CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Evaluation of open source operating systems for safety-critical applications

Master's thesis in Embedded Electronic System Design

Petter Sainio Berntsson

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2017

MASTER'S THESIS 2017

Evaluation of open source operating systems for Safety-critical applications

Petter Sainio Berntsson

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2017

Evaluation of open source operating systems for safety-critical applications
Petter Sainio Berntsson

Examiner:

Per Larsson-Edefors
Chalmers University of Technology
Department of Computer Science and Engineering

Academic supervisor:

Jan Jonsson
Chalmers University of Technology
Department of Computer Science and Engineering

Industrial supervisors:

Lars Strandén
RISE Research Institutes of Sweden
Dependable Systems

Fredrik Warg
RISE Research Institutes of Sweden
Dependable Systems

Master’s Thesis 2017
Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
SE-412 96 Gothenburg
Telephone +46(0) 31 772 1000

Abstract

Today many embedded applications will have to handle multitasking with real-time time constraints and the solution for handling multitasking is to use a real-time operating system for scheduling and managing the real-time tasks. There are many different open source real-time operating systems available and the use of open source software for safety-critical applications is considered highly interesting by industries such as medical, aerospace and automotive as it enables a shorter time to market and lower development costs. If one would like to use open source software in a safety-critical context one would have to provide evidence that the software being used fulfills the requirement put forth by the industry specific standard for functional safety, such as the ISO 26262 standard for the automotive industry. However, the standards for functional safety do not provide a clear method for how one would go about certifying open source software. Therefore, in this thesis we present identified important characteristics that can be used for comparing the suitability of open source based real-time operating systems for use in safety-critical applications together with a proposed outline for a methodology that can be used for certifying an open source real-time operating system. A case study has been done where we compared two open source operating systems for small microcontrollers with the previously mentioned characteristics in order to see which of the two is most suitable for use in safety-critical applications. The most suitable candidate is then assessed in order to see to what degree it can adhere with the requirements put forth in the widely used functional safety standards IEC 61508 and ISO 26262.

Keywords: Functional safety, open source software, real-time operating systems, software quality

Contents

1. Introduction

This thesis project will propose an outline of a methodology for how to certify and compare the suitability of open source based real-time operating systems for use in safety-critical applications with regards to the widely used functional safety standards IEC 61508 and ISO 26262. This chapter will give the background to the project as well as introducing the purpose, scope and limitations of the project. The structure of the report is also presented in this chapter.

1.1. Background

The last few years have seen a remarkable rise in the demand of embedded systems, especially in the automotive industry [1] where the automobiles are equipped with embedded electronic systems. The use of a Real-Time Operating System (RTOS) is common in modern embedded systems due to the multitasking requirement many applications have [2]. In the last two decades RTOSs have undergone continuous evolution and many commercial RTOSs have been developed [3]. A real-time system is defined as a system in which the correctness of the system does not only depend on the result of a computation but whether or not the correct result is produced within the set time constraint [3]. Real-time systems often use an RTOS that provides a framework that enables software developers to more easily create a real-time software application. By using an RTOS it becomes easier to partition the entire software into smaller tasks, executed concurrently or in parallel, which can then be separately distributed to individual programmers. An RTOS enables a proper structured way to build multitasking applications [4]. One of the most important things for an RTOS lies in its degree of predictability so that all the time-critical tasks are guaranteed to meet their deadlines. In order to achieve a high degree of predictability the RTOS has to manage time-critical tasks on priority basis and keep the time it takes to change between tasks as low as possible.

As many commercially available software products are relatively expensive people have developed their own versions of such software and made them publicly available as Open Source Software (OSS). A common use of OSS is operating systems which are normally application independent and can therefore attract a large user base and can be ported to different hardware platforms. Easy access to the internet has resulted in the growing use of OSS due to the fact that sharing code has become easier. However, a common perception is that closed proprietary software is still superior in quality because only special people with access to the code can contribute to it. Open source projects that have well-established communities (e.g. Linux and Apache) usually employ stringent development processes [5], [6] and deliver high quality software, but they do not fulfill the requirements of current safety standards such as IEC 61508 [10] and ISO 26262 [11]. These standards impose strict demands on project management, developer qualification, risk management, requirements management, quality assurance and documentation.

The use of OSS for safety-critical applications is considered highly interesting by industries such as medical, aerospace and automotive, as it enables a shorter time-to-market and lower development costs. However, in order for a piece of software to be part of a safety-critical application it requires a safety certification. A safety certification demands that evidence regarding OSS quality is supplied, and an analysis is needed to assess if the cost to produce certification evidence is worthwhile [7].

Today embedded applications with real-time constraints often use open source operating systems such as Linux, ContikiOS, ChibiOS or Android for scheduling and managing tasks. These operating systems must be able to guarantee response times within the specified time-constraints for certain tasks. Therefore, the quality of these open source operating systems needs to be actively maintained because software reliability is one of the most important aspects of software quality [8]. And even though RTOSs are a common sight in modern embedded systems, the use of OSS in the area of dependable systems is still relatively rare, especially when the products must adhere to functional safety standards. The reason for this is that standards require that the quality of the software developed can be demonstrated by adhering to the requirements put forth in the standard [9] and that a strict development process is being followed [10], [11]. This becomes a problem since many OSS projects do not follow a strict development process [12]. And therefore, some of the requirements are impossible to achieve after the software has been developed. However, in some cases software that has been developed with a development process that is not compliant with the one proposed can still qualify for certification if the software fulfills the requirements for reuse. This can be done by providing enough evidence to support its use in safety-critical applications.

1.2. Purpose

Previous studies [7] have shown that software reliability is one of the most important aspects of software quality. Therefore, this is a major concern for software being used in the automotive industry and other industries as well where standards require a certain quality of the software being developed [8]. In an automobile, one will most likely have an electronic system which needs to handle multitasking. The solution for handling multitasking is to use an RTOS for scheduling and managing the real-time tasks. There are many different open source RTOSs and if one would want to use an open source RTOS, or any form of OSS for that matter, one would have to demonstrate that the software being used can fulfill the requirements put forth by the standards required for the system being developed. So, if one would like to use a specific open source RTOS already developed, one would have to assess to what extent the RTOS can adhere to the requirements put forth in the standards.

Therefore, the first purpose of this master thesis project is to identify characteristics that can be used to evaluate, mainly from a software perspective, an open source RTOS in regards to its use in safety-critical applications. These characteristics also make it possible to compare the suitability of different RTOSs.

The second purpose of this master's thesis is to propose a methodology to certify open source operating systems for use in safety-critical applications. This certification methodology is proposed since it is desirable to have a repeatable certification process which could be used by certification bodies and others to certify and compare the suitability of multiple open source RTOSs for use in a safety-critical context. A case study will then be performed where two open source operating systems is evaluated in order to see which option is most suitable for use in a safety-critical application. This candidate will then be assessed as is, in order to see to what extent, it can comply with functional safety standards IEC 61508 and ISO 26262.

The specification of a safety-critical application will also be presented in order to show how safety functions can be specified in a given context together with its implementation.

1.3. Scope and Limitations

The two open source operating systems that have been chosen for this thesis project are ChibiOS and ContikiOS. The reason for this is because RISE Electronics – Dependable systems (RISE) are interested in getting an analysis of just these two open source operating systems since they are both open source operating systems for small microcontrollers. RISE has used ChibiOS in previous projects and it is still being used in various research projects and would therefore be interested in getting an analysis if it can possibly comply with the functional safety standards IEC 61508 and ISO 26262. ContikiOS is a small operating system designed specifically for Internet of Things (IoT) devices with support for Wireless Sensor Networks. This thesis project will only assess to what extent one of the open source operating systems can comply with the functional safety standards IEC 61508 and ISO 26262 which are used for industrial applications and road vehicles, respectively. In this thesis, the focus is mainly on software, while both of these standards are focused on systems that consists of both hardware and software components.

An integrity level (e.g. SIL/ASIL) is defined as a relative level of risk reduction provided by a safety function, or to specify a target level of risk reduction. In the IEC 61508 standard, four Safety Integrity Levels (SILs) are defined, with SIL 4 being the most dependable and SIL 1 the least. In the ISO 26262 standard there is also four Automotive Safety Integrity Levels (ASILs) defined where ASIL D is the most dependable one and ASIL A the least.

Most if not all SIL 4/ASIL D applications have hard real-time behavior defined and generally do not run under any form of operating system, therefore the maximum SIL/ASIL that is considered in this thesis is SIL 3 / ASIL C.

The international standards IEC 61508 and ISO 26262 apply to safety functions which are application dependent in a given system context whereas an RTOS is application independent and will therefore be considered as a context free software component in this thesis. The focus of the thesis is therefore reusing an existing open source RTOS for safety-critical applications. The open source RTOS will only be assessed as is and will not be modified, nor will any documentation be created. In order to see to what extent, it can adhere to the requirements put forth in the standards in their current versions.

1.4. Document structure

The workflow used in this project will be presented in chapter 2. Next, the technical background for this project will be described in chapter 3. In chapter 4 the identified characteristics are presented together with a proposed methodology for certifying an open source operating system. In chapter 5 the two open source operating systems looked at in this study are compared and the most suitable candidate is assessed. In chapter 6 an example of a safety-critical application is presented. And finally, in chapter 7 the whole project is concluded and an evaluation is made to what degree the assessed open source operating system can adhere to the functional safety standards IEC 61508 and ISO 26262.

2. Project Workflow

This chapter will describe the workflow of the project and present which parts of the functional safety standards that have been analyzed.

2.1. Literature Survey

The first phase of the project was a literature survey. The survey served as a foundation for identifying relevant characteristics that can be used for comparing the suitability of open source operating systems for a safety-critical application. The survey was also used to get an overview of what kind of requirements there are for on operating systems in safety-critical applications and how one can achieve compliance with the functional safety standards IEC 61508 and ISO 26262. The survey was also done in order to find support and maintainability issues that are commonly found in OSS projects and how the quality of OSS can be evaluated with regards to the community behind an OSS project.

2.2. IEC 61508 Compliance

In IEC 61508 we have mainly looked at Part 3: Software requirements and the supplement that has been made to Part 3 (that has yet to be accepted) regarding reuse of pre-existing software elements to implement all or part of a safety function [15]. These parts of the standard have been further analyzed in order to see how the open source RTOS can comply with the requirements concerning documentation and support that is available for it.

OSS projects are often developed and maintained by an open collaborative community of volunteering developers with no obligations and a development process that is not compliant with the requirements given in IEC 61508. Therefore, an OSS project will most likely not qualify for the first route proposed for compliance since that route specifically targets software components that has already been developed according to the IEC 61508 standard. The second route “proven in use” seems to be too hard to fulfill for an assessment to be made and there is also a limitation of the use of “proven in use software” to SIL 1 and SIL 2 in the supplement that has been made to IEC 61508-3. Therefore, route 3: assessment of non-compliant development will be used for assessing the open source RTOS to implement all or part of a safety function in this thesis.

2.3. ISO 26262 Compliance

In ISO 26262 we have mainly looked at Part 6: Product development at the software level and Part 8 clause 12: Qualification of software components and gone over the documentation needed for the open source operating system in order to see to what degree it can comply with the requirements put forth in the standard.

2.4. Quality Assessment

In order to assess the quality of the open source operating systems and the maturity of the community behind the open source project, the Capgemini Open Source Maturity Model (see section 3.3.1) will be used. This because the QualOSS (see section 3.3.2) project seems to be abandoned and the tools used for retrieval and analysis of metrics are no longer available along with the database for storing the data. The quality assessment of the two open source operating systems used in this thesis can be seen in section 5.3.

2.5. Case Study

A case study has been performed where two open source operating systems is evaluated according to the Capgemini Open Source Maturity Model together with the identified characteristics to find the most suitable option for use in a safety-critical application. The most suitable candidate is then assessed to see to what degree it can adhere to the requirements put forth in the functional safety standards IEC 61508 and ISO 26262.

The most suitable candidate will then be used in an example application in order to demonstrate how safety functions might be specified in a specific context.

3. Technical Background

This chapter will present the technical base for the project. It will start by presenting the concept of functional safety and introduce two widely used functional safety standards. This will then be followed by a summary of what open source is and what type of problems one might face when using open source software. Following this there will be an introduction to software quality and some models used for evaluating the quality and maturity of OSS projects followed by an introduction to real-time operating systems. MISRA C will then be introduced and finally, software metrics will be presented which can be used for comparing different operating systems.

3.1. Functional Safety

Functional safety is everywhere since the concept applies to everyday life and to every industry that one can imagine. It is fundamental for most safety-related systems. The oil and gas industry, nuclear plants, the manufacturing sector, automotive and transportation all rely on functional safety to achieve safety in areas where the operation of equipment can give rise to hazards.

Functional safety is related to the overall safety of a system or piece of equipment and generally focuses on electronics and related software. It looks at aspects of safety that relate to the function of a device or system and ensures that it is operating correctly in response to the commands it receives. In a systematic approach functional safety identifies potentially dangerous situations, conditions or events that could result in an accident which could potentially be of harm to someone or something. Functional safety enables corrective or preventive actions in order to avoid or reduce the impact of an accident. The aim of handling functional safety is to bring risk down to a tolerable level, but there is no such thing as zero risk. Functional safety defines risk by how likely it is that a given event will occur and how severe it would be [13]. In other words, how much harm it could cause.

Functional safety is achieved when every specified safety function is carried out and the level of performance required of each safety function is met. This is normally achieved by a process that includes the following steps as a minimum [10]:

1. Identify what the required safety functions are. This means that the hazards and the safety functions have to be known.
2. Assessment of the risk-reduction required by the safety function. This will involve an *Integrity Level* (e.g. SIL/ASIL) or performance level or other quantification assessment. An integrity level applies to a safety function of the safety-related system as a whole, not just to a component or part of the system.
3. Ensure that the safety function performs to the designed intent, including under conditions of incorrect operator input and failure modes. This will involve having the design and lifecycle managed by qualified and competent engineers carrying out processes to a recognized functional safety standard. In Europe, an important standard is IEC 61508, and one of the industry specific standards derived from IEC 61508 (e.g. ISO 26262 for the automotive industry).
4. Validate that the system meets the assigned integrity level by determining the mean time between failures and the safe failure fraction along with appropriate tests. These measures are only used for hardware.
5. Conduct functional safety audits to examine and assess the evidence that the appropriate safety lifecycle management techniques were applied consistently and thoroughly in the relevant life cycle stages of the product. These aspects are applicable for both software and hardware.

A component, subsystem or system that claims to be functionally safe needs to be independently certified to one of the recognized functional safety standards. Only when the product has been certified one can claim that the product is functionally safe to a particular integrity level (e.g. SIL/ASIL) in a specific range of applications [14].

An important element of functional safety certification is the on-going inspection by the certification bodies. This follow-up inspection ensures that the product, subsystem or system is still being manufactured in accordance with what was originally certified for the functional safety standard. Follow up surveillance may occur at different intervals depending on the certification body and will typically look at the products hardware, software as well as the manufacturer's development process.

The IEC 61508 (see section 3.1.1) and ISO 26262 (see section 3.1.2) standards define four different integrity levels for risk reduction with increasing reliability requirements for safety functions in order to have safer functionality. For each integrity level the standards define a set of requirements for the software development that can be divided in three main categories.

- **Software Safety Requirements Specification:** The safety requirements shall be derived from technical safety concept (see IEC 61508-3 subclause 7.2 and ISO 26262-6 subclause 6). Requirements must be clearly defined including the response times of safety functions.
- **Software Design and Implementation:** The requirements of this part have the objective of producing readable, testable and maintainable source code by applying rules for writing code. This part also recommends different techniques that can be used to specify the software in detail (e.g. semi-formal methods, see Appendix A).
- **Software Testing and Validation:** The objective of this category is to verify that the behavior of the software is working according to its software specification. This part also requires proof that the requirements have been met according to the appropriate integrity levels.

In the second part: Software Design and Implementation, requirements for programming are defined, e.g. the use of an adequate programming language that allows the use of structured and defensive programming and possibly assertions. The programming requirements are there to produce verifiable code. It discourages the use of machine-oriented languages (assembly). It also discourages the use of a set of features that many programming languages have, such as recursion and dynamic memory allocation. A good practice is to use programming rules and it is also a good idea to define a language subset that is considered safe. A common one is MISRA C (see section 3.5) for the C programming language.

3.1.1. IEC 61508

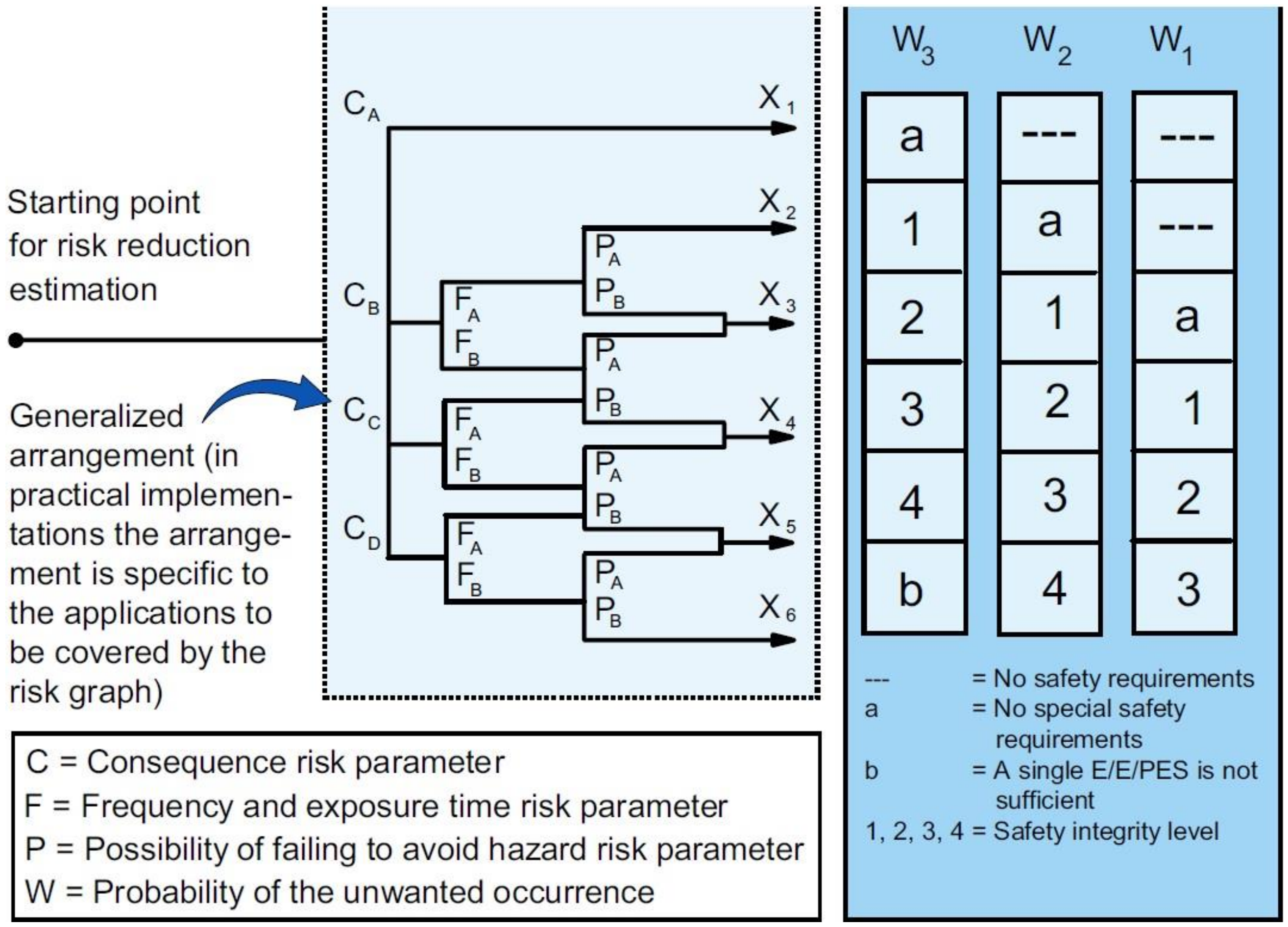
IEC 61508 is an international standard published by the International Electrotechnical Commission that consists of seven parts and is titled "Safety of Electrical / Electronic / Programmable Electronic Safety-related Systems (E/E/PE or E/E/PES)" [15]. The first version of the standard was published in the year 1998 and the second version which is the one currently used was released in the year 2010. The date for another new version is not yet decided, but a decision has been made to produce a supplement to Part 3 and a draft for this has been made regarding the reuse of pre-existing software elements to implement all or part of a safety function. The IEC 61508 standard sets forth requirements for ensuring that safety related systems are designed, implemented, operated and maintained in such a way that the system provides the necessary Safety Integrity Levels (SILs) for its safety functions [13]. It provides guidelines and requirements for both hardware and software parts of a system. The standard provides guidance for some ways to satisfy the requirements and also provides an overview of techniques and measures for determining the SIL needed [10]. The seven parts of the standard and what they contain are shown below. The first four parts are normative and the three later ones are informative [16]:

- Part 1: General requirements
- Part 2: Requirements for electrical / electronic / programmable electronic safety-related systems
- Part 3: Software requirements
- Part 4: Definitions and abbreviations
- Part 5: Examples of methods for the determination of SIL
- Part 6: Guidelines on the application of IEC 61508-2 and IEC 61508-3
- Part 7: Overview of techniques and measures

The requirements for integrity (risk reduction) of safety functions are assessed according to SIL 1-4 where SIL 4 is the highest demand for risk reduction. Suitable SIL is determined by conducting a risk analysis where the greater the risk, the greater the value of SIL is required.

There are different types of consequences to be considered and these are; harm to people, harm to equipment, harm to environment, and finally information and economic damage (short-term or long-term). Figure 1 shows how SIL is determined according to IEC 61508 Part 5 Annex E. However, note that Part 5 is informative, that is, there is no requirement to follow it. In order to obtain a SIL one uses a risk graph that is based on consequence, frequency, probability to avoid the event and the probability that the event occurs.

Because this master thesis project is about the use of OSS in safety-critical applications, we will mainly consider Part 3: Software requirements and the draft that has been made regarding the supplement to Part 3 regarding the reuse of pre-existing software elements to implement all or part of a safety function [15]. We will specifically consider aspects in the standard that can be applicable on OSS.


Starting point for risk reduction estimation
Generalized arrangement (in practical implementations the arrangement is specific to the applications to be covered by the risk graph)
C_A
C_B
C_C
C_D
F_A
F_B
P_A
P_B
X_1
X_2
X_3
X_4
X_5
X_6
W_3
W_2
W_1
C = Consequence risk parameter
F = Frequency and exposure time risk parameter
P = Possibility of failing to avoid hazard risk parameter
W = Probability of the unwanted occurrence
--- = No safety requirements
a = No special safety requirements
b = A single E/E/PES is not sufficient
1, 2, 3, 4 = Safety integrity level

W_3	W_2	W_1
a	---	---
1	a	---
2	1	a
3	2	1
4	3	2
b	4	3

Figure 1 Example of determination of SIL according to IEC 61508

In the draft that has been made to IEC 61508-3 regarding the reuse of pre-existing software elements to implement all or part of a safety function a reference is made to a requirement given in subclause 7.4.2.12 in IEC 61508-3. The subclause offers three possible routes to the achievement of the necessary integrity for pre-existing software elements that is reused to implement all or part of a safety function. The software element shall meet both requirements that are given for a) and b) below for systematic safety integrity:

a) Meet the requirements of one of the following compliance routes:

    1. **Route 1:** Compliant development. Compliance with the requirements of this standard for the avoidance and control of systematic faults in software
    2. **Route 2:** Proven in use. Provide evidence that the element is proven in use. See subclause 7.4.10 of IEC 61508-2
    3. **Route 3:** Assessment of non-compliant development. Compliance with 7.4.2.13 of IEC 61508-3

b) Provide a safety manual (see Annex D of IEC 61508-2) that gives sufficiently precise and complete description of the element to make possible an assessment of the integrity of a specific safety function that depends wholly or partly on the pre-existing software element.

3.1.2. ISO 26262

ISO 26262 is an international standard published by the International Organization for Standardization that consists of ten parts and is titled “Road Vehicles - Functional Safety”. The first version of the standard was published in the year 2011 and is the most recent one. Work is in progress regarding a new revision of the standard but there is no draft publicly available.

The standard is an adaptation of the Functional Safety standard IEC 61508 for Automotive Electric / Electronic systems. It defines a detailed safety lifecycle for developing electrical and electronic systems for production line automobiles as can be seen in figure 2. Some of the major differences are that ISO 26262 takes supplier relations into account, and that the hazard analysis and risk assessment is adapted for automotive use cases, e.g. adding the element of controllability (the possibility for the driver or other traficants to mitigate a hazard). The automotive industry has already been using safety analysis, validation and verification techniques in order to ensure safety of the vehicle. However, with the introduction of ISO 26262 this means that these techniques must now be applied as a standardized methodology that is industry-wide [11].

The ten parts and what they are about and how they connect to each other can be seen below and in figure 2:

- Part 1: Vocabulary
- Part 2: Management of functional safety
- Part 3: Concept phase
- Part 4: Product development at the system level
- Part 5: Product development at the hardware level
- Part 6: Product development at the software level
- Part 7: Production and operation
- Part 8: Supporting processes
- Part 9: Automotive Safety Integrity Level (ASIL)-oriented and safety-oriented analysis
- Part 10: Guideline on ISO 26262

A major focus of the ISO 26262 standard is the ability to consistently track how safety requirements are being refined and met from the initial concept phase to the final working product [17].

In ISO 26262 (and other standards that relate to both hardware and software requirements) the safety requirements are being assessed on a scale that is expressed in discrete risk levels and not in probabilities. Traditionally safety requirements have been expressed quantitatively, using maximum allowable probabilities for system failures that shall not be exceeded. However, techniques for quantifying the probability of software failure are immature so instead the standard uses the discrete *Automotive Safety Integrity Levels* (ASILs) to represent safety requirements regarding software and systematic failures in general [17].

For the automotive industry, ASILs are an adaption of the SIL defined in IEC 61508 [17]. The different ASILs range from A to D and represent least strict and strictest respectively, whereas QM does not represent any special safety requirement and normal quality management shall be applied. ASILs are set during the concept phase once the major functions of the system have been identified since the standard requires that a hazard analysis and risk assessment is performed for each function (called item in ISO 26262). This forces the designers to examine the different functions in order to determine how they could malfunction in various scenarios and what hazards could arise as a result of the malfunction [17]. The end result of the hazard analysis and risk assessment is a set of safety goals each with an ASIL that reflect the level of risk associated with the safety goal such that an unreasonable risk is avoided. The ASIL is determined by considering the estimate of the impact factor, i.e. severity, probability of exposure and controllability. It is based on the item's functional behavior and therefore the detailed design of the item does not necessarily need to be known.

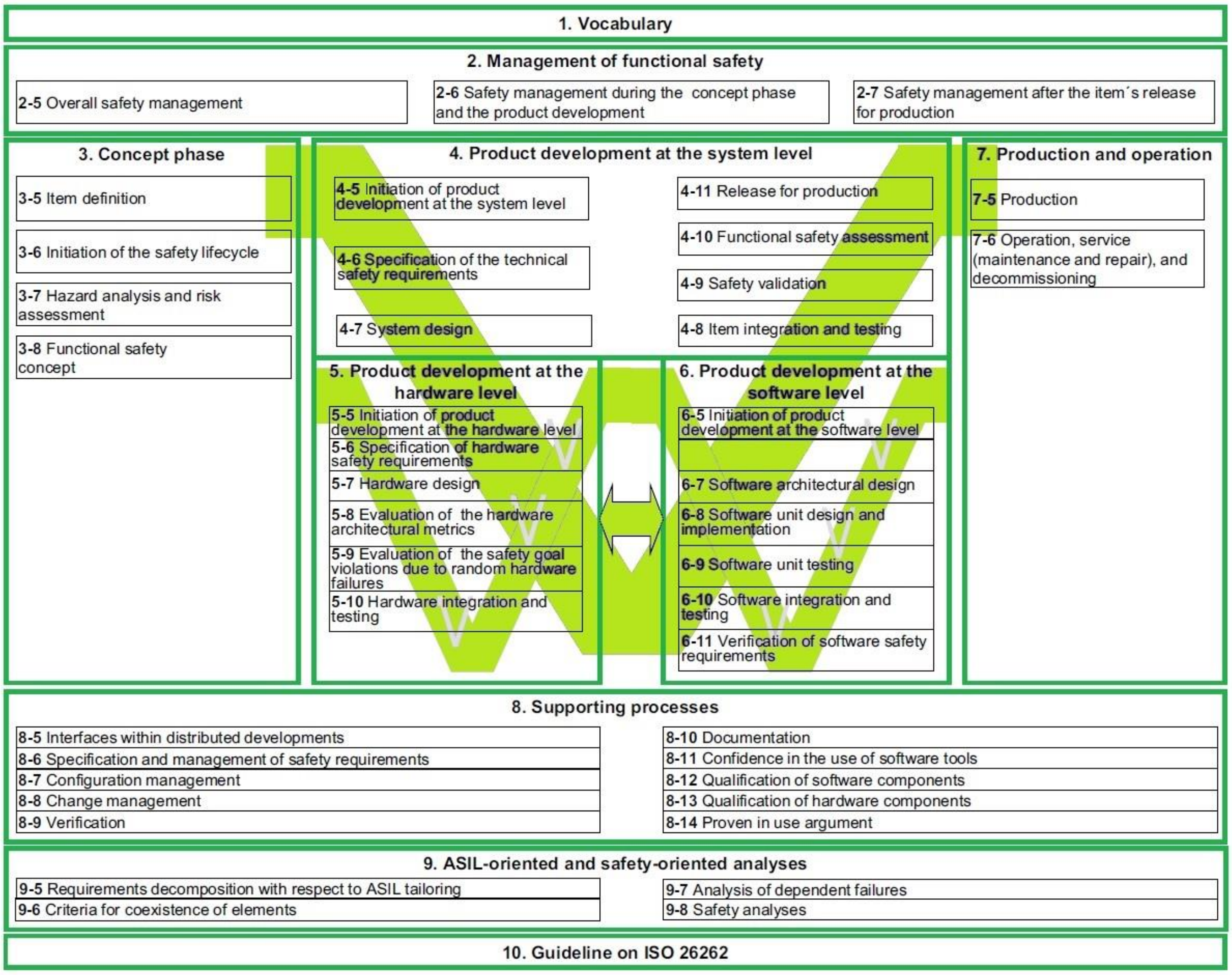

1. Vocabulary
2. Management of functional safety
2-5 Overall safety management
2-6 Safety management during the concept phase and the product development
2-7 Safety management after the item´s release for production
3. Concept phase
3-5 Item definition
3-6 Initiation of the safety lifecycle
3-7 Hazard analysis and risk assessment
3-8 Functional safety concept
4. Product development at the system level
4-5 Initiation of product development at the system level
4-6 Specification of the technical safety requirements
4-7 System design
4-11 Release for production
4-10 Functional safety assessment
4-9 Safety validation
4-8 Item integration and testing
5. Product development at the hardware level
5-5 Initiation of product development at the hardware level
5-6 Specification of hardware safety requirements
5-7 Hardware design
5-8 Evaluation of the hardware architectural metrics
5-9 Evaluation of the safety goal violations due to random hardware failures
5-10 Hardware integration and testing
6. Product development at the software level
6-5 Initiation of product development at the software level
6-7 Software architectural design
6-8 Software unit design and implementation
6-9 Software unit testing
6-10 Software integration and testing
6-11 Verification of software safety requirements
7. Production and operation
7-5 Production
7-6 Operation, service (maintenance and repair), and decommissioning
8. Supporting processes
8-5 Interfaces within distributed developments
8-6 Specification and management of safety requirements
8-7 Configuration management
8-8 Change management
8-9 Verification
8-10 Documentation
8-11 Confidence in the use of software tools
8-12 Qualification of software components
8-13 Qualification of hardware components
8-14 Proven in use argument
9. ASIL-oriented and safety-oriented analyses
9-5 Requirements decomposition with respect to ASIL tailoring
9-6 Criteria for coexistence of elements
9-7 Analysis of dependent failures
9-8 Safety analyses
10. Guideline on ISO 26262

Figure 2 Overview of the ISO 26262 standard

ISO 26262 provides three methods for using (or reusing) a software component to satisfy a safety requirement, these are:

1. **Adherent to ISO 26262:**
    a) The component was originally developed according to the ISO 26262 standard.
    b) Can be hardware or software.
    c) These components can be reviewed by a third party.
2. **Qualification of Software Components:**
    a) For software components only.
    b) Can be third party commercial off-the-shelf product, or some other already developed software component.
    c) Generally developed for use in other industries (or general-purpose).
3. **Proven in use:**
    a) For components used in systems that pre-date ISO 26262

In this master thesis project, we are focusing on the use of OSS in safety-critical applications and more specifically reuse of existing software components. The reuse of qualified software components avoids re-development for components with similar or identical functionality. In order for an existing software component to be used in an item developed in compliance with ISO 26262, evidence for the suitability of the software component shall be provided according to ISO 26262-8 sub clause 12 “Qualification of Software Components”.

3.2. Open Source Software

OSS is software that has its source code made available for free with a license in which the copyright holder provides the rights for anyone to study, change, and redistribute the software for any purpose [18]. OSS can be found in many different kinds of devices, ranging from personal computers and mobile devices to the ones used by industries such as aviation, medical, telecommunication etc. A common use of OSS is operating systems since operating systems are normally application independent and can therefore attract a large user base and can be ported to different hardware platforms.

Traditionally, software has been developed by companies in-house and the source code has been kept a secret because the company behind proprietary software thinks that if the source code of their product would be publicly available to anyone it would hold no commercial value for the company. This is called proprietary software development and proprietary software is privately owned and controlled. A lot of proprietary software has the advantage of having more “formal” software engineering methods behind them than open-source development, but this comes with significant upfront and recurring costs. Additionally, the user is dependent on the vendor to implement required changes [19]. Proprietary software development also means that the software cannot be studied openly on a larger scale which eventually could help to improve the software and its development process.

The organization of OSS projects often differs from proprietary ones in terms of their organizational structure, membership, leadership, contribution policies and quality control. Open source projects are typically created as a collaborative effort in which anyone with access to the source code can choose to improve upon it and share the changes within the project community. The easy access to the internet has resulted in a growing use of OSS due to the simplicity of starting and participating in an OSS project. As a result of this, during recent years’ companies have started to consider how to gain a competitive advantage by using OSS [20]. Software companies are releasing the source code of their commercial products and are participating in the OSS communities for further developing their products [20]. OSS is therefore no longer developed only to serve a small set of developers and their specific needs [20]. Instead there will be a growing number of users reporting bugs in the software which can be compared to beta-testing. Typically, OSS developers rely on the user base reporting bugs and volunteering co-developers that want to contribute to the project [21]. However, in this new situation where everyone has the opportunity to contribute to the project, the quality of the software becomes a very relevant issue and quality assurance must be done before a contribution reaches the final code [21].

A central aspect of any open source development is the management organization, which provides the infrastructure (e.g. repositories, etc.) and defines and monitors the processes. The contributors of course play an important role because they do the actual work on the deliverables according to the defined process. Typically, there is a lead developer or a small amount of developers together forming a core team that controls the architectural design and course of the project [22]. Volunteering developers contribute by reading the code and delivering patches of certain code sections to the core developers. The core developers then review the patches and decide if they are accepted or not.

3.2.1. Support

There is a major difference between OSS and proprietary software in how support is being handled. Generally, OSS users have multiple choices for support [23]:

1. They can choose a traditional commercial model, where they pay someone (typically a company) to provide support,
2. They can choose to provide support-in-house (designating some persons or groups to do support), or
3. They can depend on the OSS development and users' community for support (e.g. through mailing lists or forums).

These choices apply to each of the various tasks (training, installing, answering questions, fixing bugs, adding new capabilities), and the answers can even be different for the different tasks. In many cases, larger companies choose to use a more traditional support model, that is, they will pay some company to give them all that support. Choosing a traditional commercial model for support will add another problem: which company shall one choose? Unlike support for proprietary software (which is usually provided by the vendor), there may be several competing companies that are offering support. One shall favor companies that are clearly involved in the development of the software since organizations that include developers can potentially fix any problems one might have and have the fix added to the main project so that the same problem won't happen again in the future [23].

3.2.2. Maintainability

Maintainability is the ease of which a system can undergo modifications and repairs after delivery [24]. Maintainability is an important software quality attribute. Associated with this quality attribute is the maintenance process, which has long been known to represent the majority of the costs during the development of a new software component. Therefore, the maintainability of a software system can significantly impact the development costs [25].

Software maintainability is defined as "the degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers" where modifications can include corrections, improvements or adaptation of the software to changes in environment, and in requirements and functional specifications [26].
Software maintenance on the other hand is defined as "the process of modifying a software component after delivery in order to correct faults, improve performance or other attributes, or adapt to a different environment ".

From the definitions it is clear that maintenance is the process performed during the software development life-cycle whereas maintainability is the quality attribute associated with the software product. A software maintainability prediction model is a software quality model that enables organizations to predict the maintainability of their software, and therefore providing a means to better allocate their resources. This can help in reducing the maintenance effort and reduce the overall cost and time spent on a software project. Maintenance of OSS projects is quite different compared to proprietary software in general. In OSS projects, more effort is devoted on maintaining a healthy community, such as participating in threads in an issue tracking system, and answering questions through forums or mailing lists. A project which is easy to contribute to is an important aspect for OSS maintainers. OSS maintainers spend more effort on submitting good commit messages, conforming with coding styles, writing useful guides and so on [27].

Quality is the level of which a product meets the mentioned requirements or fulfils customer needs and expectations [28]. OSS developers and organizations are facing many challenges and questions regarding the quality of code when compared to proprietary software. They are worried about the level of satisfaction that can be achieved in OSS as the software is written by volunteer developers.

Both OSS and proprietary software can be assessed, utilizing basically the same methodology. Nonetheless, the way one acquires the data to assess OSS is regularly distinctive, on the ground that OSS tends to create various types of data that can be used for the assessment.

Before the emergence of OSS quality models there were already quality models in existence based on the ISO 9126 Software Engineering - Product Quality standard [29]. These models however did not factor in some quality attributes that are unique to OSS such as the community around an OSS project. Today there are quite a number of OSS quality models available. These models emerged as a result of the need to measure quality in OSS, which is quite unlike proprietary software. ISO 9126 forms the basis from which most of these models are derived from. However, ISO/IEC 9126 has been replaced by ISO/IEC 25010 [26], [30].

3.3. Software Quality

Software quality is an external software measurement attribute. Therefore, it can be measured only indirectly with respect to how the software, software development processes and resources involved in software development relate to software quality. The ISO / IEC 25010 standard [26] views quality from three different perspectives, namely:

1. **Process quality:** is the quality of any of the software lifecycle processes (e.g. requirement analysis, software coding and testing, software installation).
2. **Product quality:** can be evaluated by measuring internal metrics (metrics which do not rely on software execution) or by measuring external metrics (typically by measuring the behavior of the code when executed). Further, while the goal of quality is always to achieve end-user needs, software users' quality requirements shall play a major role in the quality assessment model. This aspect of software quality is evaluated by quality-in-use.
3. **Quality-in-use:** Measure the extent to which a product meets the needs of specified users to achieve specified goals with effectiveness productivity, safety and satisfaction in a specific context of use. In overall assessing and improving process quality improves product quality and improvements to product quality contribute to higher quality-in-use.

In proprietary software the project information is mostly confidential, whereas in an OSS development model this information is generally available to the public. Further, OSS project success / failure is highly dependent on specific software development processes and resources attributes and these types of attributes can, for the most part, be gathered for OSS and simplify their quality assessment process [31].

ISO/IEC 25010 defines:

1. A quality in use model composed of five characteristics (some of which are further subdivided into sub-characteristics) that relate to the outcome of interaction when a product is used in a particular context. This system model is applicable to the complete human-computer system, including both computer systems in use and software products in use.
2. A product quality model composed of eight characteristics (which are further subdivided into sub-characteristics) that relate to static properties of software and dynamic properties of the computer system.

The characteristics defined by both models are relevant to all software products and computer systems. The characteristics and sub-characteristics provide consistent terminology for specifying, measuring and evaluating system and software product quality. They also provide a set of quality characteristics against which stated quality requirements can be compared for completeness. The quality in use model presented in ISO / IEC 25010 defines five characteristics related to outcomes of interaction with a system, namely:

1. **Effectiveness:** The accuracy and completeness with which users achieve specified goals.
2. **Efficiency:** Resources expended in relation to the accuracy and completeness with which users achieve goals.
3. **Satisfaction:** The degree to which user needs are satisfied when a product or system is used in a specified context of use.
4. **Freedom from risk:** The degree to which a product or system mitigates the potential risk to economic status, human life, health or the environment.
5. **Context coverage:** The degree to which a product or system can be used with effectiveness, efficiency, freedom from risk and satisfaction in both specified contexts of use and in contexts beyond those initially explicitly identified.

The product quality model presented in ISO / IEC 25010 categorizes software product quality properties into eight characteristics [26]:

1. **Functional Suitability:** The degree to which a product or system provides functions that meet stated and implied needs when used under specified conditions.
2. **Performance efficiency:** The performance relative to the amount of resources used under stated conditions.
3. **Compatibility:** The degree to which a product, system or component can exchange information with other products, systems, and/or perform its required functions, while sharing the same hardware or software environment.
4. **Usability:** The degree to which a product or system can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use.
5. **Reliability:** The degree to which a system, product or component performs specified functions under specified conditions for a specified period of time.
6. **Security:** The degree to which a product or system protects information and data so that persons or other product or systems have the degree of data access appropriate to their types and levels of authorization.
7. **Maintainability:** The degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainer.
8. **Portability:** The degree of effectiveness and efficiency with which a system, product or component can be transferred from one hardware, software or other operation of usage environment to another.

The quality model proposed in ISO / IEC 25010 is well established. However, it does not provide sufficient support for assessing the quality of OSS. This is due to the particularities present in OSS development such as shared code repositories and online communities of developers and users [32]. Some quality models have been designed specifically for assessing the quality of OSS, and a comparative study [30] has been made between different OSS quality models and the ISO / IEC 25010 model. The study revealed that the Capgemini Open Source Maturity Model [33] was the most comprehensive model satisfying all eight factors under Product Quality and Context coverage under Quality in Use factors. However, it does not provide any tool support for the assessment of OSS. Tool support is an important aspect of quality assessment since it automates parts of the assessment process and reduces the time and effort for applying such methods [32]. One of the quality models compared in [30] that also offer tool support is the QualOSS quality model [34]. The QualOSS also satisfies all eight factors for Product Quality in the ISO / IEC 25010 quality model [30].

3.3.1. Capgemini Open Source Maturity Model

Capgemini Open Source Maturity Model [33] grades software against a set of product indicators and application indicators. The model points out the significance of information. The model incorporates various criteria and suggests their exhaustiveness. It includes product development, developer and user community, product stability, maintenance and training and many other factors. It was designed as a tool that could be used to compare and decide on the most suitable open source product option for an organization based on the product's maturity. The product indicators are categorized into four categories, namely:

1. **Product:** focuses on the product's inherent characteristics, age, selling points, developer community, human hierarchies and licensing.
2. **Integration:** measures the product's modularity, adherence to standards as well as options to link the product to other products or infrastructure.
3. **Use:** informs on the ease of which the product can be deployed and the way in which the user is supported in the everyday use of the product.
4. **Acceptance:** tells about the market penetration of the product and the user base formed around the product.

An open source product can't (just as any other product) be introduced into a working environment based solely on a measurement of its strengths and weaknesses. To properly assess a product, one must also take into account several environmental aspects and, naturally, the present and future demands of the user. The Capgemini Open Source Maturity Model takes these factors into account by defining the following application indicators:

- **Usability**: The intended user audience, the experience of that group.
- **Interfacing**: Required connectivity, which standards are applicable. How does this fit into the organization?
- **Performance**: The expected load and processing capability. The performance demands that must be met.
- **Reliability**: What level of availability shall the product deliver?
- **Security**: What security measures are required; what restrictions are imposed onto the product.
- **Proven technology**: Does the product use technology that has proven itself in daily production?
- **Vendor independence**: What level of commitment between supplier and user does the product demand?
- **Platform independence**: Is the product available for particular ICT environments only, or does the product allow a wide range of platforms.
- **Support**: What level of support is required?
- **Reporting**: What reporting facilities are required?
- **Administration**: Does the product allow the use of existing maintenance tools, the demands for operational management?
- **Advice**: Does the client require validation / recommendation by independent parties, if so, what is required.
- **Training**: Required training and facilities.
- **Staffing**: Is product expertise bought, taught or hired.
- **Implementation**: Which implementation scenario is preferred?

3.3.2. QualOSS

The Quality of Open Source Software (QualOSS) quality model [34] is a high-level methodology that is highly automated using measurement tools [35], [36]. The model is intended to support the quality assessment of OSS projects with a focus on evolvement and robustness. One important underlying assumption while defining the model was that the quality of a software product is not only related to the product itself (code, documentation, etc.), but also to the way the product is developed and distributed. For this reason, and since the development of OSS products is the responsibility of an open collaborative community, the QualOSS model takes both product and community-related aspects into account on an equal basis, and as comprehensively as possible.

The model groups quality characteristics into two groups: those that relate to the product, and those that relate to the community. On the product side, the QualOSS model covers the following top-level quality characteristics:

- **Maintainability*:*** The degree to which the software product can be modified. Modifications may include corrections, improvements, adaptation of the software to changes in the environment, an in requirements and functional specification.
- **Reliability:** The degree to which the software product can maintain a specified level of performance when used under specified conditions.
- **Transferability (Portability):** the degree to which the software product can be transferred from one environment to another.
- **Operability:** The degree to which the software product can be understood, learned, used and is attractive to the user, when used under specified conditions.
- **Performance:** The degree to which the software product provides appropriate performance, relative to the amount of resources used, under stated conditions.
- **Functional Suitability:** The degree to which the software product provides functions that meet stated and implied needs when the software is used under specified conditions.
- **Security:** The ability of system items to protect themselves from accidental or malicious access, use, modification, destruction, or disclosure.
- **Compatibility:** The ability of two or more systems or components to exchange information and/or to perform their required functions while sharing the same hardware or software environment.

The community side of the model, in turn, covers the following characteristics:

- **Maintenance Capacity:** The ability of a community to provide the resources necessary for maintaining its product(s) (e.g., implement changes, remove defects, provide support) over a certain period of time.
- **Sustainability:** The likelihood that an OSS community remains capable of maintaining the product or products over an extended period of time.
- **Process maturity:** The ability of a developer community to consistently achieve development-related goals (e.g., quality goals) by following established processes. Additionally, the level to which the processes followed by a development community are able to guarantee that certain desired product characteristics will be present in the product.

3.4. Real-Time Operating System

A real-time system is a system that is subject to real-time constraints. It is defined as a system in which the correctness of the system does not only depend on the result of a computation but whether or not the correct result is produced within the set time constraint [3] often referred to as deadline. A real-time system is a system that has an application-specific design with carefully specified system properties, timing constraints, high reliability and is predictable. Most real-time applications need to handle multitasking since most applications need to handle events in the operating environment that occur in parallel.

A common way of making a single-core system handle parallel events is by introducing support for concurrent programming and a common way of doing this is by using a real-time operating system (RTOS). An RTOS is a piece of software with a set of functions for users to develop applications with. It makes it easier to partition the entire software into small modular tasks that can then be executed in parallel or concurrently and separately distributed to individual programmers. It also enables the RTOS user a structured way to build multitasking applications [4]. An RTOS differs from a regular non-real-time operating system which is optimized to reduce response time while an RTOS is optimized to complete tasks within their deadlines. There are two different types of RTOSs. The most common type is called *event driven* and this type gives each task a priority and allows the task with the highest priority to execute first. The other type is called *time driven* and this type differs from event driven as it has a predefined running schedule that the tasks execute according to. By using an RTOS one is not guaranteed that the system will always meet its deadlines, as it depends on how the overall system is designed and structured.

The multitasking of an RTOS is handled by the scheduler which is a software component in the operating system. The scheduler decides which of the tasks that will be allowed to execute next depending on its priority or other factors which may vary depending on the implemented scheduling algorithm. In software systems there are also a finite amount of resources available and certain tasks may be competing for the same resources. Therefore, there is a need for task synchronization where two or more concurrent tasks ensure that they do not simultaneously execute some part of the program where they both make use the same resource at the same time since it can give rise to hazards. Task synchronization is required in a multitasking system, and where there is task synchronization there is a chance of deadlock.

3.4.1. Scheduling

An application that is running on a system can be subdivided into multiple tasks and when the tasks are executed concurrently or in parallel it is called “multitasking”. Multitasking is essential when developing real-time applications and the application must be able to respond in a predictable way to multiple, simultaneous events.

The scheduler is one of the core features of an operating system and its function is to decide which task shall execute next. If more than one task wants to use the processor at the same time, a scheduling algorithm is needed to decide which task will get to execute first. A widely used scheduling algorithm is Rate-Monotonic Scheduling (RMS) and it is a static priority assignment algorithm where the static priorities are assigned according to the periodicity of each task, the more often a task needs to execute, the higher priority it will have. Round-robin scheduling is another well-known scheduling algorithm where each task is given equal amount of execution time in a circular fashion.

3.4.2. Synchronization

In computer systems concurrently executing tasks may be either independent tasks or cooperating tasks. A task is independent if it cannot affect or be affected by the other executing tasks in the system. Any task that does not share resources with any other task is independent. A task is cooperating if it can affect or be affected by the other tasks executing in the system. Clearly, any task that shares resources with other tasks is a cooperating task [37].

Concurrent accesses to shared resources can lead to unexpected or erroneous behavior. That is why certain parts of the program where the shared resources are accessed need to be protected. This protected part is called critical region and cannot be entered by more than one task at the same time, thus enabling mutual exclusion. In a single processor system, interrupts can be disabled in order to achieve mutual exclusion. However, disabling interrupts is not a good way to handle the problem since it will block higher priority tasks that has nothing to do with the critical region from preempting the lower priority task until it is done executing its critical region. This approach also only works for single processor systems and is not feasible on multiprocessor systems. In multiprocessor systems, there is a set of instructions with the ability to atomically read and modify a memory location and these instructions are the building block of creating user-level synchronization operations such as locks, barriers and semaphores etc. [37].

3.4.2.1. Deadlock

In a computer system, there is a finite amount of resources that can only be used by one task at a time. These resources include peripherals, variables, files and so forth. In a multitasking environment, several tasks may compete for the same resources. A task that requests a resource which is not available at that time will enter a waiting state. Sometimes a waiting task is never again able to change state because the resource it has requested is being held by another task and that task won't release that resource until it has gotten a resource that is being held by the waiting task, the result of this is a deadlock, illustrated by figure 3.

A deadlock situation can arise if and only if the following four conditions hold:

1. **Mutual exclusion:** There must be some resource that can't be shared between the tasks.
2. **Hold and wait:** Some task must be holding one resource while waiting for another.
3. **No preemption:** Only the process holding a resource can release it.
4. **Circular wait:** There must be some cycle of waiting tasks T1, T2... Tn such that each task Ti is waiting for a resource held by the next task in the cycle.

To prevent deadlock, one has to ensure that at least one of the four conditions is violated at any given time. Deadlocks can also be avoided by giving the operating system additional information in advance concerning what resources a task will request and use during its lifetime. With this additional information, the operating system can decide whether the current request can be satisfied or must be delayed. The system must consider the resources currently available, the resources currently allocated to each task and the future requests and releases of each task [37].

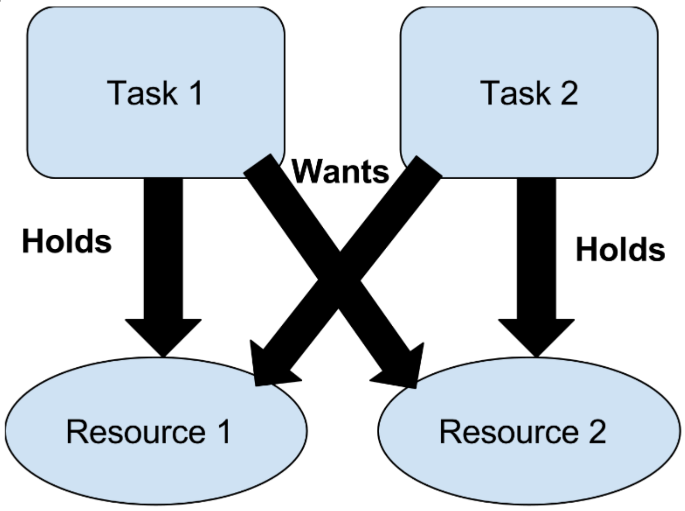


Figure 3 Illustration of a deadlock situation

3.5. MISRA C

MISRA C is a set of software development guidelines for the C programming language developed by MISRA (Motor Industry Software Reliability Association). Its aims are to facilitate code safety, security, portability and reliability in the context of embedded systems. It first appeared in the year 1998 with the objective of proving a set of guidelines to restrict features in the C programming language of known undefined or otherwise dangerous behavior. The guidelines were assembled by collaboration between vehicle manufacturers, component suppliers and engineering consultancies which seeked to promote best practice in developing safety-related electronic systems in road vehicles and other embedded systems in response to the rapidly growing use of C in electronic embedded systems [38].

MISRA C is becoming industry standard for all embedded systems, no matter their nature. The MISRA C 2012 is sometimes considered a strongly typed subset of C.

In order for a piece of software to claim to be compliant to the MISRA C Guidelines, all the mandatory rules shall be met and all the required rules and directives shall either be met or subject to a formal deviation. Advisory rules may be unimplemented without a formal deviation, but this shall still be recorded in the project documentation. In order to ensure that the source code written does conform to the guidelines it is necessary to have measures in place indicating which of the rules have been broken. The most effective means of achieving this is to use one or more of the static checking tools that are commercially available [39] and support MISRA C checking. These tools can then automatically find violations in the source code for any MISRA rules. This kind of analysis can be used as a quantitative measure to assess implementation quality. A full MISRA conformance check will require more effort than just running a static code analyzer, like a manual inspection and rigorous rule deviations. For the purpose of this study this measure is considered sufficient and indicative of code quality. As an example, Misra rule 9.2 specifies how to initialize two-dimensional arrays. Both variants shown below are valid in ISO C, but the first one does not adhere to the MISRA rule.

```
int16_t y[3][2] = { 1, 2, 3, 4, 5, 6 };                /* not compliant */
int16_t y[3][2] = { { 1, 2 }, { 3, 4 }, { 5, 6 } };   /* compliant     */
```

3.6. Software Metrics

Software metrics were first introduced in 1977 [40] and have been used to set development guidelines. These guidelines are specifically used in applications that cannot afford bugs such as safety-critical applications. In these types of applications software metrics are used during the development life cycle to assure reliability of the software [41]. Software metrics are measurements or indications of some properties of a software system; its specification and/or the software project that developed it. It is a common approach in software evaluation and development to measure the quality of software. The goal is to obtain objective reproducible and quantifiable measurements, which may have numerous valuable applications e.g. metrics used early can aid in detection and correction of requirement faults that will lead to prevention of errors later in the life cycle. However, there is still insufficient information on how to choose and select the best metrics [42].

The most important aspect of an RTOS is that it behaves according to its specification and that the intended functionality is verified. A common way of verifying software is by running it through test cases to see that it behaves correctly. Software testing is one of the most practical approaches for assuring the quality of software systems, and one of the challenges in software testing is evaluating the quality of test cases. Such quality measures are called test adequacy [43] and one of the most commonly used metrics for test adequacy is code coverage [44]. Most existing automated test generation tools use code coverage as their quality evaluator. The tools try to generate test cases that cover 100% (or as close as possible to that) of the source code [45]. It shall be noted that even with 100% code coverage it is not certain that everything has been tested, for example all variants of expressions and calculations. However, it is a good way to find places in the code that has no test cases at all. The goal of testing is to ensure correctness and predictability of a system.

Cyclomatic complexity is a metric that measures the number of linearly independent paths through a piece of code. It was introduced by Thomas McCabe in 1976 and to this day it still remains one of the most popular and meaningful measurements for analyzing code [46]. Cyclomatic complexity is simple to calculate and easily understood. Too many nested decisions make code more difficult to understand and make it more prone to errors due to the many potential flows through the code. The cyclomatic complexity value of a software component directly correlates with the number of test cases necessary for path coverage. A high cyclomatic complexity value implies high complexity and the code will have a higher defect density and the test effort is higher which also makes the code harder to maintain.

4. Method for OSS Assessment

In this chapter, a summary of the safety requirements is given followed by characteristics that are relevant for an open source RTOS in safety-critical applications. These characteristics will then be used to evaluate and compare two RTOS. A certification methodology is proposed that can be used to evaluate the suitability of an open source RTOS for use in safety-critical applications and how one shall proceed in order to certify an RTOS with regards to the two functional safety standards IEC 61508 and ISO 26262.

4.1. Safety Requirements for Operating Systems

The international standards for functional safety considered in this thesis are the IEC 61508 (see section 3.1.1) and ISO 26262 (see section 3.1.2). Relevant parts of these standards and previous studies have been analyzed in order to find attributes that can be used to assess an open source RTOS for use in safety-critical applications. A previous study [50] has been made regarding the use of Linux in safety-critical applications and three criteria were set out in order to assess the suitability of an operating system and a simplified version of these three criteria are the following:

1. The behavior of the operating system shall be sufficiently well defined.
2. The operating system shall be suitable for the characteristics of the application.
3. The operating system shall be of sufficient integrity to allow the system safety integrity requirements to be met.

Considering the first criterion, it is important that the software developer of the safety-critical application has full knowledge of the intended behavior of the operating system. This is necessary so that hazards don’t arise due to misconceptions that the application developer might have about the intended functionality of the operating system. It shall also be clear that the second criterion is necessary, since no matter how well specified an operating system might be, if it does not provide the desired functionality to support the software design chosen for the safety-critical application, it won’t be suitable for use. This can be most clearly seen in the timing domain: if the application has hard real-time requirements and the operating system cannot support deadlines then the operating system cannot be used with confidence. The third and final criterion is fairly self-evident. However, it shall be noted that what is sufficient will depend on the complete system design, including any system level measures that can mitigate operating system failures and thus allow the operating system to have a lower safety requirement than would be the case without system mitigation measures.

In order for an operating system assessment to be useful, sufficient information shall be provided that allows a reasonable judgement to be made of whether the combination of the specific software application, the operating system and the system design features meets the three criteria for a given safety-critical application.

The operating system features that were identified in [50] that are recommended to be used as a minimum to assess the sufficiency of an operating system with regard to being used in safety-critical applications can be seen below and these features have been helpful for identifying the characteristics used for comparison in this study.

1. **Executive and scheduling:** The process switching time and the employed scheduling policy of the operating system must meet all time-related application requirements.
2. **Resource management:** The internal use of resources must be predictable and bounded.
3. **Internal communication:** The inter-process communication mechanisms must be robust and the risk of a corrupt message affecting safety shall be adequately low.
4. **External communication:** The mechanisms used for communication with external devices must be robust and the risk of a corrupt message shall be adequately low.
5. **Internal liveness failure:** The operating system shall allow the application to meet its availability requirements.

6. **Domain separation:** If an operating system is used functions shall be provided that allows safety functions of lower integrity levels to not interfere with the correct operation of higher integrity safety functions.
7. **Real-Time:** Timing facilities and interrupt handling features must be sufficiently accurate to meet all application response time requirements.
8. **Security:** Only if the operating system is used in a secure application.
9. **User interface:** When the operating system is used to provide a user interface, the risk of interface corrupting the user input to the application or the output data of the application must be sufficiently low.
10. **Robustness:** The operating system shall be able to detect and respond appropriately to the failure of the application process and external interfaces.
11. **Installation:** Installation procedures must include measures to protect against a faulty installation due to user error.

In order for a piece of software to be used in a safety-critical application, the software must also be certified according to the relevant standards for the chosen field of application. For example, considering the ISO 26262 standard for the automotive industry, all the software components that are needed to perform a safety-related function must be certified with an ASIL depending on the safety requirement level required. The standard considers the use of an operating system as a component to create different execution domains, which can be used to separate the different tasks, especially the safety-relevant from the non-safety-relevant. The ISO 26262 standard also mandate that the operating system, if used, must withstand an ASIL equal or higher than the highest ASIL of all the tasks handled by the operating system.

An operating system needs to satisfy a number of requirements. These are usually similar to high-reliability systems requirements, although it must be noted that reliability and safety are two different things. While reliability is related to the percentage of time the system operates as expected, safety is related to the residual error probability in a system, i.e. errors are admitted but they must be detected, and the system must be able to react accordingly, in order to avoid damages or injuries to persons or environment.

The ISO 26262 standard has a series of requirements for how software must be developed, for every phase of the V-model as can be seen in figure 2 spanning from specification to implementation and testing. These requirements vary depending on the required integrity level.

In order to assess the suitability of using the chosen open source RTOS in a safety-critical application we will look at the desired functionality that is requested along with a number of characteristics that are considered, some of these characteristics are described below:

Implementation: Embedded software systems are often implemented in the C programming language and in order to increase the quality of the implementation it is common to define a set of coding rules that must be followed. The coding rules can include for example the conventions used to format the source code, complexity limits for modules, hierarchical organization of the modules and language subsets. For safe and reliable software written in C, a commonly used subset is the MISRA C subset (see section 2.5), which effectively defines a subset of the C programming language, removing features which are usually responsible for implementation errors or compiler dependencies. The MISRA C 2012 subset is sometimes considered a strongly typed subset of C.

Test Coverage: Test-driven development has become widespread both in proprietary software and OSS projects, in particular, the possibility to perform automated tests. The purpose of testing is to detect faults in the software component under test in order to find discrepancies between the specification and the actual behavior of the software. For example, by running the automated test after every single commit to the code base is of great help to detect bugs and ensure the intended functionality of the software. It is common to define a percentage of test coverage which is the percentage of software that is covered by the tests. There are different ways in which this percentage can be defined; the most common one is the number of lines of code executed or the number of branches taken which is often referred to as code coverage and branch coverage respectively. However, no set of test procedures can achieve 100% coverage for complex software components such as an RTOS in a practical amount of time. This has to do the number of test cases that results from choosing different combinations of input values for each test case would be too many and would be impractical. The test cases shall therefore be chosen in a manner such that the test cases test different aspects of the RTOS to maximize the test coverage. It is also recommended that an argument is made for justifying why certain test cases don't require testing.

Domain Separation: Mixed criticality systems are systems that consist of multiple safety functions where the safety functions are of different integrity levels. An important functionality, desirable especially for mixed criticality systems, is the ability to execute different tasks in different domains, where the operating system guarantee the independency of these domains. The purpose of having different execution domains is to control any additional hazards or failure conditions that may be introduced when multiple tasks are sharing the same processor, memory or other system resources. Domain separation provides fault containment between different tasks. This can make it easier to perform a safety analysis and it increases safety assurance, if properly implemented and verified. Domain separation is meant to prevent a task in one domain from overwriting or corrupting the memory of a task in another domain. A common way of implementing domain separation is through hardware support that creates different domains that allows tasks to execute in different regions of memory by using a Memory Protection Unit (MPU) or Memory Management Unit (MMU). This hardware component then performs checks whenever memory is being accessed in order to prevent unauthorized access to certain memory locations.

Another important aspect is temporal independency of tasks; this can be described using an example where a low priority task must not prevent the correct execution of a high priority task. This is usually achieved by a priority based preemptive scheduler with an appropriate priority assignment, depending for example on the criticality of the task, the deadline for periodic tasks, etc. If domain separation is implemented, other shared resources like for example a communication stack or interface must be handled so that a malfunctioning task cannot affect other tasks. An example of domain separation can be seen in figure 4.

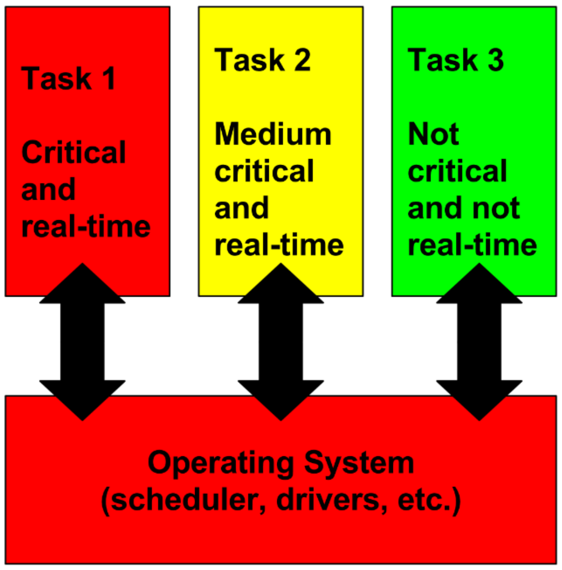

Task 1
Critical and real-time
Task 2
Medium critical and real-time
Task 3
Not critical and not real-time
Operating System (scheduler, drivers, etc.)

Figure 4 Domain separation of three tasks with different levels of criticality and real-time requirements. Color indicates the level of criticality: red high, yellow medium, green none

Real-Time Performance: In safety-critical applications, it is necessary to have timely execution of a recovery or emergency procedure, which cannot be delayed or prevented to execute. This can be given for example by an operating system with priority-based preemptive scheduling that support deadlines. However, it shall be noted that different concurrency models if appropriately used, can result in predictable execution of critical tasks. For example, often the firmware of Electronic Control Units (ECU) has no operating system, but all the tasks are implemented in the main function and executed periodically. This is similar to having a cooperative scheduler for all the tasks. In this case, if the worst-case execution time of all the tasks is known then the worst-case reaction time is also known.

These and other characteristics are evaluated in chapter 5 for the two open source operating systems looked at in this thesis. The information regarding the attributes has been gathered through information that is freely available.

4.2. Characteristics for Comparison

Based on the material from previous chapters together with information gathered from the literature survey we below present some characteristics that are relevant for an open source RTOS for use in a safety-critical context and a methodology on how one shall proceed in order to certify an open source RTOS to get compliance with the functional safety standards.

- **Coding rules:** Is the RTOS implemented with coding conventions throughout the entire code base?
- **Language subset:** Is the RTOS implemented using any defined language subset to reduce the likelihood of faults?
- **Version-control:** Is version control being used to track changes in the code base?
- **Documentation:** Is there documentation available?
  - What functionality does it cover?
- **Static resource allocation:** Does the operating system use static memory allocation? Dynamic memory allocation is not recommended in the standards since it can give rise to hazards.
- **Priority-based preemptive scheduling:** The scheduling policy needs to be priority-based preemptive in order to be used with confidence.
- **Hard real-time support:** Does the RTOS support deadlines?
- **Domain separation:** Is there support for domain separation, if so, how is it supported?
  - Hardware support.
  - Software support.
- **Synchronization primitives:** Are there synchronization primitives available (e.g. semaphores, mutexes, etc.) to allow safe inter-task communication?
- **Verification:** Are there any verification procedures to verify the functionality of the RTOS?
  - **Test suite:** Is there a test suite available?
    - Does the test suite provide test coverage metrics? (e.g. code/branch coverage)
- **Configuration (Option to turn off undesired functionality):** Is there an option to turn off undesired functionality in the operating system so that unused functionality won’t be compiled?
- **Active community:** Is the community behind the open source RTOS active?
  - **Quality assurance:** Are measures made in order to keep out “bad” code from the project code base.
  - **Bug tracking:** Is there a list of known bugs?
  - **Bug fixing:** Are bugs being fixed at regular intervals?

If the open source operating system fulfills all of the above characteristics it is a good candidate for certification since it holds some of the functionality that is desirable for a safety-critical operating system and some attributes that are preferred from an open source perspective. However, if it does not fulfill all of the above characteristics it could still be valid for certification but the effort to get compliance with functional safety standards is considerably higher.

4.3. Methodology for Certification

Based on the material from the previous sections, the process for choosing and certifying a suitable open source RTOS candidate is described below and the proposed workflow can be seen in figure 5.

First off one would have to identify open source operating system candidates. When enough candidates have been identified it is recommended to evaluate the “quality” and the maturity of the OSS projects. In our proposed methodology, we are using the Capgemini Open Source Maturity Model (see section 3.3.1) in order to see what support options there are for the project, if there are different licensing options available, does it have a stable and active community with regular updates, bug tracking and bug fixing etc. These and other criteria are then graded with certain number of points based on guidelines given in [33]. When the quality assessment has been done, one shall do a comparison of the characteristics presented in section 4.2. Based on the results produced by the quality assessment and the comparison of characteristics, one shall be able to draw a conclusion of which of the candidates is most suitable for use in a safety-critical application.

When a candidate has been identified one has to make a decision of which standard one would like to certify for. In order to assess to what degree, the open source operating system can comply with the functional safety standards IEC 61508 and ISO 26262 one has to study the available documentation of the open source operating system and other available information and compare it with the requirements for reuse of software components that are described in section 2.2 and section 2.3 However, if the required documentation is insufficient or unavailable. Then one has the option of creating the formal documentation required. It shall be noted that no documents will be created due to this in this thesis since it is out of the scope of this project and the RTOS will be assessed as is with the documentation that is currently available.

The certification process shall be carried out as follows:

1. Creation of a specification of expected system behavior. However, if a detailed specification already exists the creation of a specification is not necessary.
2. Creation of tests that are intended to verify the functionality of the operating system that can be traced back to the specification. If test cases already exist but they are insufficient, then additional test cases shall be developed.
3. Running of the test cases to validate the functionality and make corrections where necessary.
4. Manual code inspection to increase the confidence of the software. This manual inspection shall then target:
    a. Code sections of high complexity as shown by code metrics.
    b. Statements which do not conform to guidelines e.g. according to MISRA C.
5. Running the test cases with code for collecting test coverage measurements.
6. Compliance analysis, result against functional safety standard.

The output of the certification process would be the following:

1. A manual describing the certification process.
2. A detailed specification of the operating system.
3. A test suite.
4. Test results with summary of test coverage.
5. An inspection report with recommendations for code changes where anomalies have been discovered.
6. Document describing how the requirements for compliance are fulfilled according to a specific functional safety standard together with a conclusion of the certification stating if the operating system is suitable for use in safety-critical application and to what integrity level.

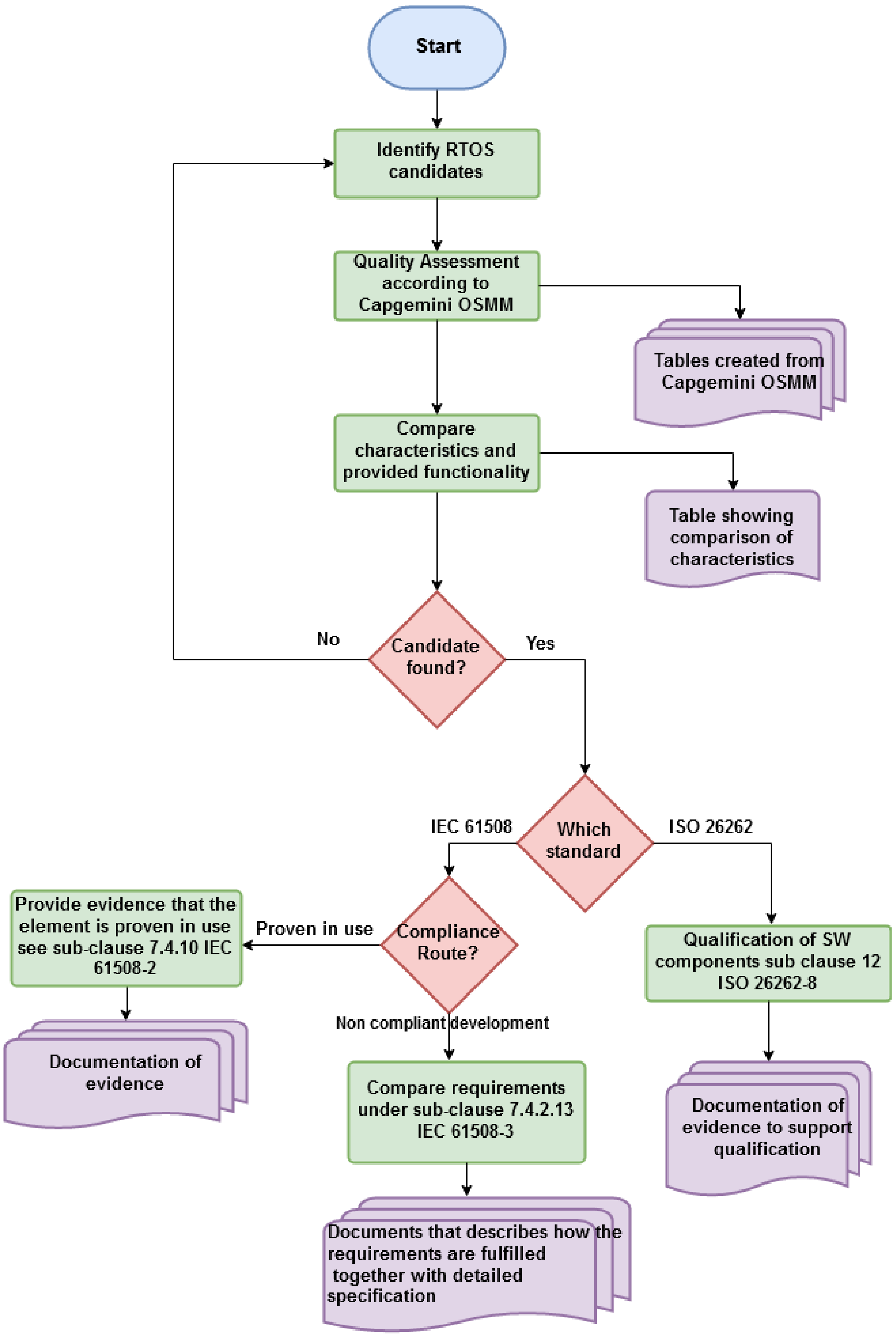

Start
Identify RTOS candidates
Quality Assessment according to Capgemini OSMM
Tables created from Capgemini OSMM
Compare characteristics and provided functionality
Table showing comparison of characteristics
No
Candidate found?
Yes
IEC 61508
Which standard
ISO 26262
Provide evidence that the element is proven in use see sub-clause 7.4.10 IEC 61508-2
Proven in use
Compliance Route?
Qualification of SW components sub clause 12 ISO 26262-8
Documentation of evidence
Non compliant development
Compare requirements under sub-clause 7.4.2.13 IEC 61508-3
Documentation of evidence to support qualification
Documents that describes how the requirements are fulfilled together with detailed specification

Figure 5 Proposed workflow for assessing an RTOS for safety-critical applications with regards to the functional safety standards IEC 61508 and ISO 26262.

5. Case Study: Evaluation of OSS RTOS

In this chapter, two open source operating systems are compared in order to find the most suitable candidate for use in a safety-critical application. The most suitable candidate is then assessed to see to what degree it can adhere to the requirements put forth in the functional safety standards IEC 61508 and ISO 26262.

5.1. ChibiOS

ChibiOS [47] is an open source project that is made up of three major software components: ChibiOS/Nil, ChibiOS/RT which are two RTOS where ChibiOS/Nil is aimed at extremely low resource usage, but with limited functionality, while ChibiOS/RT is slightly more resource demanding and has a more complete set of functionalities. The third software component is ChibiOS/HAL and it is a set of peripheral drivers for various embedded platforms which can in principle be used with any RTOS or even without an RTOS.

Implementation: The ChibiOS RTOS is implemented in MISRA C 2012, with some exceptions that are justified based on design decisions. A static code analysis has been performed using the PC-Lint analyzer, on the ChibiOS 16.1.7 release and it reported no violations of the checked MISRA rules, using the configuration provided with the source code. Additionally, the coding standard in terms of naming conventions, design patterns are explicit and consistently used on all the software. The documentation for the source code is available and generated with Doxygen which is a tool for generating documentation from source files and includes a description of all the modules and their relationship.

Test Coverage: ChibiOS has a test suite which is used to verify the functional correctness of the core mechanisms in the operating system like priority inversion, priority-based scheduling, timers and all the synchronization primitives that are offered by the operating system. The test suite can be executed both on a simulator and on real hardware and can be used for benchmarking a given hardware platform. The test suite is available for both versions of the operating system (Nil/RT) and also for the HAL so that the peripheral drivers can be tested and verified.

Domain Separation: Separation of different tasks can be done in the time domain, which is done by using priority-based preemptive scheduling. Care must be taken while using critical regions, since they can introduce unexpected latency. To avoid this, the different tasks must be developed coherently to avoid timing interference between the different tasks. The only execution model available for ChibiOS is single process-multi thread execution model, which means that all tasks share the same addressing space and memory separation is not implemented which means that mixed-criticality is not possible with ChibiOS. However, it is possible by using a Memory Protection Unit (MPU) or Memory Management Unit (MMU) in some of the supported architectures.

Real-Time Performance: The ChibiOS/RT scheduler is a priority-based preemptive scheduler with Round-Robin scheduling for tasks at the same priority level and it is suited for hard-real time systems. The time overhead of operations like context-switch, interrupts and synchronization primitives can be obtained by running the test suite on the target hardware.

5.2. ContikiOS

ContikiOS [48], [49] is a lightweight open source operating system that is highly portable and is designed to run on Internet of Things devices with limited resources such as memory. ContikiOS provides three network mechanisms: the uIP TCP/IP stack which provides IPv4 networking, the uIPv6 stack which provides IPv6 networking, and the Rime stack which is a set of custom lightweight networking protocols designed for low-power wireless sensor networks.

Implementation: ContikiOS is mainly developed in standard C (ISO C) and is portable to various platforms. The coding rules used only covers the formatting style and can be found in the documentation of the project. A MISRA C compliance check has been performed using PC-Lint static analyzer with the default configuration for MISRA C checking. All the files under the dev/, cpu/, platform/ and core/ directories of the ContikiOS 3.0 code base have been checked. In total over 70 000 messages were generated by the static analyzer most of them relating to either styling issues or errors that are easily correctable. However, there were also reports of more severe errors such as:

- Recursive functions
- Discarding of volatile or constant qualifiers
- Variable shadowing
- Uninitialized variables
- Unused return codes from functions
- Dereferencing of null pointer
- Buffer overruns

Test Coverage: On every new version of ContikiOS regression tests are performed. However, they seem to mostly cover the communication protocols and not the functionality of the operating system like scheduling, inter-task communication and synchronization primitives.

Domain Separation: Memory separation is not used because of lack of support in most supported architectures. The scheduler does not support priorities; it only handles two types of tasks: cooperative and preemptive. There are also no well-defined mechanisms for inter-task communication and concurrency issues must be handled manually.

Real-Time Performance: The scheduler in ContikiOS does not support different priorities for tasks. However, the tasks can be either cooperative or preemptive and a number of timer modules exist and these can be used to schedule preemptive tasks.

5.3. Quality Assessment of OSS

The quality of the two open source operating system looked at in this thesis have been evaluated with the Capgemini Open Source Maturity Model (see section 3.3.1) and with the help of the guidelines given in [33].

First the product indicators that were mentioned in section 3.3.1 form the basis of the model. Using these indicators, the maturity of the open source products can be determined. Product indicators receive a score valued between one and five. One is poor, five is excellent and 3 is average. All the scores are summed to produce a product score. A comparison of the two products can be seen in table 1 and this data allows us to determine a score for each group of indicators. This makes it possible to compare the products on a group basis and we can draw the graph shown in figure 6.

Table 1 Comparison of product indicators according to Capgemini Open Source Maturity Model (see section 3.3.1)

Indicator	ChibiOS	ContikiOS
Product		
Age	5	5
Licensing	5	3
Human hierarchies	5	5
Selling points	3	3
Developer community	3	3
Integration		
Modularity	5	5
Collaboration with other products	3	3
Standards	5	3
Use		
Support	3	3
Ease of deployment	3	3
Acceptance		
User community	3	1
Market penetration	3	1
Total	46	38

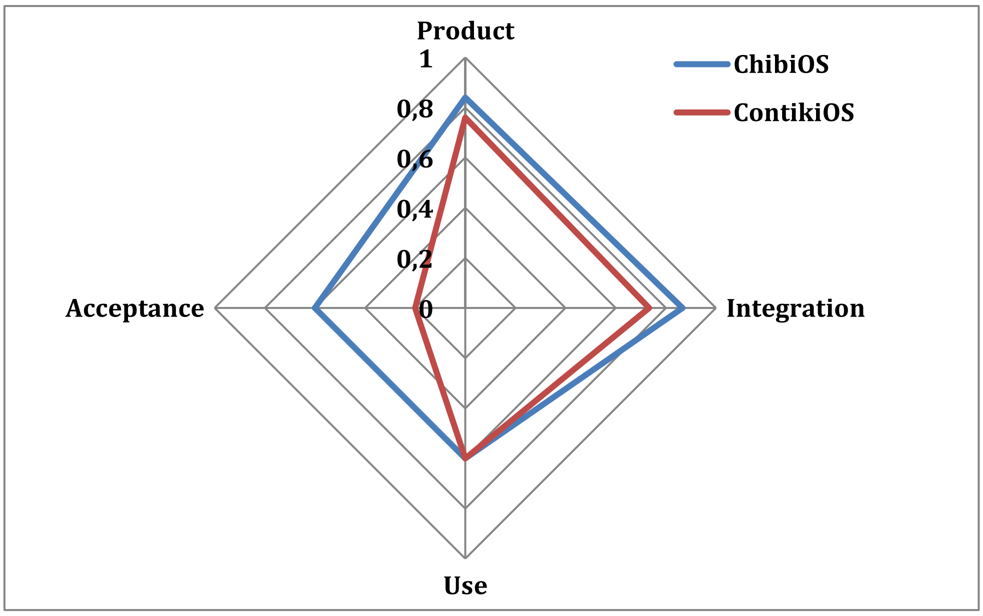


Figure 6 Score per group with data from table 1 where each axis has been normalized to the max value of 1 since the groups have different max values.

However, the data in table 1 and figure 6 are only a comparison of the products strengths and weaknesses. To properly assess the product, we must also take into account the application indicators described in section 3.3.1.

Table 2 Application indicator comparison according to Capgemini Open Source Maturity Model

		ChibiOS		ContikiOS	
Indicator	**Priority (P)**	**Score (S)**	**PxS**	**Score (S)**	**PxS**
Usability	5	5	25	5	25
Interfacing	3	3	9	3	9
Performance	5	4	20	3	15
Reliability	5	5	25	4	20
Security	3	3	9	3	9
Proven technology	3	3	9	3	9
Vendor independence	4	4	16	4	16
Support	4	3	12	2	8
Reporting	2	4	8	2	4
Administration	2	3	6	3	6
Advice	1	1	1	1	1
Training	3	3	9	3	9
Staffing	3	2	6	2	6
Implementation	3	4	12	2	6
Total			167		143

From the data presented in figure 6 and table 1 and table 2 we can see that the most suitable candidate for our given purpose is ChibiOS. However, it shall be noted that these values do not guarantee a certain quality of the product they are just indicators of which product has the highest “quality” of the development.

5.4. Comparison of RTOS Characteristics

In this section, the requirements presented in section 4.1 will be evaluated together with the characteristics presented in section 4.2 to find the most suitable open source operating system option. The most suitable operating system will then be assessed to see how well it can comply with the functional safety standards IEC 61508 and ISO 26262.

A quality assessment has already been made in section 5.3 between the two open source operating systems to indicate which option has the highest quality of development. This comparison is helpful but not nearly sufficient to make a decision. Therefore, the characteristics presented in 4.2 will be used to compare the two open source operating systems and this comparison can be seen in table 3.

Table 3 Comparison of important characteristics, an "X" indicates that the operating system is a holder of the characteristic and "-"indicates that it is not available

Characteristic	ChibiOS	ContikiOS
Coding rules	X	X
Language subset	X	-
Version control	X	X
Documentation	X	X
Static resource allocation	X	X
Priority-based preemptive scheduling	X	-
Hard real-time support	X	-
Domain separation support	X	-
Synchronization primitives	X	-
Verification	X	X
Test suite	X	X
Configuration	X	X
Active community	X	X
Quality assurance	X	X
Bug tracking	X	X
Bug fixing	X	X

5.5. Assessment of ChibiOS

As we can see from the quality assessment made in section 5.3 together with the comparison of characteristics in table 3 we can see that the most suitable choice for use in a safety-critical context is ChibiOS. Therefore, we will assess ChibiOS and see to if it can comply with the functional safety standards IEC 61508 and ISO 26262.

5.5.1. IEC 61508 Assessment

As described in section 2.2 we will for the IEC 61508 standard assess ChibiOS according to the subclause 7.4.2.13 of IEC 61508-3 since it is developed with a development process that is non-compliant with the one proposed in IEC 61508. The RTOS will be regarded as a context free software component. A safety manual shall be provided that gives sufficiently precise and complete description of the software element to make possible the integrity of a specific safety function that depends wholly or partly on the pre-existing software element. A complete description of the various functions of the RTOS is available in the reference manual [51] together with the supporting book [52].

In order to assess ChibiOS with the requirements of subclause 7.4.2.13 of IEC 61508-3 we will have to go through the documentation that is available for ChibiOS and state how these requirements are fulfilled. These requirements and how they are fulfilled can be seen below. However, a detailed description of the requirements is not given here. For a more detailed description of the requirements, see [10].

1. **There is a requirement that a software safety requirements specification shall be available for the software element in its new application and that it shall be documented to the same degree of precision as would be required by the IEC 61508 standard. This specification shall also cover the functional and safety behavior of the software element in its new application.**

   The software safety requirements specification is a specification that contains all system safety requirements, in terms of system safety function requirements and the system safety integrity requirements, in order to achieve the required functional safety. The software safety requirements specification must be valid in the system context. A pre-existing operating system is unlikely to have any specific safety requirements defined since it is not bound to a specific context, therefore in this thesis we assume that the requirement of a software safety requirement specification can be replaced by the requirement that the behavior of the operating system shall be precisely defined.

   For ChibiOS there is a detailed reference manual [51] covering all the functionality of the real-time kernel together with a supporting book [52] that describes the architectural design of the operating system and how all the kernel submodules works.

   For safety functions considered to be of SIL 1 and SIL 2 it is recommended to use semi-formal methods and for SIL 3 they are highly recommended. Semi-formal methods are used to express parts of a specification unambiguously and consistently, so that some types of mistakes, such as wrong behavior can be detected. There are multiple techniques recommended as semi-formal methods in order to model, verify, specify or implement the control structure of a system, one of which is finite state machines as can be seen in Appendix A.

   If we look at the documentation that is available for ChibiOS we can see that when the ChibiOS real-time kernel is used the system can be in one of the given multiple logical states [53]. One can also see transitions between the different states and what event that triggers the transition. However, a finite state machine containing all the different submodules that can be found in the operating system would be too extensive for one state machine; therefore, ChibiOS provides finite state machine diagrams for all their modules that are implemented as a state machine e.g. the binary semaphore implementation [54]. The IEC 61508 standard states that state transition diagrams can apply to the whole system or to some objects within it.

2. **There are requirements for implementing code and it is highly recommended that coding rules are being followed and that each module of software is reviewed in order to assure a certain quality of the code.**

   ChibiOS is an open source project that is split in two parts:

   1. The core ChibiOS codebase. This code is thoroughly tested and maintained, bugs are tracked and fixed, and the code is released in stable packages regularly.
   2. The community overlays codebase. All the community contributed code goes initially into this public repository without review. It is an incubator, the code that survives time and user's feedback can eventually be migrated to the core codebase. The repository is organized as an "overlay" over the core codebase. That is the core codebase together with community contributed code. The two parts together form the global ChibiOS project [58].

   Therefore, in order for code to be added to the core codebase, the code has to follow strict coding guidelines and go through extensive reviews and testing. Therefore, one can expect the code in the core codebase to be thoroughly tested, reviewed and documented.

3. **There are requirements that recommended practices shall have been considered for the software architectural design for safety related software and evidence shall be provided that justifies the use of a software element.**

   The recommended practices for the software architectural design of safety related software can be seen in Appendix B. Some of these are however not applicable to an operating system such as graceful degradation which is something that needs to be implemented on the application level. Some of the recommended techniques and measures for high integrity software are that the software shall be designed with a modular approach and the behavior of the software shall be modeled with semi-formal methods (e.g. finite state machines). Other recommended practices are to use static resource allocation for more predictable resource usage and it is recommended to have a time-triggered architecture with cyclic-behavior.

   ChibiOS has a modular design [55]; it is internally divided in several major independent components. It uses static resource allocation, which means that everything in the kernel is static; nowhere memory is allocated or freed. There are allocator subsystems but those are optional and not part of the core operating system. If the memory allocator subsystems are to be used, then evidence has to be provided that proves that failure as a cause of using these subsystems won't affect the safety-related behavior of the system. ChibiOS has a time-triggered architecture [62], when the processor does not currently have anything to execute it will be running the idle thread which is an endless loop waiting for an interrupt to occur which will invoke the scheduler and check if some task is ready for execution. The scheduler also implements cyclic behavior, it will prioritize tasks of higher priority and if there are multiple tasks at the same priority level it will schedule them according to the Round-Robin scheduling algorithm [62].

4. **For all integrity levels there it is highly recommended that a suitable strongly typed programming language is used together with a language subset. In section 4.1 we can see what implementation requirements there are for embedded software systems in a safety-critical context.**

   ChibiOS fulfills these requirements since it is mainly written in C with small code sections written in assembly language. C is a suitable programming language since ChibiOS is developed for embedded systems where performance is important with limited amount of resources, such as memory. Although C was not specifically designed for this type of application, it is widely used for embedded and safety-critical software for several reasons. The main properties of note are control over memory management (which, for example allows you to avoid having to garbage collect), simple, well debugged core runtime libraries and mature tool support. While manual memory management code must be carefully checked to avoid errors, it allows a degree of control over application response times that is not available with languages that depend on garbage collection. The core runtime libraries of the C language are relatively simple, mature and well understood, so they are amongst the most suitable platforms available. ChibiOS also implements the C subset MISRA C 2012 which is sometimes considered a strongly typed subset of C.

   Another important fact is that you can find a C compiler for the vast majority of embedded devices. This is not true for every high-level language.

5. **There are requirements that the software produced shall be modular and each software module shall be verifiable and testable with measurements indicating the test coverage.**

   For specifying the detailed software design, it is again highly recommended to use semi-formal methods, a modular approach with design and coding standards. If we look into the architecture of ChibiOS we can see that it has a modular design [55]; it is internally divided in several major independent components where the components themselves are divided in multiple subsystems. We can see that the behavior of different submodules is modeled with finite state machines as well as strict coding conventions.

There are also requirements that the implementation shall be verifiable and testable and this is done by the test suite provided. The kernel code of ChibiOS is subject to rigorous testing by the core development team. The test suite aims to test all the kernel code and reach a code coverage as close to 100% as possible. In addition to the code coverage, the kernel code is tested for functionality and benchmarked for speed and size before each official release. In addition to the code coverage and functional testing a batch compilation test is performed before each release. The kernel is compiled by alternatively enabling and disabling all the various configuration options, the kernel code is expected to compile without errors nor warnings and execute the test suite without failures (a specific simulator is used for this execution test, it is done automatically by a script because the entire sequence can take hours). All the tests results and benchmarks are included as reports in the operating system distribution under ./docs/reports [56].

Code that is ported to certain hardware platforms is tested by executing the kernel test suite on the target hardware. The ported code is valid only if it passes all the tests. Speed and size benchmarks for all the supported architectures are performed, both size and speed regressions are monitored [56]. The Hardware Application Layer (HAL) high level code and device driver implementations are tested through specific test applications that can be seen in [57].

The functionality of ChibiOS is subject to rigorous testing through its test suite. This test suite aims to test all the kernel code and tries to reach a code coverage as close to 100% as possible. In addition to this the kernel code is tested for functionality in order to verify that the software behaves according to its specification. The various modules of the kernel are also modeled using finite state machines as can be seen in [59] for semaphores, mutexes (for mutual exclusion) and condition variables **[**60].

6. **There are requirements on how the programmable electronics integration (hardware and software) is being handled and appropriate software system integration tests shall be available to ensure that the behavior of the software can be validated on different hardware platforms. The test specification shall state the test cases, test data and which types of tests to be performed. The integration testing shall be documented, stating the results of the tests. If there is a failure, the reason for the failure shall be documented.**

Different recommended techniques/measures that are recommended for software testing and integration can be seen in Appendix C. Software module testing is a way of verifying the functionality of a software module. The software verification shall show whether or not each software module performs its intended function and does not perform unintended functions and the results shall be documented with test coverage.

Most of the ChibiOS demos link a set of software modules (test suite) in order to verify the proper working of the kernel, the port and the demo itself. The operating system components are tested in various modes depending on their importance [56].

- **Kernel:** The kernel code is subject to rigorous testing. The test suite aims to test all the kernel code and reach a code coverage as close to 100% as possible. In addition to the code coverage, the kernel code is tested for functionality and benchmarked for speed and size at each stable release. In addition to the code coverage and functional testing a batch compilation test is performed at each release, the kernel is compiled by alternatively enabling and disabling all the various configuration options, the kernel code is expected to compile without errors nor warnings and execute the test suite without failures (a specific simulator is used for this execution test, it is done automatically by a script because the entire sequence can take hours).
- **Ports**: Code that is ported to a specific hardware platform is tested by executing the kernel test suite on the target hardware. A port is valid only if it passes all the tests. Speed and size

benchmarks for all the supported architectures are performed, both size and speed regressions are monitored.

- **HAL**: The HAL high level code and device driver implementations are tested through specific test applications under ./testhal and can be seen in [57].
- **Various**: The miscellaneous code is tested by use in the various demos.
- **External Code**: Not tested, external libraries or components are used as-is or with minor patching where required, problems are usually reported upstream.

Kernel Test Suite: The kernel test suite is divided in modules or test sequences. Each test module performs a series of tests on a specified kernel subsystem or subsystems and can report a failure/success status and/or a performance index as the test suite output. The test suite is usually activated in the demo applications by pressing a button on the target board; see the readme file into the various demos directories. The test suite output is usually sent through a serial port and can be examined by using a terminal emulator program [56]. All the tests results are included as reports in the operating system distribution under ./docs/reports.

7. **There are requirements for validation of software aspects of safety related functions.**

   Software usually cannot be validated separately from its underlying hardware and system environment and the validation of safety related functions is bound to a very specific context. Therefore, we assume that this requirement can be replaced by the requirement that the functionality of the operating system shall be precisely defined and verifiable. This verification can be done by running the test suite on a specific hardware platform as described earlier.

8. **In the IEC 61508 standard there are also requirements for how software modification shall be handled. If the modification has an impact on something that is safety-related, then a hazard and risk analysis has to be performed in order to detect any possible hazards that might arise due to the modification.**

   Due to this requirement, if one chooses to use ChibiOS in a safety-critical context it shall be used as is and shall not be modified. However, if for some reason the source code of ChibiOS were to be modified, a detailed specification is needed for the modification with a given verification plan for the software modification. An analysis shall also be carried out on the impact of the proposed software modification on the functional safety of the system context in order to determine whether or not a hazard and risk analysis is required. The results from the impact analysis shall then be documented.

9. **There are requirements that a functional assessment shall be carried out for all the safety-related functions.**

   Functional safety assessment is context bound and as stated earlier we are in this thesis considering the RTOS as a context free software component. We have looked at the recommended techniques and measures that are to be used for strengthening the evidence that a software component provides a certain level of integrity and compared them with what is provided by the ChibiOS documentation and how the software is designed and implemented. A general description of the requirements and how they are fulfilled are been presented above and a conclusion will be drawn and presented in chapter 7.

10. **When software elements are present which are not required in the safety-related system, then evidence shall be provided that the unwanted functionality will not prevent the system from meetings its safety requirements.**

    If ChibiOS is used in a safety-critical context, the configuration used shall be documented. When certain functions in ChibiOS aren't required by the safety-related system, the unwanted functions can be removed from the build by disabling them in the ChibiOS configuration file so that they won't be compiled at all. Functionality that can be disabled/enabled or modified can vary from hardware peripheral drivers, software subsystems, debugging options, speed optimization, system tick frequency etc. [63], [64].

11. **There shall be evidence that all credible failure mechanisms of the software element have been identified and that appropriate mitigation measures have been implemented.**

    ChibiOS provides support for domain separation different of tasks in the time domain, which is done by using priority-based preemptive scheduling. Care must be taken while using critical regions, since they can introduce unexpected latency. To avoid this, the different tasks must be developed coherently to avoid timing interference between the different tasks. The only execution model available for ChibiOS is single process-multi thread execution model, which means that all tasks share the same addressing space and memory separation is not implemented which means that mixed-criticality is not possible with ChibiOS. However, it is possible by using a MPU or MMU in some of the supported architectures

    Mitigation measures shall also be done at the application level if it is considered necessary in the context of use.

5.5.2. ISO 26262 Assessment

ISO 26262 does not provide a specific method for certifying OSS in safety-critical applications since certification is intended for software developed specifically according to the ISO 26262 standard. “Proven in use” is specifically for software previously used and that is not being changed in any way. This allows components developed and deployed before ISO 26262 to be reused if enough evidence is provided that support the claim of proven in use. As described in section 2.3 we will for the ISO 26262 standard assess ChibiOS with the requirements given in ISO 26262-6 Product Development at the Software Level as well as looking at the requirements given subclause 12 of ISO 26262-8 Qualification of Software Components. The RTOS shall be regarded as a context free software component.

From a general point of view, all the sections of the ISO 26262:2011 standard must be addressed (at the appropriate ASIL) to be considered compliant. The high-level steps for compliance will include creation and approval of a safety plan, safety goals and safety cases along with a complete safety lifecycle with bi-directional traceability. The activities and work products shall include verification, validation, independent assessment (by an accredited certification body) and a complete list of documentation supporting the required activities. The organization of ISO 26262-6 and other ISO 26262 parts are relevant for approval, assessment and reuse of software components intended to be used in a variety of applications. The organization of the software lifecycle requirements and all the activities that shall be carried out in ISO 26262 are staged. This means that the outputs of one phase are the inputs to the next phase. The other parts of ISO 26262 that are relevant to software components include part 2 (Management of Functional Safety) and part 8 (Supporting Processes). The software developer shall demonstrate that the internal planning processes are aligned with requirements given in part 2. At a minimum, this shall include a functional safety management plan and a quality management plan as well as evidence of safety culture and personnel who are trained and responsible for enforcing the safety culture in both the development and production phases. Additional plans that would be required (and defined in the safety plan or quality plan) are the verification plan, validation plan and test plans that demonstrate adherence to ISO 26262 requirements. Once a supplier can support the requirements of part 2, they have a framework to ensure their software developed under part 2 requirements can support the other parts of ISO 26262

In order to assess ChibiOS with the requirements for compliance in ISO 26262 we have also looked at the requirements given in subclause 12 of ISO 26262-8 Qualification of software components because that subclause specifically targets reuse of software components such as commercial off-the-shelf software or previously developed software components. The objective of the qualification process is to provide evidence for the suitability of a software component developed according to other safety standards to be reused in items newly developed in compliance with ISO 26262. Because of this, it won’t be possible for ChibiOS to fulfill the requirements for any of the ASILs since there is a lack of formal documents that would have been produced in the earlier stages of the V model as can be seen in figure 2. These documents are necessary in order to fulfill the requirements for compliance with the ISO 26262 standard.

6. Example: Safety-Critical Application

In this chapter, a specification of a safety-critical application is given followed by its implementation. This is done in order to show how safety functions can be specified in a given context.

6.1. System Specification

The specification presented here is an example of a safety-critical application where an RTOS is used. In this thesis, it is used to demonstrate how certain functionality of the RTOS is used to implement certain Safety Functions (SFs). The example application comes from an industrial context where Automated Guided Vehicles (AGVs) are used to move materials around a manufacturing facility or warehouse [65], [66]. The AGV can navigate itself through vision guidance e.g. following a visual line painted or embedded in the floor.

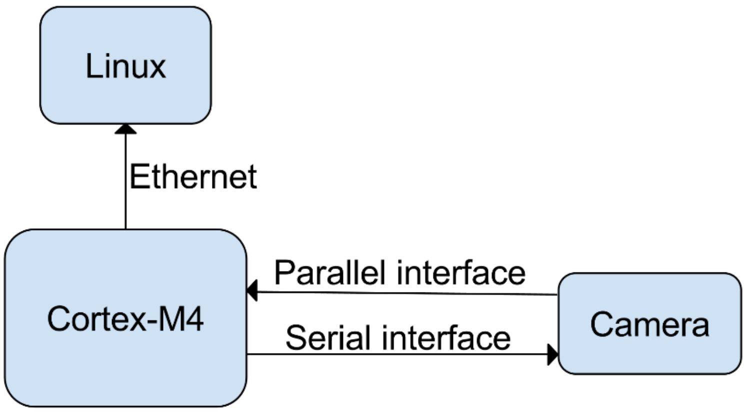


Figure 7 Overview of the system. On the cortex-M4 microcontroller, a safety-critical application shall be running on an RTOS. The application shall be capable of capturing real-time video and stream the video to a Linux computer over Ethernet at a steady frame rate. The microcontroller shall also be able to do image processing on the captured video before transmitting it over Ethernet.

The Cortex-M4 microcontroller shall run ChibiOS which was evaluated in chapter 5. The microcontroller shall configure the camera module over a serial communication interface. When the camera has captured an image, the pixel data of the image shall be transmitted over a synchronous parallel interface to the microcontroller where the image will be processed before it is to be transmitted to a Linux computer over Ethernet.

The system shall implement fault detection, if one of the subsystems fails it shall be detected and the system shall enter safe state. Graceful degradation shall be implemented, if it cannot achieve a steady frame rate of 15 frames per second (FPS) at the resolution 160x120 it shall lower the camera resolution. If it cannot lower resolution more than the minimum resolution of 160x60, then the system shall be stopped and kept stopped (safe state).

Safe state: The system shall be stopped and kept stopped with all functionality disabled and it shall be indicated that the system has failed by blinking a red LED.

Safety functions (SFs):

SF 1: If possible, the resolution shall be lowered in order to keep a steady frame rate of 15 FPS. If not possible the system shall enter safe state.

SF 2: If a subsystem fails the system shall enter safe state

SF 3: The system shall complete the image processing of one frame in 65 milliseconds. If not possible; the system shall enter safe state.

SF 4: If the system loses track of the white line on the floor the system shall enter safe state.

6.2. Implementation

The system that is specified in section 6.1 is implemented on the development board that can be seen in figure 8. On the microcontroller expansion card, there is a STM32F407VGT6 microcontroller from STMicroelectronics and together with this is an OV7725 camera module from Omnivision as can be seen in figure 9. The microcontroller used also has a watchdog timer which can be used to reset the system if something goes wrong.

While implementing the system the image processing turned out to be too heavy for the microcontroller while sustaining a stable framerate of 15 FPS. Therefore, we lower this requirement of **SF 1** to 5 FPS and increase the deadline of one frame in **SF 3** to 200 ms in our implementation. A solution to this could be to move the image processing over to the system running Linux in order to achieve this requirement, but in our example, we want the safety-critical functions to be carried out on the microcontroller. The reason is that the embedded system is less complex than a system running a full Linux distribution and would make a safety certification of such system more likely.

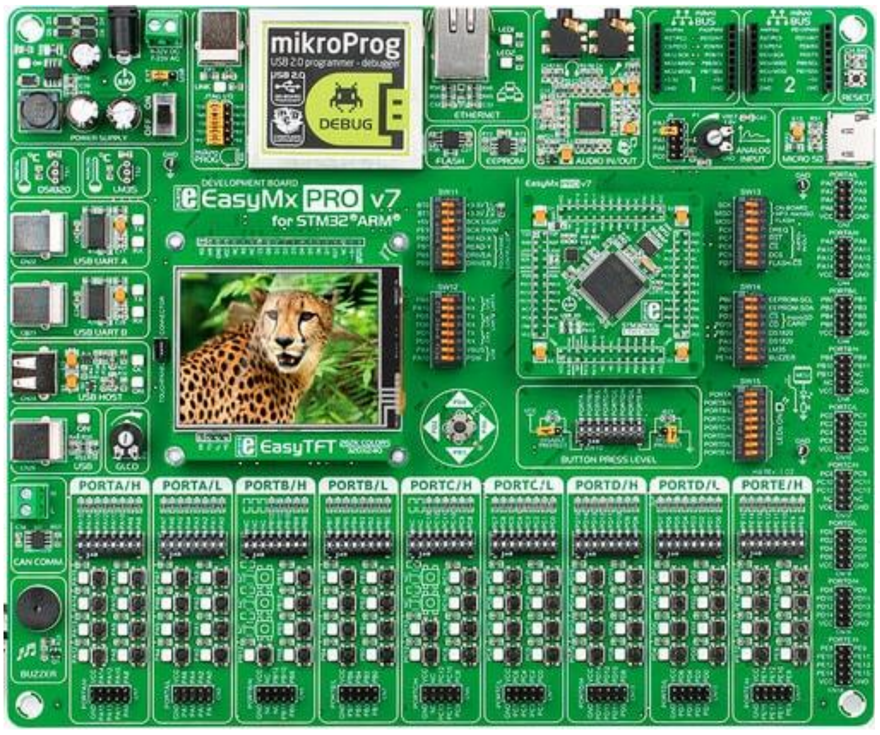


Figure 8 Easy MX PRO v7 Development Board

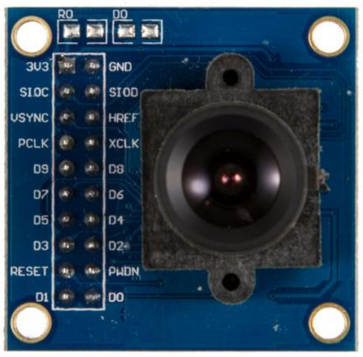


Figure 9 OV7725 Camera Module

The microcontroller initializes the camera module through the Inter-Integrated Circuit (I2C) serial communication protocol where settings like resolution, image settings (contrast, gamma, brightness etc.) and output format are configured among other things. The camera module is configured to output pixel data in the format RGB565 which means that each pixel will require two bytes of data as can be seen in figure 10.

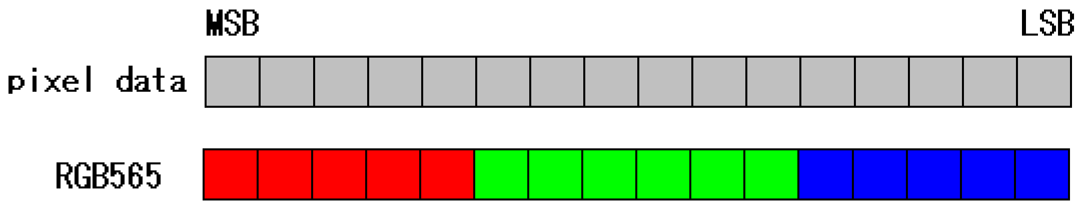


Figure 10 Pixel map of RGB565

In order to capture the pixel data received from the camera module we make use of the microcontrollers Digital Camera Interface (DCMI). The DCMI is a synchronous parallel interface that is able to receive high-speed data from an external CMOS camera module. The DCMI is configured to receive RGB565 pixel data through an 8-bit parallel interface. The data flow is synchronized by hardware using horizontal and vertical synchronization signals. The DCMI stores the received data in a 32-bit register before requesting a Direct Memory Access (DMA) transfer to a RAM buffer.

The RAM buffer is handled by the DMA controller and not by the DCMI. The DCMI is configured to capture pixel data until the buffer has been filled by the DMA controller. When the buffer is full, an interrupt is generated by the DMA controller and the DMA transfer from the DCMI to RAM is disabled and a flag is set in order to indicate that the picture is ready for processing. When the image processing has been completed the data is to be transmitted through a TCP socket over Ethernet. When the data has been sent and an acknowledgement has been received the whole process is repeated.

Before an image is to be captured the microcontroller reads from a static register in the camera module. This is done in order to make sure that the camera module is present. If the data received is correct the microcontroller will command the camera module to capture an image. However, if there is no data received or if the data is incorrect we assume that something is wrong and the system will enter safe state.

The camera module is configured to always output an image with the resolution 320x240 which is too big for the microcontroller’s memory; therefore, we have configured the DCMI to crop the captured image so we only capture a certain part of the image which we then perform edge detection on. If the microcontroller fails to complete the image processing of one 160x120 image within its deadline of 200 ms the system will first reconfigure the cropping register of the DCMI to only receive an image resolution of 160x60 in order to speed up the image processing. However, if the deadline is missed a second time the application will enter safe state.

When the image processing has been completed the embedded application will transmit the processed image over a TCP socket. The application running on Linux will then send an acknowledgement message when the picture has been received. However, if there is no acknowledgement message received within a time interval of 50 ms the embedded application will enter safe state. If the application gets stuck in any of the cases for too long the 250 ms watchdog timer will eventually reach zero and it will trigger a reset of the application. If the application ends up in the safe state the watchdog timer is disabled. A state diagram of the application can be seen in figure 11.

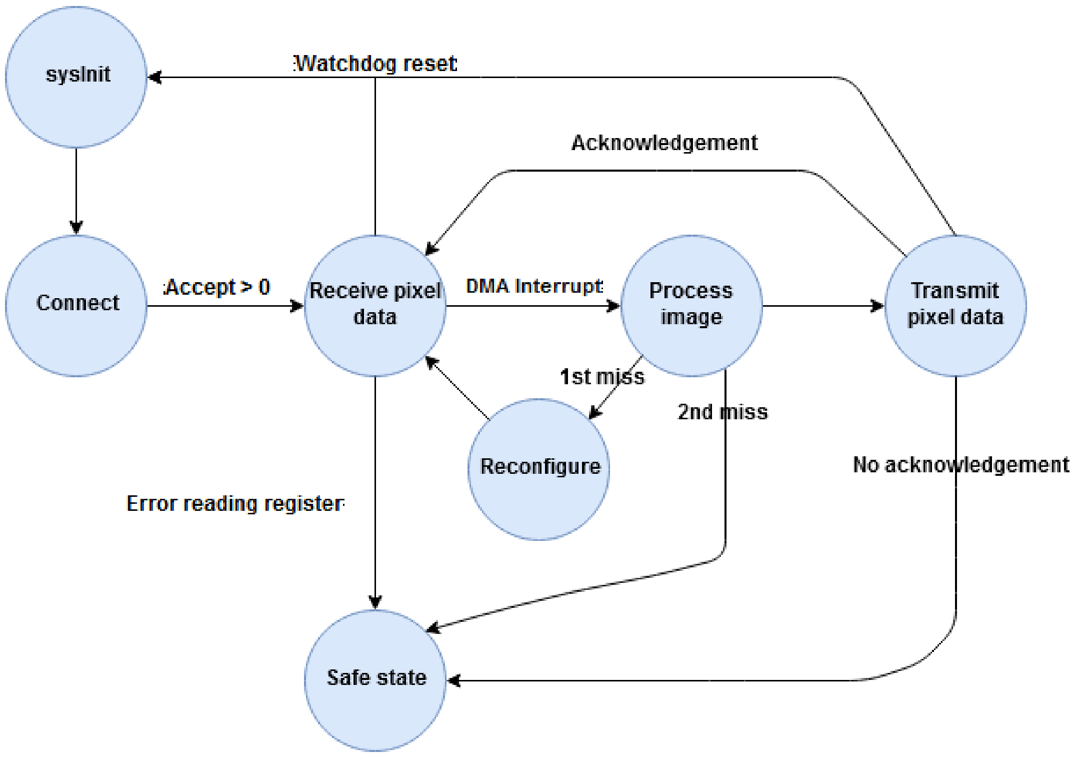


Figure 11 State diagram of example application

The RTOS used is ChibiOS and it is used for scheduling the various tasks of the application so that the most critical tasks will be prioritized and to make sure that deadlines will be met. ChibiOS have an API for peripherals such as I2C, Ethernet, DMA and GPIO that have been helpful while developing the application. ChibiOS also has support for the LwIP TCP/IP stack that has been used to handle the communication over Ethernet.

A C++ application has been developed on Linux that connects to the embedded device through Ethernet and receives the pixel data over a TCP socket and then displays the image.

6.2.1. Edge Detection

In order to locate a line in the image received from the camera module we implemented a Sobel operator using only the horizontal Sobel kernel. This allows us to only locate horizontal edges in an image as can be seen in figure 12. Sobel filters are commonly used in image processing and computer vision, particularly within edge detection algorithms where it creates an image emphasizing edges. This technique is sometimes used for detecting other vehicles in autonomous driving cars [67].

Before the Sobel operators can be applied on an image it has to be converted into grayscale as can be seen in figure 12. When we have the image in grayscale we can convolve the image with the horizontal Sobel kernel. The Sobel operator uses two 3x3 kernels which are convolved with the original image to calculate approximations of the derivatives, one for horizontal changes and one for vertical changes. If we define A as the source image and Gx and Gy are two images which at each point contain the horizontal and vertical derivative approximations respectively, the computations to distinguish the edges are as follows:

$$\mathbf{G}_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G}_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * \mathbf{A}$$

Where * denotes the two-dimensional signal processing convolution operation.

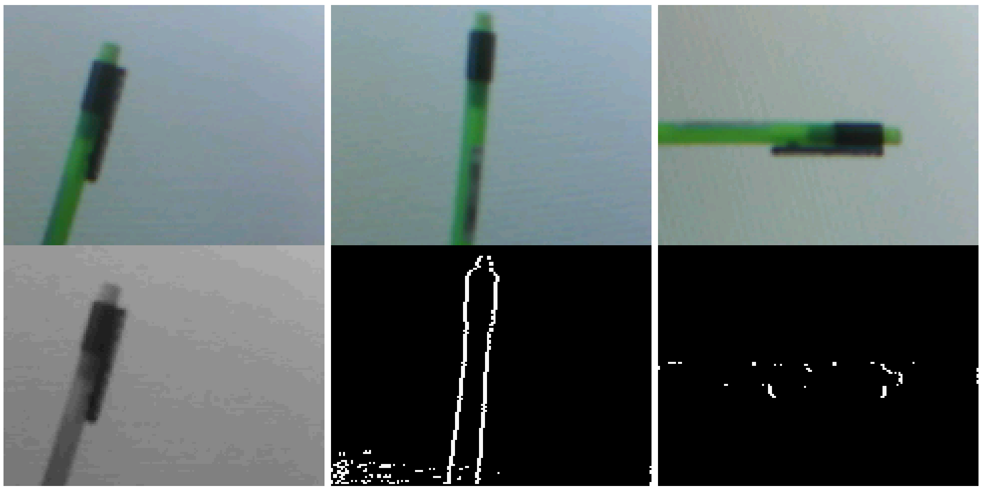


Figure 12 Picture showing RGB565 to grayscale conversion (left) and horizontal edge detection (middle and right)

6.2.2. Light-Weight Internet Protocol (LwIP)

In order to handle the communication over Ethernet the LwIP TCP/IP stack [68] was used. The main focus of the LwIP stack is to reduce memory usage and code size, making it suitable candidate for use in embedded systems with limited amount of resources such as memory. In order to reduce processing and memory demands, LwIp uses a tailor-made application programmable interface that does not require copying of data.

7. Discussion and Conclusions

In this chapter, we will first discuss how the project went, the applicability of OSS operating systems, what we could have done differently and how useful the developed application was. We will then present our conclusions based on the analysis that has been performed as a part of this thesis. Finally, possible future work is discussed.

7.1. Discussion

In this thesis project, we have identified a model that can be used for evaluating and comparing the quality and the maturity of an open source project with regards to the collaborative community behind the OSS. This model was then used together with identified characteristics for choosing which of the two open source operating systems looked at in this study was the most suitable candidate for use in safety-critical applications. This candidate was then assessed in order to see to what degree it could adhere to the requirements in the widely used functional safety standards IEC 61508 and ISO 26262.

We have also presented the outline of a methodology that could be used for certifying an open source operating system together with a proposed workflow for assessing OSS for safety-critical applications with regards to the functional safety standards IEC 61508 and ISO 26262. This is useful since it is desirable to have a repeatable certification process that can be used by certification bodies and others to certify and compare the suitability of multiple open source operating systems for use in a safety-critical context.

The safety-critical application that is specified in section 6.1 was implemented successfully with some minor modifications due to the fact that the image processing was too heavy for the target hardware platform. As a result of this we could not achieve the required real-time requirements. This could have been easily fixed by moving the image processing over to the computer running Linux or by changing the target hardware. However, the intention of the application was only to show how safety functions might be specified in a given context and that safety measures must also be implemented on the application level in order to mitigate failures.

In this study, we have only looked at two open source operating systems for small microcontrollers and assessed the one that we found to be the most suitable candidate for use in a safety-critical context. It would have been interesting to look for more suitable candidates but that was out of the scope of this thesis project.

7.2. Conclusion

The purpose of this thesis project was to identify characteristics that could be used to compare the suitability of an open source RTOS for use in safety-critical applications and propose an outline for a methodology that can be used by certification bodies and others. The results for these can be seen in chapter 4.

A case study was done as can be seen in chapter 5 where we evaluated two open source operating systems for small microcontrollers in order to find out which of the two were most suitable for use in safety-critical applications. The most suitable candidate was assessed according to the widely used functional safety standards IEC 61508 and ISO 26262 and our conclusions are presented below based on the analysis that was carried out in chapter 5.

For the IEC 61508 standard we have on the basis of the evidence presented in section 5.5.1 and the limited analysis carried out in this study concluded that ChibiOS would be acceptable for use in safety-critical application of SIL 1 and SIL 2. This statement must of course be qualified by stating that the hardware must be of suitable SIL. It may also be feasible to certify ChibiOS for use in SIL 3 applications by providing further evidence from testing and analysis. Certification in this way would also increase confidence in the use of ChibiOS in appropriate SIL 1 and SIL 2 applications.

Any safety justification for using ChibiOS in a given application shall include evidence that an analysis has been carried out in order to show that ChibiOS is suitable for the given application, and that appropriate techniques are applied on the application level to mitigate failures of the RTOS such as graceful degradation, fault detection and fault containment in order to avoid propagation of the failure.

ChibiOS holds many of the desirable characteristics that are required by an RTOS in safety-critical applications. However, as described in section 5.5.2 qualification is intended to support reuse of high quality software components where rewriting the software is not an option. ISO 26262-8 subclause 12 Qualification of software components are intended to be used for modules developed for other industries but with functionality that is also useful in the automotive industry. Qualified software components "must" then be developed according to an appropriate safety standard and open source development practices do not qualify to these requirements as of today. Therefore, the conclusion is drawn that ChibiOS won't fulfill the requirements of any ASIL in its current version since there is a lack of documentation in accordance with the ISO 26262:2011 standard.

7.3. Future Work

As a follow up project, effort could be made to perform a real assessment of ChibiOS with regards to IEC 61508 and see if it can fulfill certification for SIL 1 or SIL 2. This is because the assessment done in this thesis project is general and a real certification assessment would require trained professionals from an accredited certification body to actually perform an analysis of the available code and documentation. If ChibiOS cannot fulfill the requirements for certification supporting documentation would have to be produced. Therefore, the first certification project would be the costliest since the certification process would have to be defined and supporting materials would have to be produced.

It would also be interesting to look at other accepted standards which do not specifically have to safety related and see if OSS development could qualify for that given development process. This because subclause 12.4.1 in ISO26262-8 states that "evidence that the software development process for the component is based on an appropriate national or international standard.". And a note connected to this statement is that some "re-engineering activities" could be needed in order qualify already developed software components.

Generally speaking, one may wish to go deeper and see if the process of a given OSS project is close to an acceptable standard and then create the documentation that is missing and see how much "re-engineering" is required in order to gain compliance with the specific standard and then use this documentation and argue for ISO 26262 compliance according to subclause 12 in ISO 26262-8.

8. Bibliography

[1] G. Macher *et al*, “Automotive Embedded Software: Migration Challenges to Multi-Core Computing Platforms” in IEEE 13th International Conference on Industrial Informatics (INDIN), 2015, pp. 1386 - 1393.

[2] S. Tan and T. N B. Anh, “Real-time operating system (RTOS) for small (16-bit) microcontrollers” in IEEE 13th International Symposium on Consumer Electronics (ISCE), 2009, pp. 1007 - 1011.

[3] P. Hambarde, R. Varma and S. Jha,”The Survey of Real Time Operating System: RTOS” in IEEE International Conference on Electronic Systems, Signal Processing and Computing Technologies, 2014, pp. 34-39.

[4] A.K. Vishwakarma, K.V. Suresh and U.K Singh, “Porting and Systematic Testing of an Embedded RTOS” in IEEE International Conference on Computer and Communication Technologies (ICCCT), 2014.

[5] J. Corbet, How the Development Process Works (The Linux Foundation, San Francisco, 2011)

[6] A. Mockus, R.T. Fielding, J.D. Herbsleb, Two Case Studies of Open Source Software Development: Apache and Mozilla in ACM Transactions on Software Engineering and Methodology, 2002, pp. 306-346.

[7] D. Cotroneo *et al,*” Prediction of the Testing Effort for the Safety Certification of Open-Source Software: A Case Study on a Real-Time Operating System” in IEEE 12th European Dependable Computing Conference (EDCC), 2016

[8] H. Okamura and T. Dohi, “Towards Comprehensive Software Reliability Evaluation in Open Source Software” in IEEE 26th International Symposium on Software Reliability Engineering (ISSRE), 2015, pp. 121-129.

[9] A. Ismail and W. Jung, “Research Trends in Automotive Functional Safety” in IEEE International Conference on Quality, Reliability, Risk, Maintenance, and Safety Engineering (QR2MSE), 2013.

[10] IEC 61508, International Standard. Functional Safety of Electrical / Electronic / Programmable Electronic Safety-Related System, 2010.

[11] ISO 26262, International Standard. Road vehicles – Functional safety, 2011.

[12] L. Zhao and S. Elbaum, “Quality assurance under the open source development model” in The Journal of Systems and Software 66, pp. 65-75, 2003.

[13] J.V. Bukowski *et al*, “Software Functional Safety: Possibilities & limitations of IEC 61508-7 Annex D” in IEEE Annual Reliability and Maintainability Symposium (RAMS), 2016.

[14] Functional Safety: Essential to overall safety. [Online].
Available: http://www.iec.ch/about/brochures/pdf/technology/functional_safety.pdf
Accessed: Feb. 7, 2017.

[15] IEC 61508-3-1 (Draft), Functional Safety of Electrical / Electronic / Programmable Electronic Safety-Related Systems - Part 3-1: Software requirements - Reuse of pre-existing software elements to implement all or part of a safety function, 2015.

[16] IEC 61508 Edition 2.0. [Online].
Available: http://www.iec.ch/functionalsafety/standards/page2.htm
Accessed: Feb. 7, 2017.

[17] L. Silva Azavedo *et al,* “Assisted Assignment of Automotive Safety Requirements” in IEEE Software, 2013.

[18] A. M. St. Laurent and A. M. St Laurent, *Understanding open source and free software licensing*, 1st ed. United States: O’Reilly Media, Inc, USA, 2004.

[19] D. Watzenig and M. Horn, *Automated driving: Safer and more efficient future driving*., Switzerland: Springer International Publishing, 2017.

[20] B. Fitzgerald, “The Transformation of Open Source Software” in MISQ 30(3). Pp. 587-598, 2006.

[21] H Hedberg *et al,* “Assuring quality and usability in open source software development” in 1st International Workshop on Emerging Trends in FLOSS Research and Development, 2007.

[22] A. Mockus *et al,* “A case study of open source development: The Apache server” in Proceedings of the International Conference on Software Engineering (ICSE), pp. 263-272, 2000.

[23] D. A. Wheeler, "How to evaluate open source software / free software (OSS/FS) programs," 2011. [Online].
Available: https://www.dwheeler.com/oss_fs_eval.html
Accessed: Feb. 23, 2017.

[24] A. Avizienis *et al,* “Basic Concepts and Taxonomy of Dependable and Secure Computing” in IEEE Transactions on Dependable and Secure Computing, 2004.

[25] M. Riaz *et al,* “A Systematic Review of Software Maintainability Prediction and Metrics” in Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement, pp. 367 - 377, IEEE Computer Society, 2009.

[26] ISO / IEC 25010, International Standard. Systems and Software Engineering - Systems and Software Quality Requirements and Evaluation, 2011.

[27] H. Wu *et al,* “Maintenance Effort Estimation for Open Source Software: A Systematic Literature Review” in IEEE International Conference on Software Maintenance and Evolution, 2016.

[28] S. K. Khatri and I. Singh, “Evaluation of Open Source Software and Improving its Quality” in the 5th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO), 2016.

[29] ISO / IEC 9126, International Standard. Information Technology - Software engineering - Product quality, 2001.

[30] A. Adewurmi *et al,* “Evaluating Open Source Software Quality Models against ISO 25010” in IEEE International Conference on Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing, 2015.

[31] G. Polancic *et al,* “Comparative Assessment of Open Source Software Using Easy Accessible Data” in IEEE 26th International Conference on Information Technology Interfaces, 2004.

[32] D. Di Ruscio *et al*, “Supporting Custom Quality Models to Analyse and Compare Open-Source Software”, in the 10th International Conference on the Quality of Information and Communications Technology, 2016.

[33] F.W. Duijnhouwer, C. Widdows, "Capgemini Expert Letter Open Source Maturity Model", Capgemini, 2003, pp 1-18.

[34] M. Soto, and M. Ciolkowski, “The QualOSS open source assessment model measuring the performance of open source communities”. In Proceedings of the 3rd ESEM, pp. 498–501, 2009.

[35] R. Glott *et al*, “Quality models for Free/Libre Open Source Software - towards the “Silver Bullet”?” in the 36th EUROMICRO Conference on Software Engineering and Advanced Applications, 2010.

[36] D. Izquerdo-Cortazar *et al*, “Towards Automated Quality Models for Software Development Communities: the QualOSS and FLOSSMetrics Case” in 7th International Conference on the Quality of Information and Communications Technology, 2010.

[37] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating system concepts*. Hoboken, NJ: Wiley, 2013.

[38] L. Hatton, “Safer Language Subsets: An Overview and Case History, MISRA C” in Information and Software Technology, pp. 465 - 472, 2004.

[39] MISRA-C Guidelines for the Use of the C Language in Critical Systems Motor Industry Software Reliability Association, UK, 2004.

[40] T. Glib, *Software Metrics* 1977.

[41] L. Rosenberg, T. Hammer and J. Shaw, “Software Metrics and Reliability” in the 9th International Symposium on Software Reliability Engineering (ISSRE), 1998.

[42] Z. Bukhari *et al*, “Software Metric Selection Methods: A Review” in the 5th International Conference on Electrical engineering and Informatics, 2015.

[43] A. P. Mathur, *Foundations of Software Testing*, second edition. Pearson Education India, 2008.

[44] H. Zhu, P. A. V. Hall, and J. H. R. May, “Software unit test coverage and adequacy,” in ACM Computing Surveys, vol. 29, no. 4, 1997.

[45] H. Hemmaty, “How Effective Are Code Coverage Criteria”, in IEEE International Conference on Software Quality, Reliability and Security, 2015.

[46] C. Ebert and J. Cain, “Cyclomatic Complexity” in IEEE Software, vol. 33, 2016.

[47] ChibiOS [Online]
Available: www.chibios.org
Accessed: Apr. 6, 2017.

[48] ContikiOS [Online]
Available: www.contiki-os.org
Accessed: Apr. 18, 2017.

[49] A. Dunkels *et al*, “Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors” in the 29th Annual IEEE International Conference on Local Computer Networks, pp. 455-462, 2004.

[50] R. H. Pierce, “Preliminary Assessment of Linux for Safety Related Systems”, HSE contract research report RR011/2002. London: 2002.

[51] ChibiOS/RT Reference Manual
Available: http://chibios.sourceforge.net/docs3/rt/index.html
Accessed: Apr. 27, 2017.

[52] ChibiOS Book: The Ultimate Guide
Available: http://www.chibios.org/dokuwiki/doku.php?id=chibios:book:start
Accessed: Apr. 27, 2017.

[53] ChibiOS System states
Available: http://www.chibios.org/dokuwiki/doku.php?id=chibios:book:kernel#system_states
Accessed: Apr. 27, 2017.

[54] ChibiOS Binary semaphore
Available:
http://www.chibios.org/dokuwiki/doku.php?id=chibios:book:kernel_semaphores#binary_semaphores
Accessed: Apr. 27, 2017.

[55] ChibiOS General Architecture
Available: http://wiki.chibios.org/dokuwiki/doku.php?id=chibios:documents:architecture
Accessed: Apr. 27, 2017.

[56] ChibiOS Test suite
Available: http://chibios.sourceforge.net/html/testsuite.html
Accessed: Apr. 27, 2017.

[57] ChibiOS TestHAL
Available: https://github.com/ChibiOS/ChibiOS/tree/master/testhal
Accessed: Apr. 27, 2017.

[58] ChibiOS Community Overlay
Available: http://wiki.chibios.org/dokuwiki/doku.php?id=chibios:community:guides:community_overlay
Accessed: Apr. 27, 2017.

[59] ChibiOS Semaphores
Available: http://www.chibios.org/dokuwiki/doku.php?id=chibios:book:kernel_semaphores
Accessed: Apr. 27, 2017.

[60] ChibiOS Mutexes and Condition Variables
Available: http://www.chibios.org/dokuwiki/doku.php?id=chibios:book:kernel_mutexes
Accessed: Apr. 27, 2017.

[61] ChibiOS Integration guide
Available: http://www.chibios.org/dokuwiki/doku.php?id=chibios:articles:rt_integration_guide
Accessed: Apr. 27, 2017.

[62] ChibiOS Scheduler
Available: http://www.chibios.org/dokuwiki/doku.php?id=chibios:book:kernel_scheduler#the_scheduler
Accessed: Apr. 27, 2017.

[63] ChibiOS kernel configuration options
Avaialble: http://chibios.sourceforge.net/html/group__config.html
Accessed: Apr. 27, 2017

[64] ChibiOS HAL configuration options
Available: http://chibios.sourceforge.net/html/group___h_a_l___c_o_n_f.html
Accessed: Apr. 27, 2017

[65] B. Kotze *et al*, "Reconfigurable Navigation of an Automatic Guided Vehicle Utilising omnivision" in the 6th Robotics and Mechatronics Conference, 2013.

[66] J.H. Lee *et al*, "Indoor Navigation for an Automatic Guided Vehicle with Beacon Based Relative Distance Estimation", in the 8th International Conference on Ubiquitous and Future Networks (ICUFN), 2016.

[67] A. N. Shazwani *et al*, "Vehicle Detection Based on Underneath Vehicle Shadow Using Edge Features" in the 6th International Conference on Control System, Computing and Engineering, 2016.

[68] A. Dunkels, "Design and Implementation of the LwIP TCP/IP Stack", Swedish Institute of Computer Science, 2001.

Appendices

Appendix A – Semi-formal methods IEC 61508

Technique/Measure		SIL 1	SIL 2	SIL 3
1	Logic/function block diagarams	R	R	HR
2	Sequence diagrams	R	R	HR
3	Data flow diagrams	R	R	R
4	Finite state machines / state transition diagram	R	R	HR
5	Time Petri nets	R	R	HR
6	Entity-relationship-attribute data models	R	R	R
7	Message sequence charts	R	R	R
8	Decision/turth tables	R	R	HR
9	UML	R	R	R

NR	-	Not Recommended
R	-	Recommended
HR	-	Highly Recommended

Appendix B – Software architectural design IEC 61508

Technique/Measure		SIL 1	SIL 2	SIL 3
Architecture and design feature				
1	Fault detection	-	R	HR
2	Error detecting codes	R	R	R
3a	Failure assertion programming	R	R	R
3b	Diverse monitor techniques (with independence between the monitor and the monitored function in the same computer)	-	R	R
3c	Diverse monitor techniques (with separation between the monitor computer and the monitored computer)	-	R	R
3d	Diverse redundancy, implementing the same software safety requirements specification	-	-	-
3e	Functionality diverse redundancy, implementing different software safety requirements specification	-	-	R
3f	Backward recovery	R	R	-
3g	Stateless software design ()	-	-	R
4a	Re-try fault recovery mechanisms	R	R	-
4b	Graceful degradation	R	R	HR
5	Artificial intellegence - fault correction	-	NR	NR
6	Dynamic reconfiguration	-	NR	NR
7	Modular approach	HR	HR	HR
8	Use of trusted/verified software elements (if available)	R	HR	HR
9	Forward traceability between the software safety requirements specification and software architecture	R	R	HR
10	Backward traceability between the software safety requirement and software architecture	R	R	HR
11a	Structured diagrammatic methods	HR	HR	HR
11b	Semi-formal methods	R	R	HR
11c	Formal design and refinement methods	-	R	R
11d	Automatic software generation	R	R	R
12	Computer-aided specification and design tools	R	R	HR
13a	Cyclic behaviour, with guaranteed maximum cycle time	R	HR	HR
13b	Time-triggered architecture	R	HR	HR
13c	Event-driven, with guaranteed maximum response time	R	HR	HR
14	Static resource allocation	-	R	HR
15	Static synchronization of access to shared resources	-	-	R

NR	-	Not Recommended
R	-	Recommended
HR	-	Highly Recommended

Appendix C – Software testing and integration IEC 61508

Technique/Measure		SIL 1	SIL 2	SIL 3
1	Probabilistic testing	-	R	R
2	Dynamic analysis and testing	R	HR	HR
3	Data recording and analysis	HR	HR	HR
4	Functional and black box testing	HR	HR	HR
5	Performance testing	R	R	HR
6	Model based testing	R	R	HR
7	Interface testing	R	R	HR
8	Test management and automation tools	R	HR	HR
9	Forward traceability between the software design specification and the module and integration test specification	R	R	HR
10	Formal verification	-	-	R

NR	-	Not Recommended
R	-	Recommended
HR	-	Highly Recommended