262 is based upon a V-Model as a reference process model for the different phases of product development. The shaded "V"s represents the relations between ISO 26262-3, ISO 26262-4, ISO 26262-5, ISO 26262-6 and ISO 26262-7.



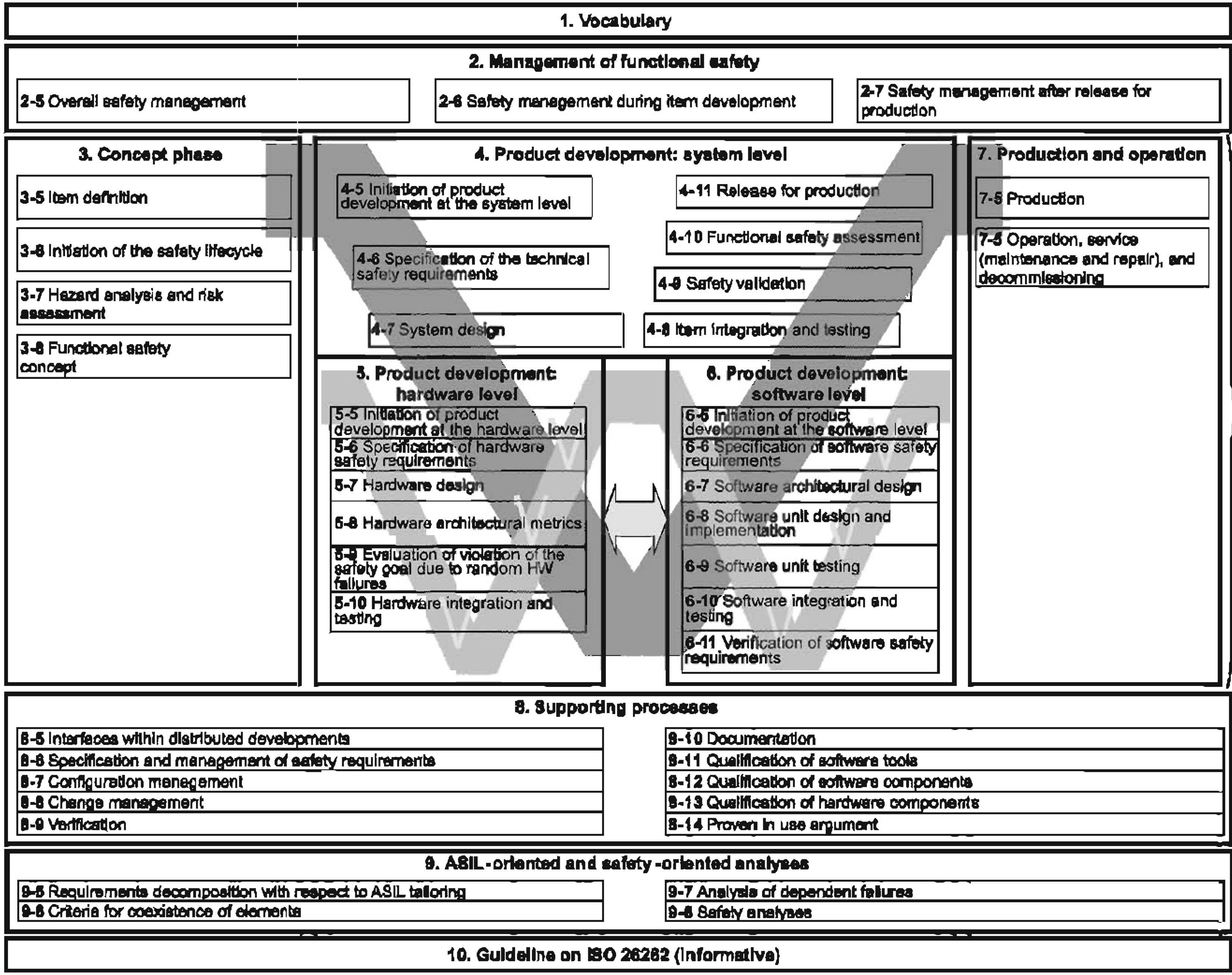



Figure 1 — Overview of ISO 26262

# Road vehicles — Functional safety — Part 6: Product development: software level

## 1 Scope

ISO 26262 is intended to be applied to safety-related systems that include one or more E/E systems and that are installed in series production passenger cars with a max gross weight up to 3,5 t. ISO 26262 does not address unique E/E systems in special purpose vehicles such as vehicles designed for drivers with disabilities. Systems developed prior to the publication date of ISO 26262 are exempted from the scope.

ISO 26262 addresses possible hazards caused by malfunctioning behaviour of E/E safety-related systems including interaction of these systems. It does not address hazards as electric shock, fire, smoke, heat, radiation, toxicity, flammability, reactivity, corrosion, release of energy, and similar hazards unless directly caused by malfunctioning behaviour of E/E safety-related systems.

ISO 26262 does not address the nominal performance of E/E systems, even if dedicated functional performance standards exist for these systems (for example active and passive safety systems, brake systems, ACC).

This Part of ISO 26262 specifies the requirements for product development at the software level for automotive applications. This includes requirements for initiation of product development at the software level, specification of software safety requirements, software architectural design, software unit design and implementation, software unit testing, software integration and testing, and verification of software safety requirements.

## 2 Normative references

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO 26262-1:—[1], *Road vehicles — Functional Safety — Part 1: Vocabulary*

ISO 26262-2:—[1], *Road vehicles — Functional Safety — Part 2: Management of functional safety*

ISO 26262-4:—[1], *Road vehicles — Functional Safety — Part 4: Product development: system level*

ISO 26262-5:—[1], *Road vehicles — Functional Safety — Part 5: Product development: hardware level*

ISO 26262-8:—[1], *Road vehicles — Functional Safety — Part 8: Supporting processes*

ISO 26262-9:—[1], *Road vehicles — Functional Safety — Part 9: ASIL-oriented and safety-oriented analyses*

[1] To be published

# 3 Terms, definitions, abbreviated terms

For the purposes of this document, the terms, definitions and abbreviated terms given in ISO 26262-1 apply.

# 4 Requirements for compliance

## 4.1 General requirements

When claiming compliance with ISO 26262, each requirement shall be complied with, unless one of the following applies:

1) Tailoring in accordance with ISO 26262-2 has been planned and shows that the requirement does not apply.

2) A rationale is available that the non-compliance is acceptable and the rationale has been assessed in accordance with ISO 26262-2.

Information marked as a "NOTE" is only for guidance in understanding, or for clarification of, the associated requirement and shall not be interpreted as a requirement itself.

## 4.2 Interpretations of tables

Tables may be normative or informative depending on their context.

The different methods listed in a table contribute to the level of confidence that the corresponding requirement shall apply.

Each method in a table is either a consecutive entry (marked by a sequence number in the leftmost column, e.g. 1, 2, 3) or an alternative entry (marked by a number followed by a letter in leftmost column, e.g., 2a, 2b, 2c).

For consecutive entries all methods are recommended in accordance with the ASIL. If methods other than those listed are to be applied a rationale shall be given that they comply with the corresponding requirement.

For alternative entries an appropriate combination of methods shall be applied in accordance with the ASIL, independently of whether they are listed in the table or not. If methods are listed with different degrees of recommendation for an ASIL the higher one should be preferred. A rationale shall be given that the selected combination of methods complies with the corresponding requirement. If all highly recommended methods listed for a particular ASIL are selected a rationale needs not to be given.

For each method, the degree of recommendation to use the corresponding method depends on the ASIL and is categorized as follows:

”++” The method is highly recommended for this ASIL.

“+“ The method is recommended for this ASIL.

“o“ The method has no recommendation for or against its usage for this ASIL.

## 4.3 ASIL dependent requirements and recommendations

The requirements or recommendations of each subclause shall apply to ASIL A, B, C and D, if not stated otherwise. These requirements and recommendations refer to the ASIL of the safety goal. If ASIL decomposition has been performed at an earlier stage of development in accordance with ISO 26262-9—: Clause 5 the ASIL resulting from the decomposition will apply.



If an ASIL is given in parentheses, the corresponding subclause shall be read as a recommendation rather than a requirement for this ASIL.

# 5 Initiation of product development at the software level

## 5.1 Objectives

The objective of this subphase is to plan and initiate the functional safety activities for the subphases of the software development.

## 5.2 General

The initiation of the software development is a planning activity, where software development subphases and their supporting processes (see ISO 26262-8 and ISO 26262-9) are determined and planned according to the extent and complexity of the item development. The software development subphases and supporting processes are initiated by determining the appropriate methods in order to achieve the requirements of the assigned ASIL. The methods are supported by guidelines and tools, which also have to be determined and planned for each subphase and supporting process.

The planning of the software development includes the coordination with the product development at hardware level (see ISO 26262-5).

## 5.3 Inputs to this clause

### 5.3.1 Prerequisites

The following information shall be available:

— Overall project plan (refined) (see ISO 26262-4:—, 5.5.1)

— Safety plan (refined) (see ISO 26262-4:—, 5.5.2)

— Item integration and testing plan (refined) (see ISO 26262-4:—, 7.5.4)

— Technical safety concept (see ISO 26262-4:—, 7.5.1)

— System design specification (see ISO 26262-4:—, 7.5.2)

### 5.3.2 Further supporting information

The following information may be considered:

— Design and coding guidelines for modelling and programming languages (from external source)

— Guidelines for the application of methods (from external source)

— Guidelines for the application of tools (from external source)

— Qualified software components available (see ISO 26262-8:—, Clause 12)

## 5.4 Requirements and recommendations

**5.4.1** The activities for the product development at the software level shall be planned, including determination of appropriate methods during design.



NOTE The overall project plan (refined) and safety plan (refined) (see ISO 26262-4 —, Clause 5) are to be updated and maintained

**5.4.2 The tailoring of the lifecycle for product development at the software level shall be performed in accordance with ISO 26262-2:—, Clause 6.4.3.4, and based on the reference phase model given in Figure 2.**

**5.4.3 When developing configurable software, Annex C shall be applied.**

**5.4.4 The software development process for the software of an item, including lifecycle phases, methods, languages and tools, shall be consistent across all the subphases of the software lifecycle and be compatible with the system and hardware lifecycles, such that the required data can be transformed correctly.**

NOTE The sequencing of phases, tasks and activities, including iteration steps, for the software of an item is to ensure the consistency of the corresponding work products with product development at the hardware level (see ISO 26262-5) and product development at the system level (see ISO 26262-4)

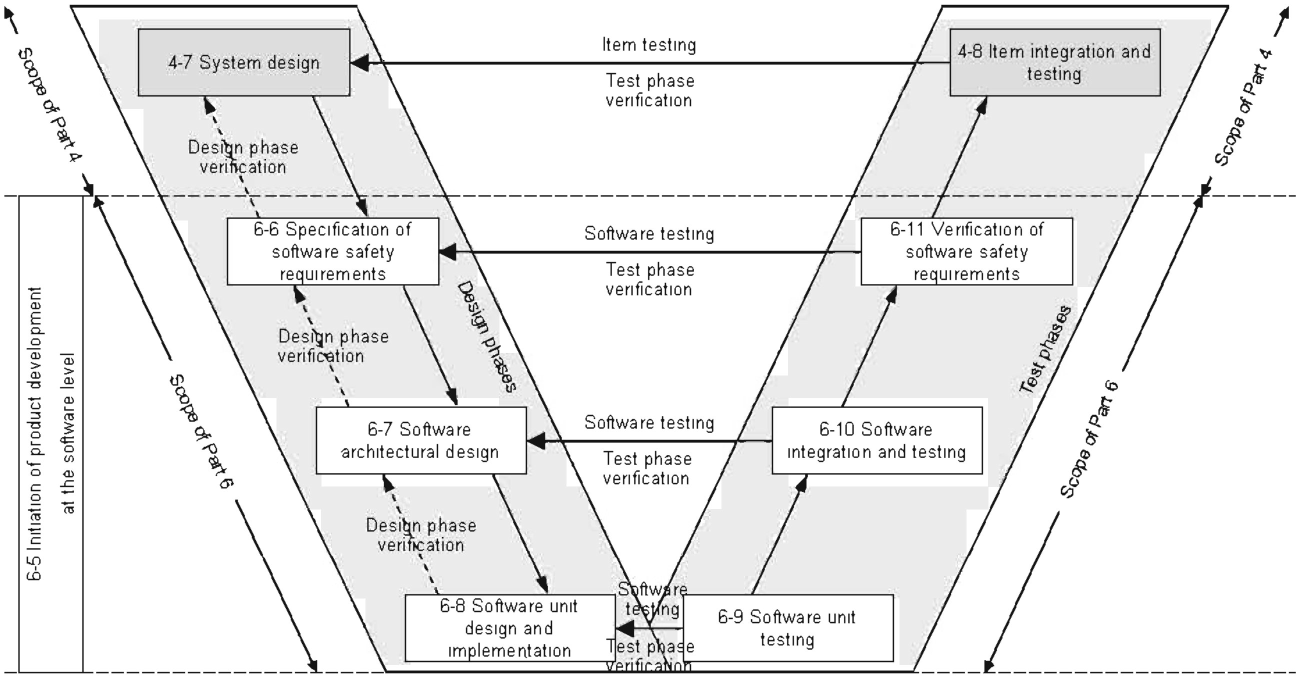



**Figure 2 — Reference phase model for the software development**

**5.4.5 For each subphase of software development, the selection of**

**a) methods; and**

**b) corresponding software tools,**

**including guidelines for their application, shall be carried out.**

**5.4.6 To support the correctness of the design and implementation, the design and coding guidelines for the modelling, or programming languages, shall address the topics listed in Table 1.**

NOTE 1 Coding guidelines are usually different for different programming languages

NOTE 2 Coding guidelines can be different for model-based development

NOTE 3 Existing coding guidelines might have to be modified for a specific item development

EXAMPLE MISRA C (see [MISRA C]) and MISRA AC AGC (see [MISRA AGC]) are coding guidelines for the programming language C.

**Table 1 — Topics to be covered by modelling and coding guidelines**

| Topics | | ASIL | | | |
|---|---|---|---|---|---|
| | | A | B | C | D |
| 1a | Enforcement of low complexity | ++ | ++ | ++ | ++ |
| 1b | Use of language subsets[b] | ++ | ++ | ++ | ++ |
| 1c | Enforcement of strong typing[c] | ++ | ++ | ++ | ++ |
| 1d | Use of defensive implementation techniques | o | + | ++ | ++ |
| 1e | Use of established design principles | + | + | + | ++ |
| 1f | Use of unambiguous graphical representation | + | ++ | ++ | ++ |
| 1g | Use of style guides | + | ++ | ++ | ++ |
| 1h | Use of naming conventions | ++ | ++ | ++ | ++ |

[a] An appropriate compromise of this method with other methods in ISO 26262-6 may be required.

[b] The objectives of method 1b are

— Exclusion of ambiguously defined language constructs which might be interpreted differently by different modellers, programmers, code generators or compilers.
— Exclusion of language constructs which from experience easily lead to mistakes, for example assignments in conditions or identical naming of local and global variables.
— Exclusion of language constructs which might result in unhandled run-time errors.

[c] The objective of method 1c is to impose principles of strong typing where these are not inherent in the language.

**5.4.7** For software tools ISO 26262-8:—, Clause 11 shall be applied.

**5.4.8** The criteria to be considered when selecting a suitable modelling or programming language are:

a) an unambiguous definition;

EXAMPLE Syntax and semantics of the language

b) the support for embedded real time software and runtime error handling; and

c) the support for modularity, abstraction and structured constructs.

Criteria that are not sufficiently addressed by the language itself shall be covered by the corresponding guidelines, or by the development environment.

NOTE 1 The selected programming language (such as ADA, C, C++, Java, Assembler or a graphical modelling language) is to fulfil the criteria given in 5.4.6. Usually programming or modelling guidelines are necessary to fulfil these criteria.

NOTE 2 Assembly languages can only be used for those parts of the software where the use of high-level programming languages is not appropriate. Examples are low-level software with interfaces to the hardware, interrupt handlers and time-critical algorithms.

## 5.5 Work products

**5.5.1 Safety plan (refined)** resulting from requirements 5.4.1, 5.4.2, 5.4.3, 5.4.4, 5.4.5, 5.4.6, 5.4.7 and 5.4.8.

**5.5.2 Software verification plan** resulting from requirement 5.4.1, 5.4.2, 5.4.3, 5.4.4, 5.4.5, 5.4.6 and 5.4.7.

**5.5.3 Design and coding guidelines for modelling and programming languages** resulting from requirements 5.4.6 and 5.4.8.

**5.5.4 Software tool application guidelines** resulting from requirements 5.4.5, 5.4.7 and 5.4.8.

# 6 Specification of software safety requirements

## 6.1 Objectives

The first objective of this subphase is to specify the software safety requirements. They are derived from the technical safety concept and the system design specification.

The second objective is to detail the hardware-software interface requirements initiated in ISO 26262-4:—, Clause 6.

The third objective is to verify that the software safety requirements are consistent with the technical safety concept and the system design specification.

## 6.2 General

The technical safety requirements are divided into hardware and software safety requirements. The specification of the software safety requirements considers constraints of the hardware and the impact of these constraints on the software.

## 6.3 Inputs to this clause

### 6.3.1 Prerequisites

The following information shall be available:

— Technical safety concept (see ISO 26262-4:—, 7.5.1)

— System design specification (see ISO 26262-4:—, 7.5.2)

— Safety plan (refined) (see 5.5.1)

— Hardware-software interface specification (see ISO 26262-4:—, 7.5.6)

— Software verification plan (see 5.5.2)

### 6.3.2 Further supporting information

The following information may be considered:

— Hardware design specification (see ISO 26262-5:—, 7.5.1)

— Design and coding guidelines for modelling and programming languages (see 5.5.3)

— Software tool application guidelines (see 5.5.4)

— Guidelines for the application of methods (from external source)

— Qualified software components available (see ISO 26262-8:—, Clause 12)

## 6.4 Requirements and recommendations

**6.4.1** The software safety requirements shall address each software-based function whose failure could lead to a violation of a technical safety requirement allocated to software.

NOTE 1 Functions whose failure could lead to a violation of a safety requirement include:

— functions that enable the system to achieve or maintain a safe state;

— functions related to the detection, indication and handling of faults of safety-related hardware elements;

— functions related to the detection, notification and mitigation of faults in the software itself;

These include both the self-monitoring of the software in the operating system and application-specific self-monitoring of the software to detect, indicate and handle systematic faults in the application software.

— functions related to on-board and off-board tests;

On-board tests can be carried out by the system itself or through other systems within the vehicle network during operation and during the pre-run and post-run phase of the vehicle.

Off-board tests refer to the testing of the safety-related functions or properties during production or in service.

— functions that allow modifications of the software during production and service;

— functions with interfaces or interactions with non-safety-related functions;

— functions related to performance or time-critical operations; and

— functions with interfaces or interactions between software and hardware elements

Interfaces can also include those required for programming, installation, setup, configuration, calibration, initialisation, start-up, shut down, and other activities occurring in modes other than the "ready to operate" mode of the embedded software.

**6.4.2** The specification of the software safety requirements shall be derived from the technical safety requirements and the system design (see ISO 26262-4:—, 7.4.1 and ISO 26262-4:—, 7.4.5) and shall consider:

a) the overall management of safety requirements (see ISO 26262-8:—, Clause 6)

b) the system and hardware configuration;

EXAMPLE 1 Configuration parameters include gain control, band pass frequency, and clock prescaler.

c) the hardware-software interface specification;

d) the hardware safety requirements and hardware architecture;

e) the timing constraints;

EXAMPLE 2 Execution or reaction time derived from required response time at the system level.

f) the external interfaces; and

NOTE 1 Examples include communication and user interfaces.

g) each operating mode of the vehicle, the system or the hardware, having an impact on the software.

EXAMPLE 3 Operating modes of hardware devices can include default, initialization, test, and advanced modes.

NOTE 2 The process of software requirements refinement and architecture development can lead to risk mitigation opportunities through changes in hardware specifications. Therefore, close communication and cooperation between the software and hardware development are necessary. Change requests that occur are handled in the system design subphase (see ISO 26262-4:—, Clause 7).

**6.4.3** The hardware-software interface specification initiated in ISO 26262-4:— Clause 7 shall be detailed down to a level allowing the correct control and usage of hardware, and shall describe each safety-related dependency between hardware and software.

**6.4.4** If other functions in addition to those functions for which safety requirements are specified in 6.4.1 are carried out by the embedded software, these functions shall be specified, or else a reference made to their specification.

**6.4.5** The software safety requirements shall include sufficient information to enable:

a) the software design and subsequent development activities to be performed effectively;

EXAMPLE Shared and exclusive use of hardware resources (including memory mapping, allocation of registers, timers, interrupts, I/O ports)

b) the software verification and the safety validation of software aspects to be performed effectively; and

c) functional safety to be assessed effectively.

**6.4.6** The verification of the software safety requirements, and the refined specification of the hardware software interface shall be planned and specified in accordance with ISO 26262-8:—, Clause 9.

**6.4.7** The persons responsible for the hardware and software development shall jointly verify the refined hardware software interface specification.

**6.4.8** The verification methods listed in Table 2 shall be applied to demonstrate that the software safety requirements are:

a) compliant with the technical safety requirements;

b) compliant with the system design; and

c) consistent with the relevant parts of the hardware safety requirements.

**Table 2 — Methods for the verification of requirements**

| Methods | | ASIL | | | |
|---|---|---|---|---|---|
| | | A | B | C | D |
| 1a | Informal verification by walkthrough | ++ | + | o | o |
| 1b | Informal verification by inspection | + | ++ | ++ | ++ |
| 1c | Semi-formal verification[a] | + | + | ++ | ++ |
| 1d | Formal verification | o | + | + | + |

[a] Method 1c can be supported by executable models.



## 6.5 Work products

**6.5.1 Software safety requirements specification** resulting from requirements 6.4.1, 6.4.2, 6.4.4 and 6.4.5.

**6.5.2 Hardware-software interface specification (refined)** resulting from requirement 6.4.3.

**6.5.3 Software verification plan (refined)** resulting from requirement 6.4.6.

**6.5.4 Software verification report** resulting from requirements 6.4.7 and 6.4.8.

# 7 Software architectural design

## 7.1 Objectives

The first objective of this subphase is to develop a software architectural design that realises the software safety requirements.

The second objective of this subphase is to verify the software architectural design.

## 7.2 General

The software architectural design represents all software components and their interactions with one another in a hierarchical structure. Static aspects, such as interfaces and data paths of all software components, as well as dynamic aspects, such as process sequences and timing behaviour, need to be described.

In order to develop a single software architectural design both software safety requirements as well as all non-safety-related requirements have to be fulfilled. Hence in this subphase safety-related and non-safety-related requirements are handled within one development process.

The software architectural design has to provide the means to implement the software safety requirements and to manage the complexity of the technical safety concept.

## 7.3 Inputs to this clause

### 7.3.1 Prerequisites

The following information shall be available:

— Software safety requirements specification (see 6.5.1)

— Safety plan (refined) (see 5.5.1)

— Software verification plan (refined) (see 6.5.3)

— Software verification report (refined) (see 6.5.4)

### 7.3.2 Further supporting information

The following information may be considered:

— Technical safety concept (see ISO 26262-4:—, 7.5.1)

— System design specification (see ISO 26262-4:—, 7.5.2)

— Design and coding guidelines for modelling and programming languages (see 5.5.3)

- Guidelines for the application of methods (from external source)
- Software tool application guidelines (see 5.5.4)
- Qualified software components available (see ISO 26262-8:—, Clause 12)

## 7.4 Requirements and recommendations

**7.4.1** To ensure that the software architectural design captures the information necessary to allow the subsequent development activities to be performed correctly and effectively, the software architectural design shall be described with appropriate levels of abstraction by using the notations for software architectural design listed in Table 3.

**Table 3 — Notations for software architectural design**

| Methods | | ASIL | | | |
|---|---|---|---|---|---|
| | | A | B | C | D |
| 1a | Informal notations | ++ | ++ | + | + |
| 1b | Semi-formal notations | + | ++ | ++ | ++ |
| 1c | Formal notations | + | + | + | + |

**7.4.2** During the development of the software architectural design the following shall be considered:

a) the verifiability of the software architectural design;

NOTE This implies bi-directional traceability.

b) the suitability for configurable software;

c) the feasibility for the design and implementation of the software units;

d) the testability of the software architecture during software integration testing; and

e) the maintainability.

**7.4.3** The software architectural design shall exhibit the following properties by use of the principles listed in Table 4:

a) modularity;

b) encapsulation and;

c) minimum complexity.



**Table 4 — Principles for software architectural design**

| Methods | | ASIL | | | |
|---|---|---|---|---|---|
| | | A | B | C | D |
| 1a | Hierarchical structure of software components | ++ | ++ | ++ | ++ |
| 1b | Restricted size of software components[a] | ++ | ++ | ++ | ++ |
| 1c | Restricted size of interfaces[a] | + | + | + | + |
| 1d | High cohesion within each software component[b] | + | ++ | ++ | ++ |
| 1e | Restricted coupling between software components[a, b, c] | + | ++ | ++ | ++ |
| 1f | Appropriate scheduling properties | ++ | ++ | ++ | ++ |
| 1g | Restricted use of interrupts[a, d] | + | + | + | ++ |

[a] In methods 1b, 1c, 1e and 1g "restricted" means to minimise in balance with other design considerations.

[b] Methods 1d and 1e can, for example, be achieved by separation of concerns which refers to the ability to identify, encapsulate, and manipulate those parts of software that are relevant to a particular concept, goal, task, or purpose.

[c] Method 1e addresses the limitation of the external coupling of software components.

[d] Any interrupts used have to be priority-based.

NOTE An appropriate compromise of the methods listed in Table 4 is to be selected since the methods are not necessarily mutually exclusive.

**7.4.4** The software architectural design shall be developed down to the level where the software units, which are to be treated as indivisible, are identified.

**7.4.5** The software architectural design shall describe:

a) the static design aspects of the software components; and

NOTE 1 Static design aspects address:

- the software structure including its hierarchical levels;
- the logical sequence of data processing;
- the data types and their characteristics;
- the interfaces of the software components;
- the external interfaces of the software; and
- the constraints including scope of architecture and external dependencies.

NOTE 2 In the case of model-based development, modelling the structure is an inherent part of the overall modelling activities.

b) the dynamic design aspects of the software components.

NOTE 3 Dynamic design aspects address:

- the functionality and behaviour;
- the control flow and concurrency of processes;
- the data flow between the software components;

— the data flow at external interfaces; and

— the temporal constraints.

NOTE 4 To determine the dynamic behaviour (e.g. of tasks, time slices and interrupts) the different operating states (e.g. power up, shut down, normal operation, calibration and diagnosis) are considered.

NOTE 5 To describe the dynamic behaviour (e.g. of tasks, time slices and interrupts) the communication relationships and their allocation to the system hardware (e.g. CPU and communication channels) are specified.

**7.4.6** Every safety-related software component shall be categorised as one of the following:

a) newly developed;

b) reused with modifications;

c) reused without modifications; or

d) a COTS product.

**7.4.7** Safety-related software components that are newly developed or reused with modifications shall be developed in accordance with ISO 26262:—.

**7.4.8** Safety-related software components that are reused without modifications or that are COTS products shall be qualified in accordance with ISO 26262-8:—, Clause 12.

NOTE The use of qualified software components does not affect the applicability of Clauses 10 and 11. However, some activities described in Clauses 8 and 9 may be omitted.

**7.4.9** The software safety requirements shall be allocated to the software components. As a result, each software component inherits the highest ASIL of any of the requirements allocated to it.

NOTE Following this allocation, further refinement of the software safety requirements can be necessary.

**7.4.10** If the embedded software has to implement software components of different ASILs, or safety-related and non-safety-related software components, then all of the embedded software shall be treated in accordance with the highest ASIL, unless the software components meet the criteria for coexistence in accordance with ISO 26262-9:—, Clause 6.

**7.4.11** If software partitioning (see Annex D) is used to implement freedom from interference between software components it shall be ensured that:

a) the shared resources are used in such a way that freedom from interference of software partitions is ensured.

NOTE 1 Tasks within a software partition are not free from interference among each other.

NOTE 2 One software partition cannot change the code or data of another software partition nor command non-shared resources of other software partitions.

NOTE 3 The service received from shared resources by one software partition cannot be affected by another software partition. This includes the performance of the resources concerned, as well as the rate, latency, jitter and duration of scheduled access to the resource.

b) the software partitioning is supported by dedicated hardware features or equivalent means. This subclause applies to ASILs (A), (B), (C) and D, in accordance with Clause 4.3.

c) the part of the software that implements the software partitioning shall have the same or higher ASIL than the highest ASIL associated with the software partitions.



NOTE 4 In general the operating system provides or supports software partitioning.

d) The correct verification of the software partitions during software integration and testing (see Clause 10) is performed.

**7.4.12** If ASIL decomposition is applied to the software architectural design, it shall be applied in accordance with ISO 26262-9:—, Clause 5.

**7.4.13** Safety analysis (see ISO 26262-9:—, Clause 8) shall be carried out to

a) identify or confirm the safety-related characteristics of software components; and

b) support specification of the safety mechanisms.

**7.4.14** An analysis of dependent failures (see ISO 26262-9:—, Clause 7) shall be carried out whenever the implementation of software safety requirements relies on freedom from interference between software components.

**7.4.15** To specify the necessary software safety mechanisms at the software architectural level, based on the results of the safety analysis carried out in Clause 7.4.13, the mechanisms for error detection listed in Table 5 and the mechanisms for error handling listed in Table 6 shall be applied.

NOTE 1 When not directly required by technical safety requirements allocated to software, the use of software safety mechanisms is reviewed at the system level to analyse the potential impact on the system behaviour.

NOTE 2 The analysis at software architectural level of possible hazards due to hardware is described in ISO 26262-5:—.

**Table 5 — Mechanisms for error detection at the software architectural level**

| Methods | | ASIL | | | |
|---|---|---|---|---|---|
| | | A | B | C | D |
| 1a | Plausibility check[a] | ++ | ++ | ++ | ++ |
| 1b | Detection of data errors[b] | + | + | + | + |
| 1c | External monitoring facility | o | + | + | ++ |
| 1d | Control flow monitoring | o | + | ++ | ++ |
| 1e | Diverse software design[c] | o | o | + | ++ |

[a] Plausibility checks include assertion checks. Complex plausibility checks can be realised by using a reference model of the desired behaviour.

[b] Types of methods that may be used to detect data errors include error detecting codes and multiple data storage.

[c] Diverse software design is not intended to imply n-version programming.

**Table 6 — Mechanisms for error handling at the software architectural level**

| Methods | | ASIL | | | |
|---|---|---|---|---|---|
| | | A | B | C | D |
| 1a | Static recovery mechanism[a] | + | + | + | + |
| 1b | Graceful degradation[b] | + | + | ++ | ++ |
| 1c | Independent parallel redundancy[c] | o | o | + | ++ |
| 1d | Correcting codes for data | + | + | + | + |

[a] Static recovery mechanisms can be realised by recovery blocks, backward recovery, forward recovery and recovery through repetition.

[b] Graceful degradation at the software level refers to prioritising functions to minimise the adverse effects of potential failures on functional safety.

[c] For parallel redundancy to be independent there has to be dissimilar software in each parallel path.

**7.4.16** If hazards not already known and reflected in a safety goal are found during this subphase, they shall be introduced and evaluated in the hazard analysis and risk assessment in accordance with the change management process in ISO 26262-8:—, Clause 8.

**7.4.17** An upper estimation of required resources for the embedded software shall be made, including:

a) the execution time;

b) the storage space; and

EXAMPLE RAM for stacks and heaps, ROM for program and non-volatile data.

c) the communication resources.

**7.4.18** The software architectural design shall be verified in accordance with ISO 26262-8:—, Clause 9 and by using the software architectural design verification methods listed in Table 7 to demonstrate the following properties:

a) compliance with the software safety requirements;

b) compatibility with the target hardware; and

NOTE This includes the resources as specified in Clause 7.4.17.

c) adherence to design guidelines.

**Table 7 — Methods for the verification of the software architectural design**

| Methods | | ASIL | | | |
|---|---|---|---|---|---|
| | | A | B | C | D |
| 1a | Informal verification by walkthrough of the design[a] | ++ | + | o | o |
| 1b | Informal verification by inspection of the design[a] | + | ++ | ++ | ++ |
| 1c | Semi-formal verification by simulating dynamic parts of the design[b] | + | + | + | + |
| 1d | Semi-formal verification by prototype generation / animation | o | o | + | + |
| 1e | Formal verification | o | o | + | + |
| 1f | Control flow analysis[c, d] | + | + | ++ | ++ |
| 1g | Data flow analysis[c, d] | + | + | ++ | ++ |

[a] Informal verification is used to assess whether the software requirements are completely and correctly refined and realised in the software architectural design. In the case of model-based development this method can be applied to the model.

[b] Method 1c requires the usage of executable models for the dynamic parts of the software architecture.

[c] Control and data flow analysis can be carried out informally, semi-formally or formally.

[d] Control and data flow analysis may be limited to safety-related components and their interfaces.

## 7.5 Work products

**7.5.1 Software architectural design specification** as a result of requirements 7.4.1, 7.4.2, 7.4.3, 7.4.4, 7.4.5, 7.4.6, 7.4.9, 7.4.10, 7.4.15 and 7.4.17.

**7.5.2 Safety plan (refined)** as a result of requirement 7.4.7.

**7.5.3 Software safety requirements specification (refined)** as a result of requirement 7.4.9.

**7.5.4 Safety analysis report** as a result of requirement 7.4.13.

**7.5.5 Dependent failures analysis report** as a result of requirement 7.4.14.

**7.5.6 Software verification report (refined)** as a result of requirement 7.4.18.

# 8 Software unit design and implementation

## 8.1 Objectives

The first objective of this subphase is to specify the software units in accordance with the software architectural design and the associated software safety requirements.

The second objective of this subphase is to implement the software units as specified.

The third objective of this subphase is to verify the design of the software units and their implementation.

## 8.2 General

Based on the software architectural design, the detailed design of the software units is developed. The detailed design will be implemented as a model or directly as source code, in accordance with the modelling or coding guidelines. The detailed design and the implementation are verified before proceeding to the software unit testing phase.

The implementation of the software units includes the generation of source code and the translation into object code.

## 8.3 Inputs to this clause

### 8.3.1 Prerequisites

The following information shall be available:

- Safety plan (refined) (see 7.5.2)
- Software verification plan (refined) (see 6.5.3)
- Software architectural design specification (see 7.5.1)
- Software safety requirements specification (refined) (see 7.5.3)
- Software verification report (refined) (see 7.5.6)

### 8.3.2 Further supporting information

The following information may be considered:

- Technical safety concept (see ISO 26262-4:—, 7.5.1)
- System design specification (see ISO 26262-4:—, 7.5.2)
- Design and coding guidelines for modelling and programming languages (see 5.5.3)

NOTE In the case of model-based development these guidelines include modelling guidelines.

- Guidelines for the application of methods (from external source)
- Software tool application guidelines (see 5.5.4)
- Safety analysis report (see 7.5.4)
- Pre-existing software components (from external source)

## 8.4 Requirements and recommendations

**8.4.1** To ensure that the software unit design captures the information necessary to allow the subsequent development activities to be performed correctly and effectively, the design shall be described using the notations listed in Table 8.

**Table 8 — Notations for software unit design**

| Methods | | ASIL | | | |
|---|---|---|---|---|---|
| | | A | B | C | D |
| 1a | Documentation of the software unit design in natural language | ++ | ++ | ++ | ++ |
| 1b | Informal notations | ++ | ++ | + | + |
| 1c | Semi-formal notations | + | ++ | ++ | ++ |
| 1d | Formal notations | + | + | + | + |



NOTE In the case of model-based development with automatic code generation, the methods for representing the software unit design have to be applied to the functional model which will serve as the basis for the code generation.

**8.4.2** The specification of the software units shall describe the functional behaviour and the internal design to the level of detail necessary for their implementation.

EXAMPLE Internal design can include constraints on the use of registers and storage of data.

**8.4.3** The design and implementation of the software unit shall achieve the following properties:

a) avoidance of unnecessary complexity;

b) testability; and

c) maintainability.

In the case of manual code development these properties shall apply to the source code. In the case of model-based development, with automatic code generation, these properties shall apply to the model.

**8.4.4** The design principles for software unit design and implementation listed in Table 9 shall be applied to achieve the following properties:

a) correct order of execution of subprograms and functions within the software units, based on the software architectural design;

b) consistency of the interfaces between the software units;

c) correctness of data flow and control flow between and within the software units;

d) simplicity;

e) readability and comprehensibility;

f) robustness;

EXAMPLE Methods to prevent implausible values, execution errors, division by zero, and errors in the data flow and control flow.

g) suitability for software modification; and

h) testability during software unit testing.

**Table 9 — Design principles for software unit design and implementation**

| Methods | | ASIL | | | |
|---|---|---|---|---|---|
| | | A | B | C | D |
| 1a | One entry and one exit point in subprograms and functions[a] | ++ | ++ | ++ | ++ |
| 1b | No dynamic objects or variables, or else online test during their creation[a, b] | + | ++ | ++ | ++ |
| 1c | Initialisation of variables | ++ | ++ | ++ | ++ |
| 1d | No multiple use of variable names[a] | + | ++ | ++ | ++ |
| 1e | Avoid global variables or else justify their usage[a] | + | + | ++ | ++ |
| 1f | Limited use of pointers[a] | o | + | + | ++ |
| 1g | No implicit type conversions[a, c] | + | ++ | ++ | ++ |
| 1h | No hidden data flow or control flow[b, d] | + | ++ | ++ | ++ |
| 1i | No unconditional jumps[a, c, d] | ++ | ++ | ++ | ++ |
| 1j | No recursions | + | + | ++ | ++ |

[a] Methods 1a, 1b, 1c, 1d, 1e, 1f, 1g and 1i may not be applicable for graphical modelling notations used in model-based development.

[b] If these compiler features are “tool qualified” in accordance with ISO 26262-8:—, Clause 10, Method 1b need not be applied if a compiler is used which ensures that there will be enough program storage allocated for all dynamic variables and objects before run-time or which inserts run-time tests for correct online-allocation of program storage, i.e. stack bounds checking.

[c] Methods 1g and 1i are not applicable in assembler programming.

[d] Methods 1h and 1i reduce the potential for modelling data flow and control flow through jumps or global variables.

NOTE For the C language, [MISRA C] covers many of the methods listed in Table 9.

**8.4.5** The software unit design and implementation shall be verified in accordance with ISO 26262-8:— Clause 9, and by applying the verification methods listed in Tables 10 and 11, to demonstrate:

a) the compliance with the hardware-software interface (see ISO 26262-5:—, Clause 6.4.10);

b) the completeness regarding the software safety requirements and the software architecture through traceability;

c) the compliance of the source code with its specification;

d) the compliance of the source code with the coding guidelines; and

NOTE The analysis for compliance with the implementation and coding guidelines is typically carried out using tools. Guideline rules that cannot be checked automatically are to be checked manually.

e) the compatibility of the software unit implementations with target hardware.



**Table 10 — Methods for the verification of software unit design and implementation**

| Methods | | ASIL A | ASIL B | ASIL C | ASIL D |
|---|---|---|---|---|---|
| 1a | Informal verification | See Table 11 | | | |
| 1b | Semi-formal verification[a] | + | + | ++ | ++ |
| 1c | Formal verification | o | o | + | + |
| 1d | Control flow analysis[b, c] | + | + | ++ | ++ |
| 1e | Data flow analysis, [b, c,] | + | + | ++ | ++ |
| 1f | Static code analysis | + | ++ | ++ | ++ |
| 1g | Semantic code analysis[d] | + | + | + | + |

a Method 1b requires an executable design or implementation model of the unit to be verified.

b Methods 1d and 1e should be applied at the source code level. These methods are applicable both to manual code development and to model-based development,

c Methods 1d and 1e can be part of methods 1c or 1g.

d Method 1g is used for mathematical analysis of source code by use of an abstract representation of possible values for the variables. For this it is not necessary to translate and execute the source code.

NOTE Table 10 only lists static verification techniques. Dynamic verification techniques (testing techniques) are covered in Tables 12, 13, and 14.

**Table 11 — Methods for the informal verification of software unit design and implementation**

| Methods | | ASIL A | ASIL B | ASIL C | ASIL D |
|---|---|---|---|---|---|
| 1a | Inspection of the software unit design | + | ++ | ++ | ++ |
| 1b | Walkthrough of the software unit design | ++ | + | O | o |
| 1c | Model Inspection[a] | + | ++ | ++ | ++ |
| 1d | Model Walkthrough[a] | ++ | + | O | o |
| 1e | Inspection of the source code[b] | + | ++ | ++ | ++ |
| 1f | Walkthrough of the source code[b] | ++ | + | O | o |

a In the case of model-based software development the software unit specification can be verified at the model level by applying Method 1c and 1d.

b In the case of model-based development with automatic code generation, methods for informal verification of the generated code can be replaced by automated methods and techniques if applicable.

**8.4.6** It shall be verified that the embedded software to be included as part of a production release (see ISO 26262-4:—, Clause 11) contains all the specified functions. If other functions are included, it shall be ensured that these cannot impair the compliance with the software safety requirements.

NOTE 1 Code used for other functions, such as for development and debugging, is only to be included in the software for production if it is developed and verified to the same requirements as the production code. The removal of such code is also a change (see ISO 26262-8:—, Clause 8).

NOTE 2 Deactivation of these functions can be ensured as an acceptable means of compliance.

### 8.5 Work products

**8.5.1 Software unit design specification** resulting from requirements 8.4.1, 8.4.2, 8.4.3 and 8.4.4.

NOTE In the case of model-based development, the implementation model specifies the software units in conjunction with other techniques (see Table 8).

**8.5.2 Software unit implementation** resulting from requirement 8.4.4.

**8.5.3 Software verification report (refined)** resulting from requirements 8.4.5 and 8.4.6.

## 9 Software unit testing

### 9.1 Objectives

The objective of this subphase is to demonstrate that the software units fulfil the software unit specifications and do not contain undesired functionality.

### 9.2 General

A procedure for testing the software unit against the software unit specifications is established, and the tests are carried out in accordance with this procedure.

### 9.3 Inputs to this clause

#### 9.3.1 Prerequisites

The following information shall be available:

— Safety plan (refined) (see 7.5.2)

— Software verification plan (refined) (see 6.5.3)

— Software unit design specification (see 8.5.1)

— Software unit implementation (see 8.5.2)

— Software verification report (refined) (see 8.5.3)

#### 9.3.2 Further supporting information

The following information may be considered:

— Guidelines for the application of methods (from external source)

— Software tool application guidelines (see 5.5.4)

— Pre-existing software components (from external source)

### 9.4 Requirements and recommendations

**9.4.1** Software unit testing shall be planned, specified and executed in accordance with ISO 26262-8:—, Clause 9.



NOTE 1 Based on the definitions in ISO 26262-8:—, Clause 9, the test objects in the software unit testing are the software units.

NOTE 2 For model-based software development, the corresponding parts of the implementation model represent test objects as they implement the software units.

**9.4.2** The software unit testing methods listed in Table 12 shall be applied to demonstrate that the software units achieve:

a) compliance with the software unit design specification (see Clause 8);

b) compliance with the specification of the hardware-software interface;

c) correct implementation of the functionality;

d) absence of unintended functionality;

e) robustness; and

EXAMPLES The absence of inaccessible software, effectiveness of error detection and error handling mechanisms.

f) sufficiency of the resources to support the functionality.

NOTE The software unit testing always comprises a combination of different test methods, as there is no test method that covers all aspects of software unit testing.

**Table 12 — Methods for software unit testing**

| Methods | | ASIL | | | |
|---|---|---|---|---|---|
| | | A | B | C | D |
| 1a | Requirement-based test | ++ | ++ | ++ | ++ |
| 1b | Interface test | ++ | ++ | ++ | ++ |
| 1c | Fault injection test[a] | + | + | + | ++ |
| 1d | Resource usage test[b] | + | + | + | ++ |
| 1e | Back-to-back test between model and code, if applicable[c] | + | + | ++ | ++ |

[a] This includes injection of arbitrary faults in order to test safety mechanisms (e.g. by corrupting values of variables)

[b] Some aspects of the resource usage test can only by evaluated properly when the software unit tests are executed on the target hardware or if the emulator for the target processor supports resource usage tests.

[c] This method requires a model that can simulate the functionality of the software units. Here, the model and code are stimulated in the same way and results compared with each other.

**9.4.3** To demonstrate the appropriate specification of test cases, for the selected software unit test methods, test cases shall be derived using the methods listed in Table 13.

**Table 13 — Methods for deriving test cases for software unit testing**

| Methods | | ASIL | | | |
|---|---|---|---|---|---|
| | | A | B | C | D |
| 1a | Analysis of requirements | ++ | ++ | ++ | ++ |
| 1b | Generation and analysis of equivalence classes | + | ++ | ++ | ++ |
| 1c | Analysis of boundary values[a] | + | ++ | ++ | ++ |
| 1d | Error guessing[b] | + | + | + | + |

[a] This method applies to interfaces, values approaching and crossing the boundaries and out of range values.

[b] “Error guessing tests” can be based on data collected through a “lessons learned” process and expert judgment.

**9.4.4** To evaluate the completeness of test cases and to demonstrate that there is no unintended functionality, the coverage of requirements at the software unit level shall be determined and the structural coverage shall be measured in accordance with the metrics listed in Table 14. If necessary, additional test cases shall be specified or a rationale shall be available.

EXAMPLE Analysis of structural coverage can reveal shortcomings in requirement-based test cases, inadequacies in requirements, dead code, deactivated code or unintended functionality.

**Table 14 — Structural coverage metrics at the software unit level**

| Methods | | | ASIL | | | |
|---|---|---|---|---|---|---|
| | | | A | B | C | D |
| 1a | Statement coverage | | ++ | ++ | + | + |
| 1b | Branch coverage | | + | ++ | ++ | ++ |
| 1c | MC/DC (Modified Condition/Decision Coverage) | | + | + | + | ++ |

NOTE 1 The structural coverage can be determined by the use of appropriate software tools.

NOTE 2 In the case of model-based development, software unit testing may be moved to the model level using analogous structural coverage metrics for models.

NOTE 3 If instrumented code is used to determine the degree of coverage, it might be necessary to show that the instrumentation has no effect on the test results. This can be done by repeating the tests with non-instrumented code.

NOTE 4 A rationale is to be given for the level of coverage achieved (e.g. for accepted dead code or code segments depending on different software configurations), or else code not covered can be verified using complementary methods (e.g. inspections).

**9.4.5** The test environment for software unit testing shall correspond as far as possible to the target environment. If the software unit testing is not carried out in the target environment, the differences in the source and object code, and the differences between the test environment and the target environment, shall be analysed in order to specify additional tests on the target environment during the subsequent test phases.

NOTE 1 Differences between the test environment and the target environment can arise in the source code or object code, for example, due to different bit widths of data words and address words of the processors.

NOTE 2 Depending on the scope of the tests, it can be useful to carry out the software unit testing on the processor of the target system. If this is not possible, a processor emulator can be used. Otherwise the software unit testing is executed on the development system.

NOTE 3 Software unit testing can be executed in different environments, for example:

— Model-in-the-loop tests;

- Software-in-the-loop tests;
- Processor-in-the-loop tests; and
- Hardware-in-the-loop tests.

NOTE 4 For model-based development, software unit testing can be carried out at the model level followed by back-to-back tests between the model and the code. The back-to-back tests are used to ensure that the behaviour of the models with regard to the test objectives is equivalent to the automatically-generated code.

### 9.5 Work products

**9.5.1 Software verification plan (refined)** resulting from requirements 9.4.1, 9.4.2, 9.4.3 and 9.4.5.

**9.5.2 Software verification specification** resulting from requirements 9.4.1, 9.4.4 and 9.4.5.

**9.5.3 Software verification report (refined)** resulting from requirement 9.4.1.

## 10 Software integration and testing

### 10.1 Objectives

The first objective of this subphase is to integrate the software components.

The second objective of this subphase is to demonstrate that the software architectural design is correctly realised by the embedded software.

### 10.2 General

In this subphase, the particular integration levels are tested against the software architectural design, and the interfaces between the software units and the software components are tested. The steps of the integration and the tests of the software components are to correspond directly to the hierarchical architecture of the software.

The embedded software can consist of safety-related and non-safety-related software components.

### 10.3 Inputs to this clause

#### 10.3.1 Prerequisites

The following information shall be available:

- Safety plan (refined) (see 7.5.2)
- Software verification plan (refined) (see 9.5.1)
- Software architectural design specification (see 7.5.1)
- Software unit implementation (see 8.5.2)
- Software verification specification (see 9.5.2)
- Software verification report (refined) (see 9.5.3)

### 10.3.2 Further supporting information)

The following information may be considered:

— Guidelines for the application of methods (from external source)

— Software tool application guidelines (see 5.5.4)

## 10.4 Requirements and recommendations

**10.4.1** The planning of the software integration shall describe the steps for integrating the individual software units hierarchically into software components until the embedded software is fully integrated, and shall consider:

a) the functional dependencies that are relevant for software integration; and

b) the dependencies between the software integration and the hardware-software integration.

NOTE For model-based development, the software integration can be replaced with integration at the model level and subsequent automatic code generation for the integrated model.

**10.4.2** Software integration testing shall be planned, specified and executed in accordance with ISO 26262-8:—, Clause 9.

NOTE Based on the definitions in ISO 26262-8:—, Clause 9 the software integration test objects are the software components.

**10.4.3** The software integration test methods listed in Table 15 shall be applied to demonstrate that both the software components and the embedded software achieve:

a) compliance with the software architectural design (see Clause 7);

b) compliance with the specification of the hardware-software interface (see ISO 26262-4:—, Clause 7);

c) correct implementation of the functionality;

d) robustness; and

EXAMPLES absence of inaccessible software, effective error detection and handling.

e) sufficiency of the resources to support the functionality.



**Table 15 — Methods for software integration testing**

| Methods | | ASIL | | | |
|---|---|---|---|---|---|
| | | A | B | C | D |
| 1a | Requirements-based test | ++ | ++ | ++ | ++ |
| 1b | External interface test | ++ | ++ | ++ | ++ |
| 1c | Fault injection test[a] | + | + | ++ | ++ |
| 1d | Resource usage test[b, c] | + | + | + | ++ |
| 1e | Back-to-back test between model and code, if applicable[d] | + | + | ++ | ++ |

a This includes injection of arbitrary faults in order to test safety mechanisms (e.g. by corrupting software or hardware components)

b To ensure the fulfilment of requirements influenced by the hardware architectural design with sufficient tolerance, properties such as average and maximum processor performance, minimum or maximum execution times, storage usage (e.g. RAM for stack and heap, ROM for program and data) and the bandwidth of communication links (e.g. data busses) have to be determined.

c Some aspects of the resource usage test can only by evaluated properly when the software integration tests are executed on the target hardware or if the emulator for the target processor supports resource usage tests.

d This method requires a model that can simulate the functionality of the software components. Here, the model and code are stimulated in the same way and results compared with each other.

**10.4.4** To demonstrate the appropriate specification of test cases for the selected software integration test methods, test cases shall be derived using the methods listed in Table 16.

**Table 16 — Methods for deriving test cases for software integration testing**

| Methods | | ASIL | | | |
|---|---|---|---|---|---|
| | | A | B | C | D |
| 1a | Analysis of requirements | ++ | ++ | ++ | ++ |
| 1b | Generation and analysis of equivalence classes[a] | + | ++ | ++ | ++ |
| 1c | Analysis of boundary values[b] | + | ++ | ++ | ++ |
| 1d | Error guessing[c] | + | + | + | + |

a This method may be used to partition possible input values of external interfaces.

b This method applies to parameters or variables, values approaching and crossing the boundaries and out of range values.

c This method considers situations usually leading to errors. Determining such test cases in an efficient way requires experience in testing as well as intuition combined with knowledge about the integrated software to be tested.

**10.4.5** To evaluate the completeness of test cases and to demonstrate that there is no unintended functionality, the coverage of requirements at the software architectural level shall be determined and the structural coverage shall be measured in accordance with the metrics listed in Table 17. If necessary, additional test cases shall be specified or a rationale shall be available.

EXAMPLE Analysis of structural coverage can reveal shortcomings in requirement-based test cases, inadequacies in requirements, dead code, deactivated code or unintended functionality.

**Table 17 — Structural coverage metrics at the software architectural level**

| Methods | | ASIL | | | |
|---|---|---|---|---|---|
| | | A | B | C | D |
| 1a | Function coverage[a] | + | + | ++ | ++ |
| 1b | Call coverage[b] | + | + | ++ | ++ |

[a] The degree of coverage claimed in method 1a requires at least one execution of every software function. This evidence can be achieved by an appropriate software integration strategy.

[b] The degree of coverage claimed in method 1b requires at least one execution of every software function call.

NOTE The structural coverage can be determined using appropriate software tools.

**10.4.6** Unspecified software components shall be identified. Such software components shall be removed or deactivated.

NOTE This applies to all components of the embedded software including non-safety-related components.

**10.4.7** The test environment for software integration testing shall correspond as closely as possible to the target environment. If the software integration testing is not carried out in the target environment, the differences in the source and object code and the differences between the test environment and the target environment shall be analysed in order to specify additional tests on the target environment during the subsequent test phases.

NOTE 1 Differences between the test environment and the target environment can arise in the source or object code, for example, due to different bit widths of data words and address words of the processors.

NOTE 2 Depending on the scope of the tests, it can be useful to carry out the software integration testing on the processor of the target system. If this is not possible, a processor emulator can be used. Otherwise, the software integration testing is executed on the development system.

NOTE 3 Software integration testing can be executed in different environments, for example:

— Model-in-the-loop tests;

— Software-in-the-loop tests;

— Processor-in-the-loop tests;

— Hardware-in-the-loop tests.

## 10.5 Work products

**10.5.1 Software verification plan (refined)** resulting from requirements 10.4.1, 10.4.2, 10.4.3, 10.4.4 and 10.4.5.

**10.5.2 Software verification specification (refined)** resulting from requirements 10.4.1, 10.4.5, 10.4.6 and 10.4.7.

**10.5.3 Embedded software** resulting from requirement 10.4.1.

**10.5.4 Software verification report** resulting from requirement 10.4.2.



# 11 Verification of software safety requirements

## 11.1 Objectives

The objective of this subphase is to demonstrate that the embedded software fulfils the software safety requirements.

## 11.2 General

The purpose of the verification of the software safety requirements is to demonstrate that the embedded software satisfies its requirements in the target environment.

## 11.3 Inputs to this clause

### 11.3.1 Prerequisites

The following information shall be available:

— Safety plan (refined) (see 7.5.2)

— Software verification plan (refined) (see 10.5.1)

— Software safety requirements specification (refined) (see 7.5.3)

— Software architectural design specification (see 7.5.1)

— Software verification specification (refined) (see 10.5.2)

— Software verification report (refined) (see 10.5.4)

— Integration testing report (see ISO 26262-4:—, 8.5.2)

### 11.3.2 Further supporting information

The following information may be considered:

— System design specification (see ISO 26262-4:—, 7.5.1)

— Technical safety concept (see ISO 26262-4:—, 7.5.1)

— Validation plan (refined) (see ISO 26262-4:—, 9.5.1)

— Guidelines for the application of methods (from external source)

— Software tool application guidelines (see 5.5.4)

## 11.4 Requirements and recommendations

**11.4.1** The verification of the software safety requirements shall be planned, specified and executed in accordance with ISO 26262-8:—, Clause 9.

**11.4.2** To verify that the embedded software fulfils the software safety requirements, tests shall be conducted in the test environments listed in Table 18.

NOTE Test cases that already exist, for example from software integration testing, can be re-used.

**Table 18 — Test environments for conducting the software safety requirements verification**

| Methods | | ASIL | | | |
|---|---|---|---|---|---|
| | | A | B | C | D |
| 1a | Hardware-in-the-loop | + | + | ++ | ++ |
| 1b | Electronic control unit network environments[a] | ++ | ++ | ++ | ++ |
| 1c | Vehicles | ++ | ++ | ++ | ++ |

[a] Examples are “lab-cars”, “rest of the bus” simulations or test benches partially or fully integrating the electrical systems of a vehicle.

**11.4.3** The verification of the software safety requirements shall be executed on the target hardware.

**11.4.4** The results of the verification of the software safety requirements shall be evaluated in accordance with:

a) compliance with the expected results;

b) coverage of the software safety requirements; and

c) a pass or fail criteria.

## 11.5 Work products

**11.5.1 Software verification plan (refined)** resulting from requirements 11.4.1, 11.4.2 and 11.4.3.

**11.5.2 Software verification specification (refined)** resulting from requirements 11.4.1 and 11.4.3.

**11.5.3 Software verification report (refined)** resulting from requirement 11.4.4.

# Annex A
(informative)

## Overview of and document flow of management of product development at the software level

Table A.1 provides an overview of objectives, prerequisites and work products of the particular phases of the product development at the software level.

**Table A.1 — Product development at the software level: overview**

| Clause | Title | Objectives | Prerequisites | Work products |
|---|---|---|---|---|
| 6-5 | Initiation of product development at the software level | Plan and initiate the functional safety activities for the subphases of the software development activity. | Overall project plan (see ISO 26262-4:—, 5.5.1)<br>Safety plan (refined) (see ISO 26262-4:—, 5.5.2)<br>Item integration and testing plan (refined) (see ISO 26262-4:—, 7.5.4)<br>Technical safety concept (see ISO 26262-4:—, 7.5.1)<br>System design specification (see ISO 26262-4:—, 7.5.2) | 5.5.1 Safety plan (refined)<br>5.5.2 Software verification plan<br>5.5.3 Design and coding guidelines for modelling and programming language<br>5.5.4 Software tool application guidelines |
| 6-6 | Specification of software safety requirements | Specify software safety requirements. The software safety requirements have to be derived from the system design specification.<br>Verify that the software safety requirements are consistent with the technical system design specification. | Technical safety concept (see ISO 26262-4:—, 7.5.1)<br>System design specification (see ISO 26262-4:—, 7.5.2)<br>Safety plan (refined) (see 5.5.1)<br>Software verification plan (see 5.5.2) | 6.5.1 Software safety requirements specification<br>6.5.2 Hardware-software interface specification (refined)<br>6.5.3 Software verification plan (refined)<br>6.5.4 Software verification report |
| 6-7 | Software architectural design | Develop a software architectural design which realises the software safety requirements.<br>Verify the software architectural design. | Software safety requirements specification (see 6.5.1)<br>Safety plan (refined) (see 5.5.1)<br>Software verification plan (refined) (see 6.5.3)<br>Software verification report (refined) (see 6.5.4) | 7.5.1 Software architectural design specification<br>7.5.2 Safety plan (refined)<br>7.5.3 Software safety requirements specification (refined)<br>7.5.4 Safety analysis report<br>7.5.5 Dependent failures analysis report<br>7.5.6 Software verification report (refined) |
| 6-8 | Software unit design and implementation | Specify the software units in accordance with the software architectural design and the associated software safety requirements.<br>Implement the software units as specified.<br>Verify the software unit design and their implementation. | Safety plan (refined) (see 7.5.2)<br>Software verification plan (refined) (see 6.5.3)<br>Software architectural design specification (see 7.5.1)<br>Software safety requirements specification (refined) (see 7.5.3)<br>Software verification report (refined) (see7.5.6) | 8.5.1 Software unit design specification<br>8.5.2 Software unit implementation<br>8.5.3 Software verification report (refined) |

| Claus e | Title | Objectives | Prerequisites | Work products |
|---|---|---|---|---|
| 6-9 | Software unit testing | Demonstrate that the software units fulfil the software unit specifications and do not contain undesired functionality. | Safety plan (refined) (see 7.5.2)<br>Software verification plan (refined) (see 6.5.3)<br>Software unit design specification (see 8.5.1)<br>Software unit implementation (see 8.5.2)<br>Software verification report (refined) (see 8.5.3) | 9.5.1 Software verification plan (refined)<br>9.5.2 Software verification specification<br>9.5.3 Software verification report (refined) |
| 6-10 | Software integration and testing | Integrate the software components.<br>Demonstrate that the software architectural design is correctly realised by the embedded software. | Safety plan (refined) (see 7.5.2)<br>Software verification plan (refined) (see 9.5.1)<br>Software architectural design specification (see 7.5.1)<br>Software unit implementation (see 8.5.2)<br>Software verification specification (refined) (see 9.5.2)<br>Software verification report (refined) (see 9.5.3) | 10.5.1 Software verification plan (refined)<br>10.5.2 Software verification specification (refined)<br>10.5.3 Embedded software<br>10.5.4 Software verification report (refined) |
| 6-11 | Verification of software safety requirements | Demonstrate that the embedded software fulfils the software safety requirements. | Safety plan (refined) (see 7.5.2)<br>Software verification plan (refined) (see 10.5.1)<br>Software safety requirements specification (refined) (see 7.5.3)<br>Software architectural design specification (see 7.5.1)<br>Software verification specification (refined) (see 10.5.2)<br>Software verification report (refined) (see 10.5.4)<br>Integration testing report (see ISO 26262-4:—, 8.5.2) | 11.5.1 Software verification plan (refined)<br>11.5.2 Software verification specification (refined)<br>11.5.3 Software verification report (refined) |



# Annex B
(informative)

# Model-based development

## B.1 Objectives

This Annex describes the concept of model-based development of in-vehicle software and outlines its implications on the product development at the software level.

## B.2 General

Mathematical modelling, which has been extensively used in many engineering domains, is also gaining wide-spread use in the development of embedded software. In the automotive sector, modelling is used for the conceptual capture of the functionality to be realised (open/closed loop control, monitoring) as well as for the simulation of real physical system behaviours (vehicle environment).

Modelling is usually carried out with commercial-off-the-shelf modelling and simulation packages. They support the development and definition of system/software components, their connections and interfaces by semiformal graphical models using editable, hierarchical block diagrams (control diagrams) and extended state transition diagrams (state charts) and provide the necessary means of description, computation techniques and interpreters/compilers. Graphical editors permit an intuitive development and description of complex models. Hierarchically structured modularity is used in order to control complexity. A model consists of function blocks with well-defined inputs and outputs. Function blocks are connected within the block diagram by directed edges between their interfaces, which describe signal flows. With this, they represent equations in the mathematical model, which relate the interface variables of different components. The connection lines represent causally motivated directions of action, which define the outputs of one block as the inputs of another. Components within the hierarchy can aggregate other components or be elementary.

Such models can be simulated, i.e. executed. During simulation the calculation causality follows the defined directions of action until the entire model has been processed. For solving the equations described by the model at certain instants of time there is a range of different predefined solvers available. Variable-step solvers are of special importance for the thorough modelling of physical systems and are used primarily for modelling the vehicle and the environment. For the development of embedded software fixed-step solvers are used, which represent a necessary prerequisite for an efficient code generation.

The modelling style described is used extensively within the scope of model-based development of embedded in-vehicle software. Typically, both an executable model of the control software (functional model) and a model of the surrounding system (vehicle model) and its environment (environment model) are created early in the development cycle and are simulated together. This way, it is possible to model even highly complex automotive systems with a high degree of detail at an acceptable calculation speed and to simulate their behaviour closely to reality. While the vehicle/environment model is gradually replaced during the course of development by the real system and its real environment, the functional model serves as a blueprint for the implementation of embedded software on the control unit through code generation.

One characteristic of the model-based development paradigm is the fact that the functional model not only specifies the desired function but also provides design information and finally even serves as the basis of the implementation by means of code generation. In other words, such a function model represents specification aspects as well as design and implementation aspects. In practice, these different aspects are reflected in an evolution of the functional model from an early specification model via a design model to an implementation model and finally its automatic transformation into code (model evolution). In comparison to code-based software development with a clear separation of phases, in model-based development a stronger coalescence of the phases Software Safety requirements, Software Architectural Design, and Software unit design and implementation can be noted. Moreover, one and the same graphical modelling notation is used during the consecutive development stages. Testing activities are also treated differently since models can be used as a useful source of information for the testing process (model-based testing). The seamless utilisation of models facilitates a highly consistent and efficient development.

NOTE The paradigm of model-based development does not depend on the existence of the model type mentioned above. Alternatively, UML-models, for example, can be used as well.



# Annex C
(normative)

# Software configuration

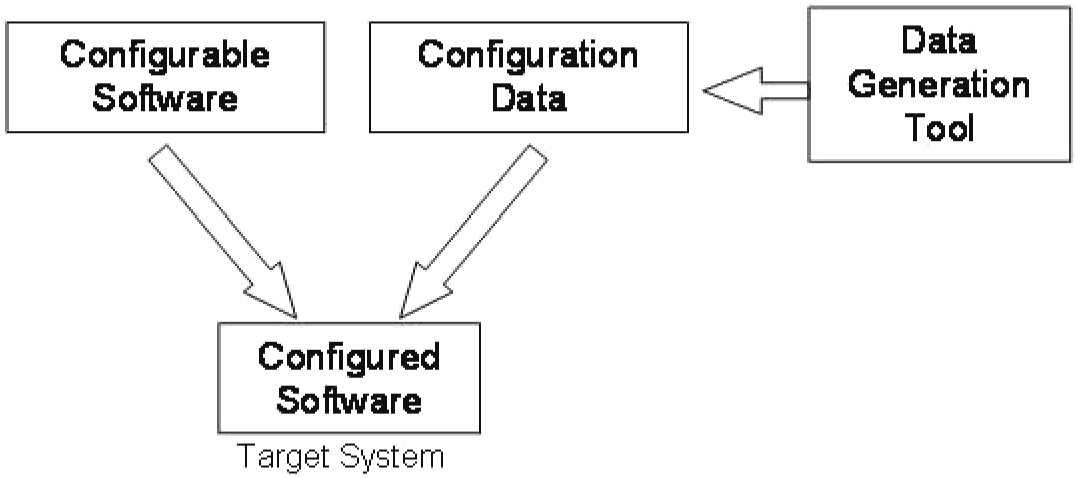


## C.1 Objectives

The objective of software configuration is to enable controlled changes in the behaviour of the software for different applications.

## C.2 General

**Description:**
Configurable software enables the development of application specific software using configuration and calibration data (see Figure C.1).



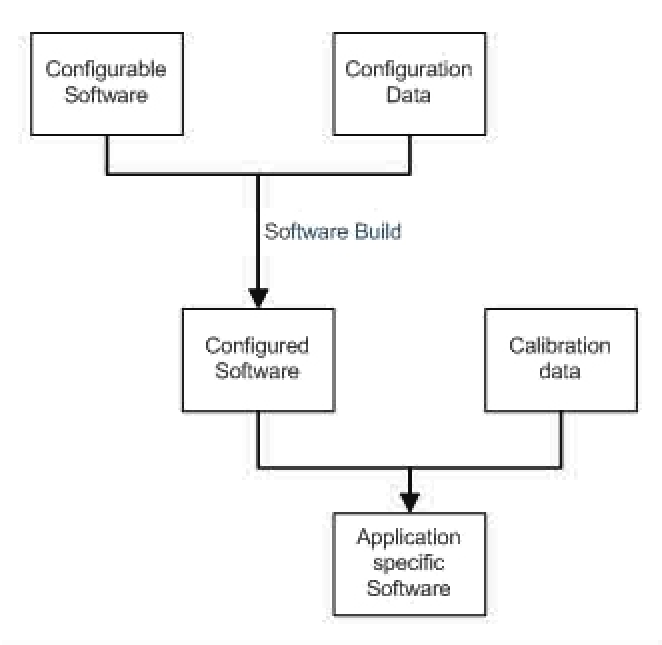



**Figure C.1 — Creating application specific software**

## C.3 Inputs to this clause

### C.3.1 Prerequisites

The following information shall be available:

— Safety plan (see 5.5.1)

## C.4 Requirements and recommendations

**C.4.1** The configuration data shall be specified to ensure the correct usage of the configurable software during the safety lifecycle. This includes:

a) the valid values of the configuration data;

b) the intent and usage of the configuration data;

c) the range, scaling, units; and

d) the interdependencies between different elements of the configuration data.

**C.4.2** Verification of the configuration data shall be performed to ensure:

a) the use of values within range; and

b) the compatibility with values of the other configuration data.

NOTE The testing of the configured software is performed within the test phases of the software safety lifecycle (see ISO 26262-6:—, Clause 9, ISO 26262-6:—, Clause 10, ISO 26262-6:—, Clause 11 and ISO 26262-4:—, Clause 8).

**C.4.3** The ASIL of the configuration data shall equal the highest ASIL of the configurable software by which it is used.

**C.4.4** The verification of configurable software shall be planned, specified and executed in accordance with ISO 26262-8:—, Clause 9. Configurable software shall be verified for each configuration data set that is to be used for the item development under consideration.

NOTE Only that part of the embedded software whose behaviour depends on the configuration data is to be verified for each configuration data set to be used.

**C.4.5** For configurable software a simplified software safety lifecycle in accordance with Figures C.2 and C.3 may be applied.

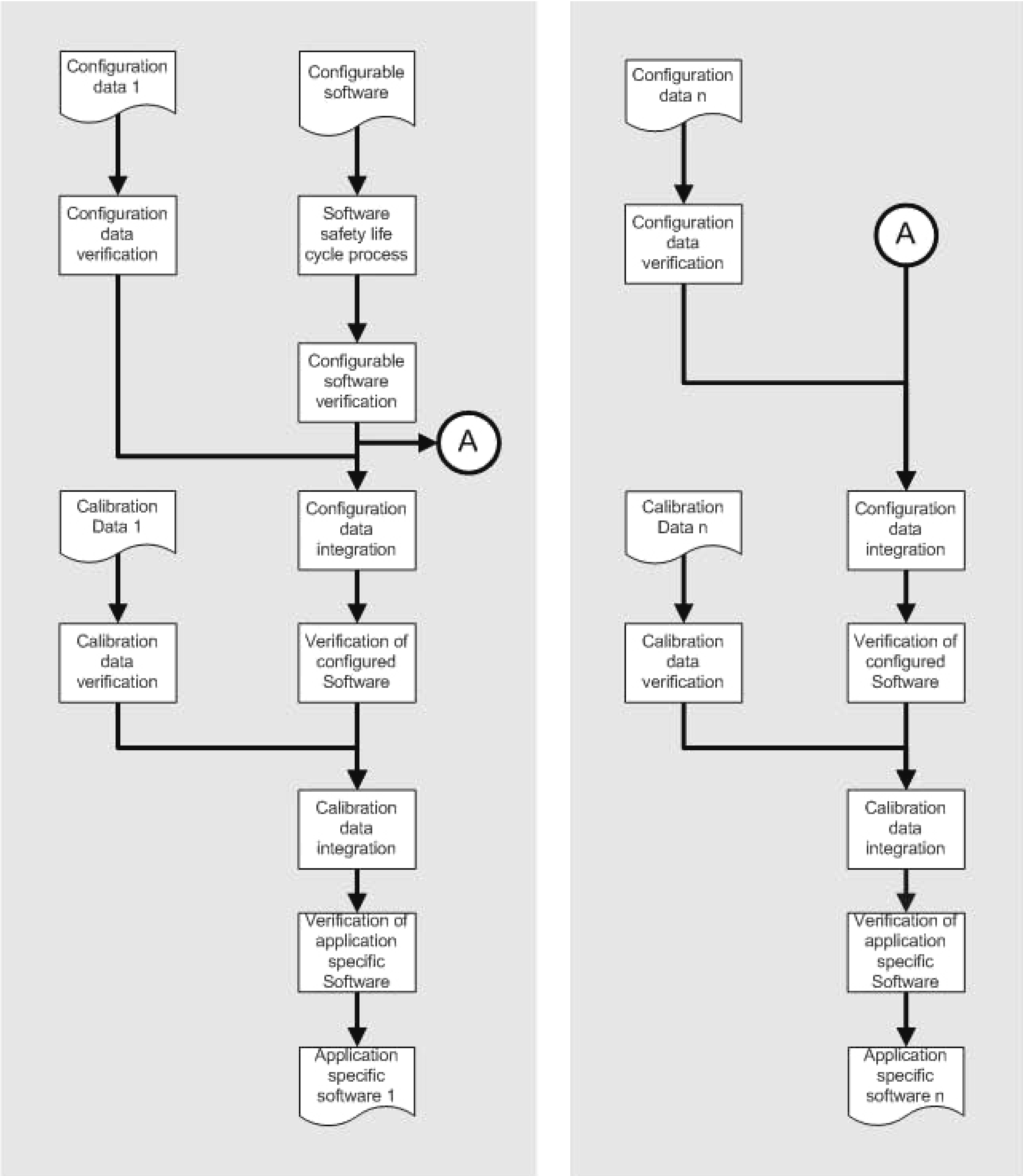



**Figure C.2 — Variants of the reference phase model for software development with configurable software and different configuration data**

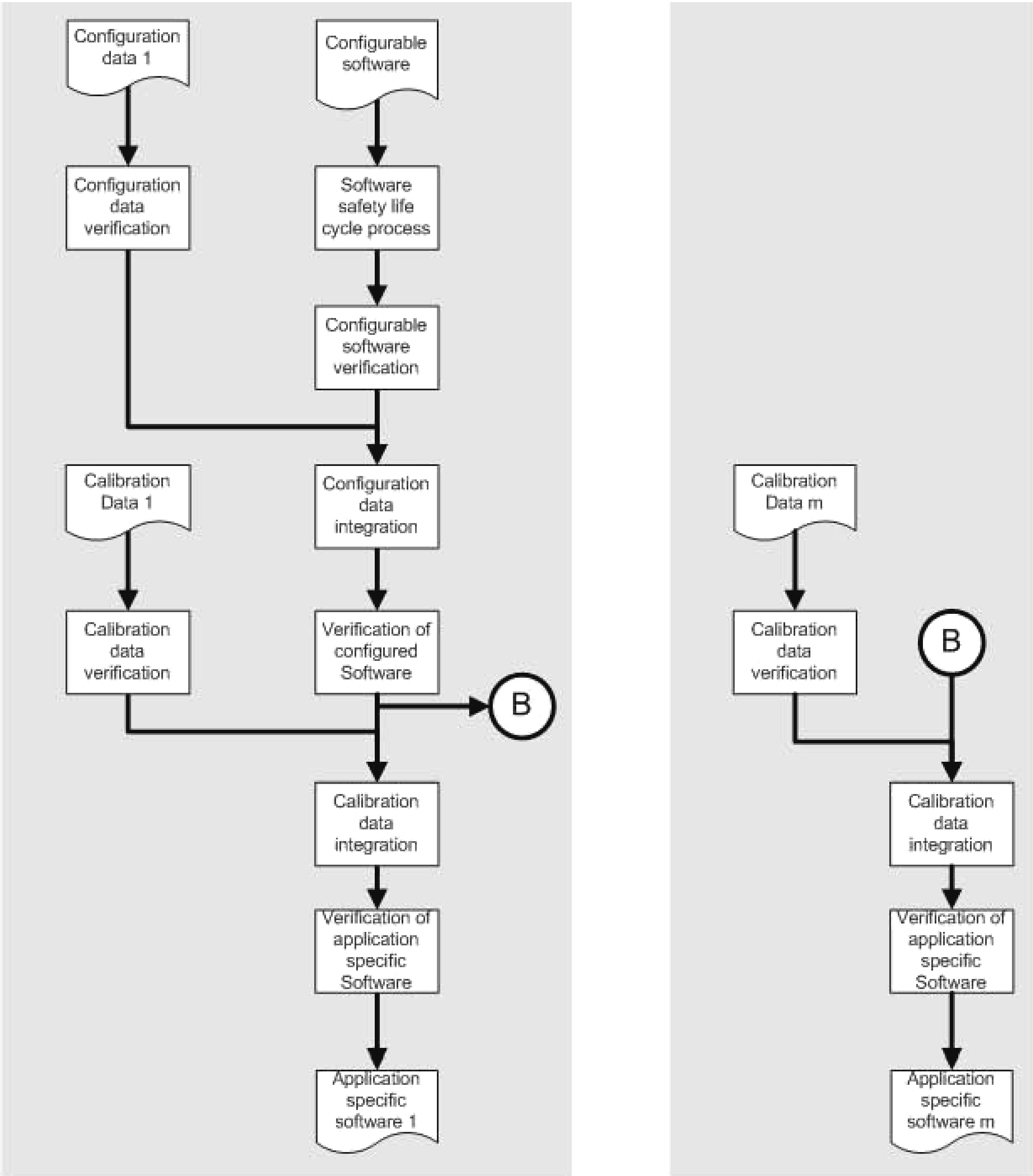



**Figure C.3 — Variants of the reference phase model for software development with configurable software and different calibration data**

**C.4.6** The calibration data associated with software components shall be specified to ensure the correct operation and expected performance of the configured software. This includes:

a) the valid values of the calibration data;

b) the intent and usage of the calibration data;

c) the range, scaling and units, if applicable, with their dependence from the operating state;

d) the known interdependencies between different calibration data of one calibration data set; and

e) the known interdependencies between configuration data and calibration data.

NOTE As configuration data has an impact on the configured software which uses the calibration data, these interdependencies are also to be considered.

**C.4.7** The verification of the calibration data shall be planned, specified and executed in accordance with ISO 26262-8:—, Clause 9. The verification of calibration data shall examine whether the calibration data is within its specified boundaries.

NOTE Verification of calibration data can also be performed partially within application specific software verification, or at runtime by the configurable software.

**C.4.8** The ASIL of the calibration data shall equal the highest ASIL of the configurable software by which it is used.

**C.4.9** To detect unintended changes of safety-related calibration data, the mechanisms for the detection of unintended changes of data listed in Table C.1 shall be implemented.

**Table C.1 — Mechanisms for the detection of unintended changes of data**

| Methods | | ASIL | | | |
|---|---|---|---|---|---|
| | | A | B | C | D |
| 1a | Performing plausibility checks on calibration data | ++ | ++ | ++ | ++ |
| 1b | Redundant storage of calibration data | + | + | + | ++ |
| 1c | Using error detecting codes[a] | + | + | + | ++ |
| [a] Error detection codes may also be implemented in the hardware in accordance with ISO 26262-5. | | | | | |

**C.4.10** The planning of the generation and application of calibration data shall specify:

a) the procedures which shall be followed;

b) the tools for generating calibration data, which shall be used; and

c) the procedures for verifying calibration data.

NOTE Verification of calibration data can be done by checking the value ranges of calibration data as well as the interdependencies between different calibration data.

**C.4.11** Software tools that are a candidate for tool qualification, including tools used for generating calibration data, shall be identified in accordance with ISO 26262-8:—, Clause 11.

## C.5 Work products

**C.5.1 Configuration data specification** resulting from requirements C.4.1 and C.4.3.

**C.5.2 Calibration data specification** resulting from requirement C.4.6.

**C.5.3 Safety plan (refined)** resulting from requirements C.4.1, C.4.4, C.4.9, and C.4.10.

**C.5.4 Configuration data** resulting from requirement C.4.3.

**C.5.5 Verification plan** resulting from requirements C.4.2; C.4.4 and C.4.7.



**C.5.6 Verification specification** resulting from requirement C.4.4.

**C.5.7 Verification report** resulting from requirements C.4.1, C.4.7 and C.4.8.

# Annex D
(informative)

# Freedom from interference by software partitioning

## D.1 Objectives

The objective is to prevent propagation of a failure in one software partition to another software partition.

NOTE Errors in the state of the executing software can occur due to systematic software faults or due to random as well as systematic hardware faults. Such errors in one partition could disturb the operation of other software partitions either due to shared resources or due to error propagation.

## D.2 General

**D.2.1** Software partitioning allows the co-existence of software partitions that use the same resources. It allows

a) software components to be free from interference from other software components; and

NOTE Different software partitions can be assigned different values of ASIL or a value of QM (see ISO 26262-9:—, Clause 5).

b) changes to be made to one software partition without the need to re-verify the unmodified software partitions.

### D.2.1 Impact on system and software design

Depending on the system architecture, two approaches can be used:

a) several software partitions within a single microcontroller (see Figure D.1) with shared resources such as CPU time, memory, I/O-devices; and

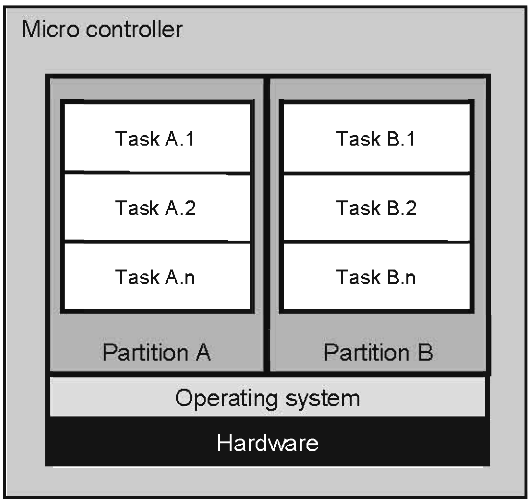



**Figure D.1 — Several software partitions within a single microcontroller**



b) several software partitions within the scope of a micro controller network (see Figures D.2 and D.3) with shared resources such as I/O-devices, especially internal and external data buses.

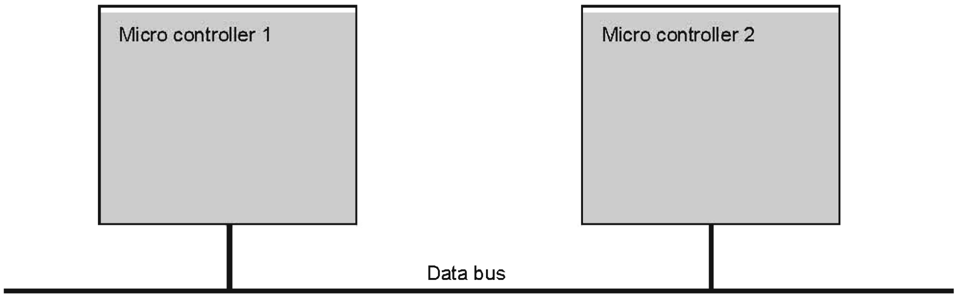



**Figure D.2 — Several partitions within the scope of a micro controller network**

NOTE The micro controller network can consist of several processors in a single electronic control unit communicating via an internal data bus (intra processor communication). This is illustrated in figure D.3.

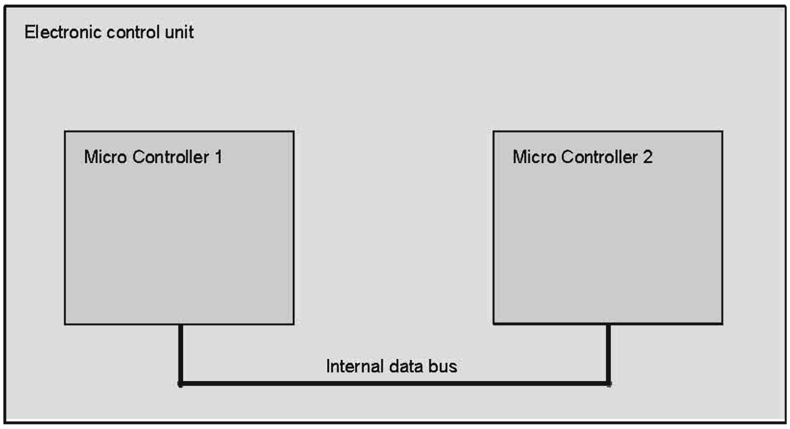



**Figure D.3 — Several partitions within the scope of a multi-processor electronic control unit**

Software components are executed within their respective software partition on their respective microcontroller as illustrated in Figures D.2 and D.3.

## D.2.2 Impact on shared resources

Software partitioning requires adequate support by system resources.

In order to isolate multiple software partitions in a shared resource environment, the hardware has to provide the operating system with the ability to restrict access to shared resources for each software partition.

### D.2.2.1 CPU time

To ensure freedom from interference of software partitions within a single microcontroller, the fault effects

— blocking of partitions due to communication deadlocks; and

— wrong allocation of processor execution time

are to be prevented by:

a) time triggered scheduling;

NOTE 1 Software partitions are considered coequally in allocating processor execution time and the same priority is assigned to all of them.

NOTE 2 Regarding the allocation of processor time, spare time is allocated in each processing cycle because of incoming interrupts.

NOTE 3 Time triggered scheduling is considered to have effectiveness “high” against protection against the fault effect “wrong processor execution time”.

b) cycling execution scheduling policy;

NOTE 4 The time triggered scheduling method specifies a scheduling algorithm based on a predetermined fixed schedule, repetitive with a fixed periodicity.

NOTE 5 Using the time triggered scheduling method the allocation of processor execution time takes place through a static allocation table. Thus, for each task, a fixed point in time is predetermined for activating the task. Usage of time triggered scheduling method precludes priority-based scheduling.

c) fixed priority based scheduling;

d) monitoring of processor execution time of software partitions in accordance with the allocation;

NOTE 6 Monitoring of each partition by software checks if all partitions are executed in conformance with the predefined static allocation table.

e) program sequence monitoring;

NOTE 7 Program sequence monitoring is based on a hardware device (see ISO 26262-5:—, Table D.10).

f) arrival rate monitoring.

NOTE 8 Monitoring of processor execution is an additive method of Program sequence monitoring. If both methods are combined the effectiveness is "high" protection against the fault effect “wrong processor execution time”.

#### D.2.2.2 Memory

To ensure freedom from interference of software partitions within a single microcontroller, the fault effect

— memory corruption due to unintended writing to memory of another partition

is to be prevented by:

a) memory protection mechanisms;

NOTE 1 The memory protection mechanisms refer to processors with Memory Management Unit or Memory Protection Unit.

NOTE 2 A Memory Management Unit enables the concept of virtual address space. This prevents a task of one partition corrupting the memory space of another task by unintended writing into that memory space, since every partition has its own address space.

NOTE 3 Usage of a Memory Management Unit requires support of the operating system.

NOTE 4 Provisions are made that the Memory Management Unit cannot be ignored, i.e. tasks are executed in a so-called user mode and the real addressing mode is not to be used.

b) verification of safety-related data;

NOTE 5 RAM locations containing safety-related data are verified by additional methods. This can be accomplished for example by using parity bits, Error Correcting/Correction Code (ECC), Cyclic Redundancy Checksum (CRC) or redundant storage.



NOTE 6 The effectiveness of these methods depends very heavily on the verification quality.

NOTE 7 Verification of safety-related data is done at run time.

c) offline analysis of code and data of other partitions;

d) restricted access to memory;

e) static analysis; and

NOTE 8 Static analysis methods defined in Table 10 can be used for reviewing pieces of code that access memory locations containing safety-related data.

f) static allocation.

NOTE 9 Static allocation means that resources are allocated statically during initialisation.

#### D.2.2.3 I/O-devices (communication)

To ensure freedom from interference of software partitions in communication microcontrollers, the fault effects

- loss of peer to peer communication;
- unintended message repetition due to the same message being unintentionally sent again;
- message loss during transmission;
- insertion of messages due to receiver unintentionally receiving an additional message, which is interpreted to have correct source and destination addresses;
- re-sequencing due to the order of the data being changed during transmission, i.e. the data is not received in the same order as in which it was been sent;
- message corruption due to one or more data bits in the message being changed during transmission;
- message delay due to the message being received correctly, but not in time;
- blocking access to data bus due to a faulty node not adhering to the expected patterns of use and making excessive demands for service, thereby reducing its availability to other nodes, e.g. while wrongly waiting for non existing data; and
- constant transmission of messages by a faulty node, thereby compromising the operation of the entire bus.

are to be prevented by:

a) identifier for communication objects;

b) keep alive messages;

NOTE 1 Keep alive messages is considered “high” effectiveness for detection of “Failure of communication peer”.

c) alive counter;

NOTE 2 Alive counter is considered “high” protection against “Unintended message repetition” and “medium” protection against “Message loss”, “Insertion of messages” and “Constant transmission of messages”.

d) sequence number;

NOTE 3 Sequence number is considered “high” protection against “Unintended message repetition”, “Message loss”, “Insertion of messages”, “Re-sequencing” and “Medium” protection against “Constant transmission of messages”.

e) error detection codes;

NOTE 4 Cyclic Redundancy Checks are used as error detection codes if the residual error rate of the CRC implemented in the bus system is considered not to be sufficient. In this case an additional CRC at the application level is recommended.

NOTE 5 Alive Counter and CRC are transmitted (embedded in the frame for instance) and checked by the receiver.

f) error correction code;

g) message repetition;

NOTE 6 Message repetition is considered “high” protection against “Message loss”, “Medium” protection against “Re-sequencing”, and “Message corruption”.

h) loop back;

i) acknowledge;

NOTE 7 Acknowledge is considered “high” effectiveness protection against the fault effect “Wrong communication peer”.

j) separated point-to-point unidirectional communication objects;

NOTE 8 Exactly two uni-directional communication objects are used between two partitions respectively for data exchange.

NOTE 9 Method j) is considered “Medium” effectiveness protection against the fault effect “Wrong communication peer”.

k) unambiguous bidirectional communication object;

NOTE 10 Unambiguous bidirectional communication object uses unique numbers for identifying communication peers and/or acknowledges receipt of messages by the communication peer.

NOTE 11 Method k) is considered “medium” effectiveness protection against the fault effect “Wrong communication peer”.

l) asynchronous data communication; and

NOTE 12 In using asynchronous data communication there is no waiting state completed by the communication itself.

NOTE 13 Method l) is considered “high” effectiveness protection against the fault effect “Blocking of partitions”.

m) synchronous data communication.

NOTE 14 Access is synchronised between both software partitions using shared memory for communication. This can be done e.g. using semaphores.

NOTE 15 Communication objects in a) and j) between software partitions are e.g. pipes, message queues, shared memory. These communication objects cannot to be used for synchronizing partitions.

NOTE 16 Blocking read or write access is to be avoided by design when using message queues.

For bus allocation within a microcontroller network the following methods have to be considered:

a) time-triggered data bus;

NOTE 1 Time-triggered data bus is considered “high” protection against “Failure of communication peer”, “Insertion of messages”, “Message delay”, and “Medium” protection against “Blocking access to data bus”.

b) event-triggered data bus;

c) event-triggered data bus with time-triggered access;

d) mini-slotting;

NOTE 2 Mini slotting is considered “high” protection against “Constant transmission of messages” and “Medium” protection against “Message delay”.

NOTE 3 Mini-slotting (see [ARINC 629]) requires each micro controller connected to the bus to wait a certain period before it is permitted to access the bus again.

e) bus arbitration by priority;

f) bus guardian.

NOTE 4 Bus guardian is considered “high” protection against “Blocking access to data bus”, “Constant transmission of messages” and “Medium” protection against “Message corruption”.

# Bibliography

[1] ISO/IEC 12207:2008, Systems and software engineering — Software life cycle processes.

[2] MISRA C Guidelines for the use of the C language in critical systems, ISBN 978-0-9524156-2-6, MIRA, Oct. 2004.

[3] MISRA AC AGC Guidelines for the application of MISRA-C:2004 in the context of automatic code generation, ISBN 978-1-906400-02-6, MIRA, November 2007.

[4] IEC 61508-SER:2005, Functional safety of electrical/electronic/programmable electronic safety-related systems — all parts.

[5] IEC 61508-3:1998 Functional safety of electrical/electronic/programmable electronic safety-related systems — Part 3: Software requirements.

[6] ARINC 629 Backplane data bus, ARINC, December 1993.