IEC TR 63201

Edition 1.0 2019-05

TECHNICAL REPORT

colour inside

Low-voltage switchgear and controlgear – Guidance for the development of embedded software

IEC TR 63201:2019-05(en)

IEC TR 63201

Edition 1.0 2019-05

TECHNICAL REPORT

colour inside

Low-voltage switchgear and controlgear – Guidance for the development of embedded software

INTERNATIONAL ELECTROTECHNICAL COMMISSION

ICS 29.130.20 ISBN 978-2-8322-7006-6

CONTENTS

INTERNATIONAL ELECTROTECHNICAL COMMISSION

____

LOW-VOLTAGE SWITCHGEAR AND CONTROLGEAR – GUIDANCE FOR THE DEVELOPMENT OF EMBEDDED SOFTWARE

FOREWORD

1) The International Electrotechnical Commission (IEC) is a worldwide organization for standardization comprising all national electrotechnical committees (IEC National Committees). The object of IEC is to promote international co-operation on all questions concerning standardization in the electrical and electronic fields. To this end and in addition to other activities, IEC publishes International Standards, Technical Specifications, Technical Reports, Publicly Available Specifications (PAS) and Guides (hereafter referred to as "IEC Publication(s)"). Their preparation is entrusted to technical committees; any IEC National Committee interested in the subject dealt with may participate in this preparatory work. International, governmental and non-governmental organizations liaising with the IEC also participate in this preparation. IEC collaborates closely with the International Organization for Standardization (ISO) in accordance with conditions determined by agreement between the two organizations.
2) The formal decisions or agreements of IEC on technical matters express, as nearly as possible, an international consensus of opinion on the relevant subjects since each technical committee has representation from all interested IEC National Committees.
3) IEC Publications have the form of recommendations for international use and are accepted by IEC National Committees in that sense. While all reasonable efforts are made to ensure that the technical content of IEC Publications is accurate, IEC cannot be held responsible for the way in which they are used or for any misinterpretation by any end user.
4) In order to promote international uniformity, IEC National Committees undertake to apply IEC Publications transparently to the maximum extent possible in their national and regional publications. Any divergence between any IEC Publication and the corresponding national or regional publication shall be clearly indicated in the latter.
5) IEC itself does not provide any attestation of conformity. Independent certification bodies provide conformity assessment services and, in some areas, access to IEC marks of conformity. IEC is not responsible for any services carried out by independent certification bodies.
6) All users should ensure that they have the latest edition of this publication.
7) No liability shall attach to IEC or its directors, employees, servants or agents including individual experts and members of its technical committees and IEC National Committees for any personal injury, property damage or other damage of any nature whatsoever, whether direct or indirect, or for costs (including legal fees) and expenses arising out of the publication, use of, or reliance upon, this IEC Publication or any other IEC Publications.
8) Attention is drawn to the Normative references cited in this publication. Use of the referenced publications is indispensable for the correct application of this publication.
9) Attention is drawn to the possibility that some of the elements of this IEC Publication may be the subject of patent rights. IEC shall not be held responsible for identifying any or all such patent rights.

The main task of IEC technical committees is to prepare International Standards. However, a technical committee may propose the publication of a technical report when it has collected data of a different kind from that which is normally published as an International Standard, for example "state of the art".

IEC TR 63201, which is a technical report, has been prepared by subcommittee 121A: Low-voltage switchgear and controlgear, of IEC technical committee 121: Switchgear and controlgear and their assemblies for low voltage.

The text of this technical report is based on the following documents:

Enquiry draft	Report on voting
121A/256/DTR	121A/287A/RVDTR

Full information on the voting for the approval of this technical report can be found in the report on voting indicated in the above table.

This document has been drafted in accordance with the ISO/IEC Directives, Part 2.

The committee has decided that the contents of this document will remain unchanged until the stability date indicated on the IEC website under "http://webstore.iec.ch" in the data related to the specific document. At this date, the document will be

- reconfirmed,
- withdrawn,
- replaced by a revised edition, or
- amended.

A bilingual version of this publication may be issued at a later date.

IMPORTANT – The 'colour inside' logo on the cover page of this publication indicates that it contains colours which are considered to be useful for the correct understanding of its contents. Users should therefore print this document using a colour printer.

INTRODUCTION

Programmable electronics are now being integrated within switchgear and controlgear. For example, soft-starters, electronic overload relays, circuit-breakers with electronic trip units, proximity switches with built in micro-controllers and some accessories such as extension modules and control panels are using programmable electronics with embedded software called firmware. This embedded software often supports the main functions (see 3.3) provided by the equipment such as overcurrent protection and other important functions, e.g. alarm detection from monitoring devices.

The integration of embedded software within switchgear and controlgear should not degrade the integrity of their main functions compared to purely electromechanical equipment. Therefore, a minimum set of standard requirements for embedded software is provided by this document.

This document takes into account the existing best practices for developing embedded software within safety functions for automation given by IEC 61508-3. Functional safety approach is mainly used in machinery, automotive, automation and process automation where safety functions are implemented with multiple components which should match a consistent level of integrity when combined. In other sectors, such as electric distribution and power control systems, key functions such as over-current release, residual current release, load monitoring, etc. should follow installation rules and coordination rules which are systematically safety and reliability related. Therefore, this document can be seen as providing the principles of the good practice given by IEC 61508-3.

This document is also intended to provide an up-to-date method with regards to the supplement SE of UL 489.

The intention of this document is to provide guidance about:

– risk assessment aspects in relation to embedded software;
– embedded software evaluation method;
– software architecture;
– basic coding rules;
– measures to control software errors;
– software verification and its relationship with the validation of the equipment or system.

In this document, the term “software” is used as a generalized term for embedded software.

LOW-VOLTAGE SWITCHGEAR AND CONTROLGEAR – GUIDANCE FOR THE DEVELOPMENT OF EMBEDDED SOFTWARE

1 Scope

This document provides information, and recommended minimum requirements related to embedded software supporting the main functions of switchgear and controlgear during the whole lifecycle of the equipment. It includes also the parameterization aspects and basics about secure coding standards.

This document can be used in addition to product standard requirements when not already covered.

This document is appropriate for new development or major changes in existing equipment.

This document is not intended to cover the functional safety of control systems for machinery or for automation which are covered by IEC 62061, ISO 13849-1 and IEC 61508 (all parts), neither the cybersecurity risk which are covered by ISO 27005, and IEC 62443 (all parts). It gives only some example of secure coding rules.

NOTE Future new publication IEC TS 63208[1] is under development for implementing embedded cybersecurity measures within switchgear and controlgear based on ISO 27005 and IEC 62443 (all parts).

2 Normative references

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements 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/IEC 2382-1:1993, *Information technology – Vocabulary – Part 1: Fundamental terms*

3 Terms and definitions

For the purposes of this document, the terms and definitions given in ISO/IEC 2382-1:1993, as well as the following apply.

ISO and IEC maintain terminological databases for use in standardization at the following addresses:

- IEC Electropedia: available at http://www.electropedia.org/
- ISO Online browsing platform: available at http://www.iso.org/obp

3.1
embedded software
software, supplied by the manufacturer, that is an integral part of the equipment and that is not accessible for partial modification

Note 1 to entry: Firmware and system software are examples of embedded software.

Note 2 to entry: An embedded software can be upgraded by an integral upload.

[1] Future publication IEC TS 63208 is currently at CD stage.

3.2
programmable electronic
based on computer technology which can comprise hardware, software, and input and/or output units

EXAMPLE The following are all programmable electronics:

– microprocessors;
– micro-controllers;
– programmable controllers;
– application specific digital integrated circuits (ASICs with programmable part);
– programmable logic controllers (PLCs);
– other computer-based devices (for example smart sensors, transmitters, actuators).

Note 1 to entry: This term covers microelectronic devices based on one or more central processing units (CPUs) together with associated memories, etc.

Note 2 to entry: The term "programmable component" is from ANSI/UL 1998:2013, definition 2.39. The definition in ANSI/UL for programmable component is: "Any microelectronic hardware that can be programmed in the design centre, the factory, or in the field". Here the term "programmable" is taken to be "any manner in which one can alter the software wherein the behaviour of the component can be altered."

[SOURCE: IEC 61508-4:2010, 3.2.12, modified – In the definition, "may" changed by "can", "be" and "of" deleted, and addition of a new Note 2 to entry.]

3.3
main function
<switchgear and controlgear> defined function of switchgear and controlgear whose failure can result in its unwanted operation which can lead to hazardous situations, in the loss of its protective function, or in the loss of a key functionality defined by the manufacturer

3.4
systematic failure
failure related in a deterministic way to a certain cause, which can only be eliminated by a modification of the design or of the manufacturing process, operational procedures, documentation or other relevant factors

Note 1 to entry: Corrective maintenance without modification will usually not eliminate the failure cause.

Note 2 to entry: A systematic failure can be induced by simulating the failure cause.

Note 3 to entry: Examples of causes of systematic failures include human error in:

– the safety requirements specification;
– the design, manufacture, installation and/or operation of the hardware;
– the design and/or implementation of the software.

[SOURCE: IEC 61508-4, 3.6.6, modified – Deletion of Note 4 to entry.]

3.5
full variability language
FVL
type of language that provides the capability to implement a wide variety of functions and applications

Note 1 to entry: FVL is normally found in embedded software and is rarely used in application software.

Note 2 to entry: FVL examples include: Ada, C, Pascal, Instruction List, assembler languages, C++, Java, SQL.

[SOURCE: IEC 61511-1:2016, 3.2.75.3, modified – Deletion of "designed to be comprehensible to computer programmers" and Note 1 to entry, remaining notes to entry renumbered.]

3.6
configuration management
discipline of identifying the components of an evolving system for the purposes of controlling changes to those components and maintaining continuity and traceability throughout the lifecycle at a specific point of time

[SOURCE: IEC 61508-4:2010, 3.7.3, modified – Addition of "at a specific point of time".]

3.7
baseline
<configuration management> agreed set of elements (hardware, software, documentation, tests...) of an equipment at a specific point in time, which serves as a basis for verification, validation, modification and changes

Note 1 to entry: If an element is changed the status of the baseline is intermediate until a new baseline is defined.

3.8
coding rules
coding standard
set of rules and guidelines for the formatting of software source code of a program intended to ensure its readability, maintainability, compatibility and robustness

Note 1 to entry: Typical aspects of coding rules are naming conventions, file naming and organization, formatting and indentation, comments and documentation, classes, functions and interfaces, allowed/forbidden standard library function usages, data types, pointer and reference usage, and testing.

3.9
software unit
separately compilable piece of code

Note 1 to entry: Software unit is also called software module and software component such as in documents from International Software Testing Qualifications Board (ISTQB).

[SOURCE: ISO/IEC 12207:2008, 4.43, modified – Addition of a new Note 1 to entry.]

3.10
integration tests
<software> tests performed during the software units and hardware/software integration process prior to the verification and system validation to verify compatibility of the software and the hardware

[SOURCE: IEC 60880:2006, 3.23, modified – Addition of “software units and”, replacement of “computer-based” and “computer” by “the verification and system validation”.]

3.11
software verification
confirmation by examination (e.g. tests, analysis) that the software, its integration or its units requirements have been fulfilled

3.12
static analysis
<software> examination of source code for features that may indicate the presence of software faults

Note 1 to entry: Static analysis typically reveals unreachable sections of code, unused, misused, doubly-defined or uninitialized variables, and unintended execution paths.

Note 2 to entry: Static analysis normally employs computer aided software engineering tools.

[SOURCE: IEC 60050-192:2015, 192-09-22]

3.13
system validation
confirmation by examination and provision of objective evidence that the requirements for a specific intended use of the equipment or the system are fulfilled

Note 1 to entry: By principle the embedded software is integrated in an equipment or a system. The validation of this equipment or system includes embedded software related verifications.

4 Risk assessment and identification of the main functions

Based on manufacturer experience, a system analysis, in the context of the intended applications of the equipment, including its different modes of operation and the reasonably foreseeable misuse, should determine the list of main functions and their associated degree of risk.

EXAMPLE Sensor detection of an object, overcurrent protective function, continuity of supply, power off a motor system.

A risk assessment method such as IEC Guide 116 should be used for this purpose.

Each software part implementing a main function should be managed according to the method provided in this document.

5 Design management

5.1 Objective

A particular organisation of the software design is necessary to ensure the complete realisation of the main function according to its original specification.

This organisation should be defined by a software management plan which may be a clearly identified part of a global design management plan.

5.2 Software management plan of the main functions

This plan is required for defining the management of the activities along the software development lifecycle for the specification and the verification of the software related to main functions (see 3.3).

The organisation should be defined with the roles and associated responsibilities for the management (starting, controlling) and the execution of the activities.

It should be drawn up, documented and updated as necessary. It is intended to provide measures for preventing incorrect specification, implementation, or modification issues.

The software management plan of the main functions should be adapted to the project.

In particular, the plan should:

a) identify the relevant design activities for the development of the parts related to the main functions (essential design activities and their organisation in appropriate sequences are illustrated in Figure 2);
b) describe the policy and strategy to fulfil the specified requirements related to the main functions;
c) describe the strategy for the development, integration, and verification which may include conformity assessment;

d) identify authorized persons, departments or other organisation units and resources that are responsible for carrying out and reviewing each of the main functions design activities; the level of appropriate competency (i.e. training, technical knowledge, experience and qualifications) should be taken into account;
e) identify or establish the procedures and resources to record and maintain information relevant to the main function of the equipment, taking into account the risk assessment results and the change management;
f) describe the strategy for configuration management taking into account relevant organizational issues, such as authorized persons and internal structures of the organization;
g) describe the strategy for modification;
h) establish a software verification plan as detailed in 7.4.3.

5.3 Configuration management

The configuration management is a system engineering process establishing and maintaining consistency of the equipment performance, functional, and physical attributes with its requirements, design, and operational information throughout its life. It consists of verifying whether the software and hardware elements of the equipment are identified and documented in sufficient detail to support its projected lifecycle. This process facilitates the management of the equipment information and the equipment changes for allowing revise capability; improve performance, reliability, or maintainability; extend life; reduce cost; reduce risk and liability; or correct defects.

The main operational aspects of configuration management are:

– **identification** of the structure of the equipment, by identifying its elements e.g. system, subsystems, functions, function blocks, management documents, tools for creating a baseline;
– **controlling** of the release of an element created during each lifecycle phase at a specific point in time;
– **recording** and **reporting** of the status of each element which is and/or will be part of a baseline;
– **audit** and **review** of all elements and maintaining consistency among all elements of a baseline.

Procedures should be developed for configuration management of the equipment during the overall development lifecycle phases of the equipment system and software, including in particular:

a) the point, in respect of specific phases, at which formal configuration control is to be implemented;
b) the procedures to be used for uniquely identifying all constituent parts of an item (hardware and software);
c) the procedures for preventing unauthorized elements from entering into service.

The configuration management procedures should be implemented in accordance with the software management plan.

The procedures for an appropriate change-control-process should consider the requirements of procedures for defining a unique baseline of each version of the equipment including all related configuration items.

5.4 Change management

If a modification related to a main function is to be implemented, then relevant activities should be identified specifically and an action plan should be prepared, documented and approved before carrying out any modification.

NOTE 1 The request for a modification can arise from, for example:

– main function requirements specification changed;

– conditions of actual use;

– incident/accident experience;

– change of material processed;

– modifications of the equipment or of its operating modes.

NOTE 2 Interventions (e.g. adjustment, setting, repairs) on the equipment made in accordance with the information for use or instruction manual for the equipment are not considered to be a modification in the context of this subclause.

The reason(s) for the request for a modification should be documented.

The effect of the requested modification should be analysed to identify the effect on the main function.

The modification impact analysis and the effect on the main function of the equipment should be documented.

All accepted modifications that have an effect on the equipment should initiate a return to an appropriate design phase for its hardware and/or for its software (e.g. specification, design, integration, installation, commissioning, verification and system validation). All subsequent phases and management procedures should then be carried out in accordance with the procedures specified for the specific phases in this document. All relevant documents should be revised, amended and reissued accordingly.

5.5 Defect management

Any software development requires a solid defect management process all along the development lifecycle.

A defect can be:

– a consequence of a coding fault;
– a deviation from the original requirement;
– an unexpected behaviour on an external event.

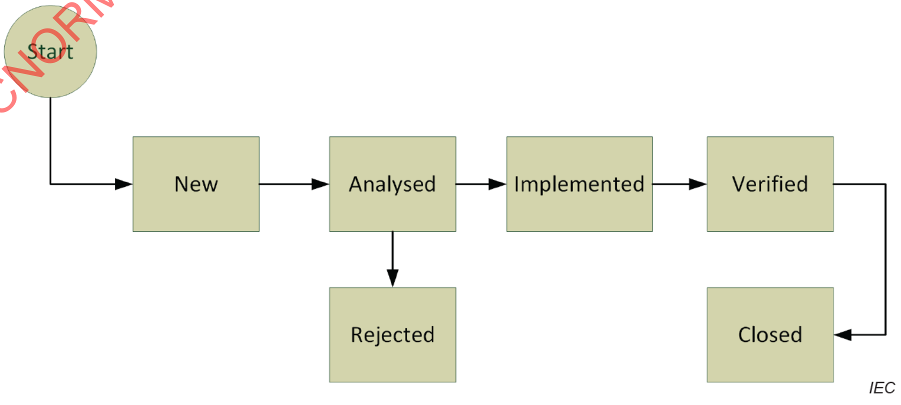

Figure 1 – Defect management process

Defect process workflow varies depending on development team practices. Upper Figure 1 gives an example of process flow for defect lifecycle with the following states:

– **New**: this will be the first step to identify the defect in the process. Most of the time the test team are the one who usually submit defects and sometimes the customer can also report about defects;
– **Analysed**: once the defect registered, the development team setup all means to reproduce and understand the defect root cause. Usually defect severity is set or modified during this step;
– **Rejected**: according to the type of defect or severity of the defect, defect resolution may be rejected;
– **Implemented**: developer fixes the defect and places it in the same place from where it was identified;
– **Verified**: as soon as the defect is fixed, the verification team verifies that defect is actually fixed and the system is working as expected;
– **Closed**: once it is fixed, documented and resolved it is marked as closed case.

NOTE Each state of the defect process workflow can be assigned to a specific project team member, with due delivery date.

5.6 System build and release processes

5.6.1 Binary generation

Building the software binaries that will run on the target (switchgear and controlgear) should be highly predictable, i.e. same sources shall generate same binary.

Hence build tools (compiler, linker, downloader...) and their used options shall be handled under configuration management.

5.6.2 Release management

Each software release should be identified by a unique version/revision number. The most common numbering convention identifies major, minor and patch releases:

– a major version is incremented when there is a compatibility break or several new features support. Usually when MAJOR=0, the software is still in development phase or in Beta Version;
– a minor revision is incremented when few additional features are supported;
– a patch release is incremented for bug-fixes.

Each software release shall be documented by notes which detail the release contents: new features, list of fixed defects and known limitations.

6 Manual parameterization of the embedded software

6.1 General

Some equipment needs manual parameterization to carry out a main function. The parameterization can be done via a remote connection (e.g. PC-based configuration tool) or a human machine interface (e.g. display, DIP-switches, potentiometer, knobs, memory-cards) connected to the equipment.

The objective of the requirements for software-based manual parameterization is to guarantee that the main function related parameters are correctly selected and transferred into the equipment performing the main function. These parameters should not affect a main function in an unintended nor an unauthorized manner.

Different methods can be applied to set such parameters; even DIP-switch based parameterization may be used to set or change main function related parameters. However, tools with dedicated parameterization software, commonly called configuration or parameterization tools, are becoming more prevalent. This subclause is limited in scope to only manual, software-based parameterization that is performed and controlled by an authorized person. These requirements apply if the parameterization occurs at initial commissioning of the equipment, or if the parameterization occurs at any point in the lifecycle of the equipment.

6.2 Influences on main function related parameters

During software-based manual parameterization, the main function parameters can be affected by several influences, such as:

– data entry errors by the person responsible for parameterization;

– faults of the software of the parameterization tool;

– faults of further software and/or service provided with the parameterization tool;

– faults of the hardware of the parameterization tool;

– faults during transmission of parameters from the parameterization tool to the equipment;

– faults of the equipment to store transmitted parameters correctly;

– systematic interference during the parameterization process, e.g. by electromagnetic interference or loss of power;

– interference due to external influences or factors, such as electromagnetic interference or (random) loss of power.

Without appropriate measures, the influences listed above can occur without notice to the person responsible for the parameterization and lead to the following:

– parameters are not updated by the parameterization process, either completely or in parts;

– parameters are incorrect, either completely or in parts;

– parameters are applied to an incorrect device, such as when transmission of parameters is carried out via a wired or wireless network.

Measures should be applied to counteract, avoid or control potential dangerous failures caused by the influences listed above.

6.3 Requirements for software-based manual parameterization

Software-based manual parameterization should use a dedicated tool provided by the supplier of the equipment or the related subsystem(s). This tool should have its own identification (name, version, etc.). The equipment or the related subsystem(s) and the parameterization tool should have the capability to prevent unauthorized modification, for example by using a password.

When the parameterization process can cause an unsafe state, the parameterization should be carried out off-line with the equipment in a safe state. The following requirements should be fulfilled in addition:

a) the design of the software-based manual parameterization should be considered as a main function related aspect of the equipment design that is described in a main function requirements specification, e.g. in the system requirements specification;

b) the equipment or subsystem should provide a data checking system that carries out e.g. checks of data limits, format and/or logic input values;

c) the integrity of all data used for parameterization should be maintained. This should be achieved by applying measures to:

   – control the range of valid inputs;

   – control data corruption before transmission;

– control the effects of errors from the parameter transmission process;
– control the effects of incomplete parameter transmission; and
– control the effects of faults and failures of hardware and software of the parameterization.

The software units used for encoding/decoding within the transmission/retransmission process and the software units used for visualization of the main function-related parameters to the user should, as a minimum, use diversity in function(s) to avoid systematic failures. Diversity in this case consists of using a totally different transmission method to reduce the probability of common cause failure.

6.4 Verification of the parameterization tool

As a minimum, the following verification activities should be performed to verify the basic functionality of the parameterization tool:

– verification of the correct setting for each main function related parameter (check against valid values);
– verification of the plausibility for each main function related parameter, for example by detection of invalid parameter combinations;
– verification that means are provided to prevent unauthorized modification of main function related parameters.

NOTE This is of particular importance where the parameterization is carried out using a device not specifically intended for this purpose (e.g. personal computer or equivalent).

6.5 Documentation of software-based manual parameterization

Software-based manual parameterization should be carried out using the dedicated parameterization tool provided by the supplier of the equipment, and should be documented according to the requirements given in the information for use. This information can originate from different parties. It can be stored in different locations (paper, digitally, on the parameterization tool, on the equipment, etc.). Protective measures against unauthorized access should be activated and used.

The initial parameterization, and subsequent modifications to the parameterization, should be documented. The documentation should include:

a) the identification of the parameterized equipment when the documentation is not stored internally;
b) the date of initial parameterization or change;
c) the date or version number of the data set;
d) the name of the authorized person carrying out the parameterization;
e) an indication of the origin of the data used (e.g. pre-defined parameter sets, as left at a previous setting after the last change);
f) clear identification of main function related parameters.

7 Design lifecycle

7.1 General

All lifecycle activities of embedded software should focus on the avoidance of faults introduced during its design lifecycle. The main objective of the following requirements is to produce readable, understandable, testable, maintainable and correct software.

Where the software performs both non-main functions and main functions, then all of the software should be treated as main function related, unless sufficient independence between the functions can be demonstrated in the design. It is therefore preferable to separate non-main functions such as basic equipment functions from main functions wherever practicable.

7.2 Tools usage

A suitable set of tools, including configuration management, simulation, and test equipment with test generator (e.g. "break out box") should be selected. The availability of suitable tools for service, updating the equipment, and parameterization over the lifetime of the equipment should be considered. The suitability of the tools should be explained and documented in the configuration management documentation.

Suitability should be proven as follows:

– an analysis carried out to identify possible effects of a failure caused by these tools in the tool chain; and

– appropriate fault avoiding and fault controlling measures be selected, applied, and their effectiveness be verified via rigorous testing and the results documented.

NOTE 1 The appropriateness of fault avoiding and fault controlling measures depends on the severity of the consequence of a failure. The basis of this evaluation is an analysis. This analysis can be achieved only when there is a certain knowledge regarding the application of the support tool and of the equipment.

NOTE 2 The effect of failures can vary between different support tools. Therefore IEC 61508-4 differentiates between three categories for off-line support tools used within the software development lifecycle. The analysis points out, if a classification for off-line support tools, which are intended to be used outside the software lifecycle, is appropriate.

NOTE 3 See IEC 61508-4 for definition of support tools and examples.

NOTE 4 This document does not specify any measures for avoiding or controlling faults of off-line support tools. For examples, see 7.4.4 of IEC 61508-3:2010.

NOTE 5 Appropriate actions can be taken to ensure that the software tools used in manufacturing (programming and configuration tools) do not alter the embedded software integrity.

7.3 Software lifecycle

7.3.1 Software lifecycle model

A software lifecycle model which is resolved into distinct phases should be used (e.g. V-model), including management and documentation activities.

Any software lifecycle model may be used provided all the objectives and requirements of this subclause are met. Embedded software should be verified as described in 7.13.

Embedded software is utilizing fully variable programming languages. The detailed V-model in Figure 2 is applicable.

NOTE 1 On the left side of the V-model the output of each phase is reviewed. Review means to check the output of a phase in the V-model against the requirements of the input of the same phase. The arrows "Review" represents the first step of the software verification.

NOTE 2 Project management techniques and processes appropriate for the size and scope of the project are the most appropriate methods.

NOTE 3 In the V-models the feedback loop arrows "Verification" represents the results of test cases according to the specification and, in addition, the need for more precise test case requirements and specifications.

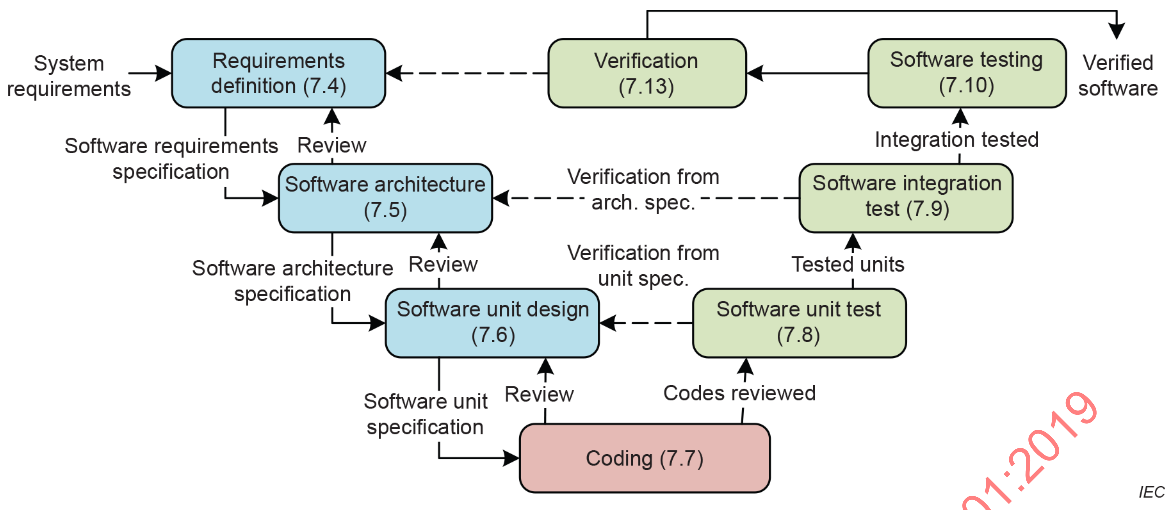
System requirements
Requirements definition (7.4)
Verification (7.13)
Software testing (7.10)
Verified software
Software requirements specification
Review
Integration tested
Software architecture (7.5)
Verification from arch. spec.
Software integration test (7.9)
Software architecture specification
Review
Verification from unit spec.
Tested units
Software unit design (7.6)
Software unit test (7.8)
Software unit specification
Review
Codes reviewed
Coding (7.7)
IEC

Figure 2 – V-model of software lifecycle

7.3.2 Independence of review, testing and verification activities

Where this document requires carrying out review or testing/verification activities, these should be performed by persons not involved directly in the design of the embedded software, i.e. who are independent of the design process. The parties concerned may be other persons, departments or bodies who/which are not subordinate to the design department within the organization's hierarchy. The level of independence should be commensurate with the risk.

The possible levels can be:

– other person;
– independent person;
– independent department;
– independent organisation.

Depending upon the company organisation and expertise within the company, the requirement for independent persons and departments may have to be met by using an external organisation. Conversely, companies that have internal organisations skilled in risk assessment and the application of main function, that are independent of and separate (by ways of management and other resources) from those responsible for the main development, may be able to use their own resources to meet the requirements for an independent organisation.

An "other person" can be involved in the same project, but may not be involved in the same software design activities. An "independent person" may be involved in the same project, but is not involved in the software design activities and does not have responsibility for project management and does not have a superior role. An "independent department" is not involved in design activities of the project. An "independent organisation" is separate and distinct, by management and other resources from the organisation responsible for the design activities of the project.

Factors that make an independent department more appropriate than an independent person are:

– better experience with a similar design;
– greater degree of complexity;
– project size.

7.4 Requirements definition

7.4.1 General

The software should be developed on basis of the system requirements and managed throughout the lifecycle of the equipment.

The software requirements specification should be detailed according to 7.4.3.

7.4.2 System requirements

To support the software development process, the following information is necessary:

a) specification of the main function(s);

b) configuration or architecture of the equipment (e.g. hardware architecture, wiring diagram, main function related inputs and outputs);

c) response time requirements;

d) operator interfaces and controls, such as: switches, joysticks, mode selector, dials, touch sensitive control devices, keypads, etc.;

e) relevant modes of operation of the equipment;

f) requirements on diagnostics for hardware including the characteristics of sensors, final actuators, etc.; and

g) effects of mechanical tolerances, e.g. of sensors and/or their sensing counter parts.

7.4.3 Software requirements specification

7.4.3.1 Design requirements

The software requirements specification should be:

– structured, reviewable, testable, understandable, maintainable and operable;

– developed for each subsystem on the basis of the equipment specification and architecture;

– sufficiently detailed to allow proper software verification and testing;

– traceable back to the specification of the system requirements of the equipment. This means that the specification is understandable such that another person (e.g. non-software specialist) can verify if the specification corresponds to the software related system requirements defined in the risk assessment;

– free of ambiguous terminology and irrelevant descriptions.

It should be possible to relate the inputs of the software requirements specification in a straightforward manner to the desired outputs and vice versa. Where appropriate, easy readable semi-formal methods such as “cause & effect” tables, logic, tables or diagrams, function-blocks or sequence diagrams should be used in the documentation.

The following should be specified within the software requirements specification:

a) logic of the main functions, including inputs and outputs and proper diagnostics on detected faults. Possible methods include, but are not limited to, “cause and effect” table, written description or function blocks;

NOTE 1 Faults can also be detected by hardware (e.g. signal discrepancy detected by input card). Such detection of faults are part of the software requirements specification.

b) test case(s) specifications that include:

   – the specific input value(s) for which the test is carried out and the expected test results including pass/fail criteria;

   – fault insertion or injection(s).

NOTE 2 For simple functions the test case(s) can be given implicitly by the specification of the main function.

c) diagnostic functions for input devices, such as sensing elements and switches, and final control elements, such as solenoids, relays, or contactors;
d) functions that enable the equipment to achieve or maintain a safe state;
e) functions related to the detection, annunciation and handling of faults;
f) functions related to the periodic testing of equipment(s) on-line and off-line;
g) self-monitoring of control flow and data flow of the equipment. On failure detection, appropriate actions should be performed to achieve or maintain a safe state;
h) functions that prevent unauthorized modification of the equipment;
i) interfaces to other equipment;
j) main function response time;
k) algorithmic model of signal processing function; and
l) coding rules.

NOTE 3 Guidance on software documentation is given in IEC 61508 (all parts) and ISO/IEC/IEEE 26512:2018.

The information in the software requirements specification should be reviewed against the system requirements specification and, where necessary, revised to ensure that the software requirements are adequately specified.

7.4.3.2 Specification of the programming language and the associated development methods

Additional requirements should be applied when a full variability language is used for the design of embedded software.

Tables of Annexes A and B of IEC 61508-3:2010 should be taken into consideration when it is appropriate to use alternative techniques and measures of an equivalent effectiveness.

The design and choice of the language chosen should take into account the following features, depending on the complexity of the application:

a) abstraction, modularity and other features which control complexity; wherever possible, the software should be based on well-proven logic functions which may include user library functions and well-defined rules for linking logic functions;
b) expression of:
   – functionality, ideally as a logical description or as algorithmic functions;
   – information flow between modular elements;
   – sequencing and time-related requirements;
   – timing constraints;
   – concurrency and synchronized access to shared resources;
   – data structures and their properties, including data types, validity of data ranges;
   – exception handling.
c) comprehension by developers and others who need to understand the design, both from a functional understanding of the application and from a knowledge of the constraints of the equipment technology;
d) software verification plan which gives the method of evaluating the required integrity of the main functions, the verification strategies and tools and the evaluation of the results and how the corrective actions will be managed;
e) related part of the system validation plan that should include in addition the intended use test cases;
f) maintenance procedures, e.g. modification.

7.5 Software architecture

7.5.1 General

The software architecture should be established for fulfilling the software requirements specification. The software architecture defines the major elements and subsystems of the software, how they are interconnected, and how the required attributes will be achieved. It also defines the overall behaviour of the software, and how software elements interface and interact. Examples of major software elements include operating systems, databases, input/output subsystems, communication subsystems, application programs, programming and diagnostic tools, etc.

The software architecture should follow a modular approach with a limited software unit size, a fully defined interface and one entry/one exit point in subroutines and functions. Each software unit should have a single, clearly understood task and should be limited to one complete main function.

7.5.2 Software architecture specification

A software architecture specification should be provided as an output of the software architecture design. This should explain the main software aspects such as indicated in the following list, for example:

– software architecture that defines the structure decided to satisfy the software requirements specification;
– global data;
– data libraries used;
– pre-existing software units used;
– diagnostic functions (internal, external);
– programming tools including information which uniquely identifies the tool;
– software integration test cases and procedures, including specification of the test environment, support software, configuration description and procedures for corrective action on failure of test.

The information contained in the software architecture specification should be reviewed against the software requirements specification.

7.6 Software unit design

7.6.1 General

Where previously developed software units are to be reused as part of the design, their suitability in satisfying the specification of requirements of the main function software should be analysed. Constraints from the previous software development environment (for example operating system and compiler dependencies) should be evaluated.

7.6.2 Input information

For software units the following information should be available in the software architecture specification:

1) software unit description;
2) software unit interface (inputs and outputs with data types), and (if necessary), with data ranges;
3) identification of the libraries used;
4) special coding rules.

7.6.3 Software unit specification

The software unit specification should contain the following information:

1) description of the logic (i.e. the functionality) of each software unit;
2) fully defined input and output interfaces assigned for each software unit;
3) format and value ranges of input and output data and their relation to software units;
4) test cases which should include normal and outside normal operation;

   NOTE Although test cases usually comprise the individually testing of parameters within their specified ranges, a varying combination of these parameters can introduce unpredicted operation.

5) documentation of the interrupts.

This information should be reviewed against the input information (see 7.6.2).

7.7 Coding

Software should be developed in accordance with the software unit specifications and coding rules. Coding rules can be either well-known industry standards or can be internal to the manufacturer. The code should be reviewed against the software unit specifications and coding rules.

NOTE 1 Coding rules are intended to restrict the freedom of programming in order to avoid the program code becoming incomprehensible.

NOTE 2 Coding rules usually define a subset of a programming language or use of a strongly typed programming language (see C.4.1 of IEC 61508-7:2010, MISRA C).

NOTE 3 Code review can be manual or automatic through static code analysis tools.

The following programming techniques should be used to avoid systematic failures:

– range checking and plausibility checking of variables and configuration parameters;
– temporal and/or logical program sequence monitoring to detect a defective program sequence;
– limiting the number or extent of global variables.

NOTE 4 See Annex G of IEC 61508-7:2010 for guidance on object-oriented architecture and design.

The output of coding should comprise:

– source code listing;
– code review report.

For limiting cybersecurity risks, secure coding rules should be applied. Depending on the cybersecurity requirements, the embedded operating system and the programming language, secure coding rules can vary significantly.

For example, in the case of a high level of integrity, a medium level of availability, an operating system limited to the manufacturer applications and C language, the following rules from ISO/IEC TS 17961:2013 and CERT/CC C should be used (each dashed item includes hyperlinks to SEI CERT C Coding Standard rules, identified by text in blue colour):

– never trust user input: sanitize data passed to complex subsystems and exclude user input from format strings;
– check boundaries when using container types: do not form or use out-of-bounds pointers or array subscripts and do not add or subtract an integer to a pointer to a non-array object;
– secure memory management: allocate sufficient memory for an object and only free memory allocated dynamically;

– call only safe function within a signal handler: call only asynchronous-safe functions within signal handlers and do not access shared objects in signal handlers;
– miscellaneous rules: default deny and adhere to the principle of least privilege.

7.8 Software unit test

Each software unit, which was not previously assessed, should be tested as required against the test cases defined in the software unit specification.

If the software unit does not pass the testing, then predefined corrective action should be taken.

The test results and corrective actions should be documented.

Software unit testing should use as a minimum the technique of dynamic analysis and testing (B.6.5 of IEC 61508-7:2010).

7.9 Software integration test

The software should be tested against the integration test cases following the software architecture specification. The results of software integration testing should be documented.

Two different steps shall be performed as follows:

1) software integration where only software modules interaction is checked (through a dedicated software integration test tool);
2) software integration with hardware where the same modules are verified with their interaction on the effective physical target (allowing effective timings measurement and global behaviour checking).

NOTE The objective of these tests is to show that all software units and software elements/subsystems interact correctly to perform their intended function and do not perform unintended functions. This does not imply testing of all input combinations, nor of all output combinations. Testing all equivalence classes or structure based testing can be sufficient. Boundary value analysis (C.5.4 of IEC 61508-7:2010) or control flow analysis (5.5.9 of IEC 61508-7:2010) can reduce the test cases to an acceptable number. Analysable programs make the requirements easier to fulfil.

7.10 Software testing

7.10.1 General

The main goal of software testing is to ensure that the functionality as detailed in the software requirements specification is achieved.

The main output of software testing is a document e.g. a test report with test cases and test results allowing an assessment of the test coverage.

Software testing should also include failure simulation and the associated failure reaction.

When previously-tested software units which incorporate failure detection and reaction are utilised (e.g. discrepancy of input signals or feedback contact of output) then the test of those failure detection and reaction is not necessary. In that case only the integration of these software units should be tested.

Software testing can be carried out as part of the system validation if testing is performed on the target hardware integrated in the equipment or the system, according to specified end-user use cases such as out-of-the-box experience, or software upgrading.

Functional testing as a basic measure should be applied. Code should be tested by simulation where feasible.

It is recommended to define general guidelines or procedures for the testing of embedded software. These guidelines or procedures should include:

– types of tests to be performed;
– specification of test equipment including tools, support software and configuration description;
– management of software versioning during testing and correcting of embedded software;
– defect tracking process with e.g. bug registering, rating, fixing, approval;
– corrective actions on failed test;
– criteria for the completion of the test with respect to the related functions or requirements;
– physical location(s) of the testing, which may be completed by the testing means such as computer simulation, bench top, or on the equipment's hardware.

7.10.2 Test planning and execution

Test planning based on test cases should include:

– definition of roles and responsibilities by name;
– installation of the testing environment;
– functional aspects of testing.

Testing of software includes two types of activities:

– static analysis: analysis of software, e.g. by manual review such as "4 eyes" code review, inspection, walk-through, control flow analysis, dataflow analysis or by automated code analysis, e.g. with MISRA-Checks. Static analysis is normally performed by the code developer during the code implementation process;
– dynamic testing: execution of the software in a controlled and systematic way, so as to demonstrate the presence of the required behaviour and the absence of unwanted behaviour. This includes, in particular, functional testing or black-box-testing.

In the early phases of the software lifecycle, verification is static. Dynamic testing becomes possible when the code is produced and is operational. For verifying the output of software lifecycle activities, both activities are required in combination. For further description of static analysis and dynamic testing, see IEC 61508-3.

The following is required for the software verification and testing of embedded software:

– static analysis shall be performed and documented;
– dynamic testing should be performed and documented, every subprogram (subroutine or function) should be called at least once (entry points) during testing;
– all statements in the code should be executed at least once during dynamic testing;
– where software is used in diagnostic functions for controlling random hardware failures, dynamic testing should address the correct implementation of the diagnostics e.g. by fault insertion testing;
– dynamic testing should include a final test on the target hardware.

7.11 Documentation

All lifecycle activities should be traceable forwards and backwards from the system requirements through the completed software verification plan.

The inputs and outputs of all software lifecycle phases should be documented and made available to the relevant persons.

The test activities results and corrective actions taken should be documented.