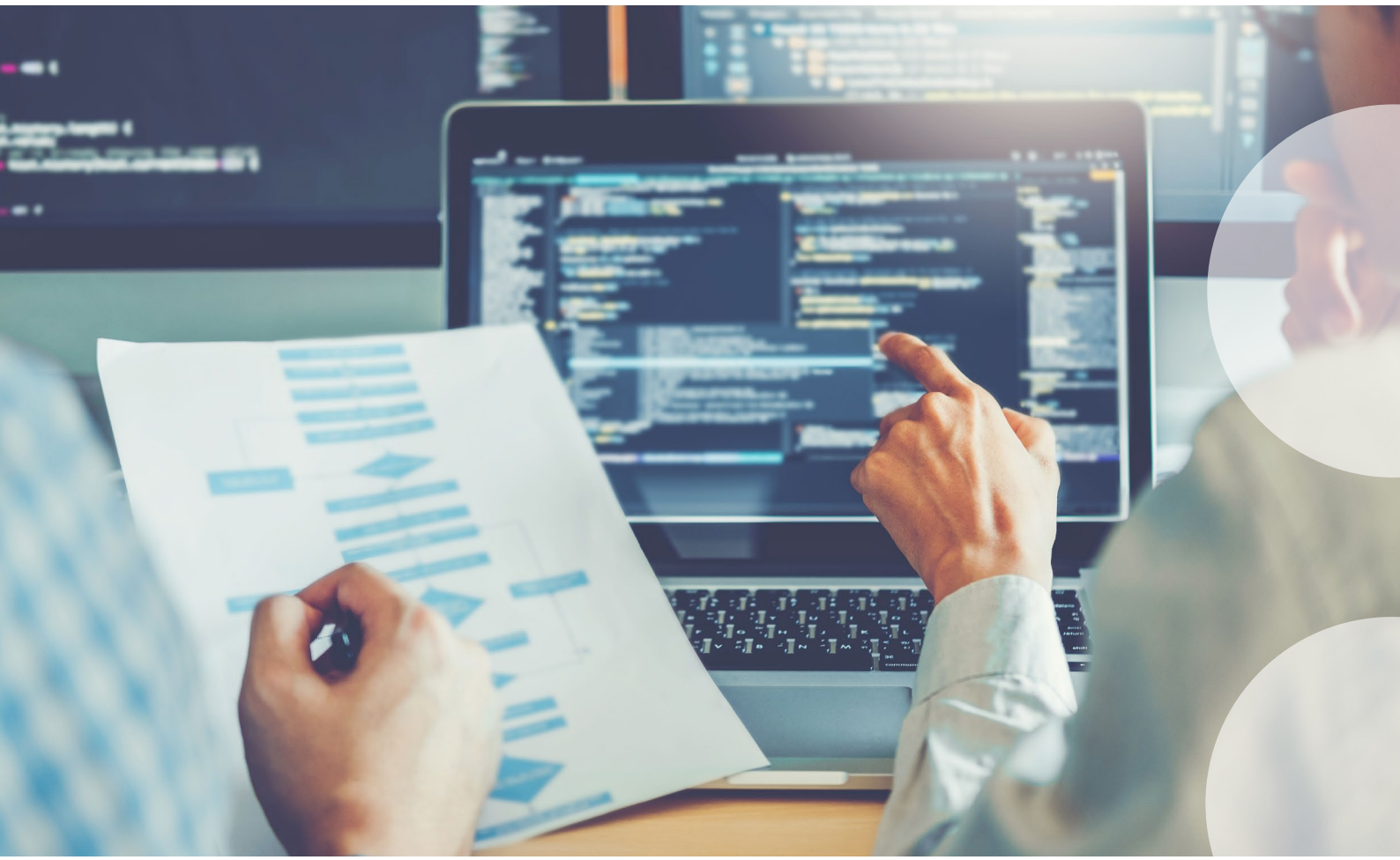
DGUV
Deutsche Gesetzliche Unfallversicherung
Spitzenverband



IFA Report 1/2020e

Practicable implementation of the requirements concerning safety-related embedded software to EN ISO 13849-1

Authors: Thomas Bömer, Karl-Heinz Büllesbach, Michael Hauke, Stefan Otto, Christian Werner;
Institute for Occupational Safety and Health of the German Social Accident Insurance (IFA),
Sankt Augustin

Published by: Deutsche Gesetzliche Unfallversicherung e. V. (DGUV)
Glinkastr. 40
10117 Berlin
Germany
Phone: 030 13001-0
Fax: 030 13001-9876
Internet: www.dguv.de
E-mail: info@dguv.de

– August 2021 –

Database of publications: www.dguv.de/publikationen

ISBN (online): 978-3-948657-00-0
ISSN: 2190-7994

Abstract

Practicable implementation of the requirements concerning safety-related embedded software to EN ISO 13849-1

This report is intended for software designers producing and checking safety-related embedded software (SRESW) for machinery in the context of EN ISO 13849-1. The explanations assist in interpretation of the normative requirements and are intended as recommendations and guidance through the various phases of the software safety life cycle.

Résumé

Mise en pratique des exigences relatives aux logiciels intégrés relatifs à la sécurité selon EN ISO 13849-1

Ce rapport s'adresse aux concepteurs de logiciels qui élaborent et contrôlent les logiciels intégrés relatifs à la sécurité (SRESW) pour des machines, conformément à la norme EN ISO 13849-1. Les commentaires servent à l'interprétation des exigences normatives et visent à servir de recommandation et de ligne directrice pour le déroulement des différentes phases du cycle de vie des logiciels.

Resumen

Implementación práctica de los requisitos de Embedded Software para funciones de seguridad según EN ISO 13849-1

Este informe va dirigido a desarrolladores de software que crean y verifican Embedded Software para funciones de seguridad (SRESW) para maquinaria en el marco de la normativa EN ISO 13849-1. Las explicaciones sirven para facilitar la implementación de los requisitos normativos y pretenden ser una recomendación y una referencia sobre el camino a seguir a lo largo de las diversas fases del ciclo de vida útil del software.

Kurzfassung

Praxisgerechte Umsetzung der Anforderungen für sicherheitsbezogene Embedded-Software nach EN ISO 13849-1

Dieser Report richtet sich an Software-Entwickler, die sicherheitsbezogene Embedded-Software (SRESW) für Maschinen im Rahmen der EN ISO 13849-1 erstellen und überprüfen. Die Erläuterungen dienen der Interpretation der normativen Anforderungen und sollen eine Empfehlung und Richtschnur für den Weg durch die verschiedenen Phasen des Software-Sicherheitslebenszyklus sein.

Contents

1 Introduction

This report is intended for software designers producing and reviewing safety-related embedded software (SRESW) for machinery within the scope of EN ISO 13849-1 [1]. The normative requirements relating to these tasks are stated in clauses 4.6.1 and 4.6.2 of the standard. However, the requirements are formulated in very general and concise terms. Whereas very detailed guidance on the programming of safety-related application software (SRASW) is already available from the IFA [2 to 4], the implications of the normative requirements for implementation of SRESW in practice are still often unclear.

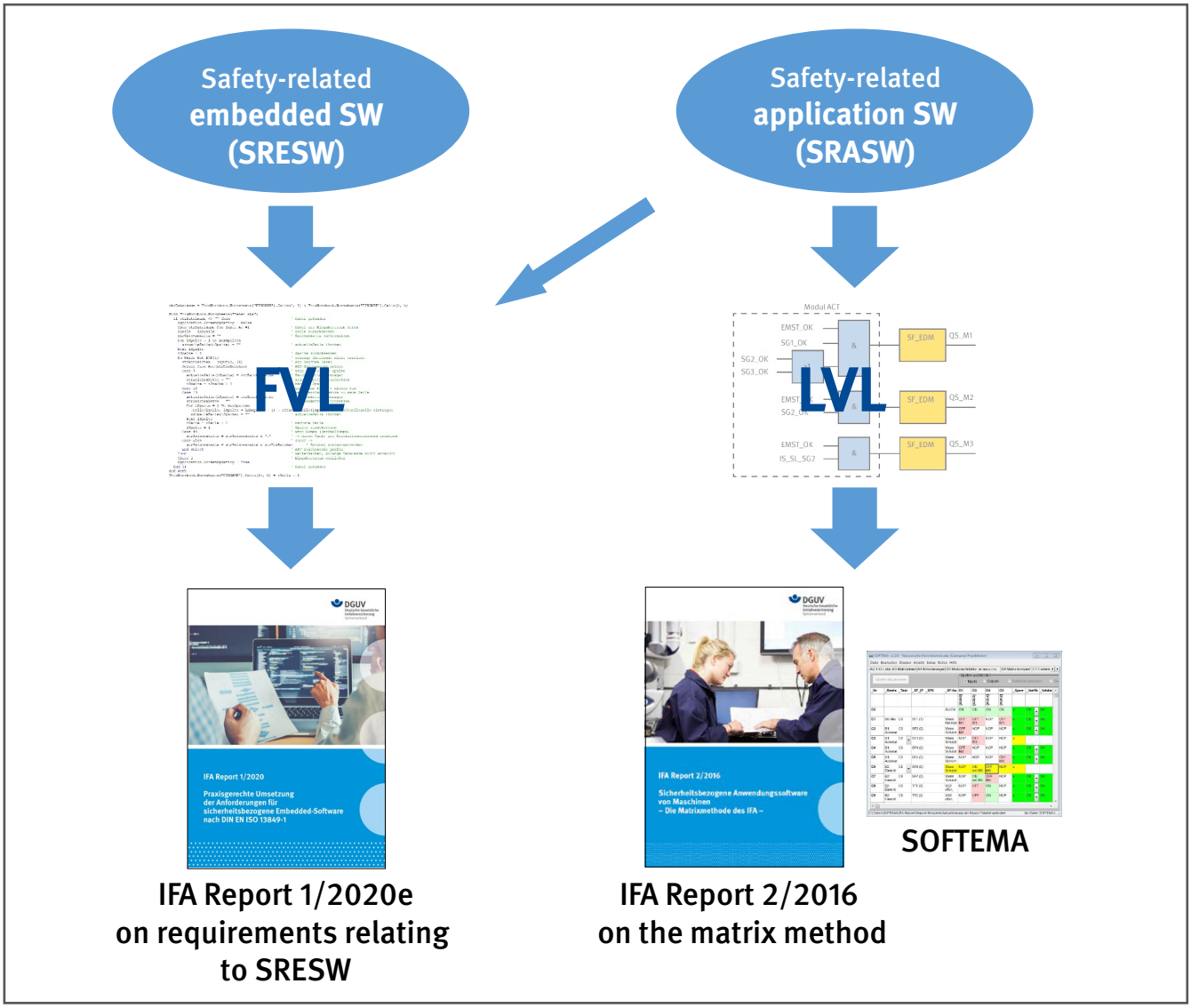
The explanations below assist in interpretation of the normative requirements and are intended as recommendations and guidance through the various phases of the software safety life cycle.

Some of the content presented below has already been addressed in the revision work currently being conducted for the standard's forthcoming fourth edition; not, however, in the level of detail described here.

Since the requirements specified for SRESW in the standard also apply to application software programmed in a full variability language (FVL), the present report also applies to the programming of such software.

Figure 1 provides an overview of the guidance already published by the IFA for the programming of safety-related software.

Figure 1:
Support from the IFA for the programming of safety-related software (SW: software)



2 Software design to EN ISO 13849-1

2.1 V model of the software safety life cycle

To illustrate the design and verification process for safety-related software, clause 4.6.1[1] of EN ISO 13849-1 uses the V model (Figure 2). The basic principle of the V model is that software design and testing are interrelated activities of equal value. This is presented figuratively by the descending and ascending branches of the letter "V".

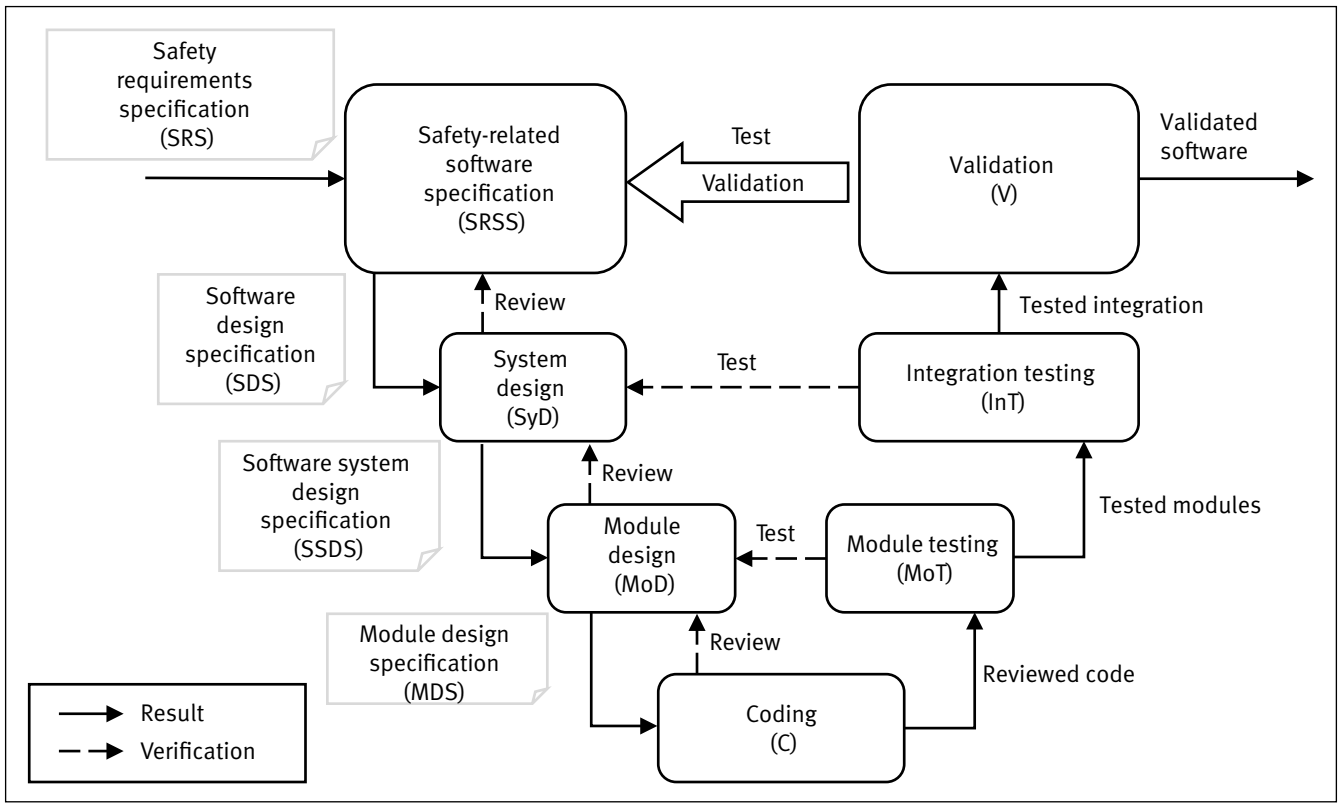
The left-hand, descending branch represents the individual design steps: from the safety requirements specification (SRS) [5], through the design phases at system and module level, to the actual coding. The right-hand, ascending branch represents the corresponding tasks in the integration and test phases. During these phases the modules are combined to form larger subsystems, a process accompanied by continuous testing. Finally, the modules are transferred to the target hardware, where the system as a whole is validated against the requirements relating to its safety functionality. Documentation is an important part of design in accordance with the V model.

The output of each level of software design (descending branch) is a specification of the requirements for the subsequent level. At the same time, the specification is the reference point against which the software is tested at each integration level following coding.

The objective of design under the V model is to obtain readable, understandable, testable and maintainable software. Continual reviews and testing at all levels of the software safety life cycle further permit validation of software design that is as complete and free of errors as possible.

The V model cannot be applied to the embedded software of purchased standard components, since such software was not designed specifically for use in safety functions, nor is it accessible to the designer of safety-related software. Alternative requirements for this scenario are described at the end of clause 4.6.2 in [1]; they do not however fall within the scope of this report.

Figure 2:
Software safety life cycle under the V model (refer to Table 1 for an explanation of the abbreviations)



1 All references are to the english edition of the standard in the version in force at the time of publication of this document (December 2015).

Table 1:
Explanation of the abbreviations in Figure 2

Abbreviation	Meaning	Result
SRSS	Safety related software specification: specification of the software with consideration for the safety functions to be analysed against the requirements of the safety requirements specification (SRS)	Software design specification (SDS)
SyD	System design: Design of the software with consideration for the safety-related hardware structure in accordance with the requirements of the software design specification (SDS)	Software system design specification (SSDS)
MoD	Module design: Design of the individual software modules in accordance with the requirements of the software system design specification (SSDS)	Module design specification (MDS)
C	Coding: Coding of the individual software modules in accordance with the requirements, followed by review for compliance with the module design specification (MDS)	Reviewed program code of the modules (including comments)
MoT	Module testing: Testing of the modules for compliance with the module design specification (MDS)	Tested modules
InT	Integration testing: Testing of the software interfaces and, if possible, of the software as a whole as transferred to the target hardware for compliance with the software system design specification (SSDS)	Tested integration
V	Validation: Review of the software and, if possible, the hardware against the software design specification (SDS) and the safety requirements specification (SRS)	Validated software

Note: Figure 2 is based on Figure 6 of EN ISO 13849-1. The design and test phases shown there have been expanded to include the associated specification documents and a more detailed classification of review and test steps. In the next chapter, the abbreviations after the design and test steps have the function of referencing the measures described to the phases of the V model.

2.2 Basic and additional measures

In addition to the V model, clause 4.6.2 of [1] lists measures to be applied in accordance with the required performance level (PL_r) during the design of safety-related software:

Basic measures (for components with a PL_r of a to d):

- Software safety life cycle with verification and validation (1.1)
- Documentation of specification and design (1.2)
- Modular and structured design and coding (1.3)
- Control of systematic failures (1.4)

- Where software-based measures are used to control random hardware failures: verification of correct implementation (1.5)
- Functional testing such as black box testing (1.6)
- Suitable activities for the software safety life cycle following changes (1.7).
- Additional measures (for components with a PL_r of c to d):
- Project and quality management comparable for example to that of the EN 61508 or ISO 9001 (2.1) series
- Documentation of all relevant activities during the software safety life cycle (2.2)
- Configuration management for identification of all configuration characteristics and documents relating to release of an item of SRESW (2.3)
- Structured specification containing safety requirements and design (2.4)
- Use of appropriate programming languages and computer-based tools with confidence from use (2.5)
- Modular and structured programming, separation from non-safety-related software, modules of limited size with fully defined interfaces, use of design and coding standards (2.6)
- Verification of the code by walk-through review with control flow analysis (2.7)
- Extended functional testing (e.g. grey box testing), performance testing or simulation (2.8)
- Impact analysis and appropriate software safety life cycle activities following changes (2.9)
- **Note:** The number in brackets after each measure indicates its reference in clause 4.6.2 of [1]. The first number indicates whether the measure is a basic measure (1) or an additional measure (2) according to the standard. The second number indicates the respective referenced bullet point in the relevant paragraph. For example, the reference 1.2 indicates the basic measure: “Documentation of specification and design”.

Since in the standard, the measures are listed uncommented, it is not always readily apparent to design and testing personnel what activities at what point in the software’s life cycle are to be inferred from the measures.

The next chapter of this report (Chapter 3) assigns the measures from clause 4.6.2 of [1] to the individual design and test phases of the V model. The activities to be inferred from each measure are also explained.

Generic measures which cannot be assigned under the V model are listed separately in Chapter 4.

Since some individual measures may be assigned to multiple phases of the V model, the relevant text from the standard is abridged in the below table to the respective relevant part; refer for example to the aforementioned basic measure: “Documentation of specification” in Section 3.1.

Note: For software with a PL_r of e, [1] refers to the requirements relating to SIL 3 in clause 7 of EN 61508-3 [6]. An exception is category 3 and 4 architectures, in which the two channels exhibit diversity in their specification, design and coding. In this case, application of the basic and additional measures described in this document is sufficient.

3 Measures in the software safety life cycle under the V model

3.1 Safety-related software specification (SRSS)

Reference	Measure	Explanation
SRSS_1	Documentation of specification (1.2)	The software design specification is to be drawn up based on the safety requirements specification (SRS), the content of which includes a detailed description of the safety functions. Independently of the specific implementation, the software design specification describes in general terms what functions the software is to perform in order to implement the safety functions. The software design specification is expected to have a structured form. This forms the basis for the subsequent phase, that of system design. To this end, its content also includes the definition of the computer-aided tools with confidence from use which are to be used (see C_5, MoT_2, InT_3).
SRSS_2	Structured specification of safety requirements and structured design (2.4)	Both a structured specification and structured design are specified. The SRSS_2 measure does not give rise to any additional activities beyond SRSS_1. It is expected that the safety requirements applicable to the software will already be provided through the software design specification (SDS). Planning of design should be reflected in project management (see QM_2). Subsequent performance of structured design is an automatic consequence of application of the V model (see QM_1).

3.2 System design (SyD)

Reference	Measure	Explanation
SyD_1	Documentation of design (1.2)	The software system design specification (SSDS) is to be produced. This describes implementation of the specified safety functions at system level in the software. The SSDS is to describe what modules are provided for and how they interact within the software architecture. The software system design specification thus forms the basis for module design. The test of the system design including the software-based measures for control of random hardware failures is to be planned and documented (see InT_1).
SyD_2	Documentation of all relevant activities during the software safety life cycle (2.2)	Supplementary to SyD_1, review of the system design against the software design specification is to be documented. Documentation of all relevant activities during this design phase assists in the avoidance of errors during creation of the safety-related software, and supports its evaluation. The documentation is to be written in understandable and natural language, supported by pictorial representations and a glossary of technical terms.
SyD_3	Modular and structured design (1.3)	The objective is to create a hierarchical description of partial requirements (“modularization”) for the safety-related functionality intended for implementation, i.e. a description extending from the coarse to the fine. This enables these partial requirements to be coded individually and verified. The use of interrupts is also to be decided as part of the system design process. Consideration is to be given to the following aspects: • Description and limitation of functionality: ideally, each software module implements a single, defined function to be performed. • Interfaces (input and output): the connections between the software modules must be limited, defined unambiguously and described clearly. • The call relationships between the individual modules must be unique. “Polymorphic” call structures, i.e. in which the module/method to be called is not determined until runtime, are to be avoided. The reasoning for exceptions is to be stated and they are to be documented transparently.

3.3 Module design (MoD)

Reference	Measure	Explanation
MoD_1	Documentation of design (1.2)	The module design specification is to be drawn up. The module design specification describes implementation of the safety-related functions in the coding of the individual modules. The description should be supported by flow charts, flow diagrams, data flow diagrams, time sequence charts, etc. Individual approaches are to be described here in detail. Where necessary, details are to be provided of the hardware-specific software environment, for example RAM/ROM ranges, I/O address ranges, the use of interrupts and their vectors, hardware reaction times (for example in relation to input filters), communication and protocol with a parallel channel, where present. The module design specification thus forms the basis for coding.
MoD_2	Documentation of all relevant activities during the software safety life cycle (2.2)	In addition to MoD_1, review of the module design against the software system design specification is to be documented. See also SyD_2.
MoD_3	Modular and structured design (1.3)	Implementation of the safety functions described in the software system design specification is to be substantiated by division into individual, self-contained subroutines (software modules). This step permits modular, structured coding (see C_3). Structured coding refers to the internal structure of a program or part of a program, whilst modular coding considers its external characteristics (such as interfaces). Besides the reduction of errors, the objective here is to reduce the complexity of a program and to avoid poor or confusing program structures. Programs should be easy to read, maintain and modify. These objectives give rise to the following requirements: • The program should be divided into reasonably small software modules (modularization). • The flow of a program should contain only the following constructs: – sequence – iteration – selection • Accordingly, the graphical representation of the program's control flow should contain only the following elements (see also MoT_3): – branch nodes – statements – merge nodes • The paths through a software module should be as few in number as possible, and relationships between input and output parameters should be simple. • The use of complicated calculations as the basis for branching and loop conditions should be avoided. • Complicated branching and, in particular, unconditional jumps (goto) should be avoided; these give rise to unstructured code. The use of dead code is not permissible. • In order for real-time influences to be reduced to a minimum and clarity of the program sequence improved, interrupts should not be used. Deviations from this design principle are permissible only where they significantly simplify the program. The reasons must be documented on a case-by-case basis. • Modular coding is characterized by constraints upon the size of software modules and by fully defined interfaces. Corresponding requirements can be found in MoD_5. For assurance of these properties, a set of rules is to be laid down in the design phase which are to serve as binding coding rules during the subsequent phase of coding of the safety-related software. Corresponding requirements can be found in MoD_6.

Reference	Measure	Explanation
MoD_4	Separation from non-safety-related software (2.6)	During subsequent coding, care is to be taken to prevent non-safety-related software from impairing the execution of safety functions. To this end, rules and design and coding standards are to be laid down during the module design phase to facilitate encapsulation of the safety-related software. Safety-related software refers to a safety function implemented in the form of software. It begins at the point in the software as a whole at which a specified, safety-related item of data is read. It further contains the part of the software that processes the item of data, and ends at the point at which a safety-related item of data is output. The safety-related software includes management of the relevant data (e.g. declaration, initialization). Safety-related software also includes the software implementations of test routines that are required for adequate fault detection (diagnostic coverage) of the functional channels.
MoD_5	Limited module size with fully defined interfaces (2.6)	Software modules should have a defined size limit and fully defined interfaces. This gives rise to the following specific requirements: • Ideally, a software module should perform a single task or fulfil a single function. • Interfaces between software modules should be limited and fully defined; each interface of a software module should, if possible, contain no parameters other than those required for it to fulfil its function. • A fully defined interface (for call parameters and return values) includes (list not exhaustive): – number of parameters passed – data types – parameter value ranges provided for • Software modules should communicate with other software modules solely through their interfaces. Variables in a module should be private. Should global or shared variables be used, they should be well structured, access to them should be controlled, and their use should always be justified. The use of meaningful names for global variables is advantageous. • All interfaces of the software modules should be fully documented (including in the code listing). • A software module should not exceed a certain size (recommended maximum: 50 lines); an exception to this is sequential code of the same functionality (such as a self-test of the individual commands of a microprocessor). • The nesting depth of module calls is to be kept as low as possible (a maximum nesting depth of five calls is recommended). • Software modules should have only one input and one output (an additional output can be provided for error handling).
MoD_6	Use of design and coding standards (2.6)	Creation and observance of coding standards is absolutely essential for coding. The use of coding standards reduces the likelihood of errors and at the same time facilitates verification at a later stage. The coding standards should include all the coding requirements stated in the sections above. The coding standards should cover at least the following areas and support them with specific requirements: • Modular approach: limitation of the size of software modules; fully defined interfaces (see MoD_5) • Comprehensibility of the code: for example, unique and coherent naming of variables; standards for documenting the code (see C_1) • Verifiability and testability: for example, protection mechanisms for critical library functions, avoidance of dead code • Good programming technique: for example, checking of pre- and post-conditions and return conditions, catching of invalid states

3.4 Coding (C)

Reference	Measure	Explanation
C_1	Documentation of design (1.2)	The module design specification serves as the basis for coding. Documentation of design refers to provision of the code listings, including comments. Adequate commenting of the code is absolutely essential, in order for it to be readable, comprehensible, testable and maintainable. Commenting of the code should provide, for example, a description of the algorithms and other special features and a module header. The module header should contain at least the following information (refer to the example in Appendix A): • Name of the author • Description of the objective and function in the overall context of the module (not merely the content) • Inputs and outputs • Configuration management history
C_2	Documentation of all relevant activities during the software safety life cycle (2.2)	In addition to C_1, review of the coding (see C_6) against the module design specification is to be documented. See also SyD_2.
C_3	Modular and structured coding (1.3; 2.6)	The measures set out in the module design (see MoD_3) are to be implemented at code level, if necessary with iterations.
C_4	Use of suitable programming languages (2.5)	The suitability of the programming language used relates to demonstration of the uniqueness and traceability of each program or program part within its program runtime. For example, unambiguous definition of a variable or of the behaviour of the module under analysis must be possible at all points during a walk-through. Where appropriate, the variability of the programming language should be limited by the exclusion of language constructs that are particularly prone to error.
C_5	Use of suitable computer-aided tools with confidence from use (2.5)	The use of computer-aided tools in the coding process is a necessary and important measure for the avoidance of errors. The tools to be used are defined in the software design specification (see SRSS_1). The term “confidence from use” is understood in this context to mean that the programmers have been trained in use of the tools and the development environment, and that the tools have been used in past projects without giving rise to faults. Evidence to this effect must be furnished.
C_6	Verification of the code by a walk-through/review (2.7)	Upon completion of coding, the individual modules are to be reviewed against the module design specification in a code review or code walk-through. The review is to be performed by the programmer(s) together with one or more persons with equivalent technical expertise. The objective of the review or walk-through is to check the code for the following: • Errors or potential errors • Quality of the comments • Compliance with coding standards (a tool may also be used for this purpose) • Clarity and readability • Completeness in terms of the functionality to be implemented A walk-through and a review are considered identical in terms of the measures to be performed.

3.5 Module testing (MoT)

Reference	Measure	Explanation
MoT_1	Documentation of all relevant activities during the software safety life cycle (2.2)	The module testing (see MoT_3 to MoT_5) is to be documented. The objective is to demonstrate that all requirements emanating from the module design specification have been implemented. See also SyD_2.
MoT_2	Use of suitable computer-aided tools with confidence from use (2.5)	Tools, wherever possible computer-aided, are also required during module testing as an important measure for preventing errors. The tools to be used are defined in the software design specification (see SRSS_1). For the term “confidence from use”, see C_5. Confidence from use must be demonstrated and documented accordingly, and refers here specifically to use during the module test.
MoT_3	Verification of the code by control flow analysis (2.7)	The control flow of the program – in this case, of a module – is to be analysed by means of a control flow graph, and evaluated with reference to the criteria stated below. Control flow analysis is a tool used for static code analysis and has the purpose of revealing possible structural flaws within a program flow. A control flow graph is a directed graph with an initial and a final node (initial and final statements of the module). Between the initial and final nodes are further nodes (statements), which are connected to each other by “edges”. The edges describe the possible control flow between the statements. Besides the initial and final nodes, the pictorial representation of the control flow may contain only the following elements: a) Statements (nodes with only one incoming and one outgoing edge each) b) Branches (nodes with at least two successor nodes) c) Merges (nodes with at least two incoming edges) The control flow graph must be reducible in steps to a single node. Should this not be possible, the code is not well structured and should be optimized.
MoT_4	Functional testing, for example black box testing (1.6)	Functional testing is to be performed at module level to verify compliance with the module design specification. The development environment for example can be used for this purpose. A test bed may need to be programmed for performance of the tests. Test data, pre- and post-conditions where applicable, and test results (e.g. outputs, changes in internal states, etc.) are to be logged. The purpose of module testing is to ensure that the modules satisfy the functionality required of them as defined in the module design specification. The functionality of a module equates in this context to its input and output behaviour. To verify this, input data adequately reflecting the expected function are supplied to each module. The data should cover at least the following ranges: • Data from permissible ranges • Data from impermissible ranges • Data from range limits Permissible ranges and range limits are to be determined from the module design specification.

Reference	Measure	Explanation
MoT_5	Extended functional testing, for example grey box testing, performance testing or simulation (2.8)	Besides the input data stated in MoT_4, invalid input data must also be considered for safety-related software in PL c and d. It is to be determined whether these data are intercepted by the programming of the module, or lead to undesired or unanticipated reactions. This concerns, for example, the following inputs: • Divisor of zero • ASCII space • Empty stack or empty list element • Full matrix • Empty table entry In test runs for error detection, as much of the code as possible should be executed. For this purpose, coverage tests are to be performed based on the elements of the control flow graph (MoT_3). The following test methods are to be distinguished: • Statement testing • Branch or decision testing • Path testing • Condition testing If possible, the objective should be for 100% coverage to be achieved, as no conclusion can be drawn regarding whether statements that have not been executed are correct. The existence of statements that cannot be executed by any test case may indicate dead code. Should testing reveal unconditional jumps, the code must be modified to avoid them.

3.6 Integration testing (InT)

Reference	Measure	Explanation
InT_1	Where software-based measures are used to control random hardware failures: verification of correct implementation (1.5)	Software-based measures may be necessary to control random hardware failures in order to satisfy the required diagnostic coverage, depending on the selected system structure (categories 2, 3 and 4). Software-based measures include, for example: • RAM, ROM and CPU tests • Tests of hardware components • Plausibility tests of calculated and input values Where such software-based fault detection measures are necessary, their efficacy and correct implementation are to be verified. For this purpose, the test cases defined in SyD_1 must be executed in order for the respective triggered test routines to be verified. The result of this verification is to be documented. The standard does not require control of random hardware failures for implementations in category B (PL a or b). Software-based measures for this purpose are therefore not normally implemented.
InT_2	Documentation of all relevant activities during the software safety life cycle (2.2)	The performance and results of the integration testing (see InT_1 and InT_4 to InT_6) are to be documented. The objective is demonstration that all requirements emanating from the software system design specification have been implemented. See also SyD_2.
InT_3	Use of suitable computer-based tools with confidence from use (2.5)	The use of computer-based tools is an important measure in integration testing for the avoidance of errors. The tools to be used are defined in the software design specification (see SRSS_1). For the term “confidence from use”, refer to C_5. Confidence from use must be demonstrated and documented appropriately, and refers at this point specifically to use in the context of integration testing.
InT_4	Verification of the code by control flow analysis (2.7)	At the level of the software as a whole, static control flow analysis of the various modules is to be performed with the aid of a call graph. The call graph has the function of displaying the call relationships between the individual modules. Excessively complex call structures should be avoided. Where polymorphic call structures have been used, all possible forms of a call must be analysed with reference to the available documentation (see SyD_3).

Reference	Measure	Explanation
InT_5	Functional testing, for example black box testing (1.6)	Integration testing in this phase concerns merging of the individual software modules to form a complete piece of software (integration testing at software level), and – if possible – integration of the software as a whole on the target hardware (integration testing at hardware level). The tests described below are to be used to verify the software's compliance with the software system design specification. Performance of integration testing is conditional upon the individual components already having been tested in the course of coding and performance of the module testing, and any defects already having been corrected. The components are then combined to form increasingly large subsystems. Testing on the software level concerns the interfaces and interaction between the individual components. Since the subsystems as such may not be executable until the software as a whole has been completed, a test framework (see also MoT_4) may have to be created for the components that are still missing. Integration-level testing is intended to detect the following errors: • Incompatible interface formats and protocol errors • Conflicts in data interpretation • Transfer of data insufficiently swiftly or at excessively short intervals The test environment for integration testing on the hardware level should resemble the anticipated operating environment as closely as possible. Accordingly, the software and hardware peripherals installed as the test environment should be as similar as possible to the peripherals that are actually to be used at a later stage. The system is to be supplied with input data that adequately reflect anticipated normal operation. Functional testing is to cover all realistically foreseeable input conditions and deliver the specified results (including for example the reaction time). The results are to be monitored and compared with the requirements set out in the software system design specification. Where applicable, requirements not previously set out in writing are to be added.
InT_6	Extended functional testing, for example grey box testing, performance testing or simulation (2.8)	Extended functional testing must be performed in addition to the tests described in InT_5. The behaviour of the specified safety functions in the event of uncommon or unspecified inputs is to be verified in the course of the extended functional testing. The following may for example be performed/executed: • Functional testing with input conditions which is rarely anticipated or which lie outside the specified properties of the safety functions • Functional tests involving boundary conditions under limit conditions (such as extreme ambient conditions, component dimensioning at the limit values, maximum thermal loading of components, maximum processor loading) • Functional testing based on knowledge of the internal coding of the software as a whole • Functional testing of hardware elements of the safety functions (such as electronic circuits) employing software simulation

3.7 Validation (V)

Reference	Measure	Explanation
V_1	Documentation of all relevant activities during the software safety life cycle (2.2)	Validation (see V_2) is to be documented. The objective is to demonstrate that all requirements emanating from the software design specification have been implemented. See also SyD_2.
V_2	Functional testing (1.6)	In the sense of the standard, validation constitutes the final check of the tests performed and their results against the requirements of the software design specification in the individual phases of the V model. Should integration already have taken place on the target hardware, validation also includes checking against the safety requirements specification (SRS). In the case of discrete components, validation is to be performed prior to placing on the market. As a rule, functional testing solely for validation purposes is generally no longer performed; integration testing and validation are combined in this case. In the case of machinery, validation forms part of installation and acceptance testing at the operator's premises. Checking against the safety requirements specification (SRS) can be performed at this point by way of additional functional testing in the real environment.

4 General measures

4.1 Quality management (QM)

Reference	Measure	Explanation
QM_1	Software safety life cycle including verification and validation, application of the V model (1.1)	The V model (Figure 2) is to be applied with the objective of structuring design of the software and demonstrating its suitability in defined phases and activities. This progressively gives rise to more detailed specifications in the descending branch of the V model. Following coding of the individual software modules, they are integrated and tested in the ascending branch.
QM_2	Project management and quality management system comparable with, for example, the EN 61508 series or EN ISO 9001 (2.1)	Project and quality management are to be applied in order for errors to be avoided during software design. Important measures for implementation of this requirement are: • Generation of a quality assurance manual describing an organizational model specifically for quality assurance • Definition of how design is to be organized, including definition of the tasks and responsibilities of the organizational units, competences of the quality assurance department, independence of quality assurance (internal inspection) from design • Definition of the schedule (phase models) specifying all activities that are relevant during performance of the project, including internal inspections and their scheduling, and updating of the project • Definition of a set procedure for internal inspection covering planning, performance and review of inspection, release mechanisms for sub-products, assurance of regular inspections • Configuration management including version management and control according to QM_3 • Introduction of computer-aided tools, conducting and demonstration of staff training • Consistent application of the second pair of eyes principle Where software is designed in accordance with EN 61508, the requirements stated are already taken into account. Where the introduction and operation of a quality management system to EN ISO 9001 has already been demonstrated, the measures stated must be implemented specifically for design of an item of software.
QM_3	Configuration management (2.3)	In order for errors to be avoided during the design of safety-related software and to permit subsequent review of the software as a whole with consideration for the safety functions under analysis, software configuration management is to be applied. Procedures must be in place: a) For the unique identification of all components of a software project, including but not limited to: • Version management of the source code of the individual software modules and the software as a whole • Specification of the software and design documents • Test plans and test results • Verification documents • Any software elements already in existence • Computer-aided tools and development environments used to produce the software b) For management of software modifications, to ensure that the specified requirements continue to be met following changes, including measures to: • Prevent unauthorized modifications • Prevent the use of non-approved components • Analyse the influence of intended modifications • Release intended modifications c) Retain the software and all related documentation In its simplest form, software configuration management can be performed manually. The use of suitable computer-aided tools and development environments for this purpose is however more intelligent and effective. d) Release of the software as a whole following completion of the design process

4.2 Systematic failures (SyF)

Reference	Measure	Explanation
SyF_1	Control of systematic failures (1.4)	In some cases, which are stated in EN ISO 13849-1, clause G.2, software engineering measures are used to control systematic failures. To verify that they have been implemented correctly and in full, the measures to be applied for the project are to be described in the software design specification based upon the safety requirements specification (SRS), and their implementation is to be documented in the software system design specification and module design specification. These requirements concern the following measures stated in clause G.2: • Program sequence monitoring: Various forms of monitoring may be possible as a function of the hardware structure selected, and based on the PL_r. Examples are an external hardware watchdog triggered from within the software, or temporal and logical monitoring of the program sequence implemented by communication with a parallel channel. Irrespective of how monitoring is implemented, the safe state (such as the de-energized state) must be assumed when a fault is detected. • Measures to control errors in the data communication process: Where execution of a safety function requires secure data communication (for example over bus systems), measures must be taken to control the following error models for safety-related messages: – repetition – loss – insertion – incorrect sequence – corruption – delay – masquerade For further details of safe data communication, refer to EN 61784-3 [7]. Correct implementation of measures for the control of systematic failures is to be considered and verified during each phase of the V model, particularly during the specification and validation phases.

4.3 Modification management (MM)

Reference	Measure	Explanation
MM_1	Suitable activities for the software safety life cycle following changes (1.7)	Before changes are made to the software, an analysis of the possible impacts should be performed and the affected software modules identified. An impermissible impact could for example be a reduction in the specified properties of the safety functions, for example in terms of reaction time or behaviour. Following performance of the impact analysis, it is to be determined, in consideration of the PL_r, what tests are to be repeated on what levels of the V model once the modification has been completed. This can be achieved with reference to the following criteria: • Repetition only of the tests designated as being of high priority • Where functional testing is performed, waiving of certain variants (special cases) • Limitation of the tests to certain configurations • Limitation of the tests to certain subsystems In order to put the conditions in place for evaluation of these criteria, detailed documentation of, reasoning for and evaluation of the test cases performed (for each software module) is required at the test levels of the V model (Sections 3.5 to 3.7). This should also identify the tests to be considered should repetition be required following changes.
MM_2	Impact analysis and appropriate software safety life cycle activities following changes (2.9)	Performance of the impact analysis described in MM_1 is imperative. The criteria stated in MM_1 are to be interpreted more strictly owing to the higher PL_r, as a result of which a greater number of tests must be repeated. The verification steps performed are to be documented.

Bibliography

[1] EN ISO 13849-1: Safety of machinery – Safety-related parts of control systems – Part 1: General principles for design (2015).

[2] *Huelke, M.; Becker, N.; Eggeling, M.:* Sicherheitsbezogene Anwendungssoftware von Maschinen – Die Matrixmethode des IFA. IFA Report 2/2016. Published by: Deutsche Gesetzliche Unfallversicherung e. V. (DGUV), (2016). www.dguv.de, Webcode: d1023063

[3] Software-Assistent SOFTEMA: Spezifikation zur IFA-Matrixmethode bei sicherheitsbezogener Anwendungssoftware. Published by: Deutsche Gesetzliche Unfallversicherung e. V . (DGUV), Sankt Augustin, www.dguv.de, Webcode: d1082520

[4] *Hauke, M.; Schaefer, M.; Apfeld, R.; Bömer, T.; Huelke, M.; Borowski, T.* et al.: Functional safety of machine controls - Application of EN ISO 13849. IFA Report 2/2017e. Published by: Deutsche Gesetzliche Unfallversicherung e. V. (DGUV), (2019). www.dguv.de, Webcode: e1179198

[5] *Apfeld, R.; Hauke, M.; Otto, S.:* The SISTEMA Cookbook 6: Definition of safety functions: what is important? Published by: Institut für Arbeitsschutz der Deutschen Gesetzlichen Unfallversicherung (IFA), Sankt Augustin, Germany (2015). www.dguv.de, Webcode: e109249

[6] EN 61508-3: Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 3: Software requirements (2010).

[7] EN 61784-3: Industrial communication networks – Profiles – Part 3: Functional safety fieldbuses – General rules and profile definitions (2021).

Annex A:
Example module header

Example module header

In accordance with the C_1 basic measure, which is mandatory for a PL_r of a or higher, each module (each method in object-oriented programming) must be preceded by its own module header, which specifies the "who, when, what, why and how". As a minimum, the module header should list the following information:

- Module name
- Brief description of the function
- Version number
- Legend of changes
- Date of last change
- Person responsible
- Input parameters
- Output parameters
- Modified registers and memory ranges
- Exit with error
- Higher-level modules (called by)
- Lower-level modules (calls)
- Stack depth (for example in Assembler)
- Changes to flags (for example in Assembler)

Also, if necessary:

- Use of a register bank (e.g. in Assembler),
- Runtime
- Timeouts
- Particular comments

Example module header

```
/*      ****************************************        */
/*                                                      */
/*      Module:                                         
/*                                                      */
/*      Version: ... Modified by: ... Date:             */
/*      Version: ... Modified by: ... Date:             */
/*      Version: ... Modified by: ... Date:             */
/*                                                      */
/*      Function of the module:                         */
/*                                                      */
/*                                                      */
/*                                                      */
/*      Input parameters:                               */
/*      Output parameters:                              */
/*                                                      */
/*      Modified registers:                             */
/*      Modified memory ranges:                         */
/*      Modified flags:                                 */
/*                                                      */
/*      Exit with error:                                */
/*                                                      */
/*      Higher-level modules:                           */
/*      Lower-level modules:                            */
/*                                                      */
/*      Stack depth:                                    */
/*                                                      */
/*      Register bank used:                             */
/*                                                      */
/*      Particular comments:                            */
/*                                                      */
/*      ****************************************        */
```

Annex B:
List of abbreviations

List of abbreviations	
C	**C**oding
CPU	**C**entral **P**rocessing **U**nit
FVL	**F**ull **V**ariability **L**anguage
InT	**In**tegration **T**esting
LVL	**L**imited **V**ariability **L**anguage
MDS	**M**odule **D**esign **S**pecification
MM	**M**odification **M**anagement
MoD	**M**odule **D**esign
MoT	**M**odule **T**esting
PL	**P**erformance **L**evel
PL_r	**r**equired **P**erformance **L**evel
RAM	**R**andom **A**ccess **M**emory
ROM	**R**ead **O**nly **M**emory
SDS	**S**oftware **D**esign **S**pecification
SRASW	**S**afety **R**elated **A**pplication **S**oft**w**are
SRESW	**S**afety **R**elated **E**mbedded **S**oft**w**are
SRS	**S**afety **R**equirement **S**pecification
SRSS	**S**afety **R**elated **S**oftware **S**pecification
SSDS	**S**oftware **S**ystem **D**esign **S**pecification
SyD	**Sy**stem **D**esign
SyF	**Sy**stematic **F**ailure
QM	**Q**uality **M**anagement
V	**V**alidation

Deutsche Gesetzliche
Unfallversicherung e.V. (DGUV)

Glinkastraße 40
10117 Berlin
Phone: 030 13001-0
Fax: 030 13001-9876
E-mail: info@dguv.de
Internet: www.dguv.de