Evaluation of Open Source Operating Systems for Safety-Critical Applications

Petter Sainio Berntsson[1], Lars Strandén[2] and Fredrik Warg[2]

[1] Chalmers University of Technology, Göteborg, Sweden
petter.berntsson@gmail.com
[2] RISE Research Institutes of Sweden, Borås, Sweden
{lars.stranden, fredrik.warg}@ri.se

Published in:

Proceeding of 9th International Workshop on Software Engineering for Resilient Systems, SERENE 2017, Geneva, Switzerland, September 4-5, 2017, pp 117-132, LNCS vol. 10479, Springer International Publishing

The final publication is available at Springer via http://dx.doi.org/10.1007/978-3-319-65948-0_8

Evaluation of Open Source Operating Systems for Safety-Critical Applications

Petter Sainio Berntsson[1], Lars Strandén[2] and Fredrik Warg[2]

[1] Chalmers University of Technology, Göteborg, Sweden
petter.berntsson@gmail.com
[2] RISE Research Institutes of Sweden, Borås, Sweden
{lars.stranden, fredrik.warg}@ri.se

Abstract. There are many different open source real-time operating systems (RTOS) available, and the use of open source software (OSS) for safety-critical applications is considered highly interesting by industrial domains such as medical, aerospace and automotive, as it potentially enables lower costs and more flexibility. In order to use OSS in a safety-critical context, however, evidence that the software fulfills the requirements put forth in a functional safety standard for the relevant domain is necessary. However, the standards for functional safety typically do not provide a clear method for how one would go about certifying systems containing OSS. Therefore, in this paper we identify some important RTOS characteristics and outline a methodology which can be used to assess the suitability of an open source RTOS for use in a safety-critical application. A case study is also carried out, comparing two open source operating systems using the identified characteristics. 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 standard IEC 61508.

Keywords: Functional safety, IEC 61508, open source software, real-time operating systems, software quality.

1 Introduction

The last few years have seen a remarkable rise in the use of embedded systems, for instance in the automotive industry where modern cars are typically equipped with dozens of embedded electronic systems. 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 [1]. The use of a Real-Time Operating System (RTOS) is common in embedded systems due to the multitasking requirement in many applications [2]. During the last two decades RTOSs have undergone continuous evolution and there are many commercially available RTOSs.

For various reasons, including cost of commercial alternatives or lack of desired features in existing products, people have developed their own versions of such software and made it publicly available as Open Source Software (OSS). A common use

of OSS is in operating systems since they are normally application independent and can therefore attract a large user base and be ported to different hardware platforms. The use of OSS for safety-critical applications is considered highly interesting by industrial domains such as medical, aerospace and automotive, as it potentially enables lower costs and more flexibility. Open source projects that have well-established communities (e.g. Linux and Apache) usually employ stringent development processes [3], [4] and deliver high quality software. However, they do not fulfill the requirements of current safety standards such as IEC 61508 [5] and ISO 26262 [6]. These standards impose strict demands on e.g. project management, developer qualification, risk management, requirements management, quality assurance and documentation. This becomes a problem since many OSS projects do not follow a strict development process [7], which makes the requirements impossible to achieve after the software has already been developed. However, software that has been developed with a non-compliant process can in some cases still be qualified if the software fulfills the requirements for reuse. This can be done by providing enough evidence to support its suitability for safety-critical applications.

In this paper we investigate criteria such as software metrics, support and maintainability issues, real-time and dependability properties that can be used to evaluate, mainly from a software perspective, an open source RTOS with regards to its use in safety-critical applications. We then propose a methodology for such evaluation; the methodology is based on using the Capgemini Open Source Maturity Model [14] together with a set of characteristics influence by earlier work in [12], but adapted to better suit an RTOS. The aim of our proposed methodology is to collect information to help determine if an RTOS is a potential candidate for use in a safety-critical application, and also to be able to choose which candidate is the most promising when several candidates exist. For use in an application that is to be certified against a functional safety standard, however, an assessment of the RTOS against the relevant requirements in the standard must also be conducted. We show a case study where the two OSS RTOSs ChibiOS [16] and ContikiOS [17] are compared using the methodology, and where ChibiOS is subsequently evaluated against the requirements for use of pre-existing software elements using non-compliant development according to the functional safety standard IEC 61508.

2 Related Work

There are multiple different open source quality and maturity models available and a comparative study [8] has been done. It showed two models that satisfied all eight factors under product quality in the ISO/IEC 25010 standard [9]. However, most of the models compared in the study seem to be abandoned, and the tools used for retrieval and analysis of metrics are no longer available. One of these is the QualOSS [10] model, which was one of the two models satisfying all eight factors under product quality in ISO/IEC 25010. High test coverage is one of the most impacting activities in order to qualify software for safety certification. The study in [11] investigates the relationship between software complexity and the effort to achieve high test cov-

erage with the objective of figuring out to what extent it is possible to predict the effort needed for certification. By looking at software complexity metrics this would enable a preliminary screening and benchmarking of OSS. A previous study [12] has been made regarding the use of Linux in safety-related systems and it has been helpful for identifying important characteristics for comparing the suitability of open source RTOS for safety-critical applications.

3 Evaluation of OSS RTOS

In this chapter, we discuss how existing software quality models can be leveraged to evaluate the maturity and quality of OSS. The rationale behind using such models is that they take into account organizational aspects that will give an indication of whether the maturity of the project is acceptable, that the software it suitable for the intended application, and that there is potential for fulfilling the process requirements of safety standards. We also look at previous work for identifying characteristics and metrics of the software which will give an indication of how well the product requirements of the standards can be fulfilled.

3.1 Software Quality and ISO/IEC 25010

Quality is the level of which a product meets the mentioned requirements or fulfils customer needs and expectations. OSS developers and organizations are facing many challenges and questions regarding the quality when compared to proprietary software, as there are worries about the level of satisfaction that can be achieved with respect to robustness, support, maintenance and other quality attributes when the software is written by volunteer developers. 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 [9] standard defines 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:

1. Functional suitability: How well the product meets stated and implied needs.
2. Performance efficiency: The performance relative to the amount of resources used.
3. Compatibility: How well the product can exchange information with other products, and if it can share hardware or software environment with other products.
4. Usability: If the product is useful in a specified context.
5. Reliability: How reliably the product performs its specified functions.
6. Security: The degree to which a product protects information and data.
7. Maintainability: Effectiveness and efficiency of product modifications.
8. Portability: Effectiveness and efficiency with which a product can be transferred from one usage environment to another.

The product can either be a complete system or a component. The characteristics are described in more detail in the standard. In addition to the product quality model there is a quality in use model which we do not further discuss or make use of.

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, specifically how to judge the impact of community, collaboration, licensing, and support aspects. Several quality models have been designed specifically for assessing the quality of OSS, but most of them predate ISO/IEC 25010, and are therefore based on its now obsolete precursor ISO/IEC 9126 [13]. A comparative study [8] has been made between different OSS quality models and the newer ISO/IEC 25010 model. The study showed that the Capgemini Open Source Maturity Model (OSMM) [14] was the most comprehensive model satisfying all eight factors under Product Quality. Therefore, we use the Capgemini OSMM in our proposed evaluation methodology.

3.2 Capgemini Open Source Maturity Model

Capgemini OSMM is a model where software is graded against a set of product indicators and application indicators. The product indicators are described as objective and measurable facts that focus on the product and the model includes scoring criteria for each indicator. The application indicators are used to assess how well the product fits a specific context. That is, they take into account environmental aspects and the present and (expected) future demands of the users. Therefore these indicators cannot be assessed without an intended context, and they are evaluated in two dimensions: the score (S) - how well the characteristic is fulfilled, and the priority (P) – how important the indicator is relative to other indicators. The final weight is the product of these (P*S). Both indicator scores and priority are judged on a scale 1-5 where 5 are the highest.

The model points out the significance of information, both of the product itself and the community that surrounds it, and thus it incorporates various criteria such as product development, developer and user community, product stability, maintenance and training. It was designed as a tool that can 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.

A brief overview of the defined application indicators:

- Usability: Taking into account the intended users.
- Interfacing: Required connectivity, applicable standards.
- Performance: The performance demands that must be met.
- Reliability: Required level of service.
- Security: Required security measures.
- Proven technology: Is the technology proven in daily production?
- Vendor independence: Level of commitment between supplier and user.
- Platform independence: Is the product available for a wide range of platforms?
- Support: What level of support is required?
- Reporting: What kind of reporting is required?
- Administration: Use of existing maintenance tools, demands for management?
- Advice: Is validation by independent parties required?
- Training: Required training.
- Staffing: How is product expertise acquired?
- Implementation: Preferred implementation scenario.

For a more described description of the indicators see [14].

3.3 Dependability-Critical Aspects of an RTOS

Relevant parts of the international standards for functional safety IEC 61508 and ISO 26262 have been analyzed, and a literature survey made (see [18] for more details), in order to find characteristics that can be used to assess and compare the suitability of an open source RTOS for use in safety-critical applications. In particular, a study [12] has been made regarding the use of Linux in safety-critical applications and three basic 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.

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 the set time constraint [1] often referred to as a deadline. However, in most RTOSs (soft real-time) one is not guaranteed that the system will always meet its deadlines, just generally. Only hard real-time systems can deterministically meet its deadlines.

An RTOS needs to satisfy a number of requirements. These are usually similar to high-reliability systems requirements. The operating system features listed below were identified in [12] as necessary to evaluate if the system is to be used in safety-critical applications:

1. Executive and scheduling: The process switching time and the employed scheduling algorithm 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 task synchronization 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 allow 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: Risk of safety implications of security issues in connected systems.
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.

These features have been helpful for identifying the characteristics that are used for comparison in this paper and these are described in section 4.1, but we have modified some characteristics and omitted some in order to better suit an embedded RTOS.

We will also look at the implementation and test coverage of the OSS. 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 organiza-

tion of the modules and language subsets. For safe and reliable software written in C, a commonly used subset is the MISRA C [15] 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.

As mentioned we will also take test coverage into account since high test coverage is one of the most impacting activities in order to qualify software for safety certification [11]. 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, running the automated test after every single commit to the code base may facilitate the detection of bugs ensuring the intended functionality of the software. Test coverage is one of the measures often required by standards to help make sure the test suite is adequate, and it is commonly defined as a percentage of the software that is covered by the tests. There are different ways in which this percentage can be defined; some common metrics are the percentage of code lines or statements executed, or the percentage of branch paths (edges) tested. These are often referred to as code (or line) coverage, statement coverage, and branch coverage respectively. For complex software components such as an RTOS the state space is potentially huge and it may be difficult to achieve 100% coverage in a practical amount of time (depending on which metric is used or prescribed in the standards). When this is the case, the test cases shall be chosen in a manner such that they test the different aspects of the RTOS, and an argument made for justifying why certain input combinations don't require testing (based on e.g. equivalence classes or other forms of analysis).

4 Selecting and Qualifying an OSS RTOS for Use in Safety-Critical Applications

In this chapter we present an approach on how to select and qualify an OSS RTOS for use in safety-critical applications.

4.1 Characteristics for Comparison

Based on the discussion in previous sections and common requirements in functional safety standards we have put together a list of characteristics that are relevant for an open source RTOS used in a safety-critical context. Although some characteristics in the list partly overlap with Capgemini OSMM indicators, this list is more focused on characteristics that will be relevant when trying to assess the RTOS suitability in safety-critical applications and map it to the requirements in functional safety standards, whereas the main purpose of OSMM is to determine whether the software is mature enough to consider in a wider perspective. It should be noted that this is a first attempt at identifying such criteria, and thus the list of characteristics should not be seen as exhaustive. In addition, we currently rate the characteristics only as 'available' or 'not

available', while in reality some of them might be somewhere in between non-existent and sufficient for the intended purpose (for instance 'documentation' and 'test suite'). This may be refined in an improved version of these characteristics, e.g. using a scoring range with specified criteria for each level, or possibly more detailed sub-characteristics. The current characteristics we use are:

- Coding rules: Is the RTOS implemented using 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 documentation available?
  - What functionality does it cover?
- Static resource allocation: Does the RTOS 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.
- Real-time support: Does the RTOS support deadlines?
- Domain separation: Is there support for domain separation?
- 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 options: Is there an option to turn off undesired functionality in the RTOS so that the unused functionality won't be compiled at all?
- 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 projects code base?
  - Bug tracking: Is there a list of known bugs?
  - Bug fixing: Are bugs being fixed at regular intervals?

If the open source RTOS fulfills all of the above characteristics it is a good candidate since it holds some of the functionality that is desirable for a safety-critical RTOS and some attributes that are preferred from an open source perspective. If it does not fulfill all of the above characteristics it could still be possible to adapt it, but the effort to achieve compliance with a functional safety standard is likely considerably higher.

4.2 Workflow

Based on the material from the previous sections, the process for choosing and assessing a suitable open source RTOS candidate against a functional safety standard is described below and a workflow can be seen in **Fig. 1**. The first step is identifying promising open source RTOS candidates. When enough candidates have been identi-

fied the quality and maturity of the OSS project is evaluated. As mentioned, we are using the Capgemini OSMM 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 a certain number of points based on the guidelines given in [14]. When the quality and maturity assessment has been made, a comparison of the characteristics presented in Section 4.1 is made. The best candidate is chosen based on the results of these two assessments. In the end, determining if there is a suitable candidate and which one to choose is not a mathematical exercise, the results must be judged within the context of what one is trying to achieve and the main point of the exercise is to gather the relevant information to make such a judgment.

When a candidate has been identified, one has to study the documentation and other information available for the open source RTOS and compare it with the applicable requirements in a functional safety standard. In our case study we have looked at the requirements for reuse of software components with non-compliant development (i.e. development has not been performed according to the standard) that are given in subclause 7.4.2.13 of IEC 61508-3. If the documentation or other material is insufficient or unavailable, the missing pieces may be supplemented by the developer of the safety-critical application, but in the case study we have only evaluated the RTOSs based on existing artefacts. The methodology is not only applicable for IEC 61508 however; other functional safety standards could also be used.

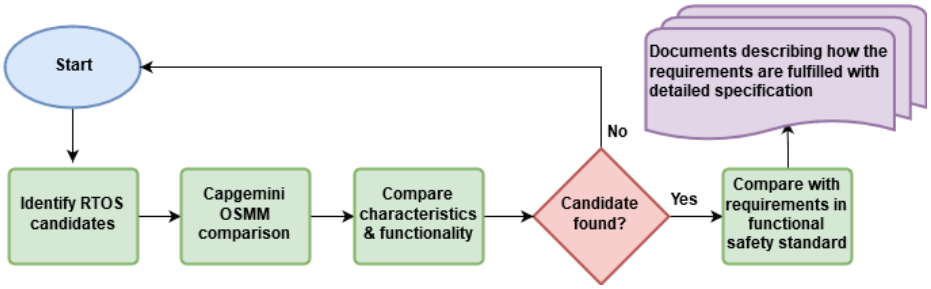

Fig. 1. Proposed workflow for assessing an RTOS for safety-critical applications

5 Case Study: Comparing ChibiOS and ContikiOS

In this chapter, the two open source operating system ChibiOS [16] and ContikiOS [17] are compared. ChibiOS is an OSS RTOS implemented compliant with MISRA C and ContikiOS is a lightweight OSS operating system that is designed to run on Internet of Things (IoT) devices with limited resources. These operating systems are compared in order to find the most suitable candidate for use in a supposed safety-critical function. The most suitable candidate is then assessed to see to what degree it can adhere to the requirements of non-compliant development put forth in Part 3: Software Requirements of the functional safety standard IEC 61508. In the case study we assume the function shall be certified according to IEC 61508 with safety integrity

level (SIL) 2. In a real case, the required SIL is determined using a hazard analysis and risk assessment for the function being implemented. SIL is a measure of the level of risk reduction provided by a safety function, and IEC 61508 specifies four levels 1-4, where higher levels require more risk reduction and hence imply more stringent demands on the electrical/electronic system (including the software). In the case study we have chosen to evaluate against SIL 2 as the higher levels have requirements that will be much more difficult to achieve for a component developed with a non-compliant development process.

Note that the evaluation has been made by one person for demonstration purposes, and not by a team of experts as recommended in Capgemini OSSM, nor by a qualified assessor as required for an actual assessment against IEC 61508. Therefore, the case study should be seen as a working example of evaluating an OSS RTOS, and not as reliable results for the two RTOSs used in the study.

5.1 Project Maturity

The quality and maturity of the two open source operating systems are evaluated with the Capgemini OSMM described in section 3.2 and with the help of the guidelines given in [14]. First the product indicators form the basis of the model. Using these indicators, the quality 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**. However, this data is only a comparison of the products strengths and weaknesses. To properly assess the product, we must also take into account the application indicators and a comparison of these can be seen in **Table 2**. From the data presented in these tables, we can see that the most suitable candidate is ChibiOS. However, it shall be noted that this comparison does not guarantee a certain quality of the product, it is just an indication of which product has the highest quality of development.

5.2 RTOS Characteristics

In this section, we will give a brief description of the two operating systems in this study and compare them based on the characteristics that are described in section 3.3 and the result of this comparison can be seen in **Table 3**. The results have been obtained by going through the documentation and test suites, running PC-Lint, and researching how community activities are performed.

Table 1. Comparison of product indicators according to Capgemini OSMM.

Indicator	ChibiOS	ContikiOS
Product		
Age	5	3
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

Table 2. Comparison of application indicators according to Capgemini OSMM.

		ChibiOS		ContikiOS	
Indicator	**Priority (P)**	**Score (S)**	**P*S**	**Score (S)**	**P*S**
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
Platform independence	2	4	8	4	8
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			175		151

Table 3. Comparison of characteristics. An "X" indicates availability and "-" non-availability.

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	-
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

ChibiOS. This is an open source RTOS implemented in MISRA C. A code analysis has been performed using the PC-Lint static 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 and design patterns are explicit and consistently used on all the software.

The scheduler of ChibiOS is implemented as a priority-based preemptive scheduler with round-robin scheduling for tasks at the same priority level and it is suited for real time systems since it supports deadlines. A test suite is also 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 RTOS. The test suite can be executed both on a simulator and on real hardware. By running the test suite on hardware, one can benchmark the given hardware platform where the time overhead of operations like context-switch, interrupts and synchronization primitives can be obtained.

Separation of different tasks can be done in the time domain by utilizing the implemented scheduling policy. However, care must be taken while using critical regions, since they can introduce unexpected latency. To avoid this, the 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. Some of the supported architectures can provide memory separation by using a Memory Protection Unit (MPU) or Memory Management Unit (MMU), while memory separation is not available in other architectures. For the case study we have rated domain separation as available, but keep in mind that this depends on the hardware to be used.

ContikiOS. This is a lightweight open source operating system that is designed to run on IoT devices with limited resources. It provides three network mechanisms: the uIP

TCP/IP stack which provides IPv4 networking, the uIPv6 stack which provides IPv6 networking, and the Rime stack [19] which is a set of custom lightweight networking protocols designed for low-power wireless sensor networks.

ContikiOS is mainly developed in standard C (ISO C) and is portable to various platforms. The coding rules only cover the formatting style. 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 dev/, cpu/, platform/ and core/ directories of the ContikiOS 3.0 code base have been checked. In total over 70 000 messages were generated, 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, buffer overrun, and unused return codes.

The scheduler in ContikiOS does not support priorities for tasks. However, they can be either cooperative or preemptive and a number of timer modules exist and these can be used to schedule preemptive tasks. There are also no well-defined mechanisms for inter-task communication and concurrency issues must be handled manually. 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. Also, memory separation is not used due to lack of support in most of the supported architectures.

5.3 Compliance with IEC 61508

As we can see by the data presented in section 5.2, the most suitable option for assessment is ChibiOS. Therefore, we have assessed ChibiOS according to the requirements for reuse of software components with non-compliant development in IEC 61508. The RTOS is in this case regarded as a context free software component and in order to assess ChibiOS we have gone through the available documentation and compared it with the requirements given under sub-clause 7.4.2.13 in IEC 61508-3. However, a detailed description of the requirements is not given here. For a detailed description of the requirements, see [5]. Due to space restrictions, this is an abbreviated version of our assessment; more details can be found in [18]. Note that we make no claim that this assessment is exhaustive.

Under sub-clause 7.4.2.13 in IEC 61508-3 there is a requirement that a software safety requirements specification shall exist for the software element. This specification shall contain 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. This specification must be valid in the specific system context and it shall also cover the functional and safety behavior of the software element in its new application. However, a pre-existing RTOS is unlikely to have any specific safety requirements defined since it is not bound to a specific context, therefore, in this study we assume that the requirement of this specification can be fulfilled if the behavior of the RTOS is precisely defined supplemented with its use in the target application. This requirement can then be fulfilled by ChibiOS since there is a

detailed reference manual covering all the functionality of the kernel together with a supporting book that describes the architecture of the RTOS and how all the submodules work.

According to IEC 61508, the use of semi-formal methods (e.g. finite state machines) to express parts of a specification so that some types of mistakes such as wrong behavior can be detected is recommended for safety functions considered to be of SIL 1 or SIL 2 and highly recommended for SIL 3. These methods are used to model, verify, specify or implement the control structure of a system, and the IEC 61508 standards states that state transition diagrams can apply to the whole system or to some objects within it. The documentation available for ChibiOS shows that the behavior of the kernel and other submodules are specified in detail with the help of UML and finite state machines.

The IEC 61508 standard provides recommended practices for verification and how the architectural design of the software shall be structured. Some of these techniques are not applicable to an operating system such as graceful degradation which is something that needs to be implemented on the application level. However, high integrity software shall be designed with a modular approach that is verifiable and testable with measurements indicating the test coverage. It is also recommended to use static resource allocation, time-triggered architecture with cyclic behavior etc. These requirements can be fulfilled by ChibiOS since it has a modular design; it is internally divided in several major independent components. It uses static resource allocation and it has a time-triggered architecture with cyclic behavior. The test suite provided is used in order to verify the proper working of the kernel and it can be used to test if a ported version of the RTOS is working and all the test results are included as reports in the RTOS distribution.

Another requirement is that when software elements are present which are not required for achieving the functional safety, evidence shall be provided that this functionality will not prevent the system from meeting its safety requirements. In ChibiOS 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 and system tick frequency. This will make it easier to prove that unwanted functionality will not interfere.

There shall also be evidence that all credible failure mechanisms of the software element have been identified and that mitigation measures exist. ChibiOS provides support for domain separation of different tasks in the time domain, which is done by using priority-based preemptive scheduling, but care must be taken while using critical regions, since they can introduce unexpected latency. The only execution model available for ChibiOS is the single process-multi thread execution model, which means that all the tasks share the same addressing space unless an MMU is used. But memory separation is not implemented on all architectures. Mitigation measures shall also be used at the application level, if considered necessary in the context of use. This requirement may therefore require significant work to fulfill.

There are requirements that coding rules are followed and each module is reviewed. Also, a suitable strongly typed language and language subset is required. In

the core ChibiOS codebase, the code is thoroughly tested and maintained, bugs are tracked and fixed, and the code is released in stable packages regularly. 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. ChibiOS also implements the C subset MISRA C 2012 which is sometimes considered a strongly typed subset of C. Although C was not specifically designed for this type of application, it is widely used for embedded and safety-critical software for several reasons. Some advantages are control over memory management are simple and 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 e.g. 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.

6 Conclusions and Future Work

ChibiOS holds many of the desirable characteristics that are required by an RTOS in safety-critical applications and for the IEC 61508 standard. On the basis of the limited evidence and analysis presented in this study, we have concluded that ChibiOS may be acceptable for use in safety-critical application of SIL 1 and SIL 2. Of course, this statement must be qualified by stating that the hardware must be of suitable SIL and the fact that we may have stretched the definitions of the standard somewhat since sub-clause 7.4.2.13 of IEC 61508-3, reuse of software components with non-compliant development, is not really intended for assessing an operating system.

The assessment done in this thesis project is by no means complete and a real assessment would require trained professionals from an accredited certification body to perform the analysis of the available code and documentation. A follow up project could therefore be to perform a real assessment of ChibiOS with regards to IEC 61508 to determine whether the requirements for SIL 1 or SIL 2 can be fulfilled. Other relevant standards such as ISO 26262 could also be considered.

It should also be noted that the list of characteristics and evaluation methodology is our first attempt for evaluating the use of an RTOS in safety-critical applications. Future work is needed to refine the characteristics and methodology to match the requirements in functional safety standards even better.

Acknowledgements. This work is from of a Master's thesis project at RISE Electronics; and is partly funded by the Swedish government agency for innovation systems (VINNOVA) in the NGEA step 2 project (ref 2015-04881).

References

1. Hambarde, P., Varma, R. Jha, S.: The Survey of Real Time Operating System: RTOS. In: IEEE International Conference on Computer and Communication Technologies (ICCCT), pp. 34-39, (2014).
2. Tan, S., Nguyen Bao Anh, T.: Real-Time operating system (RTOS) for small (16-bit) microcontrollers. In: IEEE 13th International Symposium on Consumer Electronics (ISCE), pp. 1007-1011, (2009).
3. Corber, J.: How the Development Process Works (The Linux Foundation) (2011).
4. Mockus, A., Fielding, R.T., Herbsleb, J.D.: Two case studies of open source software development: Apache and Mozilla. In: ACM Transactions on Software Engineering and Methodology (TOSEM), pp. 309-346, (2002).
5. IEC 61508, International standard. Functional Safety of Electrical / Electronic / Programmable Electronic Safety-Related System, (2010).
6. ISO 26262, International Standard. Road vehicles – Functional Safety, (2011).
7. Zhao, L., Elbaum, S.: Quality assurance under the open source development model. In: The Journal of Systems and Software 66, pp. 65-75, (2003).
8. Adewumi, A., Misra, S., Omoregbe, N.: Evaluating Open Source Software Quality Models Against ISO 25010. In: IEEE International Conference on Computer and In-formation Technology; Ubiquitous Computing and Communications; Dependable, Automatic and Secure Computing, Pervasive Intelligence and Computing, pp. 872-877, (2015).
9. ISO/IEC 25010, International Standard. Systems and Software Engineering – Systems and Software Quality Requirements and Evaluation, (2011).
10. Soto, M., Ciolkowski, M.: The QualOSS open source assessment model measuring the performance of open source communities. In: Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement (ESEM), pp.498-501, (2009).
11. Cotroneo, D., Di Leo, D., Natella, R.: Prediction of the Testing Effort for the Safety Certification of Open-Source Software: A Case Study on a Real-Time Operating Sys-tem. In: IEEE 12th European Dependable Computing Conference (EDCC), pp. 141-152, (2016).
12. Pierce, R.H.: Preliminary Assessment of Linux for Safety Related Systems. In: HSE Contract research report RR011/2002, (2002).
13. ISO / IEC 9126, International Standard. Information Technology – Software Engineering – Product Quality, (2001).
14. Dujinhouwer, F.W., Widdows, C.: Capgemini Expert Letter Open Source Maturity Model, Capgemini, pp. 1-18, (2003).
15. Motor Industry Software Reliability Association, MISRA-C Guidelines for the Use of the C Language in Critical Systems, UK, (2004).
16. ChibiOS, https://www.chibios.org, last accessed 2017/05/29.
17. ContikiOS, https://www.contiki-os.org, last accessed 2017/05/29.
18. Berntsson, P. S.: Evaluation of open source operating systems for safety-critical applications, Master's thesis, Chalmers University of Technology, (2017).
19. Dunkels A., Österlind F., He Z.: An adaptive communication architecture for wireless sensor networks. In: Proceedings of the Fifth ACM Conference on Networked Embedded Sensor Systems (SenSys 2007), Sydney, Australia, November 2007.