FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Hypervisor Based Composable Systems for the Automotive Industry

Making a Secure Platform Safe

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Master of Science

im Rahmen des Studiums

Technische Informatik

eingereicht von

Andreas Platschek

Matrikelnummer 0425291

an der

Fakultät für Informatik der Technischen Universität Wien

Betreuung: o.Univ.Prof. Dipl.-Ing. Dr. Dietmar Dietrich
Mitwirkung: Dipl.-Ing. (FH) Dr.techn. Heimo Zeilinger
Prof. Nicholas Mc Guire

Wien, 15.07.2013

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Hypervisor Based Composable Systems for the Automotive Industry

Making a Secure Platform Safe

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Engineering

by

Andreas Platschek

Registration Number 0425291

to the Faculty of Informatics

at the Vienna University of Technology

Advisor: o.Univ.Prof. Dipl.-Ing. Dr. Dietmar Dietrich
Assistance: Dipl.-Ing. (FH) Dr.techn. Heimo Zeilinger
Prof. Nicholas Mc Guire

Vienna, 15.07.2013

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Andreas Platschek
Theresiengasse 25-27, A-1180 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Acknowledgements

Thanks to all who gave me inspiration and guidance to write this thesis (knowingly or otherwise). Of course this includes everyone at OpenTech, OSADL and the whole FLOSS community - thanks for all I was able to learn from you and the guidance I got from you guys.

Special thanks to Prof. Nicholas Mc Guire, for all your guidance, patience and insight, as well as for kicking my butt at appropriate times.

Last but not least: thanks to my family and friends!

Abstract

Following the trend already set by the avionics industry, the automotive industry is reconsidering its current approach towards on-board electronics as well and starts to integrate multiple ECUs (Error Containment Units) into one hardware node. Following this approach it is vital to ensure the independence of residing applications, which often require different levels of safety and security. Independence is achieved by partitioning, that means temporal and spatial isolation, supplemented by communication mechanisms that must not violate the isolation properties. This approach allows the construction of composable systems that simplify the reuse of (legacy) software modules based on temporal and spatial isolation. Furthermore the preservation of dependability, safety and security properties of the individual modules is ensured, enabling modular validation and certification.

This thesis approaches the safety aspects of utilizing free/libre open source software (FLOSS) components, taking the constraints of the automotive industry into account. The approach taken is to use the XtratuM2 hypervisor, and to execute multiple instances of a FLOSS implementation of an OSEK (Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug) compliant operating system – each instance running one automotive application – on top of it. The resulting implementation is able to run several independent automotive applications in parallel on the same processor, rather than requiring a single hardware node for each of them. This approach reduces the number of ECUs in the car, leading to a decrease of power consumption and weight, allowing a higher utilization of the hardware nodes and simplifying inter-node communication.

Kurzfassung

Genau wie die Luftfahrt Industrie ist derzeit auch die Automobil Industrie im Umbruch, und beginnt mehrere ECUs (Error Containment Units) in einen Hardware Knoten zu integrieren. Bei diesem Ansatz ist es jedoch äußerst wichtig, dass die Unabhängigkeit der Applikationen – die meist auch unterschiedliche Sicherheitsanforderungen (sowohl Safety als auch Security) haben – erhalten bleibt. Die Unabhängigkeit wird durch Isolation, das heißt zeitliche sowie räumliche Trennung erreicht und wird durch geeignete Kommunikationsmechanismen, die keines dieser Isolationskriterien stören dürfen, ergänzt. Der Ansatz erlaubt, durch die Isolation die Safety- und Security-Eigenschaften zu erhalten und ermöglicht so die modulare Validierung sowie die modulare Zertifizierung von Software.

Die Arbeit versucht, die Safety-Kriterien, die von der Automobil Industrie gefordert werden, mit Hilfe von FLOSS (free/libre open source software) zu erfüllen. Die gewählte Vorgehensweise ist, basierend auf dem XtratuM2 Hypervisor mehrere unabhängige Instanzen einer FLOSS-Implementierung der OSEK-Spezifikation (Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug) laufen zu lassen, wobei jede dieser Instanzen eine typische Applikation aus dem Automobil Bereich ausführt. Die resultierende Plattform erlaubt es, mehrere unabhängige Applikationen parallel laufen zu lassen, anstatt einen Hardware Knoten für jede der Applikationen zu verwenden. Dieser Ansatz reduziert die Anzahl der Knoten im Fahrzeug und führt so zu einer Reduktion des Gewichtes sowie des Stromverbrauchs. Weiters wird die Ausnutzung der Ressourcen, die moderne CPUs zur Verfügung stellen, verbessert und die Kommunikation zwischen Applikationen vereinfacht.

Contents

CHAPTER 1

Introduction

Modern cars already integrate applications of the same safety integrity level into one hardware node. The following chapter explains why it will be necessary to integrate applications of different integrity level into one hardware node in the future, and gives an outline of the goals and chapters of this thesis.

1.1 Motivation

Today, the automotive industry strongly relies on software [1]. This need for more and more features stems on one side from the customer who expects a modern car to be equipped with certain features that improve comfort (e.g. navigation system, infotainment system with internet connection) and safety (e.g. ABS, ESP) and on the other side from the automotive industry which is on a journey to build autonomous driver-less cars [2] that are fully interconnected to each other (vehicle to vehicle communication - V2V) and to road side units (vehicle to infrastructure communication - V2I) via radio networks.

All of these features require a serious amount of computing power and a huge amount of program code. Currently, new features are added into vehicles by just adding another hardware node into the cars internal network. This led to the current situation were modern cars are basically highly distributed computer networks on wheels. To give a rough number, a modern higher class model contains about 70-80 hardware nodes [1].

A similar situation can be observed in the avionics industry, but there a trend to reduce the number of hardware nodes has already set in. As various articles in aviation magazines [Spi05, Ram07, Ada03, Mar06] suggest, there is a strong trend away from federated architectures towards integrated architectures (Integrated Modular Avionics - IMA), in the aviation community. Although this thesis is not targeting avionics, but automotive systems, most of the reasons why IMA is used in airplanes, are also advantageous for automotive systems. This similarity on an architectural level - not too surprisingly - can also be observed when looking at the proposed

AUTOSAR architecture which correlates with the ARINC 653 IMA architecture. Furthermore, the avionics industry started to think about IMA a long time ago[1] and is designing and implementing IMA systems for quite some time now. Therefore a lot more information on integrated systems is available from the avionics industry than from the automotive industry.

The above mentioned advantages of IMA include - amongst others: significant weight reduction, a reduced power consumption, decreased hardware costs, reduced heat generation (and therefore simpler cooling), better utilization of modern CPUs, better scalability and higher flexibility. Additionally an IMA architecture allows better reuse of software modules and makes the software highly portable. This is especially important for safety-relevant software as it lends itself towards modular certification allowing the reuse of already certified components without the need of re-certification. In the past, those certification issues were not relevant for the automotive industry, but with the introduction of ISO 26262 [STA11a] this will become one of the major issues in the automotive industry.

The above mentioned applications that are using networks like V2V, V2I or an internet connection introduce a new set of security issues that are not immanent in the current closed networks of vehicles, although there are security issues in current implementations as well. The security aspects of OVERSEE are a very important step for the automotive industry. Security already is a big issue for today's vehicles, but it has to be considered even more stringent for future vehicles.

Recent shocking news on the deficits of security in the automotive sector have been the TPMS (Tire Pressure Monitoring System) [RMM+10] which is mandatory in the US since 2008 and became mandatory in the EU in 2012 for all new cars. This system incorporates embedded systems that are located on the inside of each tire and measure the tire's pressure. The measured value is then transmitted into the car using a radio transmitter, which according to [RMM+10] can be read and spoofed from entities outside of the vehicle very easily and thus be misused to track vehicles and/or communicate non-existent problems with the tire pressure to the inside of the vehicle.

A second example showing us how stringent security is needed in the automotive sector, is a paper of a group of researchers at the University of Washington, Seattle [KCR+10]. This paper explains how the authors were able to hack a car via the On-Board Diagnostics (OBD) interface - which is conveniently located somewhere (depending on the manufacturer and model of the car) under the dash-board or the console. Via this interface the authors were not only able to write messages onto the digital dashboard, but also to do harm, like e.g. deactivate the breaks during driving or flash the motor control, while the engine was running.

Integrated systems allow the implementation of better security mechanisms that are able to monitor the access point(s) into the car more efficiently. As already mentioned above, there is a number of advantages that is gained by the shift from federated towards integrated systems:

Significant weight reduction - due to the fact that the on-board systems network consists of fewer nodes and needs less wiring a significant weight reduction can be expected.

[1] IMA has been used in jet fighters such as the F-22 or F-35 since the beginning of the 90's.

Reduced power consumption - a smaller number of CPUs that is better utilized should consume less power than a higher number of CPUs that is idle.

Decreased hardware costs - as the price difference between small microcontroller based systems and modern high performance hardware is not that big, a decrease in hardware costs can be expected.

Reduced heat generation/simpler cooling - a smaller number of hardware node does not require to pack them as densely and thus the heat generation will be reduced and cooling will be simpler.

Better utilization of modern CPUs - the performance of a modern CPU hardly justifies the use for one small application, leaving the CPU bored with a maximum utilization of few percent of its capabilities. Integrating several applications into one hardware nodes obviously leads to better utilization of the CPU.

Better scalability and higher flexibility - Using a virtualized environment, it is easier to add new applications in spare partitions and the abstraction of the network (form the view of the applications) allows to re-locate applications without any impact on the application itself, only the configuration files have to be adapted.

Easy reuse / high portability of software modules - as the software does only interact with the underlying layers via a well defined interfaces (examples could be ARINC 653, OSEK/VDX, POSIX, etc.) the software can be reused easily in new platforms, as long as the needed interfaces are available. That means, the result is a **highly composeable** system.

Modular certification allowing - the reuse of already certified components without the need of re-certification is one of the main issues discussed in this thesis, and will be discussed later. Obviously a huge reduction in time and costs can be gained if is possible to avoid the time- and money-consuming process of re-certification.

Maintainability maintenance can reach up to 90% of the total cost for a safety application, spending time and money in designing a system for maintainability will pay off in the long term.

While most of these advantages are inherent properties of an integrated system (e. g. reduction of weight) these advantages are of no value, if it cannot be shown that the integration of applications of a different safety integrity level does not impact safety of each individual application as well as the overall safety of the system. This of course cannot be shown generally and has to be part of the design and implementation of such a system.

Probably the biggest advantage of integrated systems, is the reusability of legacy code. Federated architectures make it hard to reuse legacy code, since very often the code was written for some old hardware platform which is not in use any more or for a legacy operating system that offered some weird non standardized interface to the application. All these problems are not existent for IMA systems using a standardized interface. An example for such an interface would

be the one defined in the OSEK/VDX (Offene Schnittstellen für Elektronik in Kraftfahrzeugen/Vehicle Distributed eXecutive) specification. Being compliant to a standardized interface allows to run the application on every other operating system which is compliant to this API (Application Programming Interface), and of course it gets very easy to run legacy code on the newest version of the operating system, or on another OS that offers the same standardized API. The advantage is not only the easy reusability of the code, but code that has already been certified does not have to be certified again (as long as it is not changed).

The obvious downside of strong partitioning in time (achieved by static cyclic scheduling) is, that it obviously leads to a sparse timing which limits interaction with physical endpoints/devices. This results in the change of timing properties compared to physically concurrent distributed systems which may change the system behavior even if the components are unchanged. Depending on the application this change of behaviour may not be tolerable. Consider for example the control of the interior lights, an additional latency of 100ms to adopt new settings will not be relevant. On the other hand, an additional latency of 1ms will make a huge difference for the engine control where even few μs matter.

The major part of this thesis deals with the modular certification of software. In most available publications this is limited to the VMM (Virtual Machine Monitor) that assures the independence of the applications. Of course this thesis also discusses this part of the problem, but the focus is also set on how an application has to be designed and implemented to allow modular certification and really end up with an application that can be used over a long period.

1.2 Goals

This thesis is tightly coupled with the OVERSEE project [3], a FP7 project funded by the European Commission. In short the goals of the OVERSEE project are to

- provide a single secure access point into the car,
- and allow the deployment of several independent partitions of different security levels on the same hardware node.

While safety has been descoped for the OVERSEE project, this thesis will look the safety aspects of the OVERSEE architecture. So the question this thesis is about to answer is, whether the hypervisor approach implemented in the OVERSEE platform is also suitable to fulfill the safety-requirements of the automotive industry.

This thesis also intends to make the industry aware of the advantages of open-source software and show that there is software around that is ready to be employed in safety-relevant software. While free/libre open source software (FLOSS) is entering more and more areas of computer science, (thanks to well known projects, like the Linux Kernel, the Apache Web server, the

Firefox web browser or the OpenOffice.org office software suite) and is gaining more and more acceptance [PA08], there are still industries in which it is not seen as very attractive. [2]

One of these areas are safety critical systems, because companies still think that opening their source code will result in a huge disadvantage over rival companies. While this might be true for applications and some special libraries, this is definitely not true for "infrastructural" software, like the operating systems and common libraries. The effort to develop a modern operating system is a task which can only be solved by very few, huge corporations, and even for those the benefits of doing so are not existent.

There have already been several safety critical projects employing FLOSS software (e.g. a fire detection system by Siemens Building Technologies, using the Linux kernel [Ism08], a railway signaling system [PS08] as well as a railway signaling platform [AGS08]), and there is a growing demand for FLOSS software by the safety community, and in the air and space sector. Proof for this are e.g. that NASA is doing more and more open source projects [4], and also other space programs use FLOSS software (e.g. the DLR - Deutsches Zentrum für Luft- und Raumfahrt in their MAPHEUS-1 rocket).

1.3 Methodical Approach

To show that it is in fact realistic to design and implement a system with an integrated architecture, that fulfills these safety requirements, this thesis takes a look at the problem from two different angles.

First of all a high-level safety case for the system is constructed. This safety case is structured in layers to improve maintainability and allow reusability of the high level safety case without an impact on the applications.

Secondly the design and implementation of an application is considered. To show how a design for such a safety critical application, a real-world automotive application (an indicator control) is designed with modular certification in mind.

Apart from these more theoretical problems of integrated architectures and modular certification, on a more practical level it has to be shown that it is in fact feasible to run a current automotive application in such an environment. To do so an OSEK/VDX compliant [3] operating system is ported on top of the OVERSEE platform providing a well designed interface that is commonly used in the automotive industry. On top of that runtime environment an indicator control example is implemented.

[2] The main problem still is the *Safety by secrecy* thinking that has to be abandoned. This paradigm has already been thrown over board by the security community a while ago, and is wrong for the same reasons for safety.

[3] The chosen OSEK implementation has been reviewed by me, and not deviations from the specification could be found, e.g. all parts that have been implemented are likely to be fully OSEK/VDX compliant.

1.4 Structure of the Thesis

Chapter 1 already explained why safety is such a hot topic in the automotive industry, and why there is such a desperate need for a platform that is capable of satisfying the safety as well as the security needs of the industry.

Chapter 2 is going to summarize some of the important concepts and technologies this project is based on. Furthermore it not only compares them to other concepts not suitable but also states why those other concepts are not applicable to an application like this.

Chapter 3 gives an introduction the relevant standards one has to know if he wants to build a safety-relevant system for the automotive industry.

Safety Cases are used to argue the safety of a system in a structured way. In Chapter 4 a safety case for the OVERSEE platform is given. The safety case is structured using the well established goal structuring notation (GSN) and has additionally been layered to get an even better understandable, maintainable and re-usable argument.

Chapter 5 starts of with the assessment of existing FLOSS implementations of the OSEK/VDX specification and checks the feasibility of executing them in a XtratuM runtime environment. While the reuse of components is emphasized in the above introduction, no 100% fit may ever be expected. Some adaptations to the OSEK variant used was required as well as adaptations of the toolchain to integrate it into XtratuM. The steps for transforming FreeOSEK to a para-virtualized RTE on top of XtratuM is elaborated on in this chapter.

The design of applications to allow their reuse still is a major issue in the industry. In Chapter 6 a way how a simple indicator control application can be designed to assure reusability is shown.

The main part of this thesis is concluded in Chapter 7 where a summary is given, conclusions are drawn and a outlook to future work is given.

Furthermore, Appendix A contains the code defining the interface of the turn indicator example designed in Chapter6, Appendix B presents some simple OSEK compliant example applications runnable in the runtime environment and Appendix C contains a list of papers, that have been published in the course of this thesis.

CHAPTER 2

Concepts and Technologies

This chapter gives a short introduction to the concepts and technologies that form the basis for this thesis. These concepts and technologies range from operating system principles (Section 2.1) over virtualization principles in general (Section 2.2) to virtualization in safety critical systems (Section 2.3) to the difference between federated and integrated architectures in Section 2.4.

This is followed by a description of the tool-chain used for the practical part of the thesis as well as some thoughts on code quality and coding standards in Section 2.5. Last the notations used for the safety case (Section 2.6) and the design (Section 2.7) are introduced.

2.1 Operating Systems Classification

The following section gives an overview of the most important kernel architectures that are in use nowadays, and compares the properties of the different approaches that are relevant for this thesis.

2.1.1 Metrics of Classification

The three concepts (which are described in more detail in the remainder of this section) are compared in Figure 2.1. The order in which they are presented, is from left to right:

Size of the Kernel (in decreasing order) - One discussion coming up every time when two or more operating systems programmers come together, is the question of the size of the kernel - just think about the famous discussion on monolithic vs. microkernel between Linus Torvalds and Andrew S. Tanenbaum [5] - and what part should be in the core of the OS and what part should not. When safety comes into play as an additional factor, we also have to think about the implications that a larger code base makes for certification.

While a smaller code base lends itself towards certification, there are strategies to handle big code bases as well.

Level of Hardware Abstraction (in decreasing order) While hardware abstraction is one of the main duties in a monolithic architecture, the further right we move in figure 2.1, the less abstraction is provided by the OS core, and the more knowledge about hardware specifics is required of the application and library developers.

Size of Code Run in Kernel Mode (in decreasing order) Modern CPU architectures can be operated in privileged and unprivileged modes (e.g. Ring0/Ring3 in the x86 family [6]). The different modes differ in the memory areas they are allowed to read/write from and the instruction set that is available. Generally, while in an unprivileged mode the instruction set is limited to a subset of the instruction set, and some privileged instructions cannot be used. Most OS only distinguish between kernel (privileged) mode and user (unprivileged) mode, meaning, that there are certain things that can only be done in kernel mode.

Obviously these three properties are either linked together very tightly, or exclude each other. The art is to find a way where the trade offs made in favor of one or the other approach are compensated by either technological means that allow to assure that safety properties are met, while still providing the convenient architectural properties (e.g. hardware abstraction) to the application developer.

While it is widely established that hypervisors for safety-relevant systems are small mirco- or even nanokernels, with a small code base is easier and cheaper to verify and certify, in the future also full featured operating systems could be used for hypervisors in the safety domain. In order to find out whether a monolithic approach like the one taken in the Linux kernel could be suitable for a safety-relevant system, the differences between those approaches have to be pointed out, in order to build the basis to understand the advantages/disadvantages of the properties, when it comes to safety assurance and certification.

Figure 2.1 gives a rough overview on the internals of the kernel architectures which is sufficient to illustrate the ratio between code executed in kernel mode and user mode in the four architectures.

As mentioned above, nowadays OS in the safety domain tend to be small (i.e. micro- or nanokernel approaches, often they still use no OS at all), since the verification, validation and certification of a smaller code base is already very expensive in effort, time and money, the use of huge monolithic OS seems to be not feasible - at least not with the safety assurance strategies that are used currently. Due to the high expenses in OS development, the question arises, whether an already existing OS could be used for the safety domain as well. This question is nothing new, we already experienced the same trend in the hardware area, where - back in the good old days - they had military grade (MIL) certified hardware that consisted only of expensive, high quality elements, operable in huge temperature ranges, under the harshest of environmental influences. Nowadays, this MIL certified hardware has vanished, and almost every domain is using COTS hardware even for applications where it has been unthinkable a decade a ago. The same might

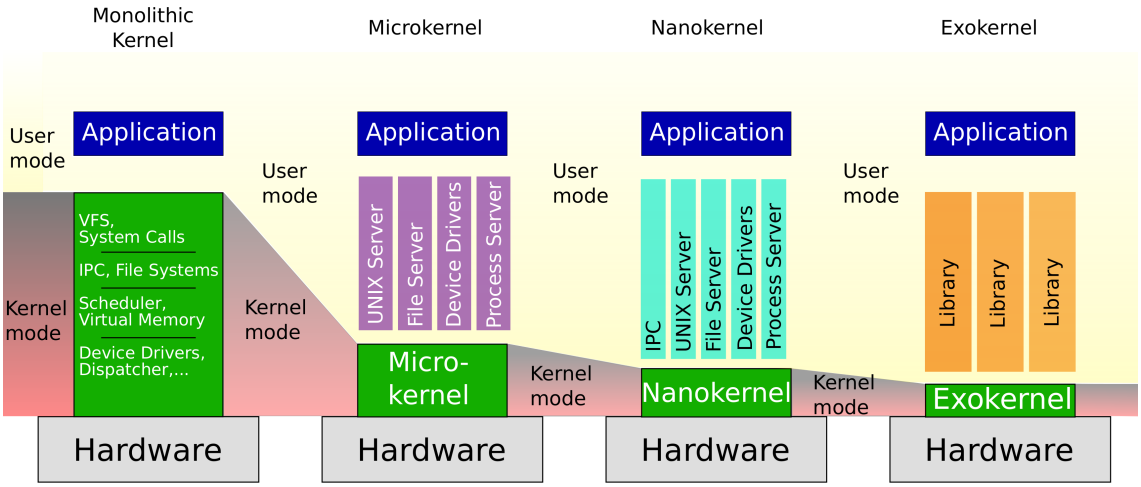

Figure 2.1: Comparison of Kernel Architectures

happen to specialized software - at least a trend of companies pushing into this direction can be recognized, viewing some recent publications [AGS08, PS08].

2.1.2 Monolithic

As the name suggests, a monolithic kernel consists of one single piece. But rather than the kernel itself being one piece - usually it is possible to dynamically load/unload kernel modules at runtime, therefore the kernel itself is not really *"one single piece (mono) of stone (lithic)"*, but rather, in a monolithic kernel all parts of the operating system run in a single address space and as a single process (multi-threading is used in all modern monolithic operating systems though). This includes not only basic functionality, but also resource allocation and management systems (e.g. memory management, file systems, network stack), power management, interprocess communication (e.g. message queues, shared memory), and so on.

This design approach of running the whole OS in one address space, requires less encapsulation than would be needed in a microkernel approach, and therefore leads to a significant reduction of system calls and context switches leading to a performance increase.

In order to being able to handle a complex, monolithic OS that consists of millions of lines of code , rigorous logical and physical de-coupling of OS subsystems is needed. This is the reason, why object-oriented design patterns can be found in modern monolithic operating systems [7,8]. If such modular design patterns, assuring the independence of subsystems are not followed, the single address space approach can easily be misused and lead to unmaintainable systems. One negative example is Windows Vista, which can be called a very large software project without hesitation with its approximately 50 million lines of code. An Ex-Microsoft employee states in his blog [9], that:

"Windows code is too complicated. It's not the components themselves, it's their inter-dependencies. An architectural diagram of Windows would suggest there are more than 50 dependency layers (never mind that there also exist circular dependencies). After working in Windows for five years, you understand only, say, two of them. Add to this the fact that building Windows on a dual-proc dev box takes nearly 24 hours, and you'll be slow enough to drive Miss Daisy." [9]

Since this statement is not verifiable and the example is a very bad one, we will now turn away from Microsoft and proprietary software and look at another monolithic operating system where the principle of de-coupling and cohesion works very well: the Linux kernel. Figure 2.2 shows an architectural map of the Linux operating system, with all its inter-dependencies. As you can see, there are only 8 dependency layers, and the inter-dependencies between the layers, as well as the inter-dependencies between the functions are not too many. Nevertheless, as this is a monolithic kernel, un-intended interference, outside of the intended interfaces is possible, but chances that someone contributing a patch that is calling around like crazy is getting away with that are very low, a very good explanation why it has to be done this way from the author would be necessary to convince the other developers.

At the moment, the Linux kernel consists of approximately 14.3 million lines of code (2.6.38). This sounds like a lot, but with Linux polymorphic configuration, this already includes the architecture specific code for all supported architectures (as of 2.6.38 there are 23 CPU architectures supported by the mainline kernel) as well as the device drivers that are part of the mainline kernel. Although this seems a lot, for a well configured, effective deployed kernel, this reduces to well under 1 million lines of code (a configuration with 2-3 file-systems, 5-10 drivers, networking and the common infrastructure will sum up to about 600k lines of code).

So how does this polymorphic configuration scheme work? At compile time you cannot only choose, which processor architecture you want the kernel to run on (and there are plenty of architectures Linux is running on, e.g. x86, x86_64, arm, ppc, mips, sparc, alpha, blackfin), but you can also decide on many other things to be built into the kernel or not. You can in example decide on the scheduler (preemptive or not), memory model being used (flat or sparse), whether to use power management or not and which function of power management you want to use, you can decide on about 20 different file-systems, or if you need any tools to debug the kernel (e.g. tracers, profilers). You can also decide to build parts of the kernel as modules instead of directly linking them into the kernel. Theses modules can be loaded and unloaded dynamically at runtime, making the kernel highly modular. This configuration system is the key element, that allows to run Linux on basically everything from embedded devices like mobile phones [10, 11], over desktop systems [12, 13] up to supercomputers [14].

If you are interested in more detail, on how much lines of code are included between two releases, how many people are involved, which companies are involved, etc. in the development of the Linux operating system, then have a look at [GKH09].

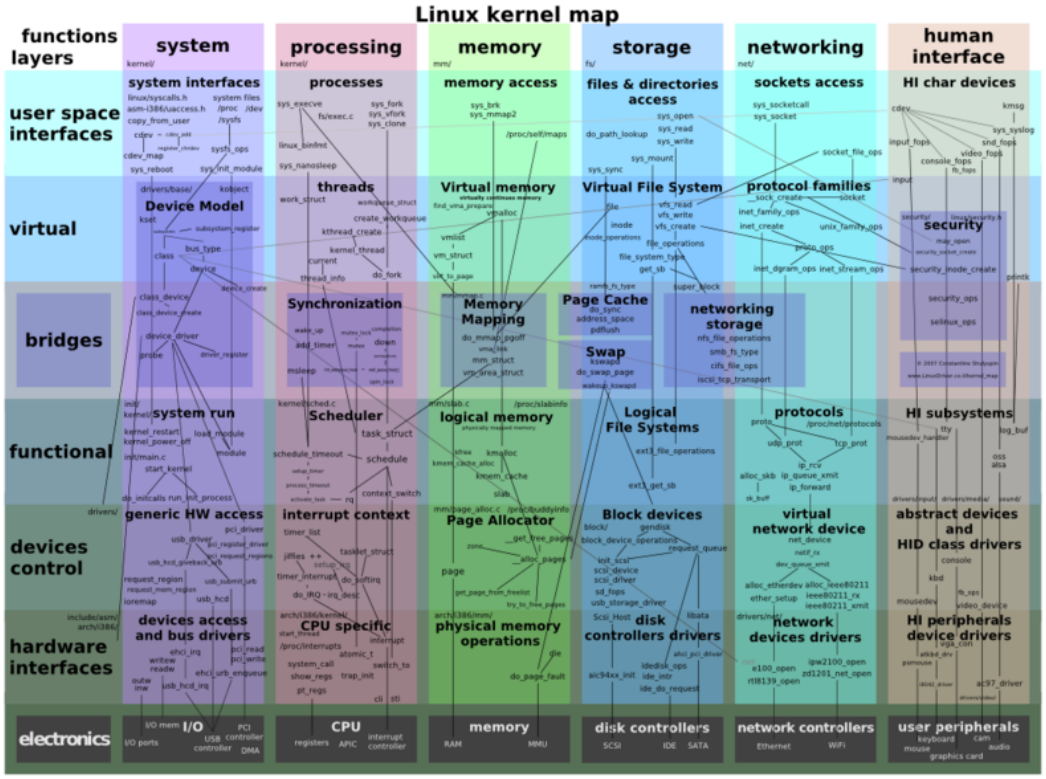

Figure 2.2: Automatically generated architectural map of the Linux operating system. This image was published under a CC-BY-3.0 license, for details see `http://commons.wikimedia.org/wiki/File:Linux_kernel_map.png`

2.1.3 Microkernel

In a microkernel architecture, the amount of code executed in kernel space is minimized. This means, that the kernel itself only contains the most essential functionality, while the rest of the operating system is shifted into separate processes (ideally into user space), where it is handled by servers (sometimes also called translators [15]). For example, while memory allocation and protection would be handled in kernel space, memory management is already a task that is shifted into user space and has to be handled by a memory management server, in a microkernel architecture.

Since the opinions about *w*hat really is essential, diverge, there are several definitions about what a microkernel contains and how big a microkernel really is, but one element has been established over all definitions, namely that the Inter Process Communication (IPC) is part of the kernel, and that the key of a successful microkernel design is the IPC performance [Lie93, Lie94, Lie96].

But even the fastest of those IPC mechanisms pose a huge penalty on latencies, due to the

overhead introduced by the need for constant switching between user- and kernel-space. For this reason, most microkernel architectures do not run their servers in user-space, but in separate processes in kernel space. This approach makes you loose more or less all advantages gained from an microkernel architecture, and therefore is not desirable. Nevertheless, this approach has been implemented in several operating systems (e.g. Windows NT/Vista/7, Mach).

Microkernels are very popular in high security applications, where a Trusted Code Base (TCB) that is as small as possible is needed. A very well known example for this is the fully verified seL4 microkernel [16]. In comparison to the Linux kernel with about 600k lines of code in a actual deployable kernel, the seL4 kernel only consists of about 8700 lines of C-code and 600 lines of assembler code. Of course at this stage, the seL4 kernel is not of much use, and all the servers needed to get a system with similar functionality as the Linux kernel would result in a comparable amount of lines of code.

2.1.4 Nanokernel

The term nanokernel was first used in [BHF+92], but the intention there was a sarcastic side blow at the Mach kernel which calls itself a Microkernel while (due to some performance enhancing opitmizations) actually being more monolithic. In contrast to a microkernel, a nanokernel is more a HAL (Hardware Abstraction Layer) than an operating system, essentially resulting in a small layer that is able to monitor and control the applications running on top of it. Often these applications tend to be full grown operating systems, making the nanokernel a VMM (Virtual Machine Monitor), that provides only hardware abstraction, monitoring, and protection from other applications to the guest OS. Since the definition of a microkernel is that *i*t includes everything that is really essential, we could say that a nanokernel contains even less [17]. This in fact is true, but this does not mean that a nanokernel is in any way incomplete, but the concept is a totally different one.

A nanokernel contains everything that is necessary to multiplex multiple runtime environments. Basically this is just memory protection, scheduling of the runtime Environments (RTEs) and interrupt virtualization. Everything else - including IPC - is shifted into one or multiple RTEs. While the this definition of a nanokernel is the one most widely in use, there are also other uses for it, e.g. [MK00] uses the term nanokernel for a kernel that supports timers with a nanosecond resolution.

Others call their operating systems nano-, pico-, atto-, ... kernel to emphasize the small code base, but most of them are not exceptional small, and from the functional point of view they clearly should be called microkernel.

2.1.5 Exokernel

The idea of exokernels [Eng98, **?**] was developed at the MIT (Massachusetts Institute of Technology) around 1994. The main design principles of this operating system architecture, are to keep the kernel itself very small and to force as few abstractions as possible on the application developer.

While previously discussed operating system architectures try to abstract the hardware as much as possible (e.g. file systems, sockets for network communication), making it as transparent to the application developer as possible (e.g. "Everything is a file" in Unix), in the exokernel architecture, as few hardware abstractions as possible are introduced. This of course makes it harder to write an application, since the application developer is forced to do much of the low-level programming himself. Of course this approach avoids the layering introduced in all the other architectures, resulting in a better performance (note that scenarios with this performance gain are very unrealistic on modern multi-core systems).

The exokernel itself is, as mentioned above, very small and in principle only checks if a hardware resource is allocatable to an application (i.e. that it is free and the application has the permissions to allocate it), but there are no abstractions forced on the application programmer. An example would be an application that reads and writes to an hard disk. The exokernel only checks if the application has the permissions to read/write that part of the disk, but it does not force the structure of the data on the application (i.e. no file system). This principle of not abstracting resources allows different applications to use their own drivers. In example, it would be possible for two processes to use two different implementations of the TCP/IP stack to send messages via the network. Libraries are used to implement the different drivers and abstraction layers, and make them accessible to the applications. This is possible because the exokernel only provides rrawïnterfaces, that can be used by the libraries to provide more abstract interfaces for the application.

Until now, two exokernel operating systems have been developed at MIT: Aegis and XOK. Both of them are explained in detail in [Eng98]. While Aegis is a proof of concept with limited support for storage, the exokernel principle is applied more thoroughly in XOK.

2.1.6 No Kernel at All

The classic approach of having no OS at all, and running the application directly on the CPU, managing the resources on its own can still be found (actually more often than one would guess), but eventually it is going to die out. The reasons for the extinction of this ancient species are simply the ever increasing complexity of modern CPUs, making it harder and harder to handle them without a full grown OS, the demand for more and more features provided by the applications that can often not be satisfied by running everything in a single `while(1)` loop and of course the portability issues of applications that are interacting with the hardware directly. Operating systems give us the abstraction needed by application developers to write portable code. If there is no OS, porting the application to a new platform is somewhere between hard and not possible at all.

2.1.7 Conclusion - Which is the Best Architecture

The question of *"Which architecture is the best?"* actually has a very simple answer: this really depends on your needs of your application. From a performance point of view, monolithic kernels are definitely the winner, the difference in the overhead of just doing a function call over invoking a microkernel's complicated IPC mechanisms and switching between user- and

kernel-space is too big to get comparable performance on a Microkernel architecture. The only way to get a microkernel based design close to the performance of an monolithic approach, is by *cheating* and running the server processes in kernel-space as well and thereby invalidating the microkernel paradigm of minimizing the code run in kernel-space.

For modern multi-core and multi-processor systems, the performance advantage of monolithic operating systems over microkernel based operating systems has proven to be even greater, but with the growing number of CPUs new problems arise and to scaling current operating systems to higher numbers of cores new approaches will have to emerge. One such approach might be to have a small operating system (micro- or nanokernel) that is doing few things well, and the RTEs running on top of that small kernel are full-featured monolithic operating systems pinned to different cores loosely coupled to each other, communicating through inter-domain communication mechanisms (e.g. for safety-critical and safety-relevant systems, such mechanisms could follow the design principles of the ARINC 653 interpartition communication - see section 3.2 for details).

From a safety point of view, the approach taken by many operating systems, is to have a small kernel (the nanokernel approach) that is easy verify-able due to its size, but contains merely a HAL as protection of the runtime environments in time and memory (see 2.4 IMA - Integrated Modular Avionics). Since the purpose of theses systems is to integrate multiple applications, that were running on multiple CPUs before, into one hardware node, the communication between those RTEs is loose and with relatively small throughput (since in an federated architecture it has probably been running over a CAN bus or something similar) even a relatively slow IPC mechanism is faster than the network connection used in an federated implementation before.

2.2 Virtualization

Virtualization is a technique that has already been in use since the 1960s [18], but which got very popular over the last years. The maybe most important example where virtualization is in use is server hosting. Since most web-, mail-, ... servers do not need the resources provided by a modern computer system[1], server hosting companies tend to run several servers on one computer system. Since each of them is rented to another costumer, it is necessary to make sure the administrators of the servers do not bother each other, and it is also essential, that if one server is hacked, the others are still secure. This basically is what virtualization systems do: they allow to run several guest systems on one hardware node and assure, that these guest systems are separated from each other appropriately. For the users inside of the virtualized environment it looks like they are working on a real hardware node.

From this short summary of main employment scenarios it becomes obvious, that the main goal of virtualization is *separation*, to accomplish *independence* between runtime environments. For applications in the server market, this *separation* has to provide mainly independence in memory, to protect the data in one runtime environment from the users of the other runtime environment. Independence in time and a communication system that discourages the impact of

[1] At least not all the time.

one runtime environment on the other one play a secondary role. Even if it is considered, this done on a quality of service level and not a strict separation level.

2.2.1 Full Virtualization

The big advantage of full virtualization is, that you can run the operating system of your choice without any further modifications, and the guest CPU architecture does not necessarily have to be the same as the host CPU architecture. QEMU as an example supports x86, x86_64, ARM, SPARC, PPC, MIPS, MIPS64, m68k(Coldfire),... as target platforms (running x86 as host plaform).

Since full virtualization allows to run unmodified operating systems, also proprietary systems can be installed as guest systems. An example how such an system could look like is given in figure 2.3, where the host operating system works as an VMM (Virtual Machine Monitor), sometimes also called hypervisor to run several guest systems. As you can see in figure 2.3, full virtualization allows us to run different operating systems like Linux, BSD, Windows, Solaris in parallel on the same computer system. The number of guest systems depends on the available hardware resources and can be up to hundreds of guest systems.

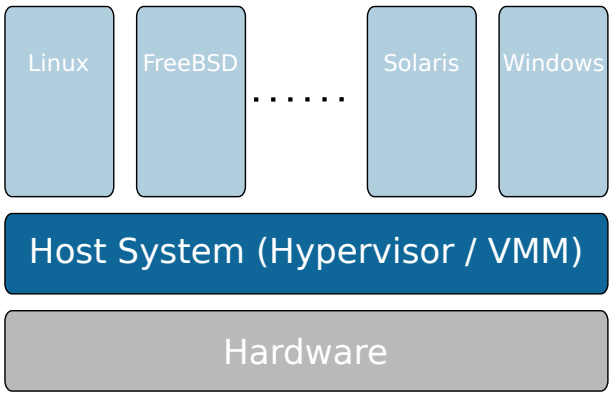

Figure 2.3: Full Virtualization

Full virtualization operates on the instruction level, and provides a specific CPU architecture to the guest operating system, by interpreting the single instruction. You can visualize this as a big switch-case statement with one case for each instruction of the CPU architecture. What the virtualization system does, is to fetch the next instruction, search the switch-case statement for the according function, which emulates the behavior of the respective guest.

Well known examples for a full virtualization FLOSS solutions are Virtualbox [19] and QEMU [20].

2.2.2 Paravirtualization

In contrast to a full virtualization system, a paravirtualized system does not operate on the instruction level, and the guest operating system has to be modified. This means, that the guest operating system is aware, that it is not running on real hardware and the boot process as well as the hardware specific functions are replaced by so called paravirtualized operations. Obviously, paravirtualization is restricted to guest operating systems, where the source code is available[2].

Furthermore, the system is usually aware on which hardware architecture it is executed on, which is the same one as the hypervisor itself. [3] Examples for paravirtualized systems are KVM [21], XEN [22] and XtratuM [23].

2.2.3 XtratuM2 and its Hypercall interface

The remainder of this section is going to explain paravirtualization in greater detail, using XtratuM2 [23] as an example. The reasons why XtratuM2 was used in this thesis, is that it is the hypervisor chosen by the OVERSEE consortium. The main reasons why it was chosen as basis for the OVERSEE platform are that it is very small, accessible to everyone (it will be released under an open-source license) and it is staying close to an operating system specification that is very well known in the safety community (ARINC 653 - see section 3.1 for details).

XtratuM is a type I (bare metal) hypervisor targeting safety related composable systems. The main guidelines for design come from one of the key IMA standards, ARINC 653 [Com03]. XtratuM is an active FLOSS project being developed at Instituto de Informatica Industrial, Universidad Politecnica de Valencia. While the OVERSEE project is focused on security aspects the goal is to provide a platform that in principle can also satisfy safety requirements. There is a strong sharing of core demands on the lowest OS layer with respect to safety and security, and while safety and security have sometimes conflicting demands at higher levels these differences are not present at the lowest level of a hypervisor [Rus99]. The key to unify the requirements at the lowest level of safety and security is to provide a sound:

- Temporal isolation
- Spatial isolation

allowing to build high-level services on top that only allows explicitly permitted sharing of resources as well as communication. XtratuM thus is intentionally reduced close to the bare minimum that is needed to allow high-level services to operate in there respective OS environments and still give strong guarantees with respect to independence.

XtratuM offers a relatively narrow interface of Hypercalls to its partitions. This simplified things a lot for our porting efforts. In this section we will only briefly outline hypercalls that were used in this porting effort, for a full list of available hypercalls we refer you to the XtratuM

[2] Of course the owner of a proprietary OS could paravirtualize it.

[3] At least the author would see little sense in paravirtualization if the architectures are different.

Reference Manual [MRC11]. The intention of this section is to show the interface size used in the XtratuM guest management for a actual example.

Time services: XtratuM provides an independent virtual time to each domain on which the guest-OS then can implement high-level timing services. In this sense the low-level services can be seen as mimicking hardware timing services.

- XM_get_time: Time entities in XtratuM are of microsecond granularity, and are maintained relative to the last system reset. There are two basic clocks in the system. Clocks in XtratuM are strictly monotonic. Clocks are maintained for the system (XM_HW_CLOCK) as well as for the partitions execution (XM_EXEC_CCLOCK)
- XM_set_timer: Interval timer service (providing one-shot behavior by setting the interval to 0). The expire time is an absolute time with respect to either hardware clock or execution clock. To a partition the expired timer is signaled as a virtual timer interrupt (emulating a hardware timer).

Interrupt services: Signaling to partitions is provided via virtual interrupts, it is up to the guest-OS to then assign suitable meaning and response to the events. Note the absence of a interrupt request hypercall - as all resources are allocated statically in XtratuM there is no need for a request_irq.

- XM_enable_irqs: globally disable interrupt delivery to this partition
- XM_disable_irqs: globally enable interrupt delivery
- XM_set_irqmask: used for masking (blocking) and unmasking of interrupts

Basic partition management functions: Much of the partition management is related to the initialization and shutdown phase of a partition. The essence of the interface is that it minimizes the state information that needs to be handled by the hypervisor - leaving more or less all state related work to the partition.

- XM_suspend_partition: This is a basic function that is only used in supervisor mode to manage a partition. It is used to block a partition (waiting on a resource) or temporarily stop a partition if errors are detected.
- XM_resume_partition: Simply the opposite to the above partition suspension.
- XM_shutdown_partition: As the hypervisor does not have information about the internal state of a partition shutdown is provided as an asynchronous notification. Basically a partition is sent a request to shut down via a dedicated interrupt and after cleaning up any internal state will then terminate it self.
- XM_reset_partition: Conversely to the XM_shutdown_partition, the XM_reset_partition is a forced shutdown of a partition whereby a warm and cold reset is differentiated, a warm reset preserves some of the partitions initialized resources (i.e. open ports and memory areas) while a cold reset clears this all and thus can have side-effects on other partitions via communication channels no longer being served.

- XM_halt_partition: A halted partition is set into an inactive state but no reclamation of resources (spatial or temporal) are done (that is left to the partition reset) in this state the partition is simply no longer scheduled by the hypervisor. The XM_halt_partition called by non-supervisor partitions can only pass self as the target of the halt.
- XM_idle_self: This allows a partition to suspend itself within its time slot. The partition will only be re-woken on its next time-slot or if a NMI is received within its current time slot. This can be used to implement donation schemes for system partitions.

Basic system management functions: Note that these are not directly related to the guest-OS as these calls are related to privileged domains - they are listed here for completeness.

- XM_halt_system: The halt partition call (also described above) is used by system partitions to manage the system as a whole as well as individual partitions. Only supervisor partitions can halt other partitions. This is used to prepare a partition reset as well as mode switching.
- XM_reset_system: Brute force system halt of the entire board after this only a hardware reset can reboot the system. No precautions are taken to put any partition into a sane state thus this is only the last step in a system shutdown as well as in extreme emergency situations.

Low level Communication related functions: In practical implementations one does not actually use the low level object class functions but uses the wrappers provided to the commonly used objects (sampling and queuing ports as specified in ARINC 653). These wrappers thus are the actual hypercalls that will be issued though they are rarely used in guest-OS code.

- XM_read_object: read the object, verifying access permissions and other low-level properties. Usage in all reading functions like XM_receive_queuing_message, XM_read_sampling_message, etc.
- XM_write_object: write the object. This is used i.e. in XM_write_sampling / queuing_message, XM_send_queuing_message.
- XM_ctrl_object: is used to create and manage objects with specific properties as well as query these objects (i.e. retrieve the id of the object). This hypercall is used in object management functions like XM_create_sampling / queuing_port, XM_get_sampling / queuing_port_status, etc.

While the overall hypercall set is a bit more elaborate than listed here, the essential calls used to implement the OSEK guest-OS are listed showing how small such a guest-OS interface actually can be constructed if the abstraction level is pulled down far enough.

2.2.4 A Formal Approach to Virtualization

So far only very practical approaches to virtualization have been mentioned. Of course there is also a theoretical approach to this topic. Especially when it comes to safety critical systems a mathematical approach to proving key properties of a system is desired. In [Rus82] John Rushby gives a *Proof of Separability*, for secure operating system kernels. Although his focus is on security, the separation properties discussed are key for safety as well.

Rushby begins by defining an abstract model of a computer:

$$M = (S, I, O, NEXTSTATE, INPUT, OUTPUT) \tag{2.1}$$

where M models a machine using the 6-tuple of S, the finite non-empty set of states this machine is able to take, I is the set of inputs, and O is the set of Outputs. While O is available continuously at every point in time, it can only be set once in the beginning. NEXTSTATE is a function that defines the transition between states, i.e. $NEXTSTATE : S \rightarrow S$, and $INPUT : I \rightarrow S$ and $OUTPUT : O \rightarrow S$ are functions to process input and generate output.

Furthermore, we have to consider a system shared by multiple users (otherwise there would be no need for security), this is done by defining C as a set of colors representing multiple users. Each of this users has his own input and output, which shall be secure and must not be leaked to other users in C. To indicate that, superscripts are used to indicate the user while subscripts are used to indicate the value at a point in time of S, I and O.

Next, Rushby gives a formal definition for security, the conclusion of which is:

"... each user of a C-shared machine must be unaware of the activity , or even the existence, of any other user: it must appear to him that he has the machine to himself." [Rus82, p.9]

Building on these basic definitions, Rushby then proofs that:

Theorem 1 A C-shared machine M is secure, if for each $c \in C$, there exists an M-compatible private machine for c.

Theorem 2 If $M = (S, I, O, NEXTSTATE, INPUT, OUTPUT)$ is a C-shared machine and $COLOUR : S \rightarrow C$ is a total function, then a private machine $M^c = (S^c, I^c, O^c, NEXTSTATE^c, INPUT^c, OUTPUT^c)$ is M-compatible. (that is, the user c does not recognise the difference).

Theorem 3 Such an M-compatible machine exists for user c.

From those three theorems, Rushby derives six conditions for his security verification technique *Proof of Separability*:

" Using 'RED' as a more vivid name for the quantified colour c, these conditions may be expressed informally as follows:

- *When an operation is executed on behalf of the RED user, the effects which that user perceives must be capable of complete description in terms of the objects known to him.*
- *When an operation is executed on behalf of the RED user, other users should perceive no effects at all.*
- *Only RED I/O devices may affect the state perceived by the RED user.*
- *I/O devices must not be able to cause dissimilar behaviour to be exhibited by states which the RED user perceives as identical.*
- *RED I/O devices must not be able to perceive differences between states which the RED user perceives as identical.*
- *The selection of the next operation to be executed on behalf of the RED user must only depend on the state of his regime."* [Rus82, p.15]

The restrictions made by this list of conditions is, that it requires total isolation between the runtime environments (or regimes, as Rushby calls them). In a real-world application some form of communication between runtime environments will be required.

2.3 Virtualization in the Safety Domain

Although most virtualization solutions focus on Server applications, virtualization is also interesting for embedded and safety applications. The reasons for this are, that if you have a hypervisor or VMM (virtual machine monitor), that is very small (few lines of code), chances are good that it is relatively easy to assess. If you have such an hypervisor that has been approved by the authorities, it makes it very easy to build your system on top of it, since the application partitions are independent of the hypervisor and of each other, assessing them becomes easier too. The other big advantage is, that the reuse of legacy code becomes much easier and at the same time the development and integration of new applications can be done faster.

In this section, the different levels of virtualization are introduced, and some examples for each level are given. There are various reasons why virtualization is becoming more and more popular:

- Better utilization of hardware resources is achieved, if more applications run on one big computer, instead of several smaller ones.
- To secure a computer system. This technique is called sandboxing [24].

- Software development - if you write a new piece of software you can first test it on a virtual machine, the advantage is, that you don't crash your real hardware, and some of the virtualization systems allow you to debug the guest systems (e.g. QEMU provides a gdbserver).
- NooM (e.g. TMR) systems can be done in a virtualized way, to prevent CCF's in the application
- In safety critical systems, virtualization is used to keep independent applications from influencing each other.

This last point the above list, is exactly what an IMA (integrated modular avionics) does, and it is the reason why different virtualization approaches are explained, and examples of virtualization software are given. The IMA approach and its advantages will be discussed in more detail in section 2.4.

2.4 Integrated Modular Avionics

At the moment, most safety-critical and safety-relevant software is based on a federated architecture, but in recent years there was a shift towards integrated architectures, and there are some examples - most out of the avionics industry - that use such an integrated architecture. This section explains the difference between a federated and an integrated architecture and lists some real world examples which use an IMA software architecture. After that a short rational why the integrated approach is also interesting for applications other than avionics is given.

2.4.1 Federated Architecture

Today, most automotive systems in operation follow a federated approach. This means, that they have one node for each software module. This makes many things easier. E.g. the FCU's (fault containment units) can be identified easily - usually an FCU is one node. The downside is, that a federated approach leads to a highly distributed system very fast, and even relatively simple applications need many processors. One simple example - originating in [Wat06b, Wat06a, CBW07] - of a federated system is shown in figure 2.4. This example could be a simple control application, where one node is reading data from some senors, based on this data a new set value is calculated by a controller located on a second node, and the new set value is applied to the effectors by a third node. In order to get data required by the nodes out of one and into the other, a communication network is needed. To get a fast responding controller with a tight control loop, it is necessary to have a high speed communication network

As you can see, this very simple example already needs three CPU's. One CPU to read the sensors, one to calculate the set value, and one to operate the effectors. Running this application on three modern CPU's would be a waste since none of the three tasks uses even 1% of the CPU power of a modern CPU. So it would be nice to find a way to integrate the three parts of the system into one CPU, and not only achieve a better utilization of the CPU, but also save hardware and reduce the power consumption.

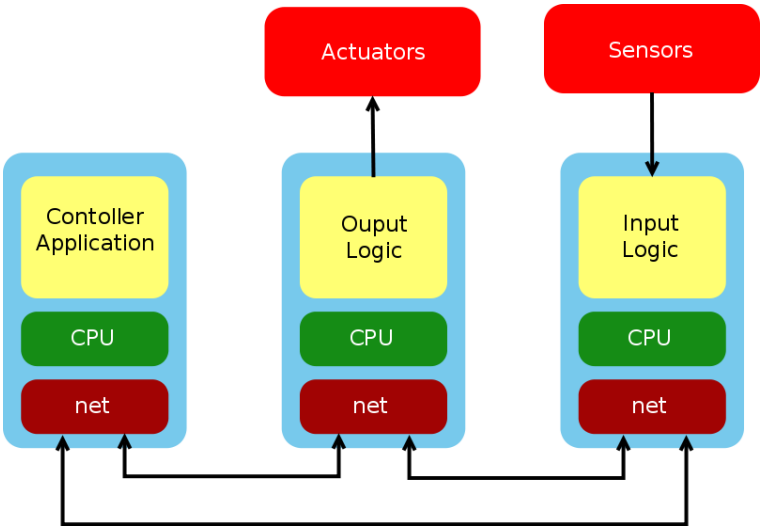

Figure 2.4: Simple example of a Federated System

2.4.2 Integrated Architecture

In contrast to a federated architecture, the integrated approach allows several independent parts of the software to reside in one hardware node. Figure 2.5 shows our simple controller application designed in an integrated manner, running input logic, controller and output logic on one common CPU. So instead of three bored CPU's we were able to reduce our hardware to one bored CPU. Furthermore the network connecting the three nodes is replaced by a virtualized network, increasing the savings even further.

Of course it would now be possible - depending on the application - to take further advantage of our integrated approach. Let's for example assume, that our controller is controlling the motor of a CNC mill, then our CNC mill would have 3 such controllers (one for each axis), meaning that a federated approach would require 9 hardware nodes to get the job done. With our integrated approach, all 3 controllers could be located on the same node, thus allowing a massive saving in hardware. But these savings in hardware are only the most obvious advantages here is a list of the most important advantages of integrated architectures:

Weight reduction: Reducing might not be the biggest problem in the automotive industry (unless you are designing sports cars), but a weight reduction of course is also of advantage when it comes to fuel consumption

Reduced power consumption: Inherent to the reduction of nodes is a reduction of power consumption.

Decreased hardware costs: Modern COTS CPUs already are in the same price-range as special purpose microcontrollers. Sparing CPUs therefore will reduce the hardware costs.

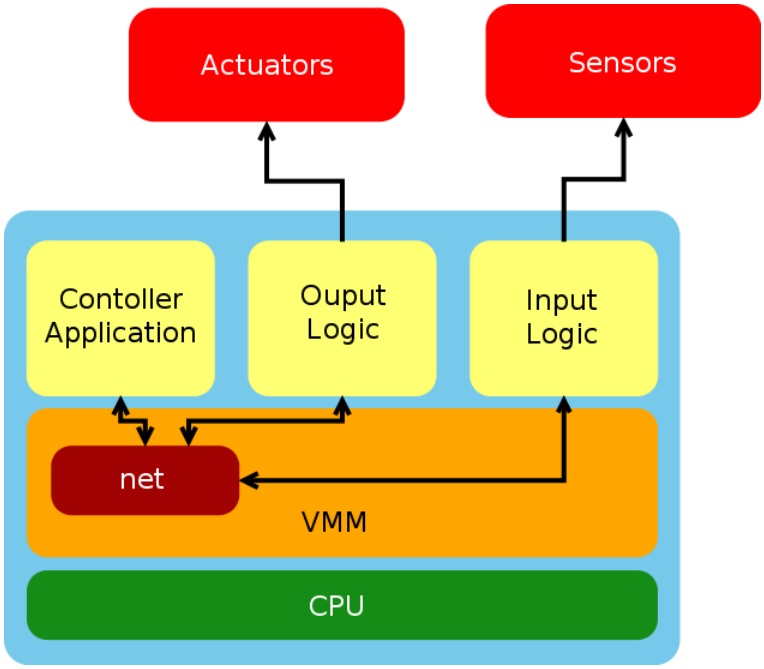

Figure 2.5: Example of an integrated architecture.

Reduced heat generation: In some industries (notably avionics) cooling can get an issue when lots of nodes are packed in a very tight space. Reducing the number of nodes in that space simplifies cooling, or might even eliminate some of those problems.

Better utilization of modern CPUs: With the availability of very chip high performant CPUs, many nodes in a modern vehicle are utilized at few percent of their capabilities. Integrating applications into the same HW node will increase the utilization of the individual node.

Better scalability / higher flexibility: It is questionable if the increase in applications will come to an end any time soon. Having platforms that support the integration of new platforms are the key to increase the flexibility and get systems that scale well - even if applications that we cannot even think of today have to be integrated into future systems. In example, consider the above discussed example. At some point a costumer could require to add a webcam to visually monitor the system. In a federated architecture, this would require you to introduce a new hardware node into the system, despite the fact that the available nodes have the resources (e.g. CPU time left, USB ports for the camera etc.) available. If an integrated approach had been taken from the very beginning, the system in figure 2.5 can easily be extended by a new application container and extended by the webcam application, depicted in figure 2.6

Better reuse of software modules: The development of applications for the automotive industry is very expensive, thus a the desire to reuse applications is very high. Having an

integrated architecture with well defined interfaces that eliminates the hardware dependencies of the application itself helps to increase the reusability of newly developed as well as legacy applications.

Increase of application portability: In order to increase the reusability portability is a cructial pre-requisite. Of course writing highly portable code is a lot more initial effort,

but it will pay off in the long term.

Communication capabilities: Integrated architectures not only allow to dynamically add applications, but also to add communication channels as desired.

Security: Although security is often neglected in safety critical systems, it will become very important in the future. That said, virtualized (interpartition) communication is easier to protect than wired internode communication.

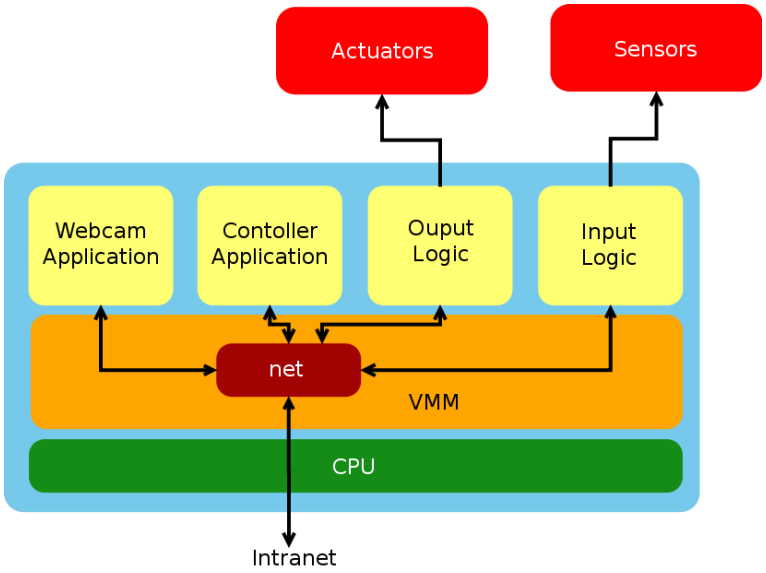

Figure 2.6: Example of an integrated architecture with an additional application

2.4.3 Related Work

In the following a short list of operating systems and applications that follow the IMA approach is given.

2.4.3.1 Integrated Modular Avionics Operating Systems

The following list of OS that are following the IMA paradigm shows, how diverse the implementations can be, and how various approaches for the OS can be ranging over all the classes in 2.1.

POK: the *partitioned operating system* [25] is an ARINC-653 compliant microkernel, that focuses on safety and security. POK has emerged from teaching activities at different schools and universities and is licensed under a BSD license.

XtratuM: is a mostly ARINC 653 [23] compliant [4] hypervisor based on an nanokernel architecture. While the first version was strongly dependent on the Linux kernel (initialization of the hardware), XtratuM2 is already standalone and can be used without Linux, although Linux can be run as a guest inside of one or multiple partitions.
According to the homepage XtratuM is licensed under the GPL, but only version 1 can be downloaded directly at the moment.

VxWorks: WindRiver's VxWorks [26] is a proprietary real-time operating system. It is widely used in smaller electronic devices (e.g. digital cameras), and there is also a ARINC 653 extension available, which has been used in many air- and space applications (e.g. Mars pathfinder mission) as well as in other mission critical and safety related applications.

Integrity: Greenhill offers another ARINC 653 compliant operating system named Integrity [27].

2.4.3.2 Applications

The IMA approach has first been used in avionics in the fourth generation of jet fighters. Examples are the Lockheed Martin *F-22* and *F-35*, as well as Dassault Aviation's *Rafale*. More recently, IMA has found its way into civil aviation. Two very well known modern airplanes using IMA technology are the Airbus A380 and the Boeing 787. Although they both use IMA, the approach taken is very different [Ram07], as we will see in the following.

While the A380 integrates only few applications into one LRU, the approach taken in the 787 is a lot more radical, and uses a Common Core System (CCS), which integrates over 100 LRUs into one hardware node.

2.4.4 The Integrated Approach Outside of Avionics

This thesis uses a lot of examples from the avionic sector, since the concept of integrating several software modules into one hardware module is already accepted in avionics, while other industries currently have not concept comparable to IMA. Although it can only be speculated on the reasons for this, it is obvious that integrated architectures would make a lot of sense in many applications. For example railway signaling, where lots of ECUs could be merged into one big TMR server, or machine controlling where e.g. all axis of CNC mill could be controlled by a single ECU.

With AUTOSAR [28], the automotive industry already started using the integrated approach, and the trend can be expected to go towards integrated components inside of vehicles very quickly. Of course one example for the research into the suitablity of integrated architectures in automotive is the OVERSEE project [3], and as we will see later in chapter 4, also ISO 26262

[4] Actually its ARINC 653 with extensions.

mentions the concept of integrated architectures, allows the usage of multiple applications of different integrity levels if separation is guaranteed and suggests runtime environments that are compatible to ARINC 653.

2.5 Toolchain - used tools

One of the first things to do in the development phase of a safety critical software project, is to select the (subset of a) programming language(s) that shall be used as well as the used tools. Furthermore the suitability of the selected languages and tools for the given application has to be shown. A very short summary of the selection used in this thesis is giving in the following.

2.5.1 Language Selection

In order to being able to select the toolchain, first a language has to be selected. In this case a long language selection process and discussion of the same can be skipped, since the language is inherent, due to preexisting code. From the XtratuM nanokernel itself to all OSEK OS implementations that were investigated in this thesis, the C programming language is used. Furthermore, C is in wide usage in the automotive industry, but to meet the high quality demands of the industry some restrictions are made. These restrictions are standardized and published in the MISRA-C:2004 [MIS04] coding guidelines.

In the practical part of this thesis not only MISRA-C compliant code will be written, since in some parts, this would not make sense. E.g. potential necessary changes (if any) in the XtratuM2 nanokernel will not be MISRA-C compliant. For the OSEK RTE as well as the example applications MISRA-C compliance is the goal, but lacking a MISRA checker it will not be possible to prove the compliance. The code in the OSEK RTE on the other hand will be MISRA-C compliant, as the current standalone implementations are MISRA-C compliant as well current standalone implementations are MISRA-C compliant as well.

The restrictions in MISRA-C are mostly dedicated to the avoidance of uncertainties in the C programming language itself. Such uncertainties, also called *undefined behavior* [29, 30], are differences in compilers that arise due to inaccurate specifications in the C programming language itself, and many experts in the safety domain criticise C as unsuitable for safety applications due to this undefined behaviour. The reasons why C is still a valid and good choice for safety critical and safety relevant software are:

High Availability - Compilers for the C programming language are available for almost all computer architectures and platforms on the market. Therefore, porting the system to a new architecture is a lot easier.

High Performance - Modern C compiler optimize the code, making it faster and more efficient than an human written assembler code could be.

Low Complexity - The C programming language allows structuring of the Code, making it better readable and therefore better maintainable. Of course it is always possible to write

"smart" code using perverse code constructs, but this only depends on the programmers awareness of the importance of well structured code.

High Portability - as mentioned above, C compilers are available for almost any computer architecture, therefore porting the application to a new architecture takes less time and is not that expensive.

But still, the critics of using C in the safety domain is rectified to the point, that programmers using C have to be more aware of what they are doing, and cannot just rely on the compiler to straighten out their mistakes and shortcomings. Using C in the safety critical and safety relevant domain requires the programmer to not only know the language, but also to have a lot of knowledge about the compiler in use and the behavior of the compiler, when in comes to *undefined behavior*. The C FAQ [Sum96] defines *undefined behavior* as follows:

"Anything at all can happen; the Standard imposes no requirements. The program may fail to compile, or it may execute incorrectly (either crashing or silently generating incorrect results), or it may fortuitously do exactly what the programmer intended." [Sum96, p.189]

Using a FLOSS compiler gives the developer the advantage of getting all the information he needs, when he needs it. To prove this point, consider the example, given in [30], where running optimization stages of the compiler in a different order leads to quite different results. Problems like these are really hard to find when using proprietary products, and as they only happen in corner cases they can stay unrecognized in the code of the compiler for a long time

Also the usage of coding guidelines like MISRA-C prevents the appearance of such problems, by assuring the necessary code quality. E.g. the following (MISRA-C compliant) version of the code given in [30] would not have led to the problem:

```
void contains_null_check(int *P) {
    if (P == NULL) {
       return;
    }
    else {
       *P = 4;
    }

    return;
}
```

Going for code quality is especially important when writing software for safety critical systems. An appropriate code quality not only prevents many of the most common programming errors (see the example above), but also makes it easier for the successor of the author to make changes, reuse the code or port it to some new hardware architecture.

2.5.2 Tools

This section lists all the tools used for Development and Debugging, as well as the version used during development. The version used is basically the default version in debian squeeze except for some of the code checkers that are used in the latest version to gain maximum information.

make [31] is used to automate the build process, thus assisting to make the system usable by everyone who does not know which exact compiler-flags to use and which objects to link together to get the resulting binary.

gcc the GNU compiler collection [32] is one of the most widely used compilers, offering support for a variety of programming languages and a lot of different hardware platforms. For this thesis only the C compiler is used.

git [33] is a version control system, designed by Linux inventor Linus Torvalds. Using a revision control system for this thesis is not only a matter of convenience, but as this thesis has a strong safety context, it provides a high level of traceability.

doxygen [34] is a source code documentation generator. The comments in the code are annotated with tags. These tags are used by doxygen to generate a source code documentation. The usage of in-code documentation systems like doxygen have the advantage that it is easier to maintain the code documentation, as interface changes are recognized automatically, and the in-code documentation can be done while introducing those changes.

coccinelle [35] is a semantic patching tool that allows to handle collateral evolution (i.e. it is able to propagate complex API changes throughout the system thus saving lots of time). Although the capabilities are beyond that, it can be expected that this will be the most important usage of coccinelle for the practical part of this thesis.

2.6 Goal Structured Notation

The argumentation of why a safety-critical or safety-relevant system can be regarded as *s*ufficiently safe is a process that is carried out starting from the very first high-level design all the way until the dissemination of the system. The document containing this argumentation is called a **safety case** (see chapter 4 for the safety case developed in this thesis) . For many systems this safety case has to be maintained over years, more often even decades as many safety-critical systems are meant to be employed over long timer periods (just think e.g. about a nuclear power plant).

To gain the needed level of maintainability, a structured approach is the key to success and for structuring safety cases, the goal structured notation (GSN) is a very popular approach. Although GSN has been around for quite some time, it has just recently been standardized into a community standard [OCYL11].

Goal: The goal describes a requirement or sub-specification of the system that is to be satisfied. This could be a high-level goal like *"The system can tolerate all single component failures"*, it also could be a very specific goal corresponding to a low-level requirement, like

"Searching the list of active tasks is of complexity O(1)". While goals generally express requirements or sets of requirements they also commonly represent derived requirements (subgoals) that are needed to develop the top goal. From the safety case perspective it is complete if all the goals are satisfied by the elements they reference to (Assumptions, Justifications, Solutions and Contexts).

Undeveloped Goal: GSN is by definition incomplete, but it makes this incompleteness explicit, and thus mitigateable. Undeveloped goals may allow assessing the consistency of a system under assumption of satisfying an undeveloped goal. As an example one might have an undeveloped goal *occurrence of double faults is covered by architectural means*. While the component being described might not allow coverage of double faults, it may be sufficient to have an undeveloped goal for this case. The advantage would be that a double fault scenario now can be developed by having a TMR to develop this goal but without the need to change the rest of the structure and thus covering different integrity requirements with a single GSN structure.

Strategy: Strategies refer to general solutions to a class of problems. Examples of strategies are TMR, timeouts, retransmission or encodings. These strategies can be applied to many different situations and have general properties e.g. a TMR has a single fault coverage or retransmission can cover transient random faults. Strategies are generally used as a constraint on the specific solution in the sense that it is not needed to elaborate the detailed limits of a solution that is based on a TMR as this is well established know how.

Solution: A solution describes the concrete mitigation of a particular problem (in example a 16bit Cyclic Redundancy Check (CRC)). A solution may be part of a strategy and it may be based on a set of assumptions.

Assumption: are always present in any abstract description of systems and system components. The intent of having assumptions explicitly stated in GSN is to allow re-assessment of a safety case in a changed environment or scenario. In this sense they are not only constraints but part of the safety case management capability of GSN. An assumption can be viewed as an inherently satisfied goal. During safety case development it may be necessary to use assumptions to constrain the complexity of the safety case or simplify validating the logic of a safety case component by providing a set of assumptions for a particular goal and showing that all of the cases would be covered.

Justification: A justification can be seen as the bottom line of a safety case. The generic process for safety (see Figure 2.7) is represented in GSN and the justification object serves to provide the necessary overall rational for the path through the argument structure. It also may serve as justification of a specific solution if this is a well accepted and generic method e.g. referring to a *H*AZOP result. Note that justifications in this form make assumptions about methods and procedures listed as justification - this will generally not be explicitly listed (i.e. nobody would state HAZOP compliant with MOD 00-58 [Min00a, Min00b] result). At a higher level a safety case needs to provide this information to prevent cluttering the lower levels.

Context: Safety is a system property (discussed in more detail in section 3.5) which means nothing else but that the actually provided margins or residual failure probabilities depend on the specific system configuration and the environment in which it is operating. Any change of the operational constraints and/or system environment would potentially impact the conclusions of the safety case. These assumptions are documented in the contextöbjects of the safety case.

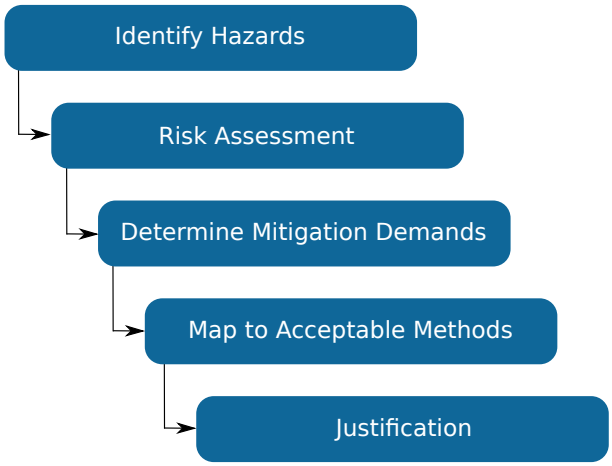

Figure 2.7: A generic safety process.

These elements described in this section are the basic elements of GSN, allowing the unfamiliar reader to understand the GSN diagrams used in section 4. For details or more advanced elements of GSN, please refer to [OCYL11].

Note that while GSN is a notational method, it also lends itself to development of patterns – in his seminal work Fan Ye developed a number of GSN based safety case patterns for COTS systems. These patterns can be viewed as a meta-strategy - for details of the strategies suitable for COTS seeof [Ye05, Appendix A] (*"A COTS safety argument pattern language"*) .

2.7 Real-Time Structured Analysis and Design

This section gives a short introduction to RTSAD - the *Real-Time Structured Analysis and Design* [DeM81, Goo01, KS92, Coo03] method, that is used for the design in the practical part of this thesis. RTSAD is a method for system design and specification. This is in not restricted to software but to the whole system including electric, electronic, mechanical, etc. parts, as well as the interaction of humans with the system. RTSAD is a well established specification method that is based on the Yourdon style variant of *structured analysis and system specification*, created by Tom DeMarco [DeM81] in the 70s. The simple, yet powerful notation of Yourdon allows to quickly understand the system specifications. The extension of Yourdon to RTSAD, making

it even more suitable for real-time and safety-related systems is therefore a very common system specification method. Although the RTSAD method is not explicitly recommended by IEC 61508, Yourdon is, and since RTSAD only an extension of Yourdon it can be seen as highly qualified and recommend by the standard as well.

As mentioned above, the notation itself is rather simplistic, but this simplicity can be seen as the strength of this method. In comparison with other design methods like UML, which tend to loose themselves in a plethora of different styled diagrams and charts, RTSAD has a very small set of design elements for diagrams complemented by data dictionaries.

The main strategy of RTSAD is a *divide and conquer* approach. First the system is designed on a rather abstract level, describing the interaction of the system with its environment in a *data context diagram*. This data context diagram is then broken down into subsystems, and the interactions between the subsystems are designed and specified. This step is repeated as long as necessary or practicable, i.e. until a level is reached, where further breaking down is no longer possible, or not practicable. The key elements of RTSAD are:

Data context diagram (DCD): As already mentioned above, the data context diagram is a high level specification of the system, describing the function behavior of the system, as well as its interactions with its environment. A DCD only contains one bubble - representing the system - as well as boxes representing external entities and arrows describing the data flow between the system and the external entities (i.e. the system's environment).

Control context diagrams (CCD): The control context diagram describes the interaction triggers and system response in a diagrammatic way similar to the DCD. The main focus though is to de-couple data from signals - both in and out of the system. While these signals could be described as data entities as well - any signal does need to transmit some information, even if it were only a single bit, it makes sense to de-couple the description as signals primarily impact control flow while data impact the processing steps more. It should be noted that for many projects the CCD is integrated in the DCD and no separate diagram is provided. In the case of joint diagrams the process is diagrammatically presented as a data transformation with control inputs.

Data flow diagrams (DFD): The key principle in RTSA is the decomposition of the system into subsystems. This decomposition is done diagrammatically by extending the original DCD/CCD to reveal its internal structure while maintaining the external context as described in the DCD/CCD. The DFD included data and control flow objects, each of them is labeled and numbered and each of the internal relations (data or control connections) gets a unique label. These names are retained throughout the entire hierarchy and the numbering is used to make the hierarchy visible in the next layer of decomposition which is described by further DFDs.

Control flow diagrams (CFD): Similar to the joining of DCD and CCD it is quite common to join CFD with DFDs, in cases where the control flow is of significant complexity though a separate diagram may be used. The only real difference is that CFDs do not contain data transformations, while DFDs commonly do incorporate control transformations.

Data dictionary: As noted in the DFD description each connection between entities in the diagrams (control transformations, data transformations and data stores) is labeled with a unique and meaningful name. These names are the first level of completeness and consistency checking in RTSA. While the labels describe the logical data/control flow, RTSA also anticipates limited quantified analysis, for this purpose each label (which corresponds to control signal or data flow) is described with detailed attributes in the data dictionary. This description generally needs to be adjusted a bit to the problem at hand. The goal though always is to have a sufficient description of logical and physical (if applicable) properties of each connecting entity so that one can also perform basic quantitative analysis.

State Diagrams (SD): Problems in control systems can be broken down to finite state machines or a hierarchy of the same. This does not only pertain to actual state machine algorithms but also to control flow in a system of coupled entities - as an example the transition from an initializing to an operational state of the system might be bound to the successful allocation of resources - this de-facto describes a state machine even though this will generally not be directly visible in the code (see figure 2.8).

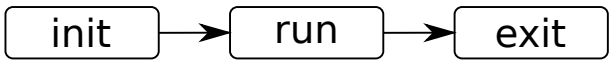

Figure 2.8: A simple finite state machine (FSM).

This state diagrams may then be further detailed by the use of control transformations in describing the criteria that lead to the respective state change (see figure 2.9).

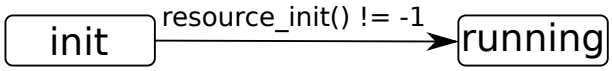

Figure 2.9: A finite state machine with state change criteria.

Event-action diagram (EAD): Event action diagrams are quite similar to FSM descriptions just that they are relieved of the strict formal model employed in FSMs. Actions to FSMs are confined to state transitions basically - EAD allow more general actions. Such an action could be a mandatory procedure or an externalized action like *call the cops*. EADs are though used as de-formalized FSMs in many designs.

Response time specification (RTS): One of the weaknesses of the data dictionary identified early on in the use of RTSA was that it did not connect information from the SD or EAD and the DFD directly in the temporal domain. Temporal specifications of periodic signals are covered by data dictionaries, signal impact on internal state is covered by SD/EAD and direct control transformations are visible in the CFD/CCD as represented by control transformations (and the associated control specification CSpec) - but all this does not

cover the system response bounds to non-periodic events in a quantified way - this is basically the role of RTS.

Equipped with the basic concepts, tools and techniques presented in this chapter, we can now proceed and also gather the basic knowledge about relevant safety standards in Chapter 3. In combination, these two big topics are then used to analyze and argument the safety of the OVERSEE platform in Chapters 4 to 7.

CHAPTER 3

Relevant Standards

Since this thesis has a strong safety-relevance, it is vital to be familiar with the needed standards. In this case this is ARINC 653 (Section 3.1) as the first standard to introduce an integrated approach to the safety industry, furthermore XtratuM2 is using it as a guiding standard and is almost compatible to ARINC 653. Section 3.3 gives an introduction to OSEK/VDX, the currently most widely used standard in the automotive industry. A mapping between those two standards is given in section 3.4.

While the first thre sections discuss operating systems specifications that are used to build safe operating systems for the avionic and automotive domain, sections 3.5 and 3.6 are dedicated to standards that help to determine and achieve the necessary level of safety for software systems. The chosen standards here are the IEC 61508 (often called *the mother of all functional safety standards*), as well as IEC 26262 the new functional safety standard which has been derived from the IEC 61508 to suite the automotive industry.

The last section in this chapter 3.8 contains a short summary to MISRA-C - a standardized coding guideline used in the automotive industry.

3.1 ARINC 653

This section gives a brief introduction to the ARINC 653 [Com03] specification and its most important parts. The title of the ARINC 653 specification is **Avionics Application Software Standard Interface**, and as the abbreviation says, it is published by the Aeronautical Radio Incorporated (ARINC). As the longer, more descriptive name says, this specification introduces a software interface designed for safety critical avionic applications. ARINC 653 backs up the idea of an Integrated Modular Avionics (IMA) system (but does not necessarily claim it).

One might read the ARINC 653 [Com03] specification and find it restrictive. But these restrictions follow the effort to specify a way to build a safe system architecture, which is highly composeable. Exactly these restrictions are the key to the success and popularity of the ARINC

653 specification. Not only the separation of software modules (partitions), but also a communication system which does not allow application partitions to influence each other in a way not intended by design, is important to achieve this goal. All these essential conditions are provided by the ARINC 653 specification, and are the reason why it is a suitable standard for our purpose, which is to provide a generic application platform for multiple independently developed applications while ensuring the high-availability of the overall system.

The central person in a ARINC 653 based development is the **system integrator**. His job is to allocate the hardware resources required by the (application) partition developers and create the XML configuration file. This means, that the application developers ask the system integrator to allocate the resources (e.g. CPU time, memory, communication channels) needed by the respective application. It is then the system integrator's responsibility to ensures that everyone gets the needed resources.The ARINC 653 specification contains the following sections:

Partition Management In ARINC 653 each software module is called a partition. There are two types of partitions: *Application* and *System Partitions*. The main difference is, that while the first kind is allowed to communicate with the OS only via the *APEX Interface* (APplication EXecutive), the second one is also allowed to directly call core functions. Partitions are the units that are scheduled by the operating system. For the partition scheduling, ARINC 653 uses a static cyclic scheduler, which is configured in the XML configuration file. Partitions are not allowed to preempt each other (i.e. in a partitions time slot, only this partition can be scheduled).

Process Management Every partition can have several processes. These processes are created and initialized at partition initialization time. The processes are managed and scheduled from within the partition and are invisible outside of the partition:
" The partition should be responsible for the behaviour of its internal processes. The processes are not visible outside of the partition." [Com03, Section 2.3.2]

Time Management includes functionality to set timers (e.g. timed wait, periodic wait). These functions operate on processes within a partition, i.e. if one process performs a "timed wait", another process of the same partition is scheduled (if the partitions slot is not over).

Memory Allocation The required memory for each partition is statically allocated via the XML configuration file. ARINC 653 does not provide dynamic memory allocation, though partitions are free to provide dynamic memory allocation to processes within the partition.

Inter- and Interpartition Communication ARINC 653 provides different ways of communication within and between partitions:

- Interpartition communication is the communication between two processes in different partitions. There are two communication modes: *queuing ports* and *sampling ports*. It is also very important to note, that it is not important for the sender partition, whether the receiving partition is on the same hardware module or on a different one the way to send a message is always the same. As interpartition communication is very important for this thesis, it is explained in detail in Section 3.2.

- Intrapartition Communication is the communication between two processes of the same partition. The intrapartition communication is achieved by *buffers, blackboards, semaphores* and *events*.

Health Monitor The Health Monitor uses knowledge about the behavior of the system (e.g. Partition X sends a message at least once per second) to find out if a partition is misbehaving. It is then able to restart the partition(s), or even the whole system.

Configuration In ARINC 653 all the configuration is done via an XML file. In this file, the System Integrator is able to manage all the resources, discussed previously. His job is to make the static schedule, configure all the necessary communication channels, and decide whether it is necessary to add more CPU's or not.

Verification and Validation Although Verification and Validation is listed as a main part of ARINC 653 the only paragraph in the related section is:
"*The system integrator will be responsible for verifying that the complete system fulfills its functional requirements when applications are integrated and for ensuring that availability and integrity requirements are met. Verification that application software fulfills its functional requirements will be carried out by the supplier of the application.* "

[Com03, Section 2.6, p.41]

3.2 Interpartition Communication in ARINC 653

Integrated systems contain a number of software modules with different safety levels on a single hardware module. Therefore it is essential to partition these software modules in time and memory, so that errors are contained in the module where they show up, and to prevent the propagation into other software modules (of even higher criticality).

Nevertheless many applications require the different software modules to communicate with each other. To guarantee a safe communication channel, which does not allow erroneous software modules to influence the other software modules in the system, the interpartition communication system is used to monitor the communication and make sure that everything is alright. To prevent possible faulty software modules from directly or indirectly influencing other software modules, the interpartition communication system has to be designed properly.

To achieve this independence of the partitions, ARINC 653 introduces an interpartition communication system that is strictly built on polling. That means that there is no way to signal the receiver, that he should fetch a message. Assume that there is a signal notifying the receiver that he got a new message. This would make it possible for a faulty sender (e.g. babbling idiot) to indirectly influence the receiver, which would exhaust a lot of its computing time handling signals, instead of getting real work done. Of course this strictly polling based intercommunication system has a huge impact on the performance, but performance is not what ARINC 653 is about. As explained above we are trying to establish a communication channel between two partitions, that does not allow either of the partitions to influence the other partition.

The ARINC 653 specification uses different properties to classify the different modes of communication. Properties common to all forms of interpartition communication are that they are message based, the data is transparent to the message passing system, sending can be periodic or aperiodic, and the partitions access the communication channel via ports. The ports, as well as the communication channels are defined by the system integrator in the XML configuration file, and configured at initialization time.

Basically ARINC 653 differs between two types of communication modes: Sampling Ports and Queuing Ports. These two categories are explained more precisely in Section 3.2. Both of these two communication modes are message based. The data is transparent to the message passing system. The application partitions access the communication channels via ports, which are configured at initialization time, and specified by the XML configuration file. All messages are atomic at the application level. That means, that either the complete message is received, or nothing is received. Messages can be sent periodic or aperiodic, but this is up to the application, and the ARINC 653 interpartition communication subsystem has nothing to do with that. What the subsystem has to do, is to check whether the message is outdated or not. This done with an apriori configured REFRESH_RATE, which specifies the maximum age of a message, before it becomes invalid.

According to ARINC 653 [Com03] section 2.3.5.2, the communication ports are used to communicate within a core module, between the core modules, as well as between a core module and a non-ARINC 653 component. This project, only handles the first of these three levels of communication. The other two are out of scope, and might be part of a following project.

Sampling Ports

Sampling ports (see [Com03], p.28) can be used to send messages at any time, but they are restricted to fixed length messages. There exists only one copy of the message, which is overwritten every time a partition sends a new message. There is no buffering supported, therefore messages can be lost, but the data a reader gets, always contains the newest available instance of the message.

This communication mode is pretty simple, there is no need for buffer management because the length of the message is known a priori (from the XML configuration file), therefore the buffers can be set up at initialization time. Messages which are not of the right length are discarded, because ARINC 653 says in section 2.3.5.1:

" At the application level, messages are atomic entities i.e., either the whole message is received, or nothing is received. Applications are responsible for assuring, data meets requirements for processing by that application. This might include range checks, voting between different sources, or other means." [Com03, Section 2.3.5.1]

This paragraph precisely defines the border of jurisdiction between the application and the message passing system. The message passing system has to make sure that the message arrives in one piece at the receiver (which is definitely not possible, if the message has the wrong length), but it has no influence on the data itself, which is application specific.

Another possibility how a message on a sampling port could get invalid is ageing. Each port

gets assigned a REFRESH_RATE at configuration time. This rate defines the maximum age of the message which is acceptable. If the message gets older, it becomes invalid.Sampling ports support uni-, multi- and broadcast. That means a message can be sent to one, multiple or all other nodes.

Queuing Ports

In contrast to sampling ports, queuing ports (see [Com03], p.28) buffer the messages. The buffer shows FIFO behaviour and supports fixed as well as variable length messages. Even if variable length messages are used, the total resources usable are hard limited per queue by a fixed maximum length of messages. Marking the message boundary/length is up to the application - no support from ARINC 653 is given on that matter.

One important part of queuing ports is the buffer management. The buffering of the messages makes sure that no messages are lost, i.e. several instances of the same message are recognized by the receiving partition. ARINC 653 mandates two independent buffers, one where messages from the sender are stored, as well as a second one, where checked, correct messages are stored. Only messages from the second buffer can be read by the receiving partition. In contrast to sampling ports, queuing ports support only unicast communication.

3.3 OSEK/VDX

In the following, the open OS specification OSEK/VDX [36] is summarized, looking at the highest conformance class extended conformance class 2 (ECC2). The lower conformance classes are subsets of ECC2, the relation between the conformance classes can be found in [Con05], Figure 3-3.

3.3.1 OSEK OS

The most important part of OSEK/VDX to understand the context of this thesis is OSEK/OS. It specifies an OS, well suited for the needs of the automotive industry. The standardized API and well defined behavior of OSEK/VDX compliant operating systems, allow high portability of applications developed for such an operating system.

The following summarizes the essential points of OSEK/VDX, for more details, please refer to the homepage [Con05], where all parts can be downloaded without charge, since it is an open standard.

Task Management OSEK/VDX distinguishes between two different types of tasks, basic tasks (BT) and extended tasks (ET). While a BT can only release the processor if it terminates, or if it is preempted by a higher prior task or an interrupt service routine (ISR), an ET can also go into an *waiting state*, allowing the scheduler to dispatch a lower priority task, without terminating the higher priority task. An example for this would be, if the ET is waiting for some kind of event to happen. Instead of just polling and wasting CPU time, it can go into the waiting state, in which it is not scheduled, before the event is signaled (more on signaling below).

OSEK/VDX provides a Task state Model ([Con05], section 4.2) that describes the states a task can be in, and the transitions between those states the task state model for extended tasks is shown in figure 3.1. For basic tasks the task state model is essentially the same, but without the *waiting* state.The states a task can be in are the following:

- **running** - a task in the running state is currently active and executed. At all times only one task can be in the running state. (OSEK/VDX is specified for single core CPUs only, multi-core solutions are covered by newer versions of AUTOSAR [28])
- **ready** - all schedule-able tasks are in the ready state, waiting for their turn to transition into the running state.
- **suspended** - tasks in the suspended task are currently inactive and wait for their activation to become ready.
- **waiting** - extended tasks that are waiting for some event to happen can decide to go into the waiting state instead of wasting CPU time. A task in the waiting state will be *released* from the waiting stated as soon as the desired event has happened.

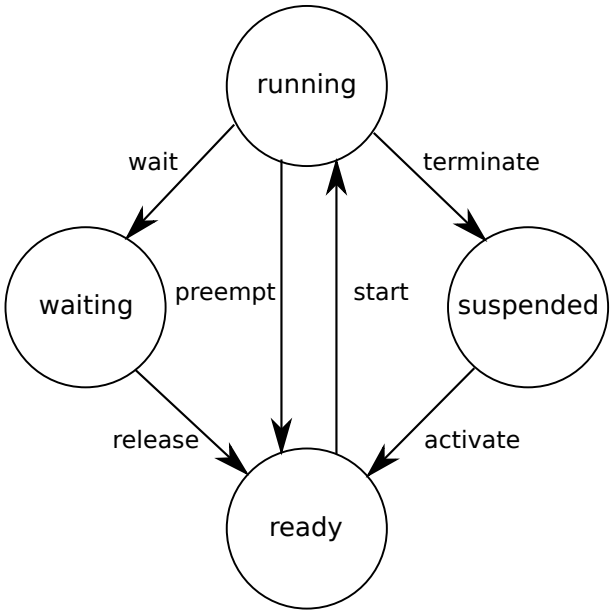

Figure 3.1: OSEK/VDX Task state model for extended tasks [Con05, Figure 4-1, p.17]

In the OSEK/VDX task management, the scheduling policy is assigned by the system integrator. A systems scheduling policy can be configured to be fully preemptive, non-preemptive or mixed (both preemptable as well as non-preemptable tasks are running at the same time). The scheduling decision itself is based on priority scheduling, with static priorities (where 0 is the lowest priority and bigger numbers denote higher priorities).

Depending on the conformance class, one or more tasks of the same priority can exist at the same time. If the system is configured to allow preemptive tasks, a priority ceiling

protocol is provided to prevent priority inversion. For non-preemptive tasks, rescheduling happens only in the following cases:

- the *running* task terminates successfully
- explicit call of the scheduler by the *running* task
- the *running* task transitions into the *waiting* state

Interrupts - OSEK/VDX distinguishes between 2 types of interrupts:

- Category 1 ISRs do not use operating system services, and after they are finished, execution continues exactly at the point where it was before the ISR has been called (no influence on task management).
- Category 2 ISRs are allowed to use operating system services that are concerned with handling interrupts (enable, disable, etc.), these ISRs prepare the system for a RTE to run a dedicated user routine (comparable to Bottom Halves). After a category 2 ISR has been executed, the execution does not return to the last point before the interrupt, instead the scheduler is invoked, in order to check if a dedicated user routine (bottom halve) has a higher priority than the current running task.

Events are a means of synchronization. They are only available for extended tasks, since they are used to transition tasks into and out of the *waiting state*. Events are objects assigned to tasks, and uniquely identified through their name and the task they belong to. At task activation of an extended task, all the events are cleared automatically. Events can be set by any task (also basic tasks) as well as category2 ISRs, to change the task state of the events owner from the *waiting* to the *ready* state, but only the owner of the event is allowed to clear the event after-wards.

Depending on the scheduling policy, the point of rescheduling is either, when the event is *set* (fully preemptive) or at next point of rescheduling in non-preemptive mode (listed above in Task Management).

Resource Management In order, to allow the concurrent task execution model described above, resource management has to be provided, in order to assure

- mutually exclusive access to resources
- prevent priority inversion
- detect and prevent deadlocks
- and prevent the transition into a *waiting* state while holding resources

All these problems are high probable error sources, the goal of the OSEK resource management system is to do everything possible to prevent them from the operating system side. To reach these goals, the following mechanisms are specified by OSEK/VDX:

- **OSEK Priority Ceiling Protocol** [Con05], Section 8.5, introduces the OSEK Priority Ceiling Protocol, used to avoid priority inversion and deadlocks between tasks. This protocol provides a ceiling priority for each resource (this ceiling priority is statically assigned at system generation), which shall be set to priority of the highest-prior task using the resource. If a task with lower prior task access the resource, its own priority is risen to the resources ceiling priority temporarily. After the task releases the resource, its priority is set back to its old priority.

  Section 8.6 of [Con05] introduces an optional extension of the OSEK Priority Ceiling Protocol, that includes ISRs.

- **Restrictions when using Resources** OSEK/VDX defines restrictions on the system calls that may be used, while a task is holding a resource. The calls forbidden while holding a resource are *TerminateTask, ChainTask, Schedule* and *WaitEvent*. As can be inferred from the names, those are the calls that invoke the scheduler and might lead to the scheduling of another task are the ones prohibited while holding a resource. This is a simple an effective way of assuring the mutual exclusivity of resources, furthermore it helps to prevent deadlocks between tasks.

- **Scheduler as a Resource** If a task wants to prevent itself from being preempted, it can lock the scheduler. In case a task chooses to do so, the scheduler is still invoked, but not allowed to schedule any other tasks. Interrupts are received and processed independently of the state of the scheduler.

Alarms are special events, offered by the OSEK OS, to activate tasks after a counter has experienced. A counter in OSEK is represented as a counter value measured in *ticks*, if the counter reaches a predefined value, the alarm expires and the alarm-event is set off. The predefined value can be specified either relative to the actual counter value (relative alarm) or as an absolute value (absolute alarm).

A tick (i.e. an increment of he counter value) can be triggered by all kinds of external events (i.e. interrupts). One counter source that has to be available by specification on OSEK/VDX compliant systems is a counter incremented by the real-time clock.

While any number of alarms can be assigned to the same counter, each alarm has exactly one counter and exactly one alarm-callback routine is statically assigned at system generation time.

Error Handling OSEK/VDX defines *hook routines* which can be used for a variety of tasks.

Hook Routines are part of the operating system, although implemented by the applications developer. They can be seen as a possibility for the application developer to extend the functionality of the operating system. The *hook routines* are called by the OS at pre-configured events. The choice of events amended by hook routines is left to the operating system deisgner. Since *hook routines* are part of the OS, they have higher priority than all tasks, and they cannot be interrupted by category2 ISRs. While the interface for *hook routines* are standardized, functionality is not, and thus is up to the application developer.

Error Handling OSEK/VDX distinguishes between two categories of errors - *application errors* and *fatal errors*. In case of a *fatal error*, the integrity of the operating systems internal data can no longer be guaranteed, and the operating systems shuts down. If an *application error* occurs, a system call could not be serviced properly, but the internal data of the operating system is still assumed to be correct. If a system service routine returns an error code, an *error hook routine* is called. This hook routine has to be provided by the user, who has the responsibility to bring his application back on track.

System Startup/Shutdown All low level (hardware) initialization is up to the application developer, the specifications of the OSEK/VDX concern only the platform independent parts and start with the call to *StartOS*.

Shutdowns are a little more complicated, since each task has to be informed of the shutdown, so it can bring potential actuators into a safe state. Therefore before the system can actually shutdown, a *shutdown hook* is called.

Debugging is done via a *PreTaskHook* and a *PostTaskHook*, which are called on task switches. These hooks can be used for debugging and measurement purposes [1].

Standardized API in [Con05], sections 12 and 13, the system services provided by the API of an OSEK/VDX compliant operating system are specified. This API must be the only way for the application to use the above described operating subsystems, like alarms, events, etc.

3.3.2 Other parts of OSEK/VDX

OSEK/VDX consists of multiple parts, OSEK OS described previously is the most important one for this thesis, while the other parts do not really play a role in this context, as the intent is to port a OSEK OS compliant system as a proof of concept. Full compliance to all parts of OSEK/VDX is out of scope. But for completeness, here is a short list of the other parts:

OSEK COM - Communication Layer specifies a message based communication for interprocessor communication - it shows a stunning resemblance with ARINC653 interpartition communication. Details about this can be found later in section 5.4.

OSEK NM - Network Management provides a standardized way for configuring networks of OSEK/VDX nodes, initialization of networking peripherals, network start-up, network monitoring and a lot more. Everything that is needed to start, maintain and diagnose a network of nodes running OSEK/VDX compliant nodes.

OSEK OIL - OSEK Interpretation Language specifies a standardized configuration mechanism for OSEK/VDX compliant nodes. The configuration files as defined by OSEK OIL are per node (single CPU nodes only), and do not include network configuration.

[1] An example experienced during this thesis was a stack overflow that happend due to a poorly implemented first shoot at the context switch. The stack overflow happened after several task switches. Using hooks, it was very easy to locate the problem.

OSEK Time - Time-Triggered OS specifies a time-triggered variant of OSEK VDX, the differences are e.g. time-triggered scheduling and the like. It is also possible to run a mixed variant, were a standard OSEK OS is run in one or multiple time slots of the time-triggered OS.

OSEK FTCom - Fault-Tolerant Communication provides a standardized time-triggered networking variant that in order to achieve better fault-tolerance than with the standard OSEK/VDX networking layer.

3.3.3 ISO17356

As mentioned above, OSEK/VDX is an open standard, that can be obtained for free at [36]. In addition to this open standard, it has been transferred into an IEC standard, with the official ISO number ISO 17356. The following list shows the document version of the open standard, that has been the basis for IEC 17356:

- OSEK Glossary (located in OSEK Binding 1.4.1 [Con99a], part of ISO 17356-1, which consists of a ISO-style introduction and the glossary)
- OSEK Binding Specification (base: OSEK Binding 1.4.1 [Con99a], ISO 17356-2 with exception of the glossary)
- OSEK OS (base: OSEK OS 2.2.1 [Con05], ISO 17356-3)
- OSEK COM (base: OSEK COM 3.0.2 [Con04], ISO 17356-4)
- OSEK NM (base: OSEK NM 2.5.2 [Con99d], 17356-5)
- OSEK OIL (base: OIL 2.4.1 [Con99c], ISO 17356-6)

3.3.4 AUTOSAR

While OSEK/VDX is currently the most used operating system standard in the automotive industry, its successor AUTOSAR [28] is on its way to take over. The main reason for this is definitely the fact that the newest release - AUTOSAR 4.0 - is the first operating systems standard taking multi-core CPUs into account. Since the days of single-core CPUs are counted, this a real important topic that will shake the safety-community over the next years.

For this thesis AUTOSAR will not be used, the reason is simply its size and the fact that AUTOSAR is based on OSEK/VDX, so every application written for an OSEK/VDX compliant operating system can also be executed on a AUTOSAR compliant operating system.

3.4 Establishing a Mapping between ARINC653 and OSEK

Since OSEK/VDX does explicitly support an integrated approach, a mapping against ARINC653 - which is a very common standard within the avionics community - will show that an integrated approach can be applied to OSEK/VDX as well. The mapping is done in several steps:

- A *dictionary* that maps expressions used in both standards to each other.
- A table showing a rough outline of which mechanisms are mapped against each other.
- To map mechansisms that can be directly mapped, the requirements of the OSEK/VDX runtime environment will be used.
- Mechanisms that are only described in one of the two standards have to be considered, and it has to be shown they do not contradict the standard in which they are not included.

Dictionary

Table 3.1 contains a list of expressions that describe the same concept in ARINC 653 and OSEK/VDX respectively. It is important to have those expressions as a basis for the rest of the mapping of the two standards.

ARINC653	OSEK/VDX
process	task
port	message object
sampling port	unqueued message
queuing port	queued message
intrapartition communication	internal communication
interpartition communication	external communication

Table 3.1: Dictionary of matching vocabular: ARINC 653 $\Leftrightarrow$ OSEK/VDX

ARINC653	OSEK/VDX
partition management	mixed OSEKtime / OSEK/VDX System
process management	task management
	interrupt processing
memory management	
buffers and blackboards	Interaction Layer
Message Communication Levels	Internal/External Communication
queuing ports and sampling ports	Interaction Layer + virtual Network Layer
events	event mechanism

Table 3.2: Matching components in ARINC 653 and OSEK/VDX

Mapping ARINC653 / OSEK

Next the components of the respective standards that map well together are listed in table 3.2. As you can see some of the entries do not have a counterpart. They will be explained in more detail in the next section, but most of the time this means that this component is hidden from the

application through abstraction. In example interrupt processing in OSEK/VDX does not have counterpart in ARINC 653. This does not mean that ARINC 653 compliant operating systems do not use interrupts at all, it just means that interrupts are abstracted through the ARINC 653s APEX interface.

Mechanisms without a match

While the previous section handled those mechanisms that are described in both standards, thus allowing a 1:1 mapping, this section is handling those mechanisms described in only one of the two standards.

To complete the mapping, the important thing in this section will be to give a rational, why these mechanisms are no contradiction to the standard they are not in. The order in this section will be first mechanisms that are in ARINC653 but not in OSEK/VDX, and then those which are in OSEK/VDX and not in ARINC653.

Mechanisms without a match – ARINC653

Partition Management The fact that parition management is part of the ARINC653 specification and that it is not included in OSEK/VDX is the reason for this whole document.

Although OSEK/VDX does not support partition management as described by the IMA approach in ARINC653, in Section 7 of [Con01], the possibility to run an OSEK OS on top of an OSEKtime OS is described. The mode of operation described in such an mixed OSEKtime / OSEK/VDX System is very similar to a hypervisor (OSEKtime OS) taking control over the system and telling a guest (OSEK OS) when it is allowed to run. Nevertheless, at this very high level view, the similarities end, since the OSEK OS is not scheduled at preconfigured points in time, but when OSEKtime OS is idle.

From mapping all the other parts of the two standards to each other, we can conclude that OSEK/VDX is suitable for an integrated approach, just as ARINC653 is.

Memory Allocation Since OSEK/VDX does not contain parition management, it does not include memory allocation either. In an integrated approach, it will be necessary to ensure the independence of OSEK OS instances. Therefore memory is allocated statically during configuration time.

Semaphores While ARINC653 defines semaphores to synchronize process, OSEK/VDX does not explicitely define semaphores. But since OSEK/VDX uses semaphores e.g. to explain the problems of synchronisation mechanisms [Con05, Section 8.4.1], Semaphores cannot be a contradiction to OSEK/VDX.

Message Communication Levels ARINC 653 defines in section 2.3.5.2 three levels of communication, depending on the communication boundaries they cross:

- Within core modules (communication between processes)
- Between core modules (communication between partitions)
- Between core modules and a non-ARINC653 component (communication with external entities)

OSEK/VDX on the other hand only distinguishes between internal and external communication [Con04, Sections 2.3.2 and 2.3.3]. Clearly the first level of communication in ARINC653 maps directly to internal communication in OSEK/VDX, which describes the communication between tasks in OSEK/VDX and between processes in ARINC653. Furthermore the two other levels of communication in ARINC653 can be considered as special cases of external communication, and can hence be both be mapped to external communication in OSEK/VDX. The only difference between those two in a paravirtualized OSEK/VDX runtime environment will be that the Network Layer has to transfer them via a different communication medium.

Mechanisms without a match – OSEK/VDX

Interrupt Processing As already mentioned, while ARINC 653 does not define interrupt processing to the application through the APEX interface, OSEK/VDX gives some guidance in Section 6 of [Con05]. It introduces 2 categories of interrupts - category 1 is not allowed to use most of the OS services and does not pose a point of rescheduling (no impact on task management), while category 2 is allowed to use all of the OS services and when the ISR is exited the scheduler is triggered to choosing the next task to be scheduled.

Alarms While OSEK/VDX defines alarms for event and time management, ARINC 653 has those 2 split up into two parts - time management covering those alarms triggered by the passage of time, and event services covering those alarms triggered by other events (be it software or hardware driven).

3.5 IEC 61508

The IEC 61508 is a generic standard for functional safety, giving guidance to developers of safety critical applications. Generic in this case means nothing else than that it is not an application sector standard, but it could be used in all industries. Usually it is used if the industry has no standard for functional safety. Being generic, it often is used to derive new, industry specific standards from it. This is why it is sometimes referred to as “the mother of all functional safety standards“.

Unfortunately safety standards as the IEC 61508 are often (especially by beginners) seen as a “list of requirements that have to be fulfilled to get my system certified“ when it should be seen as a common starting point between the developers and the authorities, kind of a common basis that establishes the current state of the art and gives guidance on how the process of developing a safe system should be approached.

IEC 61508 consists of a set of 8 documents, these 8 parts are numbered from 0 through 7, where part 0 is an introduction to functional safety, parts 1-4 are basic safety publications, and parts 5-7 give additional guidelines to other parts (details below). Parts 1-4 are the normative part of the standard, while parts 0 and 5-7 are only informative. The following list gives a short overview on the content of the various parts, to give an idea what this is all about.

Part 0: Functional Safety and IEC 61508 [IEC10a] As mentioned above Part 0 is and informative part of the standard and gives a lot of valuable definitions and explanations about safety and especially functional safety. Although the definitions given in Part 4 are more exhaustive than those in Part 0, the basis for understanding the purpose of the IEC 61508 and the impact it has on system design and development is build here. While some of the other parts might not be of outmost importance for everyone involved in the life-cycle of the system, this is the part that is mandatory for everyone to read and understand. Therefore, some of the most important definitions given in Part0 are given in the following. First we start with the term *safety* which is defined as:

"Freedom from unacceptable risk of physical injury or of damage to the health of people, either directly or indirectly as a result of damage to property or to the environment."

[IEC10a, Clause 3.1, p.13]

Right afterwards the term *functional safety* is defined as:

"Functional safety is the part of the overall safety that depends on a system or equipment operating correctly in response to its inputs."

[IEC10a, Clause 3.1, p.13]

During the design phase, two important steps have to be carried out, the *hazard analysis* and the *risk assessment*. From these two steps in the design phase, two sets of requirements can be derived:

Safety function requirements are identified during the *hazard analysis*, these requirements answer the question *What safety function has to be performed?*

Safety integrity requirements are identified during the *risk assessment*, these requirements answer the question *What degree of certainty is necessary that the safety function is carried out?*.

These two sets of requirements assure that the goal of functional system safety - the correct function of the electronic system - is reached.

Part 1: General requirements [IEC10b]

In order to make development of a safe system even possible, some general pre-requisits have to be satisfied. Those are mainly of organisational nature and include general problems such as competence of personel, configuration management and qualtiy assurance.

Furthermore the objectives that have to be met by the system development lifecycle are contained in this part, that means, tha this part is concerned with all parts of the system including mechanic, pneumatic, ... parts. The goal is to analyze the system as a whole, identify the hazards imposed by the system, assign a system integrity level and find mitigations at the system level to mitigate the identified system level hazards.

Part 2: Requirements for E/E/PE safety-related systems [IEC10c] While part1 looked at the overall system, part2 establishes the objectives an E/E/PE (electric/electronic/programmable electronic) (sub-)system has to fulfill. Those objectives are really specialized due to the nature of problems arising from the use of E/E/PE (sub-)systems and also due to the huge increase in complexity introduced by those systems. As an example think of a simple analog PID controller compared to a modern computer hosting a controller application.

Part 3: Software Requirements [IEC10d] The above example already mentions the controller application running on a modern computer based control unit, indicating that a programmable electronic device is nothing without the software running on it. Again the nature of software - its flexibility on the one and its error prone nature on the other hand - is very different and the objectives that are especially important for software thus are handled in its own part.

Part 4: Definitions and Abbreviations [IEC10e] A full list of definitions and abbreviations is found in this part. Although the terms used in standards are very often the same, their exact meanings differ very often which makes it necessary to have such a dictionary in every single standard.

Part 5: Examples of methods for the determination of safety integrity levels [IEC10f] One of the first and probably hardest problems in developing a safety critical application is to determine the hazard the application imposes and eliminate hazards where possible. Then the safety integrity level is assigned according to the criticality of the remaining hazards. Mitigations to those hazards are found, to assure that the probability of an accident is *reasonable low* provided it is below the intolerable threshold. This is called the ALARP (*as low as reasonable practicable*) principle.

Part 6: Guidelines on the application of IEC 61508-2 and IEC 61508-3 [IEC10g] (Safety) Standards are usually written in a really abstract and slightly cryptic manner that has to be interpreted. In order to help with the interpretation, part6 gives some guidance on how objectives should be understood.

Part 7: Overview of techniques and measures [IEC10h] parts 1-3 list a number of methods that are considered state of the art and that are suggested to be used for development. Part 7 gives short introductions as well as a list of pointers to more elaborate literature of those methods. As the title says, this is just an overview to get an idea of how a method works. In order to use it a lot more additional literature and experience is needed. Part 7 is mapped to parts 2 and 3 via tables (Annex A and B) that allow to select the adequate methods for the problem at hand.

3.6 ISO 26262 *Road vehicles – Functional safety*

Although the automotive industry produces systems that are highly safety critical, complex (highly coupled with dependent subsystems) systems there was no mandatory safety certifi-

cation for road vehicles. Therefore no industry standard was available for a very long time [2] although some of the most critical systems built on this planet are road vehicles. When engineers talk about risks, failures and failure modes they often state that aeroplanes are of higher criticality than road vehicles because they do not have a safe state. But on the contrary a crash of an airplanes subsystem leaves usually (except for take-off and landing) [3] enough time to just restart the system. A restart that takes several seconds is no big deal for an airplane, if a critical subsystem (breaks, engine control, ...) of a car is down for several seconds, an accident is almost inevitable.

This all changed with the release of ISO 26262 [STA11a] in fall 2011. Now finally the automotive industry has to comply to the standardized safety requirements set by ISO 26262 and has to proof it the safety of its systems to the authorities.

Being a derivative of IEC 61508 the aim it takes is very similar to that of IEC 61508, summarized in the last section. Therefore just a short list of the most important things that ISO 26262 brings into the automotive industry is given:

- A full development life-cycle at system, hardware and software level, that helps to reduce the number of systematic faults by forcing a rigorous system of verification and validation activities on the application designers and developers.
- It provides the functional safety basis for modular and compositional systems which have traditionally not been in use in the automotive industry.
- A sector specific risk classification and risk mitigation categorization is introduced - the *Automotive Safety Integrity Level (ASIL)*. ISO 26262 takes the strong role that the operator (driver) plays in the system into account. Notably, IEC 61508 explicitly de-scopes human factos and humans as part of the system being considered, this shift of focus was needed to fit the application sector.
- Gives guidance on objectives to meet the safety requirements in order to make the systems resdiual risk as low as reasonable practicable (ALARP).
- Gives guidance of objectives for design, developement, verification and validation of modern automotive applications.

The goal of applying all those methods, is to assure that the residual risk in the vehicle is reduced to a minimum and maximzing the safety of the users. More details and clauses important

[2] Some might argue, that there is no need to make vehicles safer, as the majority of accidents are caused by the drivers anyway - the author will not go into this discussion, although this might have been true for the first ABS systems, but the author cannot agree for modern road vehicles that contain assisted parking, comfort braking assistents or other *"safety features"* that change the vehicles behaviour depending on sensor readings.

[3] Although there is no hard evidence as studies comparing those two areas, this is a safe statement as airplanes can - depending on the altitude and their aerodynamic properties - travel for quite a long distance, even if all engines are lost. This distance is defined by the Lift/Drag ratio (see [KM07, p.123] for details). In example: a plane with a Lift/Drag ration of 15:1 can travel 15 000m in distance for each 1 000m of altitude.

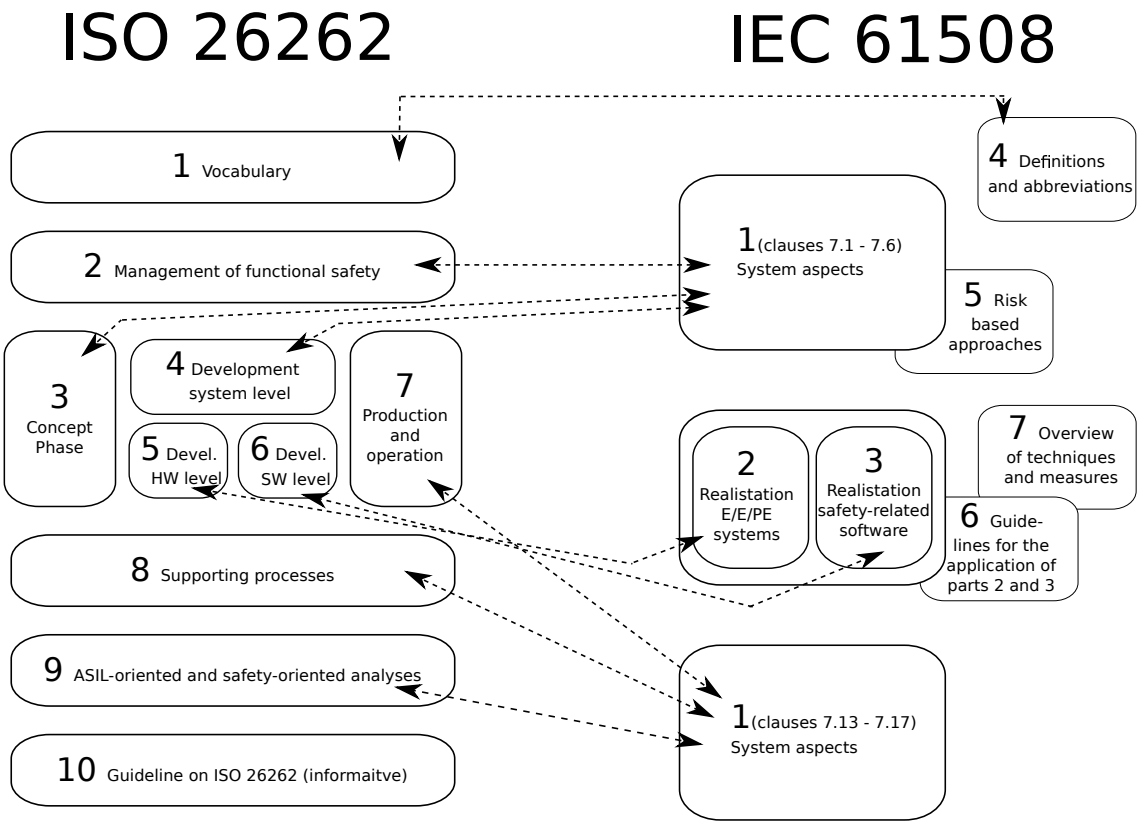

Figure 3.2: Overview of how ISO 26262 and IEC 61508 normative parts map to each other.

to this thesis will be introduced later in chapter 4. In order to show the relationship between IEC 61508 and ISO 26262, there is a mapping between the normative parts of those two standards in figure 3.2, where the parts related to each other are connected with dashed lines.

Concluding it should be noted that the importance of ISO 26262 for the industry was definitely triggered by the ever increading complexity of automotive systems. Over recent decades more and more electronic subsystems are added to road vehicles essentially making it necessary to *"running 100 million lines of code to get a premium car out of the driveway"* (see [1]). New approaches and techniques are needed to being able to handle this increase in complexity. The goal of this thesis is to look into some of the techniques that are suitable to reach this goal (see chapters 4 to 7).

3.7 EN 50128

The EN 50128 (*Railway applications - Communications, signalling and processing systems - Software for railway control and protection systems* is a standard on functional safety in the

railway domain issued by CENELEC, the *European Committee for Electrotechnical Standardization*.

Although the standard itself is not relevant to the thesis itself, it is included, since it is the very first safety standard that defines open-source software as COTS software, and gives directive on how open-source software shall be treated in safety-critical applications. Therefore the open-source components used in this thesis could in principle be used for safety-critical applications[4]. The exact clause refered to so far is:

"pre-existing software
developed software not compliant with this European Standard, e.g.
– COTS (commercial off-the-shelf) and open-source software,
– software previously developed but not in accordance with this European Standard
– software that was developed according to a previous version of this European Standard"

[WGA08, Clause 3.18]

3.8 MISRA-C

The idea of avoiding undefined behavior of the C programming language by using only a subset of the programming language during development, has been around for quite some time [Hat94]. The automotive industry managed to standardize such a subset that fits the needs of the industry. This standard is published by the *Motor Industry Software Reliability Association (MISRA)* and therefore is called *MISRA-C* [MIS04]. Software that is MISRA-C compliant, can be safer than non-MISRA-C compliant software, since *undefined behavior* (see section 2.5.1) is avoided by following a set of rules defined by the standard. In case a rule cannot be followed, this has to be documented and justified in the code documentation. To prove MISRA-C compliance, several checker are commercially available. The rules themselves are mostly best practice anyway. Some of them seem strange at first, and it is not that easy to understand their meaning, unless you understand undefined behavior in the C programming language well.

Knowledge of a wide range of standards is a powerful tool in safety engineering. Knowing the standards allows one to choose the right standard(s) and use the guidelines given by the standards to ones advantage. Standards– including safety standards – contain the domain knowledge collected over decades by the experts that wrote that standard. Even if some constructs seem odd at a first glance, thinking about them usually reveals their intention and helps to deepen ones own safety knowledge

[4] Of course only, if their suitability is shown.

CHAPTER 4

Safety Case

" A safety case communicates a clear, comprehensive and defensible argument (supported by evidence) that a system is acceptably safe to operate in a particular context."

[STA11b, clause 9.1]

Although this definition already gives a good explanation on the purpose of a safety case, Section 4.1 gives a more detailed introduction to safety cases, necessary to understand why a safety case is used in this thesis to analyze the OVERSEE platform. This introduction is followed by the proposal of safety case layering in Section 4.2 and a high level safety case for the OVERSEE platform in Section 4.3.

4.1 Introduction to Safety Cases

Although the above cited clause in ISO 26262 defines the purpose of a safety case and is followed by the suggestion (later in the same clause) of using a graphical argument notation such as GSN [OCYL11], the standard does not mention one of the big problems of how safety cases are constructed today. Unfortunately, safety cases very often tend to be badly structured or even unstructured monoliths, that are seen to be unique for each individual system and have to be completely redone for each (re)certification. But already today and even more in the future, a well structured safety case is required, as

- Safety-critical and safety-relevant systems are becoming more and more complex - both hardware and software are gaining more and more features and/or use COTS components that were not explicitly constructed for the safety domain.
- Security is a topic relevant in many safety critical systems in operation today, as everything is connected to the internet.

- FLOSS software is used for some of the building blocks of the system.

- Applications are ported to other OSs or hardware platforms without touching the actual application code.

As these points are important to understand why the high level safety case for the OVERSEE [3] platform (4.3.2) was constructed this way, the problems emerging from them are elaborated.

Due to the ever increasing complexity in safety critical systems[1] , it will be of outmost importance for the future, to find new strategies to handle this complexity. While there are strategies at the design level, the construction of safety cases is still done in a monolithic way, demanding for a review of the whole safety case even if only small changes are made. Also, by definition, a more complex concurrent software has more bugs in the code base to be found than a comparatively simple single threaded software application, thus the chance that a recertification due to bug-fixes will be needed increases with the complexity of the system as well. Another property of modern safety critical systems that increases the probability of fixes and recertification is the need for security. While an unknown security related bug can be considered as very improbable to be exploited (e.g. just as a safety related bug that has not been found during testing, static analysis, code reviews), the probability of the bug being exploited rises abruptly as soon as the bug gets known (in comparison to a purely safety related bug whose probability of striking stays the same after being found). Although this elevation of probability is not very easy to grasp at first, it becomes obvious if you think about an example. Assume that a safety critical application, which is employed in an isolated environment (read as: does not have any security requirements at all) and has been in operation for one year uses some FLOSS library. After the first year of operation a bug in this library is found and made known publicly. Although you might want to apply the patch to fix the bug, nothing has actually changed, and the probability of the bug getting you into trouble stays the same. Now assume the very same system is connected to the internet and the bug is a security bug. As long as it was unknown, the probability of it being exploited was very small - someone not involved in the development of the library would have to find it before the developers of the library did - but as soon as the issue becomes public knowledge, the probability of someone exploiting the bug becomes very high very fast. And of course if someone exploits this security related bug, the safety of the system is at stake as well.

Furthermore FLOSS is by its nature more dynamic than proprietary software - at least formally this is the claim - so any safety strategy must take into consideration the effects of this dynamics from the very outset or it will result in no more than an unmaintainable monolithic safety case. FLOSS is not suitable for monolithic safety cases - and IEC 61508 contrary to common claims, is sufficiently flexible to accommodate for the needs of utilizing FLOSS.

The next section of this chapter suggest a layered approach on constructing a safety case, with the intend to render handling the complexity possible, as well as increasing the maintainability. After that an example of such a layered safety case is given on the example of a partitioned system based on the XtratuM hypervisor, namely the OVERSEE [3] platform.

[1] Especially the automotive industry - see [1]

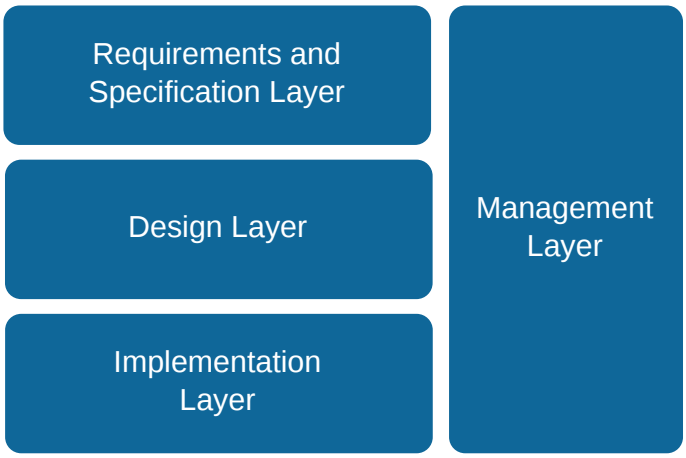

Figure 4.1: Proposed Layering for the Safety Case

4.2 Implementing a Layered Safety Case

Structuring a safety case is nothing new, it has been done for a long time. One of the ways used is a layered safety case as introduced in [**?**]. To the contrary of the layering proposed in the following, Bishop and Bloomfield are talking about one big safety case that is organized hierarchically. In the following the term *layered safety case* is more partitioning the safety case into different levels (layers) of abstraction, each of which is then argumented in a hierarchical way. The layering of the safety case proposed in this thesis is depicted in Figure 4.1. The layers this approach includes are the following:

Requirements and Specification Layer At this layer, the safety case investigates the requirements and the specification as introduced by the used standards. In this case this could be ISO/IEC17356 (OSEK/VDX), POSIX, SuSv3,...

Design Layer At this layer the design and its conformance with the standards introduced in the requirements and specification layer are investigated.

- If only subsets of chosen standards or specifications are used, these subsets have to be well documented and rationalized.
- Requirements for the system are derived from (a subset of) the chosen standards.
- Selection of an appropriate design methodology - an accepted design method (e.g. RTSAD) is chosen

Implementation Layer At this layer, the actual implementation is evaluated. Strategies to argue the compliance with standards and specifications can be found in the GSN diagram of this layer. The goal of this layer is to clarify two major components of the overall approach, first it has to describe the conformance evaluation of the systems components and secondly it shall contain a well specified integration process.

Management Layer One more layer has to be added, to assure the consistency throughout the development. This *management layer* contains all the organisational kind of things that hold the other three layers together, like in example the definition of the development life-cycle or a strategy on how to maintain the traceability over all documents and source code.

As already mentioned, for each of those four layers, a GSN (Goal Structured Notation) diagram and a data dictionary are used to document the decisions in a structured and maintainable way. To produce those GSNs a special tool for safety case development with GSN was used: D-Case [37].

The big advantage of layering the safety case in this manner, is that it is not necessary to touch the higher layers if the implementation changes. As a matter of fact it will not even be necessary to touch the *requirements and specification layer*, until e.g. a new version ISO/IEC 17356 is used. This is especially important for being able to keep up with highly dynamically developed software.

One might argue, that this approach is not desirable, since the SIL level can only be constituted for the whole system, and since the application is unknown this is not possible here. But if you turn your argumentation around and say that *"as long as the application sticks to (this subset of) the API provided by the OSEK RTE, SIL X can be reached"* it is possible to benefit from this layering.

The safety case will be represented using GSN [OCYL11, KM98, Ye05] an introduction to GSN can be found in section 2.6. While some resources on GSN give a more detailed description of the entities using a table, most of them do not give any guidelines on that at all. In order to get a complete methodology *data dictionaries* as described and used in structured design [DeM81, Goo01] are used.

The last thing that has to be established before diving into the example, is the enumeration scheme used to create unique identifiers for the items in the GSN. The identifiers consist of 3 parts, the first part identifies the kind of GSN item this is (e.g. **G** for goal, **S** for solution, **ST** for strategy). The second part is the layer this item belongs to (e.g. **RS** for requirements and specification layer, **D** for design layer, **I** for implementation layer and **M** for the management layer) and the third part is a consecutive number for all items where the first two parts are the same.

4.3 High-Level Safety Case

Since developing a complete and rigorous safety case is a very time consuming task, it is clearly out of scope for this thesis. Thus only the initial iterations of a safety case for the proposed design is are presented, with a focus on the most critical part - the high-level safety case. The safety case is developed as described above, and uses the GSN as described in section 2.6 and amends GSN by data dictionaries similar to the ones used in RTSA.

4.3.1 Requirements and Specification Layer

The goal of the requirements and specification layer is to make sure, that the chosen standards and guidelines are appropriate for this kind of system. Basically the goal here is to justify why the chosen standards and specifications are relevant, and to make sure that these chosen standards and specifications do not contradict each other. The GSN of the *requirements and specification layer* is shown in Figure 4.2, the companioning data dictionary is in the remainder of this section.

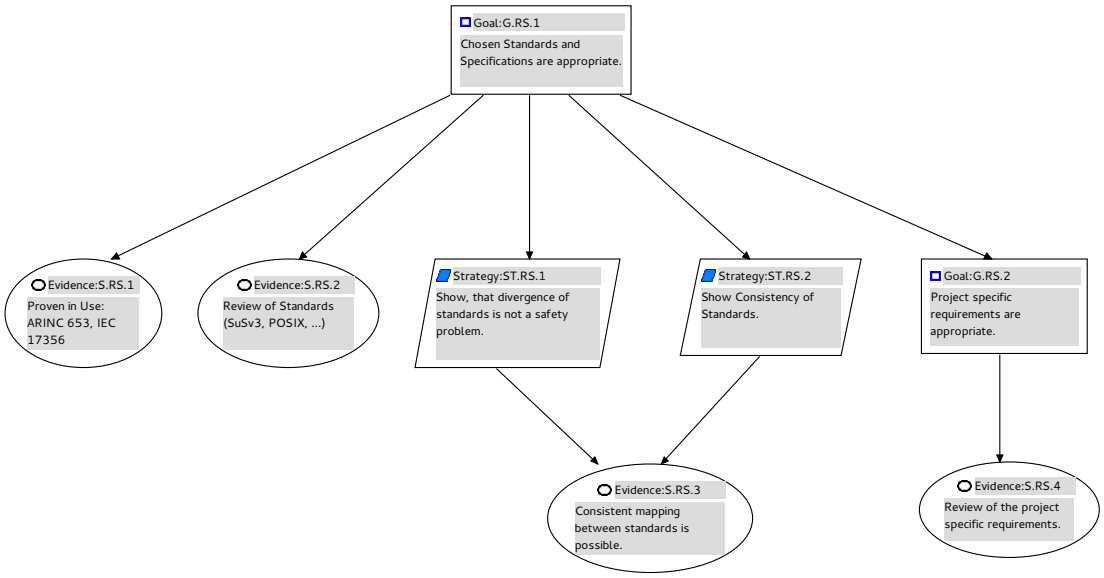

Figure 4.2: GSN of the Requirements and Specification Layer

G.RS.1 The overall goal of this layer, is to show that the chosen standards and specifications are appropriate. For those that are in wide use in the community, this will be very easy, for others a little more work has to be done to assure that they are in fact suitable.

S.RS.1 Those standards and specifications that are in wide use in the industry, or other safety-relevant industries can potentially be declared as *proven in use*. The necessary standards for the approach taken in this thesis are listed in the following with a short rational why they are considered.

- ARINC 653 originates from the avionics industry. As shown in Section 3.4 it can be cleanly mapped to OSEK/VDX. The advantage of ARINC 653 over OSEK/VDX is that it includes hard requirements on the independence of partitions (applications), and explicitly allows an integrated approach. On the functional level, OSEK/VDX is a subset of ARINC 653, so using ARINC 653 instead of OSEK/VDX makes sense, as it is suitable for automotive as well (since it is a superset of OSEK/VDX), it guarantees us the independence of applications that is needed.

- IEC 17356 is the IEC standardized version of OSEK/VDX (see 3.3.3), and the most used OS specification in the automotive industry at the moment.
- IEC 61508 is the generic standard for functional safety. Since the derived standard for the automotive industry (IEC 26262) is a bit vague in some parts, back referencing to the original standard will be necessary.
- IEC 26262 [STA11a] is the new standard for functional safety for the automotive industry, which has been released in November 2011. Wide endorsement of this standard in the automotive industry is expected within the next years, thus this is a key standard to consider if the concepts should be suitable for the automotive industry.

S.RS.2 Standards and specifications that are not in wide use in the automotive industry, or any other safety-relevant industry, need to be reviewed. If parts are not needed or not suitable for an automotive application, these restrictions have to be documented and the chosen subset justified.

ST.RS.1 Although this is not expected, it might happen that at some point there will be a slight divergence from the standards in use. These divergences have to be documented and it has to be shown that these divergences do not pose a safety related problem.

ST.RS.2 The Consistency of the chosen standards is important, in order to guarantee that no ambiguous interpretations and mixtures of standards can happen.

S.RS.3 One way to meet ST.RS.1 and ST.RS.2 is by establishing a mapping between the standards in question. An example for ARINC 653 and OSEK OS is done in the mapping in section 3.4. Although a 1:1 mapping is not possible, it can be shown that the both are compliant on large parts, and that OSEK/VDX is a subset of ARINC653. Basically, imposing additional requirements on OSEK (e.g. *no shared address spaces of tasks*) could result in an 1:1 mapping.

G.RS.2 Applying the right standards and specifications, only makes sense if the requirements are appropriate, so it has to be assured the requirements are correct and complete.

S.RS.4 The requirements have to be validated by review, deficiencies have to be documented and fixed (iterate if necessary!).

4.3.2 Design Layer

In this layer, the chosen standards and specifications from the *requirements and specification layer* have to be investigated further. While the *requirements and specification layer* has already justified why the chosen standards and specifications are relevant and appropriate, the *design layer* has to define which subsets of the chosen standards are relevant and will therefore be used.

The GSN of the *design layer* is shown in Figure 4.3, the companioning data dictionary is in the remainder of this section.

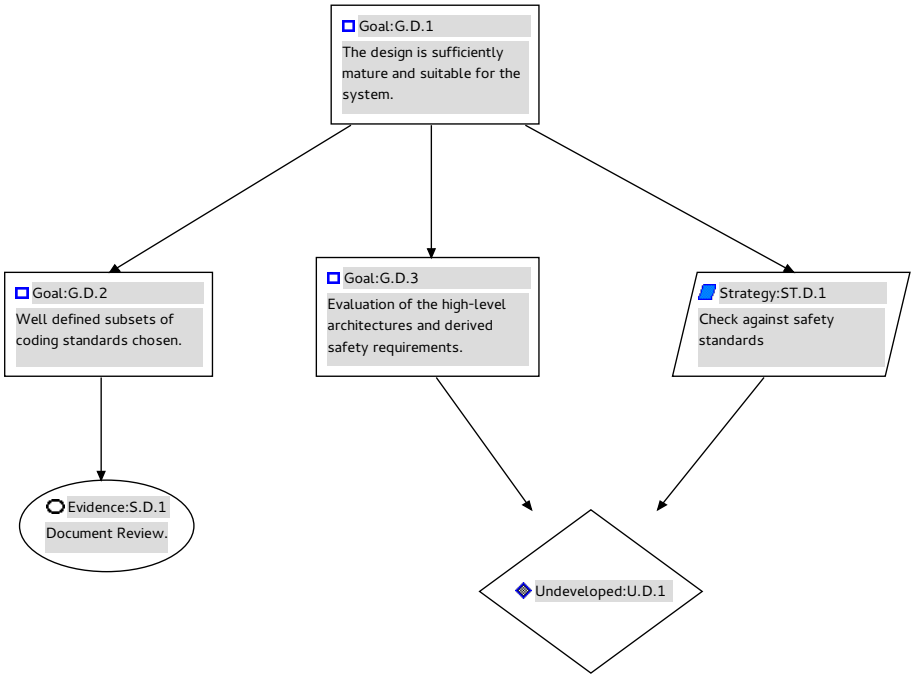

Figure 4.3: GSN of the Design Layer

G.D.1 The overall goal of the *design layer*, is to show that the design is suitable and sufficiently mature and compliant with relevant standards. In this layer, the standards and specifications chosen in the *requirements and specification layer* are further investigated and used to derive the requirements that must be met to reach the necessary safety integrity level.

G.D.2 The first step is to choose an appropriate subset of the chosen standards and specifications.

S.D.1 The used subsets of chosen standards and specifications shall be documented and rationalized.

G.D.3 The approach to evaluate the appropriateness of the design process is given in [IEC10d, clause 7.4] .

- The software architecture has to fulfill the requirements for software safety to meet the anticipated SIL level.
- The requirements imposed on the software by the hardware have to be evaluated and reviewed.
- The toolchain used for the project has to be suitable (appropriate programming language, compiler, linker as well as appropriate tools for verification, validation, assessment and modification).

- It has to be assured that the resulting software is analysable and verifiable.

ST.D.1 Safety standards give some guidance on the design, document the compliance with the standards guidelines or argue why deviations are necessary/permissible and show that all the objectives are met (see [IEC10c, 7.4] and [IEC10c, 7.4]).

U.D.1 Note, that G.D.3 as well as ST.D.1 have been left undeveloped due to time reasons but that is fine as requirements development is an inherently iterative process - the documented state of requirements development is thus just the first few iterations. Essential though is to observe that these intermediate states can, and should, be validated and verified allowing early detection of potential issues.

4.3.3 Implementation Layer

The implementation layer contains the part of the safety case, that is concerned with assuring, that the actual implementation is compliant with the requirements, the specification and the design of the system. The GSN of the implementation layer is depicted in Figure 4.4 and has three parts, with the intention to show that

- the independence between partitions (applications) is guaranteed,
- the hypervisor is safe,
- and the OSEK runtime environment for the applications is safe.

Due to its size this GSN was split into two parts depicted in Figures 4.4 and 4.5, were Figure 4.4 is the starting point, and Figure 4.5 is the extension of 4.4, visualized by the overlap of goal G.I.6.

G.I.1 The overall goal is to argue, that the implementation is able to allow several independent applications (possibly with different safety integrity) to run on the designed platform.

J.I.1 The pre-requisite for the safe execution of an safety-relevant application on top of this architecture, is, to be able to guarantee that the environment of this application is safe. To show that the environment is in fact safe, the overall argument is broken into three part parts, that show that:

- the hypervisor itself is implemented in a safe manner,
- the independence of the partitions is given, in order to contain errors in those partitions and prevent them from influencing independent partitions,
- the OSEK runtime environment (RTE) inside of the partition is safe.

These three parts cover all parts of the system's software, except for the application itself.

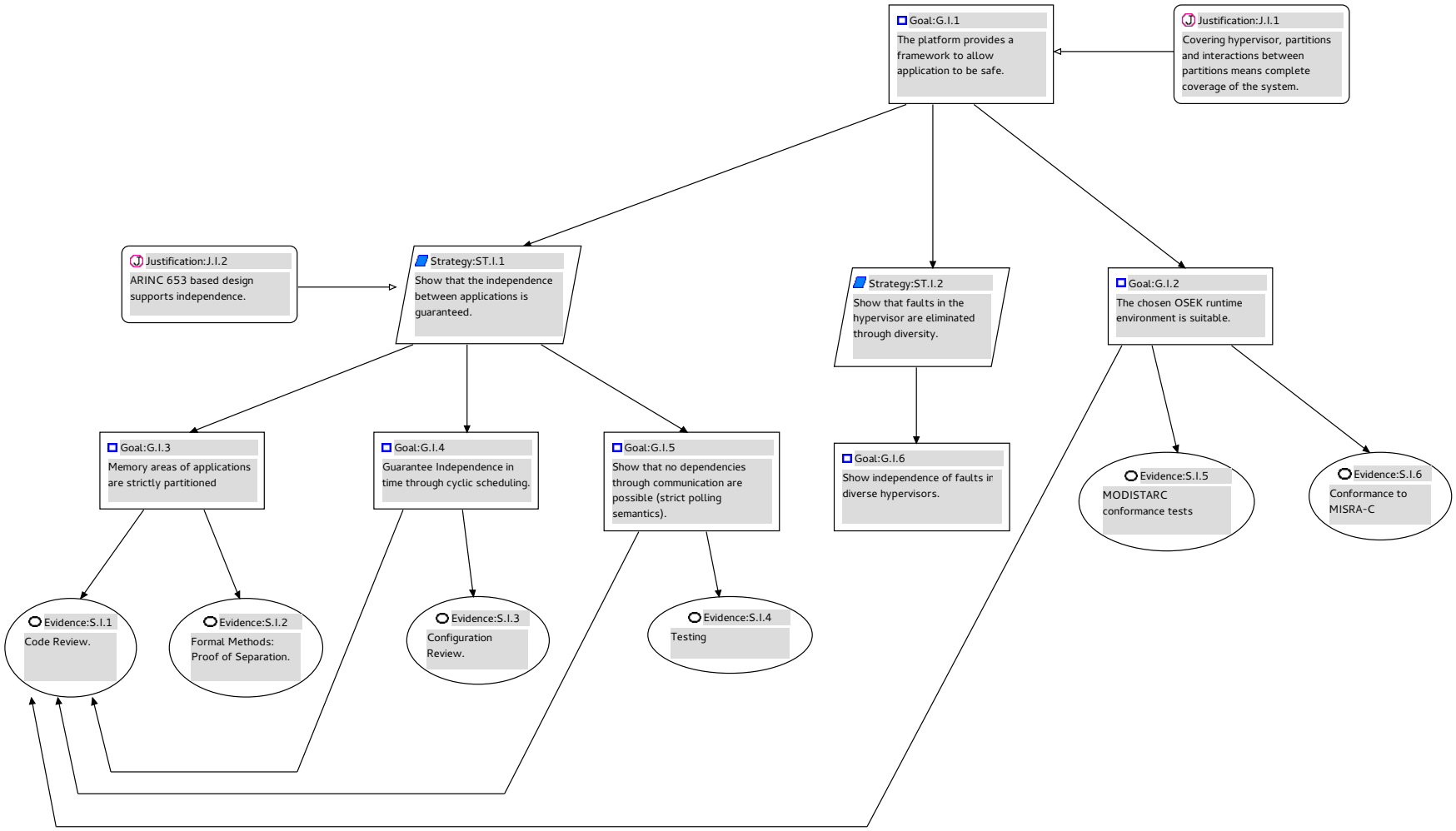

Figure 4.4: GSN of the Implementation Layer - Part 1/2

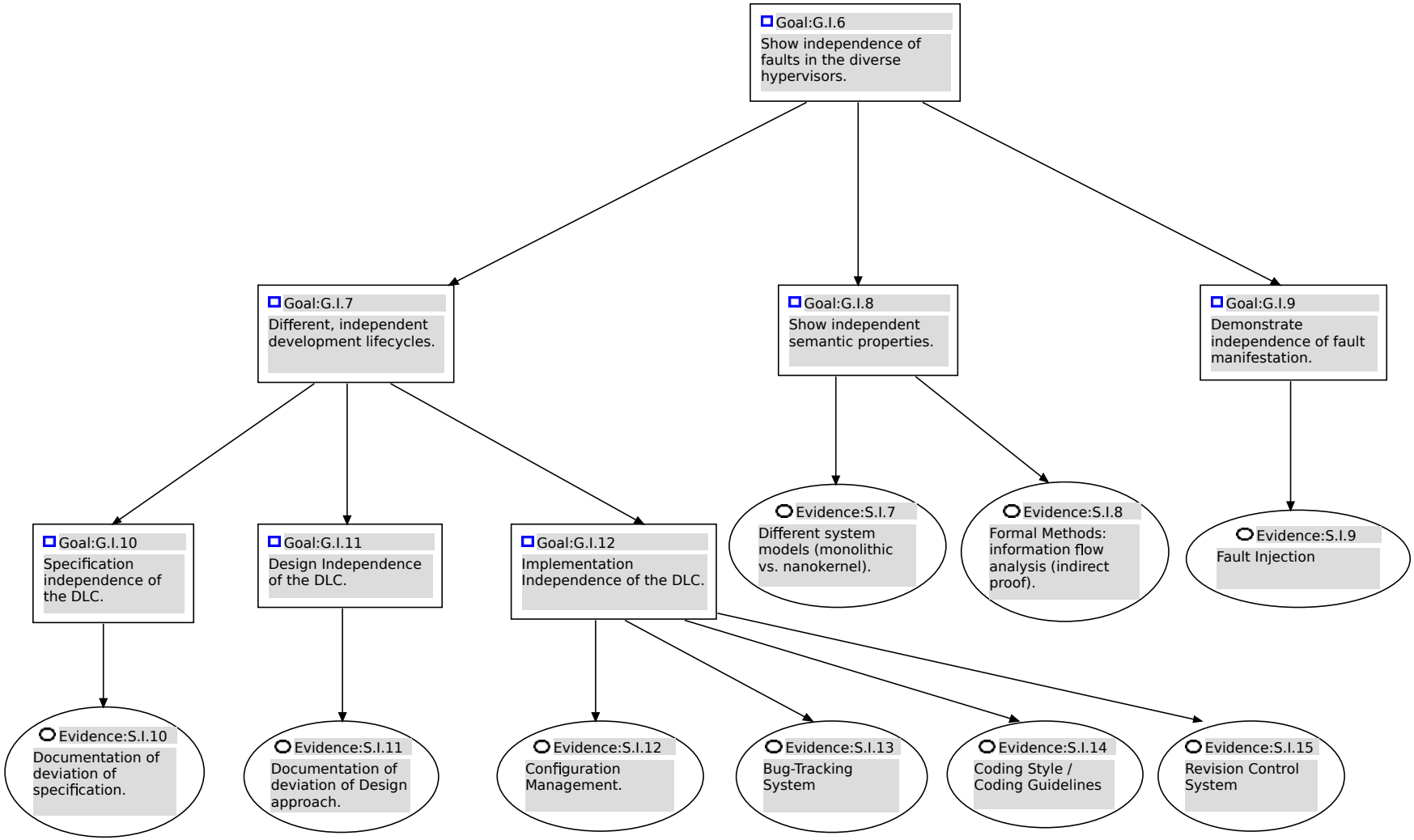

Figure 4.5: GSN of the Implementation Layer - Part 2/2

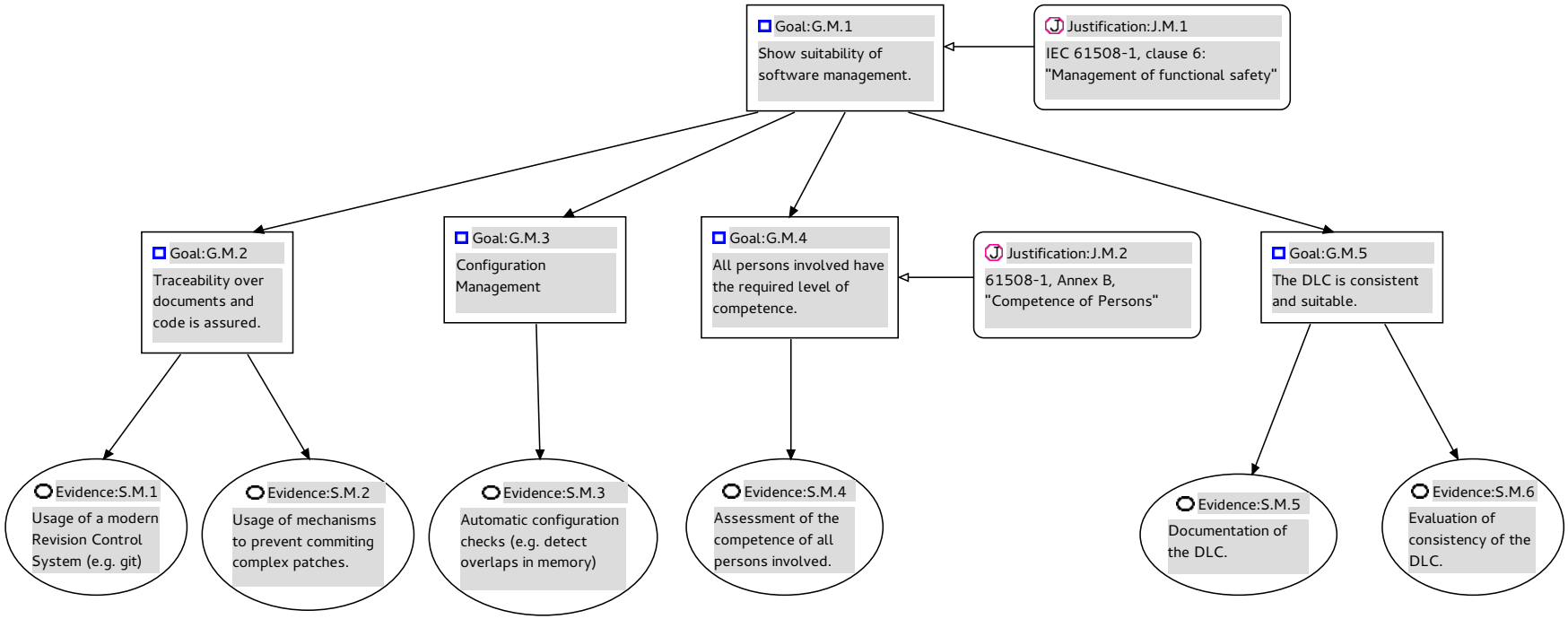

Figure 4.6: GSN of the Management Layer

ST.I.1 To allow the execution of several applications of different criticality, it is vital to assure the independence of the partitions. To show that the partitions are independent, three properties have to be proven:

- spatial partitioning (independence in memory - G.I.3)
- time partitioning (independence in time - G.I.4)
- and that no dependencies can result from the communication amongst partitions (independence in communication - G.I.5)

If all three of these properties can be shown, the hypervisor guarantees that the partitions can run completely independent of each other.
For the proof of independence, Rushby's [Rus81] proof of separability will be used.

J.I.2 Showing that the partitions (applications) are independent is explicitly stated in IEC 61508-3 [IEC10d]:
*"***7.4.2.7** *Where the software is to implement both safety and non-safety functions, then all of the software shall be treated as safety-related, unless adequate independence between the functions can be demonstrated in the design.*
7.4.2.8 *Where the software is to implement safety functions of different safety integrity levels, then all of the software shall be treated as belonging to the highest safety integrity level, unless adequate independence between the safety functions of the different safety integrity levels can be shown in the design. The justification for independence shall be documented."*

[IEC10d, Clauses 7.4.2.7 and 7.4.2.8]

These two clauses show us, that this approach - if done correctly - is also OK from the functional safety side.

G.I.3 As stated in ST.I.1, the first property that has to be proven, is the independence of the partitions in memory. [2]

S.I.1 Code reviews to assure that appropriate memory protection mechanisms are in place shall be conducted.

S.I.2 To assure the independence in memory, formal methods using the rules defined in Rushby's proof separability [Rus81] shall be used.

G.I.4 The second important property, that has to be shown is the independence in time. This independence in time is achieved by using a statically configured, cyclic scheduler. The configuration is done in an XML configuration file which is read at compile time. *NOTE: Shared devices were an important topic here. If devices are shared between different partitions, they need to be handelted at the highest SIL of all interacting partitions.*

[2] *NOTE: This will also require hardware analysis, which is descoped for this thesis.*

S.I.1 Code reviews of the configuration tools and the scheduler itself shall be done.

S.I.3 Automatic checks of the XML configuration file shall be done at compile time, in order to assure the consistency and validity of the configuration.

G.I.5 Apart from partitioning in time and memory, it has to be guaranteed, that faulty partitions cannot influence non-faulty partitions. To reach this goal, only pre-defined (XML-configuration file) communication channels are available for interpartition communication. Furthermore, these channels follow strict polling semantics. This means, that a faulty partition (e.g. babbling idiot) cannot influence a non-faulty partition, since the non-faulty partition reads messages only on its own will. You can find a more detailed description of these communication mechanisms in 3.2.

S.I.1 Code reviews shall be conducted, to assure that the strict polling semantics are adhered to.

S.I.4 Rigorous testing is sufficient to complete the argument of independence at the communication level. Due to the simplicity of the communication semantics, exhaustive testing of the communication channels polling methods seems feasible.

ST.I.2 Depending on the targeted safety integrity level, a diverse approach on hypervisor level can be used to increase the safety integrity (see [IEC10g, E.3] for an example on diversely implemented software).

G.I.6 The goal is to argue, that a fault is sufficiently unlikely to manifest itself in diverse hypervisor's in the same behavior.

G.I.7 Differing development life-cycles are the first indication of the diversity amongst the chosen hypervisor's.

G.I.10 If the hypervisor's follow a different specification, this is already the first step towards diversity.

S.I.10 Documentation of the different specifications (e.g. ARINC 653, SuSv3, AUTOSAR) used as basis for the hypervisor, and the relevant deviations between those specifications.

G.I.11 Diversity at the design level highly increases the level of diversity between hypervisors.

S.I.11 The different design approaches shall be documented.

G.I.12 A different approach at the implementation level is taken.

S.I.12 Document how configuration management is handled differently.

S.I.13 Document the use of different bug-tracking systems.

S.I.14 Document whether the hypervisor's follow different coding styles/guidelines.

S.I.15 Document which RCS (revision control systems) are used and in which way the flow of code into the main repository defers.

G.I.8 The independence of the semantic properties of the hypervisor's in question shall be shown.

NOTE: This excludes the inter-partition communication system, as the strict polling semantics of this mechanism is a key element to reach independence between partitions.

S.I.7 Using hypervisor's with a different system model decreases the probability of faults manifest themselves in the same way. So in example, employing a two out of two system where one hypervisor is based on nanokernel architecture, while the second one uses a monolithic approach is already a big step to introduce a high degree of diversity. A classification of operating systems and available system models can be found in section 2.1.

S.I.8 To show the impact of different system models, a indirect proof based on an information flow analysis [DD77] shall be done.

G.I.9 The divergence in fault manifestation shall be demonstrated.

S.I.9 Fault injection is used to demonstrate die divergence in fault manifestation.

G.I.2 The OSEK RTE shall be suitable. The solutions to reach this goal are:

S.I.5 The MODISTARC conformance tests, to make sure that the implementation is functionally compliant to the OSEK standard.

S.I.1 A code review to assure the quality and correctness of the code is possible, since the code base for the OSEK RTE is relatively small.

S.I.6 A checker can be used to prove the compliance to the MISRA-C coding guidelines.

4.3.4 Management Layer

Now that the three layers that contain all the technical justifications are defined, a fourth layer that holds them together and contains all the management stuff needed in the other three layers is established. The GSN of the management layer is shown in Figure 4.6, the companioning data dictionary is in the remainder of this section.

G.M.1 The overall goal of the management layer, is to assure that the development is managed in a well defined way, and that all the requirements for a successful certification process are met.

J.M.1 The management of functional safety is defined in [IEC10b, Clause 6]. The important clauses are:

- *6.1.1* and in more detail *6.2.1* state that a well defined development life-cycle has to be defined.
- *6.1.2* states that the responsibilities of the involved persons has to be well documented.

- *6.2.1 o)* gives some guidance on configuration management.
- *6.2.2* requires us to rigorously document all actions.

G.M.2 In order to assure that all actions taken during the development life-cycle, as required in [IEC10b, Clause 6.2.2], the traceability over all documents and source code as to be assured.

S.M.1 To reach *G.M.2*, it is vital to use an modern RCS (revision control system), in order to get a full history over the complete development life-cycle. In this thesis the tool chosen for this task is `git` [33].
A document to describe the integration of the RCS system into the development life-cycle shall be done, this document needs to specify formal things like naming conventions for tags and the like.

S.M.2 In order to extend the capabilities of `git`, hooks to assure things like:

- compliance to coding style or guidelines
- no empty commit messages
- no submission of giant patches

All these can be used to enforce permanent quality assurance measures. A document listing and documenting the used hooks shall be provided. Changes to the hooks have to be logged into this document.

G.M.3 As stated in [IEC10b, Clause 6.2.1 o)], the procedures for configuration management have to be well specified.

S.M.3 Since the approach taken by ARINC 653 is a static configuration, automatic checks on the correctness of the XML configuration file can be implemented. Such automatic checks can in example be used to verify, that there are no overlapping memory areas or that the a priori defined cyclic schedule is feasible. But also other basic properties like the validity of the static, cyclic schedule, duplicated interpartition communication channels or channels with only one partition attached can be checked easily.

G.M.4 To assure the quality of the resulting system, the competence of the persons involved has to be appropriate.

J.M.2 A list of skills that shall be assessed is given in [IEC10b, Annex B].

S.M.4 An assessment of the competence of persons shall be done. The requirements of skills and the depth of the assessment heavily depends on the targeted SIL level.

G.M.5 The basis for a successful development of every safety-relevant system lies in the definition of and compliance with an appropriate development life-cycle. It has to be assured, that the employed DLC is appropriate for a safety relevant system.

S.M.5 The first thing to do, to even being able to show the consistency of the DLC is to fully specify and document it.

S.M.6 The specification of the DLC from *S.M.4* shall now be used to evaluate the consistency of the DLC.

This section has shown, that it is indeed possible to argue the safety of a platform with a virtualized environment, and thus that the OVERSEE platform can be made safe, although some adjustments might be necessary (e.g. dual channel diverse hardware) to reach higher safety integrity levels.

CHAPTER 5

Implementation Details

In the following the efforts that have to be taken to create an OSEK/VDX compliant runtime environment in a XtratuM partition are described.

First an assessment (Section 5.1) of existing FLOSS implementations is done. The rest of the chapter (Sections 5.2 to 5.5) can be seen as a step-by-step guide of things to do when porting the chosen implementaion (FreeOSEK) to a new (hardware) platform, as the same steps would be necessary to do a port to another hardware platform - except that very often instead of low-level assembly language a XtratuM hypercall is sufficient. On the other hand, this also gives an insight on the problems and efforts for everyone who wants to port a new runtime environment to the XtratuM hypervisor or another para-virtualized environment with a similar hypercall API.

The following sections also include a detailed description of which steps already have been achieved successfully, and gives an insight into the parts that will need more work. To anticipate the most important thing first: As of this writing, FreeOSEK can be used as an XtratuM runtime environment, but more work will be needed to make a full compliant version possible, most notably in the task management and communication subsystem some (re)work will be necessary.

5.1 Assessment

This section is going to document the assessment that was done to chose the components for this thesis. Due to the connection to OVERSEE project [3] some of the components (e.g. XtratuM2) are a given and do not have to be chosen. The only software component that is left still open at this point, is a suitable FLOSS implementation of the OSEK operating system specification.

5.1.1 Selection Criteria

Before a selection can be done, the criteria that are of importance and relevance for the use-cases intended to run in the OSEK/VDX compliant runtime environment have to be established. The selection of an FLOSS licensed OSEK/VDX implementation shall then be based on these

criteria. The following criteria are a mixture of indicators for code OSEK/VDX compliance, usability and quality as those are the things of importance for an OSEK/VDX compliant runtime environment.

Code size: To get a feeling on the size of the code base, David A. Wheeler's tool *SLOCCount* will be used, to count the lines of code in the project. Furthermore, a short estimation on the distribution of the code (OIL parser, OS core, OS communication, arch dependent code,..) will be done. The most important value here will be the architecture dependent code, since it gives an idea on how much work a port to a new platform (namely the XtratuM2 hypervisor) will be.

Code quality: The questions raised in this section, are: *Does the Project follow any coding guidelines or standards? How well are deviations from those standards documented? Are data types defined by OSEK used thoroughly?*

Code Documentation: The consistency of documentation / code will be evaluated in this section.

Level of OSEK/VDX compliance: One important aspect of this assessment will be, whether OSEK/VDX is implemented fully (if parts are missing - which ones?). Furthermore deviations from the standard as well as non-standard extensions and their impact will be evaluated.

Safety considerations: A very important criteria is, whether or not a design choice was made that has an obvious (good or bad) influence on safety. I.e. if there is any property of the design or implementation that makes a certification very hard/easy.

5.1.2 FLOSS OSEK implementations

Before a selection can be done, the OSEK/VDX implementations that are up for debate are analyzed in regard of the criteria established above, so that a decision can be made after wards.

5.1.2.1 Trampoline

Trampoline [1] is mainly developed by people from the Real-Time Systems group of IRCCyN (Jean-Luc Béchennec, Mikaël Briday, Sébastien Faucou and Yvon Trinquet). The project has external contributors: Jean-François Deverge from IRISA, Trame group from ESEO (Jonathan Ilias and Jérôme Delatour) and Greensys. Trampoline is available for a number of hardware architectures (e.g. arm, avr, c166, ppc) and is licensed under a BSD license[2].

Lines of Code: Running sloccount on the top level directory reveals the following numbers on lines of code in the project. The result of shown in Table 5.1

[1] trampoline.rts-software.org

[2] For details see the License file in the subversion repository: https://trampoline.rts-software.org/svn/trunk

Programming Language	Lines of Code (Percent of Total)
ansic	66897 (91.5%)
asm	2430 (3.33%)
python	1498 (2.05%)
java	1004 (1.37%)
sh	549 (0.75%)
cpp	484 (0.66%)
objc	193 (0.26%)

Table 5.1: Total number of LoC of Trampoline grouped by language (dominant language first)

This means, a number of different languages have been used to implement the different parts of trampoline as well as the tools that are packaged with trampoline and used to configure / compile / run it. The numbers that are really interesting here are the numbers for code written in C and assembler. Since several architectures are supported, a detailed table over three architectures has been done (see 5.2 for the results), in order to get the C and assembler code of the architecture dependent part. This helps to find out how much of the code will need some work, and how much is generic code that (hopefully) should not need any attention.

Arch	NR Boards	ansi-c	asm
ARM	5	12392	1204
PPC	3	901	324
166	1	335	0

Table 5.2: LoC of architecture dependent code in trampoline.

Level of OSEK/VDX compliance / Deviations from OSEK/VDX: During the quick review performed on the documentation and code, there were several deviations from OSEK/VDX that were spotted. The sections mentioned in the deviations are sections in the trampoline manual, which can be found in `trunk/documentation/Manual/main.pdf`.

In section 2.5.3 two additional task states “for internal management“ are introduced. There is an “autostart“ state for tasks that shall be run automatically when the OS is (re)started, and a “ready_and_new“ state for tasks that are ready but have not been initialized yet. Both of those additional states are not necessary. The “autostart“ should not be necessary, since there is an autostart flag defined in the oil, allowing to mark processes that should be started automatically at compile time. These processes should then be loaded into ready automatically, and no dedicated states are necessary, and both states are neither mentioned in OSEK/VDX nor in AUTOSAR.

In addition it has to be noted for the “ready_and_new“ state, that tasks that are preempted are rescheduled at the point where they were preempted, so putting initialization at the beginning of the process is just fine, a new state is not required to get this functionality.

Of course there could be reasons for introducing those additional states, but in that case a rational should be provided why this extension of the specification is needed. Such a rational is not available, this is not acceptable. Manual section 7.1 mentions *expiry points of a schedule table*, but OSEK OS [Con05] does not specify anything about expiration of schedule tables or the like in OSEK. Apparently this has been introduced when the developers started to introduce autosar into trampoline. These non-compliance issues introduced by those extensions have to be considered very critical as it implies incompability with existing implementations. Thus these noted, very low-evel deviations from the OSEK/VDX specification are to be considered severe and will make an impact on the final decision.

Code Quality: Trampoline seems to follow the MISRA-C (see Section 3.8 coding guidelines. Although no official statement on that matter was found there are a couple of comments explaining why some MISRA-C rules have been broken. Nevertheless it has to be noted that a full MISRA-C compliance would be expected to be documented in a report (including a deviation matrix).

Code Documentation: Trampoline's documentation is done inline using the doxygen code documentation framework. This includes both, the users manual as well as the developers manual. The developers version contains more detailed information. This doxygen output contains just the code documentation, there are other documents describing different subsystems (interrupts, task state model, api) and there is the trampoline manual has already been used above.

5.1.2.2 FreeOSEK

FreeOSEK [3] [38] is a OSEK implementation started by Mariano Cerdeiro. It currently runs on ARM and on POSIX compliant platforms, so you can test it on your Linux desktop machine. FreeOSEK is licensed under GPLv3 with link exception. This means, that you can link your code into FreeOSEK and can still license your code under whatever license you want (free or proprietary). According to the FreeOSEK homepage, they currently run about 80% of the OSEK conformance tests, and of those about 95% of the tests pass. In addition, FreeOSEK is tested, using the static code checking tool splint.

Lines of Code: As above with trampoline, sloccount is run on the top level directory. The result is shown in Table 5.3.

So this time we got fewer different programming languages and the assembler part is a lot smaller (in comparison to the lines of C code) than in trampoline. The difference is by a factor of 4.8 which means the probability that trampoline includes a lot more architecture dependent code is very high.

[3] opensek.sourceforge.org

Programming Language	Lines of Code (Percent of Total)
ansic	19219 (83.00%)
php	2626 (11.34%)
perl	1135 (4.90%)
asm	145 (0.63%)
ml	31 (0.13%)

Table 5.3: Total number of LoC of FreeOSEK grouped by language (dominant language first)

FreeOSEK/Drv/Adc/inc/arm7	ansic:	26
FreeOSEK/Drv/Adc/src/arm7	ansic:	7
FreeOSEK/Drv/Dio/inc/arm7	ansic:	26
FreeOSEK/Drv/Dio/src/arm7	ansic:	33
FreeOSEK/Drv/Mcu/inc/arm7	ansic:	15
FreeOSEK/Drv/Mcu/src/arm7	ansic:	47
FreeOSEK/Drv/Pwm/inc/arm7	ansic:	22
FreeOSEK/Drv/Pwm/src/arm7	ansic:	63
FreeOSEK/Drv/StartUp/asm/arm7	asm:	71
FreeOSEK/Drv/StartUp/src/arm7	asm:	74
FreeOSEK/Os/inc/arm7	ansic:	285
FreeOSEK/Os/src/arm7	ansic:	89
FreeOSEK/Posix/inc/arm7	ansic:	5
FreeOSEK/TestSuite/inc/arm7	ansic:	8
TOTAL:	asm:	145
	ansic:	626

Table 5.4: Architecture specific code in FreeOSEK

Arch Specific LoC: In FreeOSEK, the arch dependent code is sprinkled all over the tree, in the Table 5.4 you find the numbers for arm7, listed with the according directory. From the directory names, you can see, that this includes really all the arch dependent code, even drivers (Adc, Dio), System Initialization, etc.

Code Quality: The Code Quality of FreeOSEK is comparable to Trampoline, the code seems to stick to the most important MISRA-C rules (e.g. those that are hard to satisfy after wards). Except from those minor MISRA-C deviations the code is very readable and easy to understand.

Code Documentation: The code documentation in FreeOSEK is done using the doxygen inline documentation system. This documentation contains not just the code documentation, but also manuals and HowTos are woven into the doxygen output.

5.1.3 Result and Rational for Choice

Based on the four key properties that were evaluated per choice, this section is making a selection and gives a rational for the actual choice. As a short reminder, the four key properties were:

- semi-formal metrics
- code review (code quality and documentation)
- OSEK conformance
- safety considerations

As the approaches taken by the two projects are very different, the comparison seemed to be very hard at the beginning. After trying to do the first steps of porting for both of them, it became clear very quickly, that one is easily portable to new platforms and the other is not. The Problem of lies in the approach taken how the OIL configuration file is woven into the implementation:

Trampoline uses its own compiler called `goil` to read the configuration file and generate architecture dependent code. That means that all the architecture dependent stuff is actually inside the logic of `goil` and preliminary tests made it clear that getting into `goil` requires a high expertise in compiler programming.

Furthermore code generators are amongst the tools that are really hard to argument for safety related project. Using a tool like this makes certification really hard, as there are no easy ways to prove that `goil` works as specified.

FreeOSEK uses a simple php script to parse the configuration file and generates header files that are used to initialize statically allocated data structures. This is a very simplistic approach but does exactly what it is supposed to do. Furthermore the architecture specific parts of FreeOSEK are very small and easily accessible as they are stored in normal C source and header files instead of a complicated compiler.

This also simplifies the certification process, as the code generated by the configuration generator and the output of the tool can be verified by code review. The compiler used for compiling the binary is gcc which is one of the most used C compilers today and a argument around ïncreased confidence from useßhould be sufficient - at least for projects with low and medium criticality.

According to this assessment FreeOSEK was chosen, basically for the smaller part of architecture dependent code, the fact that FreeOSEK sticks to OSEK/VDX and does deviate (except for mechanisms not implemented yet) from it. Furthermore the approach for the code generation from the OSEK OIL configuration file is a lot more reasonable. It is not only easier to understand and modify on a technical level, but also very easy to justify in an safety argument as the code that is produced by the code generator is just the initialization of some structs, which can easily be justified by a code review of the output.

5.2 Adaptation of the Build System

Following the example of the Linux kernel, the architecture dependent portions of FreeOSEK are located in directories, nicely separated from each other as well as from the generic parts of the code. In the FreeOSEK source tree, the directories containing platform dependent code are easily identifiable, as the directories name is the name of the platform (e.g. `arm9`), so the very first step in porting FreeOSEK to a new architecture, was to create directories for the new platform (`xm`), and the files included in those directories (basically the same filenames as in the directories of the other platforms). Those new files contain declarations and definitions of the needed functions and macros - those were preserved as they are needed for the XtratuM architecture as well, but the Hardware dependent (i.e. assembly) code was removed. This step does not sound too interesting, but finding out which parts of the OS should be kept and what has to go is not always an easy decision. Furthermore, this step really helped to deepen the knowledge about the internals of FreeOSEK acquired during the assessment phase.

The next step to running FreeOSEK inside of an XtratuM partition, was to adapt FreeOSEK's build system, so that the resulting binary would be accepted by XtratuM. The most important thing here is, that FreeOSEK must not be compiled as an executable binary, but instead it has to be compiled as an relocatable object, that can be linked into an XtratuM partition - if necessary even in multiple partitions - at a memory address that is specified at configuration time in the XtratuM configuration file. The following section describes the efforts that have to be taken to run FreeOSEK as an runtime environment in a XtratuM partition.

Later, this relocatable object will be moved by the xmpack tool into its place in the XEF [4]. The XEF format is basically a binary format in which the binaries of all partitions, the configuration as well as XtratuM itself are incorporated into a single container. This container can then be downloaded into the targets memory.

```
CFLAGS += -Wall -c -m32 -fno-strict-aliasing -fomit-frame-pointer

CFGFILES += examples$(DIR)xm_hello$(DIR)etc$(DIR)FreeOSEK.oil

LIBXM_PATH=/home/andi/xm_oversee/xm/user/libxm
LFLAGS += -static -nostdlib $(COMMON_PATH)/std_c.o \
     $(COMMON_PATH)/traps.o $(ARCH_PATH)/arch.o \
     $(ARCH_PATH)/boot.o \
     -T/home/andi/xm_oversee/xm/user/examples/ia32/loader.lds \
     -L$(LIBXM_PATH) --start-group $(LIBGCC) \
     --end-group -m elf_i386 -Ttext=0x800000
```

This set of compiler and linker flags has been taken from the examples in the XtratuM tree. For production code, it might however be necessary to add flags that provide safety relevant

[4] Xtratum Executable Format - for details see: [MR11, Section 6.5]

features to the resulting binary. For now, this set is sufficient and no flags are changed or added, as this might easily cause problems not relevant for the rest of the porting efforts.

After this stage it is already possible to boot into FreeOSEK, and to put some `xprintf`'s[5] into the init code. Since most of the initialization code is generic (e.g. load the data of the application's task) this is already done without any changes to the FreeOSEK code base. The next point that really needed attention, was the x86 specific code for the task management.

5.3 Task Management

In order to assure a flawless scheduling of tasks, it has to be assured,that for each possible point of rescheduling, the transition from the old to the new task is done properly (properly meaning broadly that the task has no trace of its interruption at the logical or data level - at the temporal level it may be visible). Which actions have to be performed during dispatching, depends on the event that led to the rescheduling - that is on the point of rescheduling itself. OSEK OS lists the following 4 points of rescheduling for non-preemptive scheduling:

- Task Termination
- explicit activation of successor task
- explicit call of the scheduler
- a transition into a waiting state takes place

A quick look at those four points of rescheduling reveals, that the first two can be handled with little effort. For those two, the task context of the old task does not have to be saved, since it terminates, before the new task is scheduled. Therefore, all that was needed to get a basic version of FreeOSEK running on XtratuM, was to set the stack pointer to the stack of the new task, and jump into task itself. This way, simple examples that activate non-preemptive tasks, and chain non-preemptive tasks can already be run. If preemptive scheduling is desired, the following extended list of points of rescheduling has to be considered:

- Task Termination
- Explicit activation of successor task
- Activation of a task at task level
- Explicit call of the scheduler
- A transition into a waiting state takes place
- Setting a task to a waiting state

[5] xprintf is a library function of `libxm` wrapping a XM_write_console, giving the application programmer a way to use formatted printing to the raw console for debugging.

- Release of a resource at the task level
- Return from interrupt level to task level

In order to allow preemption of tasks (either voluntarily by going into waiting states or involuntarily by hitting one of the points of rescheduling from the above list), the context has to be saved before and restored after rescheduling, this part of the task management is not clean yet and will need some rework so it can be considered done. For a proof of concept as necessary by the OVERSEE project, other parts of OSEK are more important and will therefore need to be handled before finishing up task management.

5.3.1 Counters and Alarms

As described above, one way a task can be activated, is if an alarm has expired. Each alarm is triggered by exactly one counter. Counters can be incremented by all kinds of events but one of the most common ones are timers, in order to allow timed activation of tasks. All that was to do, to allow alarms that wake up tasks, was to add an IRQ handler which is triggered by the virtualized XM timer interrupts. Inside of this IRQ handler a counter is incremented, using the OSEK defined `IncrementCounter()` call. The virtualized timer is configured in the initialization code of FreeOSEK. Now one or more alarm(s) can be associated with the counter in the OIL configuration file of the application, to make those alarms go off as soon as the counter has reached its limit. An example for such configuration looks like this (only the part that deals with counters and alarms):

```
COUNTER HardwareCounter {
        MAXALLOWEDVALUE = 100000;
        TICKSPERBASE = 1000;
        MINCYCLE = 1;
        TYPE = HARDWARE;
        COUNTER = HWCOUNTER0;
};

COUNTER SoftwareCounter {
        MAXALLOWEDVALUE = 100000;
        TICKSPERBASE = 100;
        MINCYCLE = 1;
        TYPE = SOFTWARE;
};

ALARM IncrementSWCounter {
        COUNTER = HardwareCounter;
        ACTION = INCREMENT {
                COUNTER = SoftwareCounter;
        };
```

```
        AUTOSTART = TRUE {
                APPMODE = AppMode1;
                ALARMTIME = 1;
                CYCLETIME = 1;
        };
};

ALARM ActivateTaskA {
        COUNTER = SoftwareCounter;
        ACTION = ACTIVATETASK {
                TASK = TaskA;
        }
        AUTOSTART = FALSE;
};
```

The first section describes a hardware counter, that is incremented by the ticks from the virtualized XtratuM timer interrupts. This counter is used to increment a software counter using an alarm *IncrementSWCounter*. If this software counter has an overflow, the *ActivateTaskA* alarm is triggered, and the OSEK task `TaskA` goes from state *suspended* to state *ready*.

This looks like a waste of resources, but if you want different Alarms triggered by the same hardware timer, you have to configure multiple software counters, which then activate the various tasks.

5.4 Interpartition Communication

Communication in OSEK is defined in [Con04], which defines the main goals of this specification as follows:

"It is the aim of the OSEK COM specification to support the portability, re-usability and interoperability of application software. The API hides the differences between internal and external communication as well as different communication protocols, bus systems and networks."

[Con04, Section 1.1, p.5]

From this paragraph we already can deduce, that connecting FreeOSEK to the message passing interpartition communication system that is provided by XtratuM is conforming to the specification. More importantly, the latter part stating that communication protocols as well as communication media should be transparent to the application implies, that it has to be possible, to run a legacy OSEK compliant application, that uses OSEK COM. Specifically it can be run in an FreeOSEK runtime environment communicating via the XtratuM interpartition communication system instead of let's say a CAN bus, without even knowing it, and without the need of changing a single line of application code.

One further thing we can take into account, is the mostly ARINC 653 compliant interpartition communication system provided by XtratuM, and the resemblance of the OSEK Com system and the ARINC 653 interpartition communication system. This similarity in communication mechanisms leads to a huge simplification in the external communication which can be done with wrapper functions for the XtratuM hypercalls, which configure the XtratuM (ARINC 653) ports to behave the way expected from FreeOSEK and allow an OSEK Com compliant interface. Things like FIFO buffers for queueing messages do not have to be implemented, since they already are implemented in the XtratuM core.

5.4.1 Internal Representation of Communication Channels

Before the mapping of XtratuM's interpartition communication channels can be done, it is necessary to understand FreeOSEK's internal representation of messages, and the OSEK compliant API calls that are used to invoke communication. First we focus on receiving a message [6], and the OSEK compliant call for receiving messages looks as follows:

```
StatusType ReceiveMessage {
  MessageIdentifier Message,
  ApplicationDataRef DataRef
  }
```

The details of the behavior of this call can be found in [Con04, p. 51] - everything omitted here can be found there. For now it is important to find out how FreeOSEK implements the `MessageIdentifier` type as well as the `ApplicationDataRef` type.

While the API is well documented in [Con04], the internal data structures are documented in the FreeOSEK code using doxygen tags, unfortunately some of the code is not present before running the generator scripts, so the available choices are to either read the PHP code of the generator scripts or write a simple example configuration file and use the generated code to learn about the internals of FreeOSEK.

The configuration in this example defines one message with the name `MessageA`. In the code that is generated by FreeOSEK this message is simply mapped to a numeric value by a `#define` macro in the header file `Com_Cfg.h`:

```
#define MessageA 0U
```

The configuration data defining the properties of `MessageA` are found in an array of type `Com_RxMsgObjCstType`:

```
extern const Com_RxMsgObjCstType Com_RxMsgObjsCst[1];
```

In this example, only one message is defined in the configuration file, therefore the table only contains one entry - the entry for `MessageA`. Now we can find out what data this internal type

[6] The sending part is almost the same anyway.

`Com_RxMsgObjCstType` provides - flags defining the kind of message this is (external/internal, periodic/direct/mixed, as well as the notification callback type), furthermore the size of a message, a pointer to the data and the network used to receive the message from are given.

```
/** \brief Receive Message (and network) Object Const type
 **        definition
 **
 ** \param Flags   Receive Flags, for more details see
 **                Com_MsgFlagsType type definition
 ** \param Size    Size of the network message in bits
 ** \param Data    pointer to the memory to stor the data
 ** \param Net     network message
 **/
typedef struct {
        Com_MsgFlagsType Flags;
        uint16 Size;
        uint8* Data;
        Com_NetType Net;
} Com_RxMsgObjCstType;
```

Note, that the message does not provide any fields dedicated to protection against faults (neither does XtratuM for that matter), therefore a protection mechanism against faults (e.g. CRC, sequence numbers) would have to be implemented at the application level.

5.4.2 Mapping FreeOSEK Channels to XtratuM Channels

The assumption in OVERSEE [Con10] is that all communication to a physical bus or network is conducted via the *secure I/O partition*. This also means that no FreeOSEK partition has access to a physical bus and all communication can be assumed to be via the XtratuM communication mechanisms.

To achieve this kind of communication, a mapping between OSEK messages and XtratuM's ports has to be done. Internally this handled via the struct `Com_RxPortsType`, which maps an OSEK message represented by the pointer to the message object *MsgObj* to a port represented by the portname and the portdesc. Furthermore key properties of the port are stored here as well.

```
typedef struct {
        Com_RxMsgObjCstType *MsgObj;
        int portdesc;
        char portname[MAX_PORT_NAME_LENGTH];
        uint32 maxMsgSize;
        uint32 maxNumMsgs;
} Com_RxPortsType;
```

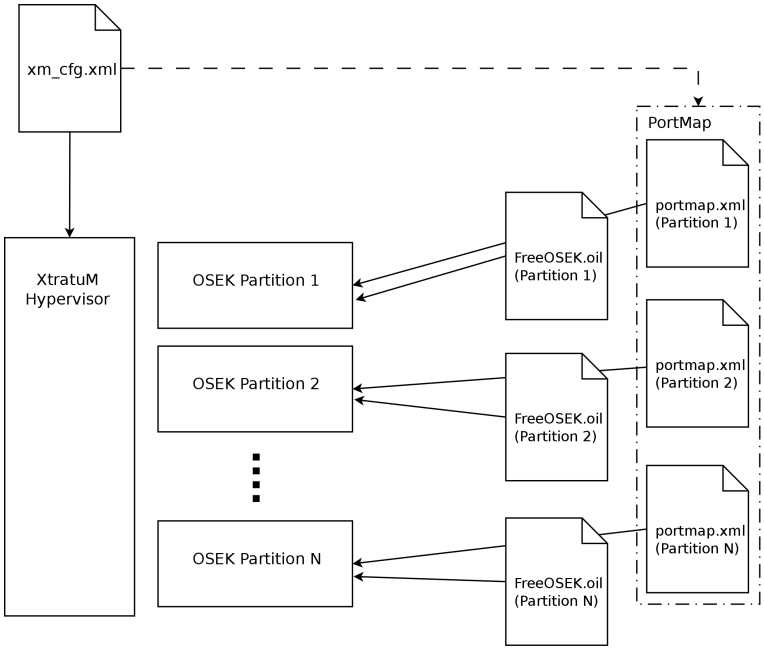

Figure 5.1: Configuration of a System with multiple OSEK partitions

In order to allow configuration of this messages to ports mapping, a simple port-mapping tool was developed that allows to do the configuration of the ports in an XML file and automatically generates the configuration of the ports and paste it into the code generated by the PHP generator script.

The principle of the resulting configuration scheme is shown in figure 5.1, where the system consisting of the XtratuM hypervisor and N partition is depicted int he lower left corner. At compile time the XML configuration file for XtratuM (xm_cfg.xml) is read and statically compiled into the binary. In addition a portmap.xml file for each OSEK partition maps the ports of the partition (configured in xm_cfg.xml) to the messages that can be used in the OSEK compliant application code. The following shows a simple example of such a mapping:

```
<PORT_MAPPING>
        <PORT id="0" portname="readerS" messagename="MessageA" \
        type="SAMPLING" direction="DESTINATION" portsize="10"  \
        msgsize="512" />
</PORT_MAPPING>
```

This simple OSEK runtime environment uses only one sampling port, which is available in XtratuM under the name `readerS`, and is mapped to the OSEK message with the name `MessageA`.

5.5 Implementation Summary

Even if this prototype of the FreeOSEK runtime environment has restricted capabilities, the implementation shows the feasibility of running an OSEK compliant operating system as a XtratuM runtime environment, fulfilling one of the main missions of OVERSEE - reuse of existing automotive applications in a security enhanced environment with minimum effort.

Furthermore, the theoretical mapping between OSEK compliant communication and ARINC 653 compliant communication could be proven valid and compatible to the point where a legacy OSEK compliant application can be moved into a XtratuM runtime environment with a virtualized communication system replacing a legacy physical communications system, without the need of adapting the application itself.

The next steps in the port of FreeOSEK to XtratuM will be the cleanup of the context switch, in order to allow fully preemptive task scheduling. This is also the pre-requisite for most of the MODISTARC [Con99b] tests which are already implemented in FreeOSEK and help to show the compliance with the OSEK/VDX specifications and finally the integration of the FreeOSEK port into the overall OVERSEE architecture proof-of-concept framework.

CHAPTER 6

Design of an Example Application

There is little doubt that there is a wealth (literally) of code and solutions that were built around OSEK in the automotive industry. To allow reuse of these components and the intellectual property behind it, the practical part of this thesis is providing an OSEK partition on top of the OVERSEE platform, to effectively demonstrate the feasibility of migrating an OSEK based application to an integrated platform based on partitioning, a typical OSEK application is to be included in the OVERSEE demonstrator. This chapter designs such a generic application than can be run in other OSEK compliant operating systems.

6.1 The Lifecycle of a Safety Critical Application

While the application itself may seem trivial, which many safety related applications actually are, the intention is to demonstrate key qualities of the integrated approach:

- Safety and security properties retaining composeability
- Reuse of OSEK applications

As mentioned above, this chapter contains the full design of this simple safety critical application. The approach taken is to collect the requirements (Section 6.2), do a naive, preliminary high-level design (Section 6.3, use this naive design to perform a hazard analysis (ection 6.4) and use the output of the hazard analysis as an input for a refined, high-level design (Section 6.5). Then, from this finalized high level design a detailed design is done (Section 6.6). This detailed design again is analyzed for potential hazards - this time a FMEA (Section 6.7) is performed on the interface of the detailed design.

Note, that this approach of starting with a preliminary design is done by choice even encouraged by safety standards to choose this way of approaching the problem at hand. For example

in IEC61508-1 Ed.2 this approach can be found in Section 7 **Overall safety lifecycle requirements**, more specifically in clause 7.3:

"7.3 Overall scope definition
NOTE - This phase is Box 2 of Figure 2.

7.3.1 Objectives

7.3.1.1 The first objective of the requirements of this subclause is to determine the boundary of the EUC and the EUC control system.

7.3.1.2 The second objective of the requirements of this subclause is to specify the scope of the hazard and risk analysis (for example process hazards, environmental hazards, etc.).

7.3.2 Requirements

7.3.2.1 The boundary of the EUC and the EUC control system shall be defined so as to include all equipment and systems (including humans where appropriate) that are associated with relevant hazards and hazardous events. NOTE - Several iterations between overall scope definition and hazard and risk analysis may be necessary.

7.3.2.2 The physical equipment, including the EUC and the EUC control system, to be included in the scope of the hazard and risk analysis shall be specified. NOTE - See references [9] *and* [10] *in the bibliography C.*

7.3.2.3 The external events to be taken into account in the hazard and risk analysis shall be specified.

7.3.2.4 The equipment and systems that are associated with the hazards and hazardous events shall be specified.

7.3.2.5 The type of accident-initiating events that need to be considered (for example component failures, procedural faults, human error, dependent failure mechanisms that can cause accident sequences to occur) shall be specified.

7.3.2.6 The information and results acquired in 7.3.2.1 to 7.3.2.5 shall be documented."

[IEC10b, Clause 7.3]

So how does this section support the approach taken in designing this indicator control application? The interpretation of this clause, and the reason one can be confident that it is suitable for this task is as follows. First of all it defines two objectives:

- Determination of the system boundary - the system boundary is formally specified in the data context diagram of the naive design.

- Specification of the hazard analysis - this is done in the preamble of the analysis. In this thesis two hazard analysis with different scopes are performed, a HAZOP on the system level and a FMEA on the API of the application.

So the objectives can be met by doing a naive design first. The requirements on the Overall scope definition are met by the following properties of the preliminary design:

Clear definition of the boundaries - everything inside of the bubble in the data context diagram is part of the system

Anticipate external events - all external interrupts as well as data passed from the environment into and out of the application are defined in the data context diagram

Associate (sub)systems with hazards - in data flow diagram 0 the preliminary (naive) design is broken into three modules, the association of hazards to those modules is done during the HAZOP that is carried out on this design in Section 6.4

Accident initiating events - consideration of accident initiating events will be handled in the form of a list after the preliminary design.

Also, as noted in clause 7.3.2.1 several iterations might be necessary, in order to get to an appropriate level of the overall scope definition, this leads to a process for getting to a high-level design as shown in Figure 6.1, where the iterative approach used in this Chapter is sketched.

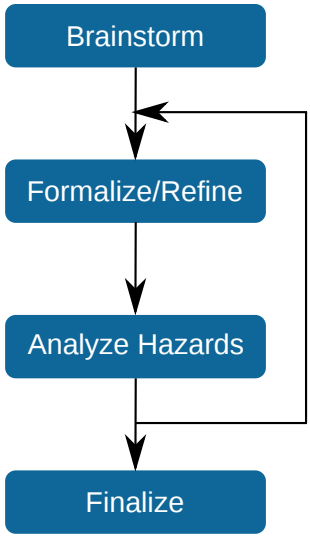

Figure 6.1: Process of defining the Overall Scope

The steps taken in the process (depicted in Figure 6.1) map to the sections of this chapter as follows:

Brainstorm this is an informal collection of requirements done in Section 6.2

Formalize/Refine depending on whether it is the first time or an iteration:

- in the first run the informal collection from the *brainstrom* are used to do a preliminary design that fulfills all the functional requirements but may be insufficient from a safety point of view (this is done in Section 6.3)
- in each further iteration the output of the preceding hazard analysis is taken to refine the design and adds needed safety functions to fulfill the safety requirements. In this thesis only one iteration is done (see Section 6.5).

Analyze Hazards in Section 6.4 a HAZOP is conducted on the preliminary design from Section 6.3.

Finalize the refinement in Section 6.5 also constitutes the finalization of the overall scope

6.2 Requirements Analysis

The application used to demonstrate the above properties in the OVERSEE demonstrator is a software implementation of a turn indicator control. Basically this is a timer application and some simple I/O. But while the functional requirements seem trivial there are a few side conditions that make this a bit more complex.

- Turn indicators and emergency flashers share the same resource but at different priorities - these priorities must be respected
- Turn indicators can fail (broken cabling or light-bulb/LED failure) and so a monitoring functionality needs to be provided
- The turn angle of the wheels, respectively the turning back of the wheels is an indicator for stopping turn indicators automatically. This automatic stop of the indicators is restricted to higher speeds, as the indicator should stay on at very low speeds (e.g. during parking maneuvers).
- Acoustic indication of turn indicator activation is needed

Thus this use-case should demonstrate that a simple function can be cleanly modularized by abstracting signal inputs to inter-domain communication primitives and thus de-coupling functions from the physical implementation, that is, it is not relevant if the failure of the lights is detected by physical circuit and signaled to the application level or if it is generated in software by some form of sensor application, essentially the integrated approach allows to provide a simple and sound implementation of direction indicator control, satisfying safety demands, under consideration of security requirements and retaining composeability by de-coupling from specific implementation details. Modularizing complex software in this way is crucial for decreasing the complexity of the safety function itself.

In the rest of this section an (almost) complete design process for a safety critical software application will be performed on this turn indicator example. First a naive high level design is done using structured design, then a HAZOP is conducted on the high level design, revealing

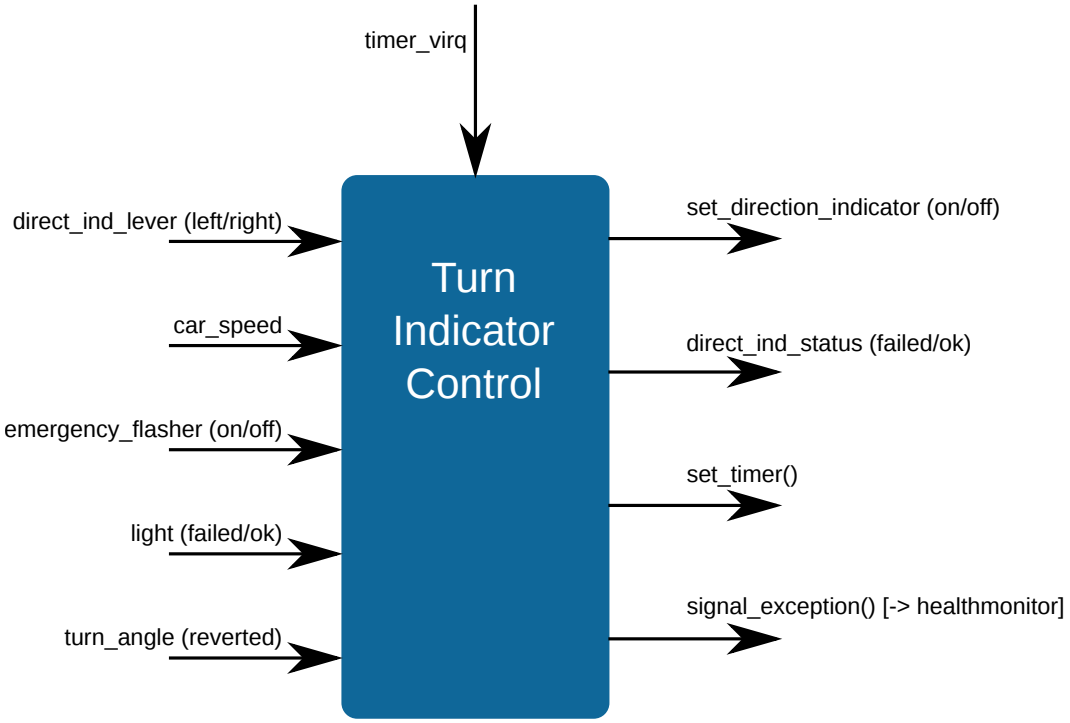

Figure 6.2: Example Application: turn indicators

some problems that had not been thought of during the high level design. Thus the high level design is redone and a detailed design is done. After that an FMEA is done - for time and space reasons only on the detailed design of one component.

Figure 6.2 shows just a simple block diagram with inputs and outputs to/from the application. This maps nicely Figure 6.3 which is a little closer to reality and shows the high level view of the resulting partition. The turn indicator control application is running on top of FreeOSEK inside of an isolated application partition, which is only accessible from the outside world via sampling ports.

6.3 High Level Design

In the following a RTSD as described in Section 2.7 is performed on the above described example. This RTSD includes a high level design of the turn indicator example. This is a naive approach to the problem, that is later on used for a hazard assessment. Of course some of the problems are anticipated because they were thought of “by accident“ while doing the design. Later on a HAZOP on the high level design is conducted, which is expected to reveal problems that were not anticipated during this first shot at the problem.

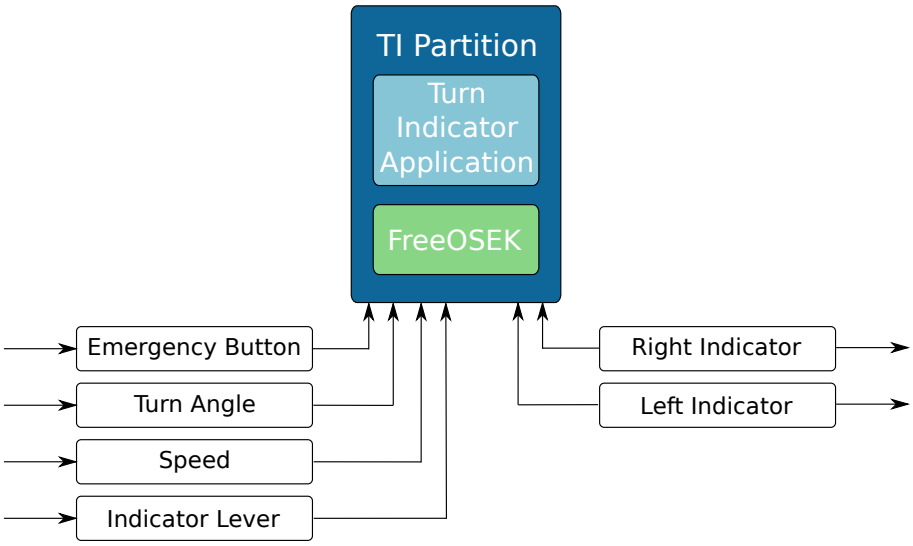

Figure 6.3: Example Application: Partition Interface

6.3.1 Data Context Diagram

The data flow diagram in Figure 6.4 shows that the turn indicator control gets a number of signals from its environment. Those signals contain the following information, summarized in Tables 6.1 and 6.2.

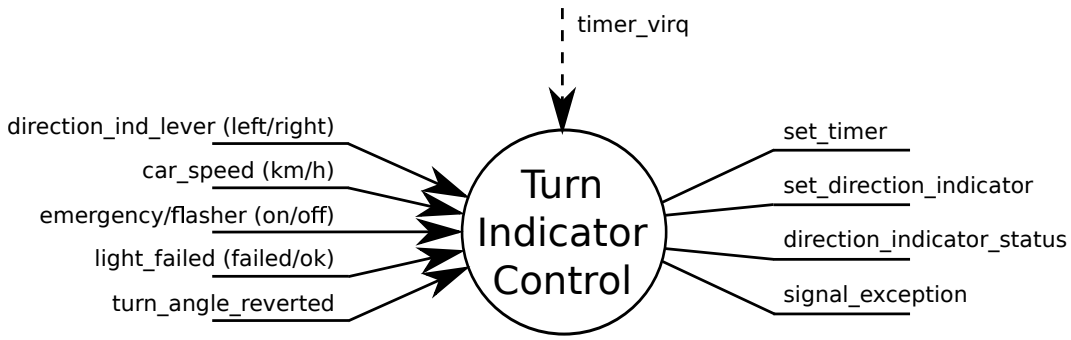

Figure 6.4: DCD - Data Context Diagram of the High Level Architecture

Inputs	Description
timer_virq	The virtualized timer interrupt is used to toggle the indicators when they are in use.
direction_ind_lever	The turn indicator lever allows the driver to (de-)activate the turn indicators
turn_angle_reverted	If the car has turned and the turn angle of the wheels has reverted to its neutral position, the indicators are turned off automatically.
car_speed	In some situations (e.g. parking maneuvers) the indicators should not be turned off when the turn angle has reverted. These situations are limited to small speeds, thus the speed of the vehicle is used in addition to the turn angle to decide whether or not the indicator lights should be turned off.
emergency_flasher	The emergency flasher uses the same set of lights that are for indicating a change in direction, but the priority is higher, as the emergency has already been identified. Therefore if the emergency flasher is turned on, the state of the turn indicators becomes unimportant and is overwritten by the state of the emergency flasher.
light_failed	If one of the lights fails, the indicator frequency is increased to indicate the failure to the driver.

Table 6.1: Inputs of the turn indicator application (see Figure 6.4)

Outputs:	Description
set_direction_indicator	This output is passed on to a suitable driver to actually physically enable the light to be set to on or off.
direction_indicator_status	Indication of the turn indicators being active to the driver is done by flashing suitable direction signals on the dash-board as well as an acoustic indicator.
set_timer	Direction indication must be done within a standardized, , frequency range. A timer is used to implement the periodic signal. Further, the frequency is used to indicate light failures (by doubling the frequency), not only to the driver in the car but also to the environment (i.e. other drivers).
signal_exception	Any software, even if seemingly trivial , has the potential to enter some inconsistent internal state that it can't handle on its own any more. As this application is running in an isolated partition some of the unhandled situations can be resolved by escalating the event to the platform that is able to perform a forced reset of the partition as well as provide failure information to the driver.

Table 6.2: Outputs of the turn indicator application (see Figure 6.4)

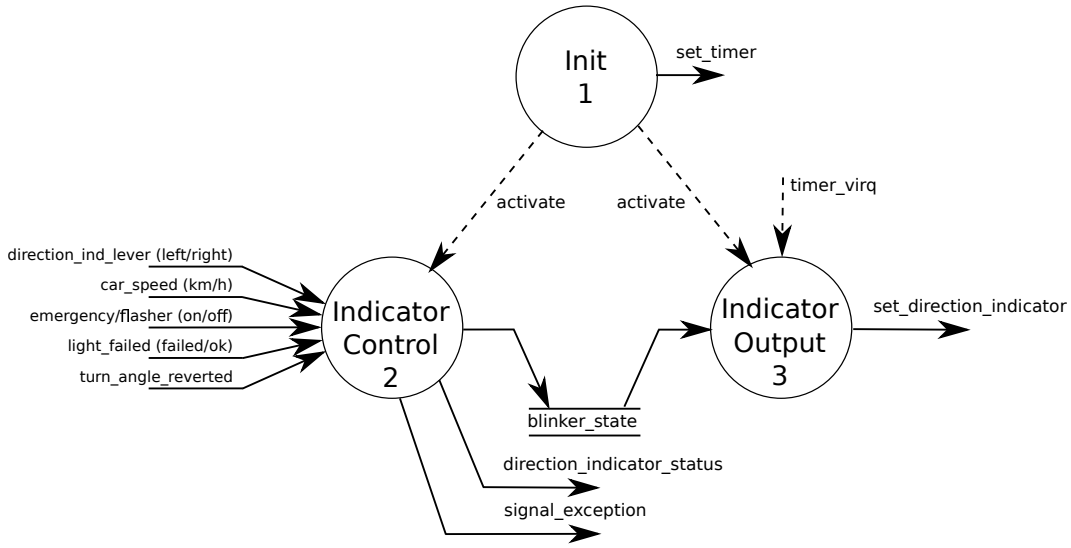

Figure 6.5: DFD0 - High Level Composition (Task Level)

6.3.2 Data Flow Diagram

The RTSAD design, as introduced in Section 2.7 is done no. The design consists of the structured diagram in Figure 6.5 as well as the data dictionary listed in the following. The items in the dictionary/diagram are enumerated to allow a clear mapping.

1 Init

Description	Initialization task
Rate	Executed only once at partition boot time.
Comment	The initialization task sets the timer for the periodic `Indicator Output` task and sets up communication channels to other partitions.

2 Indicator Control

Description	processes the input data and determines indicator output
Rate	Main Task, it is consuming all of the partitions time slot, only interrupted by the higher prior `Indicator Output` Task.
Comment	This is the task were the decision of whether to flash the indicators or not is made.

3 Indicator Output

Description	sets the indicator output calculated by the `Indicator Control`
Rate	periodic task, the period is 1.5Hz [1] in normal mode, and 3Hz [2] if one or more of the lights are damaged, in order to make the driver aware of the damage.
Comment	Manages the timer and writes the data into the sampling ports.

6.4 High Level Hazard and Operability Study

Now as the preliminary high-level design is done, this design is used to conduct a hazard analysis on the turn indicator application. This HAZOP will hopefully reveal all corner cases not anticipated in the preliminary design.

6.4.1 Pre-Requisites

During the course of doing the HAZOP in the next section, several things were realized. First of all, although a naive approach to the problem, in the form of the high level design from 6.3 is available, this is just a preliminary design and problems that have been taken into account intuitively should nevertheless be part of the hazard analysis for completeness.

Furthermore, it is important to really stick to the high level of abstraction, and avoid getting too deep into the details. Some hazards pointed out in a high level design might not be relevant, as mitigation of those hazards is outside of the scope, nevertheless on the high level design they have to be included. An example for this would be if it were possible that only the front indicators are flashing, due to bad wiring, then there is no mitigation for this in software, this is a safety condition that has to be fulfilled by the environment. From a very high level of abstraction, the *turn indicator control* system has only two functions it implements:

- Turn indicator - flash left or right (depending on the lever)
- Emergency indicator - flash all indicators if button pressed

So the goal of the HAZOP is to point out all possible hazards that are able to compromise this functionality.

Throughout the HAZOP table, some abbreviations are used, to reduce the size of the table. Please note, that these abbreviations may not be commonly used abbreviations and are possibly only valid for this section of this thesis but as any safety related document should be complete either by referencing normative sources or by meticulously defining everything internally , this is a sound approach. The abbreviations are listed in Tabele 6.3.

EF	emergency flasher
TI	turn indicator
TIC	turn indicator control (includes EF and TI)
TI on	for the TI to be on can mean flash right as well as left
SAC	safety application condition

Table 6.3: Abbreviations used during the Hazard and operability study.

[1] In Europe the frequency of the indicators has to be 1.5HZ $\pm$ 0.5Hz.
[2] European Law only defines that it has to be *much faster*.

HAZOP item	Item	Attribute	Guide Word	Cause	Consequence/ Implication	Indication/ Protection	Question/ recommendation
1	EF	ON	No	.) lights don't work .) input not handled .) battery empty	unsafe situation cannot be indicated to other drivers		**SAC:** .) if battery gets empty, TIC has priority over other systems of the vehicle .) LED lights
2	EF	ON	More		No Meaning		
3	EF	ON	As Well As	priority of functionality is incorrectly assigned.	TI overwrites	assign higher priority to EF	
4	EF	ON	Part Of		only one side flashing	handle as if light broken (flash faster)	
5	EF	ON	Early/Late	time management incorrect	wrong flashing period very slow, very fast (permanent on)	sanity checks based on wall clock timestamps	**SAC:** timer irqs provided by XM correct.
6	EF	ON	Before/After		No Meaning		
7	EF	OFF	No		No Meaning		
8	EF	OFF	More	Spurious EF on signal and higher priority of EF	EF overwrites turn indicator relevant information for other drivers suppressed.	.) see item 3 .) flash emergency button	
9	EF	OFF	As Well As	higher priority of EF over TI	turn indicator turned off as well relevant information for other drivers suppressed	see item 3	
10	EF	OFF	Part Of	.) light broken	only one side off,	read back	

				.) bad wiring .) wrong output	other driver sees indicator signal.	values and double check	
11	EF	OFF	Early/Late		No Meaning		
12	EF	OFF	Before/After		same as more.		
13	TI	ON	No	.) EF overwrites indicator .) dead battery .) lights don't work .) input not handled	Indicator not flashing.		
14	TI	ON	More	.) EF instead of TI. .) left and right with different periods.	both left and right flash at the same time.		
15	TI	ON	As Well As		same as More		
16	TI	ON	Part Of	bad wiring	only front or rear indicator flashes		**SAC:** wiring is correct.
17	TI	ON	Early/Late	time management incorrect	wrong flashing period.		**SAC:** see item 5
18	TI	ON	Before/After		No Meaning.		
19	TI	OFF	No		No Meaning.		
20	TI	OFF	More		No Meaning.		
21	TI	OFF	As Well As		No Meaning.		
22	TI	OFF	Part Of		No Meaning.		
23	TI	OFF	Early/Late	can be due to incorrect time management	delay in switch-off can be interpreted as second turn.		**SAC:** see item 5
24	TI	OFF	Before/After		No Meaning		

Table 6.4: Hazard and Operability Study

6.4.2 Summary of the HAZOP

After conducting the HAZOP in Table 6.4, it is now time to analyze the lessons learned and analyze the notes from the table in a more elaborate way. This summary of the HAZOP picks out the safety application conditions as well as the indication and protection mechanisms shortly noted in the table and discusses them, so they can be integrated into a new, refined design in the next section.

6.4.2.1 Safety Application Conditions

One of the most important outputs of this HAZOP are the *safety application conditions*. These conditions are pre-requisites that have to be fulfilled by the environment to allow the system itself to work properly. Only if these conditions are met, the system can behave correctly. Otherwise problems that cannot be detected and/or corrected by the system itself will arise eventually.

For the longterm usage of the software component these SACs are important, as they are used to determine whether the component is suitable to be deployed in a new environment, or if it cannot be used in the new environment as an important pre-requisite is not met.

Correct Wiring: In order to make it possible to flash the lights that are intended, the wiring has to be correct. If e.g. instead of the front left and the rear left indicator, front left and right indicators are connected to the output for the left indicators, the two front indicators will flash when the driver tries to indicate a left turn. These kinds of problems are not too hard to detect, and should be tested after assembly.

Dead Battery: If the battery is dying, it has to be assured that vital elements of the vehicle - e.g. the emergency flasher - function properly as long as possible, while others can be taken out of operation to save energy (e.g. the parking assistant). This kind of energy control has to happen on a vehicle wide level and can therefore not be handled by the indicator control system.

Correct Time Management: The indicator control system depends on the correct (virtual) timer interrupts generated by the hypervisor. If the time management is not correct due to a fault in the hypervisor itself, cases that cannot be detected by the application partition arise. We can however catch some cases where the fault is only affecting the timer interrupts, but the hypercall for the wall clock still is working properly. Details for these cases can be found below.

LED lights: Over recent years LED lights are used for the indicator lights as well as the rear and braking lights. The big advantage of LED lights is their long life span, as well as the fact that the single point of failure - namely the filament of the light bulb - is replaced by multiple LEDs. So even if a considerable number of LEDs die, the rest is enough to give clear signals to the other drivers, and the broken ones will be fixed by the mandatory yearly maintenance and thus this hazard is covered short term and long term. So from a strict point of view, the yearly maintenance is also a SAC here.

6.4.2.2 Indication and Protection Mechanisms

In order to get the system to detect and indicate all hazards that can be covered by the system itself, a variety of mechanisms can be employed. These mechanisms can either be pure indication mechanisms, i.e. just make the user (here: the driver as well as other drivers) aware that something bad is going on (in this case this would be flashing with a higher frequency). Other mechanisms can be real protection mechanisms like correct priority assignment and enforcement - i.e. the emergency flasher has higher priority than the turn indicator if it is switched on, otherwise it must not bother the turn indicator.

Nevertheless, the goal should be to eliminate all hazards detectable by the system, and only those hazards that cannot be handled should be passed on into the responsibility of the driver or to other drivers.

Furthermore the following list contains fault detection mechanisms that are used to check the behavior as well as the data in order to detect faults and make the indication of the fault to the user possible.

Correct Priority Assignment: The basis for resolving problems where the turn indicator can overwrite the emergency flasher or the other way round, is correctness of the priority assignment.

Sanity Checks: In order to check the correct functionality of the turn indicator control, different sanity checks on the data, as well as the timely behavior can be performed.

- *Data Checks* – a monitoring task inside of the turn indicator control application checks whether the actual output is the same as the intended output.
- *Timely Behaviour* – as already stated above (see 6.4.2.1, "Correct Time Management"), the timely behaviour of the system depends on the correctness of the hypervisors time management. Although the partition has no influence on that correctness, some failures of the correctness of the virtual interrupts can be checked easily by sanity checks. To perform these sanity checks, the application has to compare the wall clock time of the current and the last interrupt, in order to find out if the period of the interrupts is correct, that is if the difference of the two timestamps is in the boundaries of the period.

6.5 Refinement of the High-Level Design

Now that the results of the hazard analysis of the preliminary design has shown the weak spots in the design, it is time to improve the design, and to handle those hazards. This means that we are iteratively improving our design. This iterative approach is a very common approach in the safety domain, as it is an obvious assumption that the first design [3] will miss some important corner cases. These corner cases can only be found by systematically finding the weak spots in

[3] or even the first few designs for that matter

the design, the implementation, etc. and by learning from the results and redoing the whole thing with the new knowledge at hand. IEC61508 advocates this approach of iterative improvements in the following clause:

" 7.1.1.4 The overall, E/E/PES and software safety lifecycle figures (figures 2 to 4) are simplified views of reality and as such do not show all the iterations relating to specific phases or between phases. Iteration, however, is an essential and vital part of development through the overall, E/E/PES and software safety lifecycles. "

[IEC10b, Clause 7.1.1.4]

The basic idea for the refined version of the Design, is to introduce a new task. This new task is a monitoring task that calculates - based on the input and the history of previous inputs - a set of valid new states. If the state calculated by the control task does not match any of those valid new states, the monitor task switches the indicators frequency from normal to double frequency, indicating that an error has happened. The resulting new high level design can be seen in Figure 6.6, this diagram is supplemented by the following data dictionary.

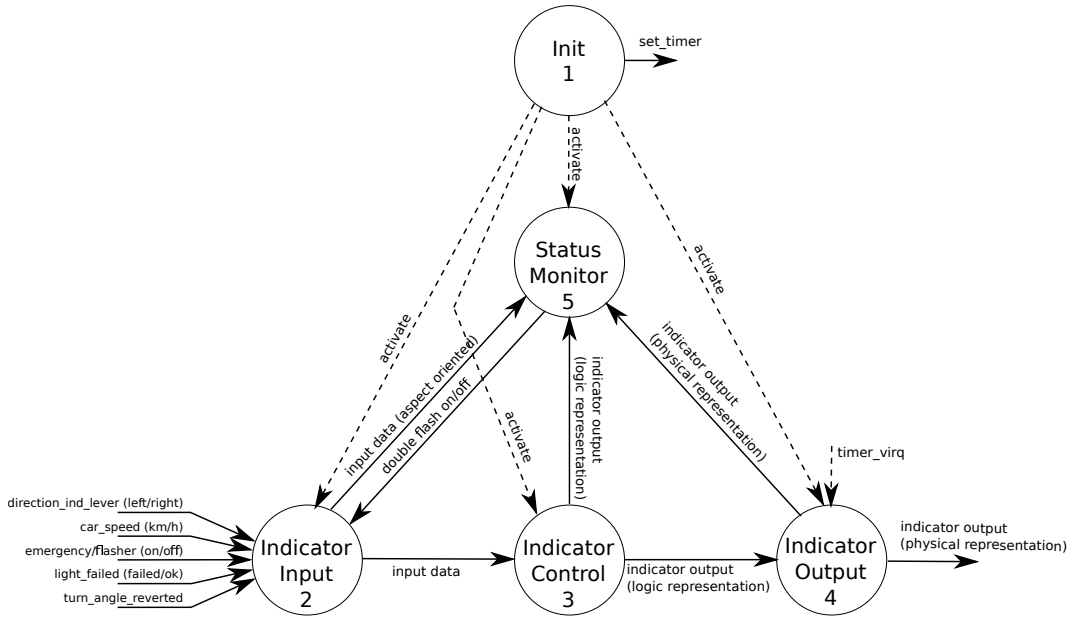

Figure 6.6: DFD0 - High Level Composition (Task Level) - Refined Version

1 Init

Description	Initialization task
Rate	Executed only once at partition boot time.
Comment	The initialization task sets the timer for the periodic `Indicator Output` task and sets up communication channels to other partitions.

2 Indicator Input

Description	reads the input data and passes it to *Control Task* and *Monitor Task*
Rate	10Hz (indicators are flashing at 1.5 Hz / 3 Hz, to satisfy Nyquists sampling theorem 6 Hz would be appropriate, adding a safety margin leads to a rate of 10Hz - this might be changed at testing time)
Comment	The *Indicator Input* Task reads the input data from the inbound communication channels. To each data set a unique sequence number is added, and the data is passed on to *Indicator Control* as well as to the Status Monitor Task.

3 Indicator Control

Description	processes the input data and determines the indicator output
Rate	10Hz (indicators are flashing at 1.5 Hz / 3 Hz, to satisfy Nyquists sampling theorem 6 Hz would be appropriate, adding a safety margin leads to a rate of 10Hz - this might be changed at testing time)
Comment	The *Indicator Control* Task gets the Input Data from the *Indicator Input* Task. This Input Data is fed into the Control Algorithm that decides whether the left, right or both (emergency) side of the indicators should flash. This logical representation of the output is passed on to the *Indicator Output* and the *Status Monitor* Task.

4 Indicator Output

Description	sets the indicator output according to the new state calculated by the *Indicator Control* Task.
Rate	10Hz (indicators are flashing at 1.5 Hz / 3 Hz, to satisfy Nyquists sampling theorem 6 Hz would be appropriate, adding a safety margin leads to a rate of 10Hz - this might be changed at testing time)
Comment	The *Indicator Output* Task transforms the logical representation of the Output calculated by the *Indicator Control* Task into a physical representation. This physical representation is then provided to the driver partition that controls the vehicles indicator lights via a Sampling port. In addition the physical representation is passed to the *Status Monitor*, where it is used to compare the physical and logical representation (e.g. to check the *Indicator Output* Task).

5 Status Monitor

Description	The *Status Monitor* task collects data from all all the other tasks and performs failure detection checks on the data.
Rate	10Hz (indicators are flashing at 1.5 Hz / 3 Hz, to satisfy Nyquists sampling theorem 6 Hz would be appropriate, adding a safety margin leads to a rate of 10Hz - this might be changed at testing time)
Comment	The *Status Monitor* keeps record of the last N input records, and uses them to calculate all possible next states. It then compares the logical (from the *Indicator Control* task) and the physical (form the *Indicator Output* task) representation of the output data and determines if this data represents a possible state. If a failure is detected, the *Status Monitor* doubles the flashing frequency, thus indicating a problem to the driver as well as to the other vehicles drivers.

This refined design is the result of only one iteration of the process introduced in the beginning of this chapter (Figure 6.1). For an application that is subject to certification more iterations might be necessary to catch even more subtle failure modes.

6.6 Detailed Design

Now that the high-level design has been developed, a more detailed design is needed. As this step is fairly straight forward for such a relatively simple example, and still produces lots of paper, this detailed design is shown by example of the *indicator input* (Id 2 in Figure 6.6).

Figure 6.7 shows the structure chart of the *indicator input* entity is further decomposed into its functional modules, and the data- and control-flow are specified. These functional modules are already very simple elements that can be transformed into code very easy. They will just be explained in prose, but other representations (e.g. state transition diagrams, truth tables, etc.) could be used at this level as well.

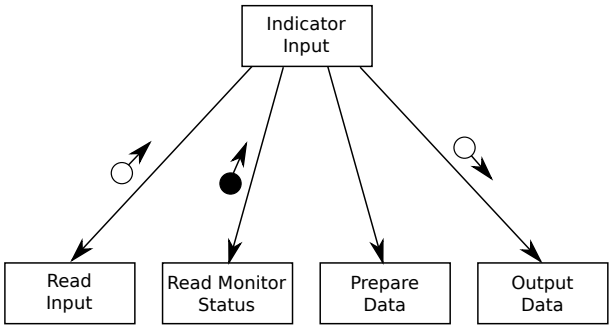

Figure 6.7: Structure Chart of *Indicator Input*

The four functional modules of *Indicator Input* are listed and explained in the following list. The structure chart can be read as *Indicator Input* calling the four functional modules one after the other. The frequency of *Indicator Input* has already been specified in Section 6.5. In the following a number of identifiers are used, their meaning described in Table 6.5.

Identifier	Description
cur_ls	current lever state (left/neutral/right /dead)
lst_ls	last lever state (left/neutral/right /dead)
cur_lis	current left indicator state (on/off/dead)
lst_lis	last left indicator state (on/off/dead)
cur_ris	current right indicator state (on/off/dead)
lst_ris	last right indicator state (on/off/dead)
cur_ang	current wheel angel (degree)
lst_ang	last wheel angle (degree)
emerg	emergency flasher button state
health	state of input (as perceived by *indicator input* indicates e.g. if the input is outdated.)
seq	sequence number
ts	timestamp
crc	a data CRC on all the above

Table 6.5: Identifiers used during the detailed design phase.

Read Input: gets the current data from the input messages. This data includes (see Figure 6.6): $[cur\_ls, cur\_lis, cur\_ris, cur\_ang, emerg]$ All these values are read via sampling messages via the OSEK/VDX API and stored locally so they can be packed and passed on to the *indicator control* and *status monitor*.

Read Monitor Status: The flashing frequency of the indicators is determined by the *status monitor*. The current setting is read from the *status monitor*, and add to the data set passed to the *indicator control*.

Prepare Data: Before the read data can be sent to the *indicator control* and *status monitor* it has to be packed into structures that are expected by the respective entity. The data is packed up as follows:

- Indicator Input sends the full state to the controller (but no timestamp): $[lst\_ang, cur\_ang, lst\_ls, cur\_ls, cur\_lis, lst\_lis, cur\_ris, lst\_ris, emerg, health, seq, crc]$
- Indicator Input sends only the current state to the monitor (including timestamp) $[cur\_ang, cur\_ls, cur\_lis, cur\_ris, emerg, health, seq, ts, crc]$

Output Data: the data that has been packed in the *Prepare Data* Module is now passed on to the interfacing bubbles, namely to the *indicator control* and the *status monitor*.

enum	members
indicator_state	IND_ON = 0 IND_OFF IND_DEAD
lever_state	LEV_NEUTRAL = 0 LEV_LEFT LEV_RIGHT LEV_DEAD
flash_state	NORMAL_FLASH = 0x55 DOUBLE_FLASH = 0xAA

Table 6.6: Detailed Design - Enumerations

struct	members
in_data	enum lever_state ls enum indicator_state lis enum indicator_state ris uint16_t ang bool emergency_button
controller_data	uint16_t lst_ang uint16_t cur_ang enum lever_state lst_ls enum lever_state cur_ls enum indicator_state cur_lis enum indicator_state lst_lis enum indicator_state cur_ris enum indicator_state lst_ris bool emergency_button enum flash_state health uint16_t seq_nr uint32_t crc
monitor_data	uint16_t cur_ang enum lever_state cur_ls enum indicator_state cur_ris bool emergency_button enum flash_state health struct timespec ts uint16_t seq_nr uint32_t crc

Table 6.7: Detailed Design - Data Structures

Function Name	Output Type	Parameters
read_input	StatusType	struct in_data *input
read_monitor_status	StatusType	enum flash_state *mon_status
pack_data	StatusType	struct in_data current struct in_data *last struct flash_state mon_status struct controller_data *to_controller struct monitor_data *to_monitor
write_output	StatusType	struct controller_data to_controller struct monitor_data to_monitor

Table 6.8: Detailed Design - Function Prototypes

Putting all this into C code, the interface of the *indicator input* task is shown in Appendix A. A summary of this interface is found in the form of enumerations in Table 6.6, data structures in Table 6.7 and function prototypes in table 6.8.

6.7 Risk Assessment of the Detailed Design

Finally, the detailed design is analyzed by performing a FMEA on the interface that has been established in the detailed design done in the previous section. The process of the FMEA is roughly consistent with [DoD80, 4.4.2], the objectives (a-h) of the process defined in MIL-STD-1629A are handled in this thesis as follows:

(a) The system definition has been done above, at the very beginning of this chapter, both in a very informal way in the introduction to this chapter, and in a more structured way in the requirements analysis 6.2.

(b) Although no block diagram was constructed, a structured high-level design (Figures 6.7 and 6.6) has been done, that *illustrates operation, interrelationships, and interdependencies of functional entities*. Therefore, although a different notation was chosen, the objective clearly is met.

(c) As the FMEA is performed on all the function prototypes (interface) the full coverage of items is definitely reached, furthermore a full coverage of potential failure modes is reached by iterating through all input parameters and their classes of input. The FMEA is performed in Section C.

Of course logical correctness is a pre-requisite (provided by the process) that has to be defined in the interface specification and tested in the testing process. The FMEA does not deal with that part of the application, here only non-functional correctness is checked.

(d) The indicator control example is a generic component, therefore the severity analysis is restricted to a qualitative severity analysis of *negligible*, *tolerable* and *critical*, where *critical* would mean that the severity would have to be assessed against the actual deployment scenario (which is unknown here).

The allocation of the severity to the potential failure modes is done in Section D.

(e) Failure mode detection methods are listed in Section E.

(f) Corrective design or other actions are listed in Section F.

(g) This objective is DOD specific and not relevant in this context.

(h) The documentation of all this is done in the form of this description of the process and the following summary of the FMEA. The result of this FMEA is summarized in H where a severity classification of all failure modes after mitigation has been applied is done.

As mentioned, the Points (a) and (b) are considered to be handled in previous sections. The open Points (c-h) are conducted in the following, where (c-h) map to (C-H) respectively.

It should also be noted, that this FMEA is restricted to systematic faults, and that other fault types are not considered.

(C) **Failure Mode Identification**

The following contains the failure mode identification part oft the FMEA. The approach was to iterate over the function prototypes of the detailed design (Section 6.6) and consider all possible values of the parameters that get the application into a failure mode.

function: `StatusType read_input (struct in_data *input)`

- *input == NULL

  what happens: access to invalid memory, application terminated through health monitor, health monitor is expected to signal error.

  effect: restart of partition

  mitigation: check for null pointer, return ERROR_CODE

- *input == 0xDEAD (also known as pointer to lala-land)

  what happens: access to invalid memory, application terminated through health monitor, health monitor is expected to signal error.

  effect: restart of partition

  mitigation: component is in undefined state, internal mitigation not possible $\Rightarrow$ has to be handled by health monitor.

- *input points to valid address space, but wrong data structure

  what happens: access to valid memory, but wrong data.

  effect: no detection by system

mitigation: extension of data structure by unique magic number.

function: `StatusType read_monitor_status( \`
`enum flash_state *mon_status)`

- *mon_status == NULL

  what happens: access to invalid memory, application terminated through health monitor, health monitor is expected to signal error.

  effect: restart of partition

  mitigation: check for null pointer, return ERROR_CODE

- *mon_status == 0xDEAD (also known as pointer to lala-land)

  what happens: access to invalid memory, application terminated through health monitor, health monitor is expected to signal error.

  effect: restart of partition

  mitigation: component is in undefined state, internal mitigation not possible ⇒ has to be handled by health monitor.

- *mon_status points to valid address space but wrong address

  what happens: access to valid memory, but wrong data.

  effect: no detection by system

  mitigation: extension of data structure by unique magic number.

function:
```
StatusType pack_data(struct in_data current,\
          struct in_data *last,\
          struct flash_state mon_status,\
          struct controller_data *to_controller,\
          struct monitor_data *to_monitor);
```

- (*last == &current) || (last == valid address space but wrong address)

  effect: valid but wrong data

  mitigation: _NEW_ unique magic number for *last data

- *last == NULL

  what happens: access to invalid memory, application terminated through health monitor, health monitor is expected to signal error.

  effect: restart of partition

  mitigation: check for null pointer, return ERROR_CODE

- *last == 0xDEAD (also known as pointer to lala-land)

  what happens: access to invalid memory, application terminated through health monitor, health monitor is expected to signal error.

  effect: restart of partition

  mitigation: component is in undefined state, internal mitigation not possible ⇒ has to be handled by health monitor.

- *to_controller == NULL

what happens: access to invalid memory, application terminated through health monitor, health monitor is expected to signal error.

effect: restart of partition

mitigation: check for null pointer, return ERROR_CODE

- *to_controller == valid address space but wrong address

  effect: no detection by system

  mitigation: extension of data structure by unique magic number.

- *to_controller == 0xDEAD (also known as pointer to lala-land)

  what happens: access to invalid memory, application terminated through health monitor, health monitor is expected to signal error.

  effect: restart of partition

  mitigation: component is in undefined state, internal mitigation not possible ⇒ has to be handled by health monitor.

- *to_monitor == NULL

  what happens: access to invalid memory, application terminated through health monitor, health monitor is expected to signal error.

  effect: restart of partition

  mitigation: check for null pointer, return ERROR_CODE

- *to_monitor == valid address space but wrong address

  what happens: access to valid memory, but wrong data.

  effect: no detection by system

  mitigation: extension of data structure by unique magic number.

- *to_monitor == 0xDEAD (also known as pointer to lala-land)

  what happens: access to invalid memory, application terminated through health monitor, health monitor is expected to signal error.

  effect: restart of partition

  mitigation: component is in undefined state, internal mitigation not possible ⇒ has to be handled by health monitor.

function:
```
StatusType write_output( \
        struct controller_data to_controller,\
        struct monitor_data to_monitor)
```

- to_controller and to_monitor are switched - different data types are recognized by the compiler. Compiler warning is assumed to be handled accordingly.

(D) **Severity without Mitigation**

Since all possible failure modes have been identified in Section C, it is now necessary to assign a severity to each possible failure mode. As this example is a generic component the severity analysis is restricted to a qualitative severity analysis of *negligible*, *tolerable* and *critical*, where *critical* would mean that the severity would have to be assessed against the actual deployment scenario (which is unknown here).

function: `StatusType read_input(struct in_data *input)`

- *input == NULL : **tolerable** – In this case the application segfaults, the application stops doing anything, which will quickly be discovered as a fault.
- *input == 0xDEAD (also known as pointer to lala-land) **critical**
- *input points to valid address space, but wrong data structure **critical**

function:
```
StatusType read_monitor_status( \
        enum flash_state *mon_status);
```

- pointer == NULL **tolerable**
- pointer to lala-land **critical**
- pointer to valid address space but wrong address **critical**

function:
```
StatusType pack_data(struct in_data current, \
        struct in_data *last, \
        struct flash_state mon_status, \
        struct controller_data *to_controller, \
        struct monitor_data *to_monitor);
```

- (*last == &current) || (last == valid address space but wrong address) **critical**
- last == NULL **tolerable**
- last is a pointer to lala-land **critical**
- *to_controller == NULL **critical**
- *to_controller == valid address space but wrong address **critical**
- *to_monitor == NULL **tolerable**
- *to_monitor == valid address space but wrong address **critical**
- *to_monitor is a pointer to lala-land **critical**

function:
```
StatusType write_output( \
        struct controller_data to_controller,\
        struct monitor_data to_monitor)
```

- to_controller and to_monitor are switched **critical**
- *to_monitor == NULL **critical**

(E) **Mitigation Methods**

Null Check – a typical method of defensive programming is to check every pointer passed to a function against NULL before it is used (usually right at the top of the function) and return an appropriate return value in case a NULL pointer has been passed to the function.

Escalate Memory Region Check to HM – XtratuM provides us a memory protection between applications. If a memory violation is detected this is signalled to the health monitor which can take appropriate action (e.g. to restart the application).

(F) **Corrective Design**

Magic Number – in order to assure that a memory area really contains the type of data structure that is expected by the application, a magic number unique to this type of data structures is added at the beginning of the memory are. If a wrong magic number is read, the function knows that a pointer to a wrong data structure has been passed by the caller.

SAC – problems that cannot (or only with a very high effort) be mitigate in the application itself are passed to the environment for handling. The way to do this are SACs (safety application conditions). The SACs identified for this application are listed in 6.4.2.1.

(G) This objective is DOD specific and not relevant in this context.

(H) **Severity with Mitigation**

function: `StatusType read_input(struct in_data *input)`

- pointer == NULL **negligible** - easy to mitigate, goes back into system (error code)
- pointer == 0xDEAD (also known as pointer to lala-land) **negligible** - easy to mitigate, goes back into system (error code)
- pointer points to valid address space, but wrong data structure **tolerable** - probability small enough due to 32 bit magic number!

function:
```
StatusType read_monitor_status( \
        enum flash_state *mon_status);
```

- pointer == NULL **negligible** - easy to mitigate, goes back into system (error code)
- pointer to lala-land **negligible** - easy to mitigate, goes back into system (error code)
- pointer to valid address space but wrong address

function:
```
StatusType pack_data(struct in_data current, \
        struct in_data *last, \
        struct flash_state mon_status, \
        struct controller_data *to_controller, \
        struct monitor_data *to_monitor);
```

- (*last == &current) || (last == valid address space but wrong address)
- last == NULL **negligible** - easy to mitigate, goes back into system (error code)
- pointer to lala-land **negligible** - easy to mitigate, goes back into system (error code)
- *to_controller == NULL **negligible** - easy to mitigate, goes back into system (error code)
- *to_controller == valid address space but wrong address
- *to_controller is a pointer to lala-land **negligible** - easy to mitigate, goes back into system (error code)

- *to_monitor == NULL **negligible** - easy to mitigate, goes back into system (error code)
- *to_monitor == valid address space but wrong address
- *to_monitor is a pointer to lala-land **negligible** - easy to mitigate, goes back into system (error code)

function:
```
StatusType write_output( \
          struct controller_data to_controller,\
          struct monitor_data to_monitor);
```

- to_controller and to_monitor are switched
- *to_monitor == NULL **negligible** - easy to mitigate, goes back into system (error code)

6.8 Design Summary

In conclusion it can be said, that no critical problems were found after the application of mitigations. Of course this does not mean that this design can be taken as is and expected to be ready for certification, as some of the objectives on the development process could not be fulfilled. This especially includes those steps in the development that should be a team effort (e.g. HAZOP) and the steps that require to be performed by a person other than and independent from the person who performed the step before (e.g. the person doing the FMEA should not be person who did the detailed design).

Ignoring the fact that lacking a team all the steps in the development were carried out single handedly, the example application can be considered safe as

- no more critical problems were found after the mitigations were applied
- a number of SACs has been found and collected to make the system integrator aware that it is his responsibility to ensure that those problems that cannot be mitigated in software are handled outside of the application
- the mitigations for found problems have been proposed, so far they have not been applied, this would be the next step - to update the detailed design according to findings in the FMEA.

Thus the turn indicator control application designed in this chapter can be considered safe, while still being independent of its environment. The only requirements on the environment are:

- an OSEK/VDX compliant runtime environment - no matter whether it is in virtualized environment on directly running on a hardware node
- the list of SACs found during development have to be fulfilled to ensure safety properties.

CHAPTER 7

Conclusion

The goal of this thesis was to analyze the safety properties of the OVERSEE platform, which was developed to suite the security needs of the automotive industry while the equally important safety properties were de-scoped.

7.1 Summary

There are many reasons why such a platform that provides a high level of safety and security is needed in the industry, some were given in Chapter 1. Those reasons basically boil down to the demand for new features by all parties - the end user, the manufacturer as well as the motor clubs. These desired new features include e.g. comfort braking, V2V and V2I communication, parking assistance and they lead to a tremendous increasing in complexity of the in-vehicle programmable electronics [1]. The industries question is, whether or not this demands can be (partially) satisfied by using legacy applications or FLOSS components, and if they are able to satisfy the functional requirements - what adoptions in the safety process are needed to assure the safe usage of legacy and FLOSS components?

Before a thorough analysis is possible, the appropriate methods and industry standards have to be found. The concepts and methods that are relevant throughout this thesis, like a discussion on operating system architectures, an introduction to virtualization and especially virtualization in the safety domain, as well as design and implementation methods (e.g. GSN, RTSAD) are considered in Chapter 2.

In the safety area one of the most important sources for guidance is given to engineers by well established industry standards, therefore the whole Chapter 3 is dedicated to this topic and gives an introduction to ARINC 653 as an operating system specification that introduces the concept of virtualization and OSEK/VDX as an operating system specification widely used in the automotive industry. Furthermore, as important functional safety standards IEC 61508 and ISO 26262 are introduced. Although the latter can not be seen as a well established standard

yet[1], it is based on the principles of IEC 61508 and the high quality standards of the automotive industry. Although standards give developers some guidance, they are no cookbooks, so the challenge of this thesis was the interpretation of the used standards. This thesis represents one possible arguable interpretation.

The problem of analyzing the safety properties of the OVERSEE platform was approached on different levels, starting in chapter 4 with a high level safety case of the platform itself and validating the possibility of such an platform in the context of the relevant, industry specific functional safety standards. This analysis was conducted on the highest abstraction level of the platform, and structured not only into one but several GSNs. This was done in order to improve the maintainability as well as the reusability of the safety case. The idea behind this structuring, is to argue the safety of the system on different levels of abstraction. This way, if changes are necessary at a lower level of abstraction - lets say in the implementation - these changes only have an impact at the lower levels of the layered safety case, but those layers concerned with higher levels of abstraction (e.g. the requirements) stay untouched and do not have to be altered. With the increasing complexity of modern ECUs this reuse of previous work becomes more and more important to not only develop systems that are safe and secure but can be used for a long times and even on future (hardware) platforms.

In order to allow the reuse of applications on the code level, one other pre-requisite had to be achieved. This pre-requisite is the support of standardized interfaces used in the automotive industry. To show that such an interface can be provided, a prototypical OSEK OS compliant runtime environment has been established in Chapter 5. Even if this FreeOSEK runtime environment is only a prototype at the moment, the implementation shows the feasibility of running an OSEK compliant operating system as a XtratuM2 runtime environment. Furthermore, the theoretical mapping between OSEK compliant communication and the ARINC 653 compliant communication could be proven valid and compatible to the point where a legacy OSEK compliant application can be moved into a XtratuM runtime environment with a virtualized communication system replacing a legacy physical communications system, without the need of adapting the application itself. Using these virtual communication mechanisms only requires changes in the configuration files but one has to be careful and consideration and assessment of possible semantic differences between physical concurrency and pseudo-concurrency of the virtualized system will be necessary during the migration into the virtualized environment. Therefore this prototype shows that one of OVERSEEs main missions - the reuse of existing automotive applications in a security enhanced environment with minimum effort is definitely feasible. Thus this part of the thesis was successful in reaching its goal, although a major rework would be necessary to make it ready for certification and ultimately for the use in a road vehicle. This rework would need to address the XtratuM2 hypervisor, the FreeOSEK runtime environment as well as the security components provided by the GNU/Linux “SecIO“ partition.

On the application level - in order to value the economic significance of reuse of legacy applications in the automotive industry - the possibilities of modular certification were explored. It was shown that it is indeed possible to reach this goal of certified software modules, an example application was designed and analyzed in Chapter 6. Although the main goal of this chapter is

[1]ISO 26262 was released in November 2011.

to show the feasibility of designing and implementing applications for modular certification, a byproduct of this is a full example of designing and implementing an application for the safety domain. Appropriate examples are sometimes really hard to find and often they just take too many facts that just magically appear into account. The goal was to avoid that kind of annoyance in this Chapter.

7.2 Conclusion

With the introduction of functional safety into the automotive industry (the release of ISO 26262), new challenges for the automotive engineering sector have emerged. Partitioning has been demonstrated by the avionic industry to be an effective mitigation of economic an technical issues related to certification and the increasing dynamics of development. Binding the functional, procedural and regulatory demands together is the job of the safety case – a safety case plays a central role as it is the single point to store compliance information. At the same time this seems to be one of the problematic issues as a modular system building on generic components are intended for the reuse in different configurations. Thus, adaptations of the safety case process need to be made – not only outlining but executing this adapted process is one of the key contributions of this thesis.

Although arguments in safety cases can always be made more accurate and detailed, it is well known that safety case development is a complex task with many pitfalls along the way (see [Gre06] for some not so obvious and many very obvious examples), but the author is confident that the suggested layering of the safety case can be worth a lot. Structuring the safety case itself not only in an hierarchical fashion but also structuring it on different levels of abstraction increases maintainability as well as readability of the argument and will help to understand and find the problems and inaccuracies. Furthermore, a safety case is also just a component in a broader system and should be considered for reuse - appropriate structuring and layered abstractions can support this reuse significantly.

Designing a simple application for a partitioned hypervisor based system showed that this application has to be treated as a generic component, that cannot make any assumptions about its environment. To the contrary, it has to induce requirements (SACs) onto its environment in order to guarantee that the targeted safety integrity level will be reached even if the application is used in a new environment.

In essence this thesis shows that isolation of safety-related applications through well specified interfaces and a high-level compositional approach, as outlined in ARINC 653, are not only suitable to provide component based safety related systems, but that these components can be treated independently in the safety argumentation as well. This is in the authors opinion a long sought goal of automotive industry (and other industries as well) and is the key result of this diploma thesis: Compositional safety is effectively and efficiently possible based on well specified interfaces.

7.3 Future Work

Although this thesis tries to tackle the problem of showing that it is possible to build a platform that is as safe as well as secure, on different levels, It still leaves a lot of room for future work.

This includes the research and refinement of the safety case layering, as pointed out numerous times this approach needs to be tested in real world. Only that way the experience as well as the input from the authorities that is necessary to find the weak spots can be gained. Building on this information, adjustments of the method might be necessary in order to allow the safety managers using it to fully exploit its advantages.

Furthermore, this separated analysis of safety and security properties will need to be addressed, as a separated handling of the issues is neither economically nor technically optimal (if possible at all). With the increasing security demands on automotive systems, this will be a topic that will need a lot of work - not only by the manufacturers and component suppliers but also by the committees working on safety standards, in order to integrate the handling of security issues into the development processes of safety relevant components.

From a more practical point of view, the used software modules are prototypical implementations that are used to show the feasibility of the approach. In order to allow the integration into a product for mass production a lot of re-work will be necessary to bring it to the point where the high quality standards of the automotive industry are met.

Modularity of software has been firmly established in software development the next step is to establish equally potent methods for modularity of safety and security - at the technical and procedural level - the feasibility was successfully demonstrated with this thesis.

APPENDIX A

Detailed Design

```
enum indicator_state {IND_ON = 0, IND_OFF, IND_DEAD};
enum lever_state {LEV_NEUTRAL = 0, LEV_LEFT, LEV_RIGHT, LEV_DEAD};
/* Reference to MISRA-C Rule 9.3!! */
enum flash_state {NORMAL_FLASH = 0x55, DOUBLE_FLASH = 0xAA};

struct in_data {
        enum lever_state ls;
        enum indicator_state lis;
        enum indicator_state ris;
        uint16_t ang;
        bool emergency_button;
};

struct controller_data {
        uint16_t lst_ang;
        uint16_t cur_ang;
        enum lever_state lst_ls;
        enum lever_state cur_ls;
        enum indicator_state cur_lis;
        enum indicator_state lst_lis;
        enum indicator_state cur_ris;
        enum indicator_state lst_ris;
        bool emergency_button;
        enum flash_state health;
        uint16_t seq_nr;
        uint32_t crc;
};
```

```
struct monitor_data {
        uint16_t cur_ang;
        enum lever_state cur_ls;
        enum indicator_state cur_ris;
        bool emergency_button;
        enum flash_state health;
        struct timespec ts;
        uint16_t seq_nr;
        uint32_t crc;
};

StatusType read_input(struct in_data *input);
StatusType read_monitor_status(enum flash_state *mon_status);
StatusType pack_data(struct in_data current, struct in_data *last, \
        struct flash_state mon_status, \
        struct controller_data *to_controller, \
        struct monitor_data *to_monitor);
StatusType write_output(struct controller_data to_controller, \
        struct monitor_data to_monitor);
```

APPENDIX B

Code Examples

This appendix contains two very simple examples, that have been written to test the various steps that have been taken during the port for FreeOSEK on top of XtratuM. But apart from test-cases, they also can be seen as a representation of the current state of FreeOSEK on XtratuM. While each example usually only demonstrates the presence of a small piece of OSEK compliant functionality, the possible combinations already allow to write real-world OSEK compliant applications.

If you are interested in more test examples (e.g. communication) or an implementation of the indicator control example from chapter 6, don't hesitate to contact me. It is planned to release all code that has been produced in the course of this thesis under an open-source license, but at the time of this writing it is unclear how and where this code will be released.

B.1 xm_hello

The first simple example - as always - is a *Hello World!*. But in comparison to a classical hello world program it got much more to say. As you can see in B.1.1, the example consists of 6 tasks, one init task and tasks A-E. While all tasks are activated at start-up (`ACTIVATION = 1;`), only the init task is actually runnable (AUTOSTART = TRUE) at start-up.

The code for example is shown in B.1.2, and the first function in this application is called `PartitionMain()`. This is a mandatory function, it is the entry point for XtratuM into FreeOSEK, and is called before the initialization of the OS. Indeed you can see the call of the function `StartOS()`, which is the entry point to the initialization of FreeOSEK. After the initialization of FreeOSEK, the init task is scheduled, and the flow of execution never returns into `PartitionMain()`.

Below of that you can see the code for the `InitTask()` which does nothing but print the message `Hello World!` on the screen and activate `TaskA()` by using the `ActivateTask()` OSEK API call. After that it terminates itself.

The tasks `TaskA .. TaskE` then chain each other in an infinite loop, printing their name onto the screen whenever they are called.

This example demonstrates that non-preemptive scheduling works without problems - that means no stack-overflows or that kind of problems.

B.1.1 Configuration File

```
OSEK OSEK {

OS ExampleOS {
        STATUS = EXTENDED;
        STARTUPHOOK = FALSE;
        ERRORHOOK = FALSE;
        SHUTDOWNHOOK = FALSE;
        PRETASKHOOK = FALSE;
        POSTTASKHOOK = FALSE;
        MEMMAP = FALSE;
        USERESSCHEDULER = FALSE;
};

TASK InitTask {
        PRIORITY = 1;
        SCHEDULE = NON;
        ACTIVATION = 1;
        AUTOSTART = TRUE {
                APPMODE = AppMode1;
        };
        STACK = 1024;
        TYPE = EXTENDED;
};

APPMODE AppMode1;

TASK TaskA {
        PRIORITY = 2;
        SCHEDULE = NON;
        ACTIVATION = 1;
        AUTOSTART = FALSE;
        STACK = 1024;
        TYPE = EXTENDED;
};

TASK TaskB {
```

```
        PRIORITY = 3;
        SCHEDULE = NON;
        ACTIVATION = 1;
        AUTOSTART = FALSE
        STACK = 1024;
        TYPE = EXTENDED;
};

TASK TaskC {
        PRIORITY = 4;
        SCHEDULE = NON;
        ACTIVATION = 1;
        AUTOSTART = FALSE
        STACK = 1024;
        TYPE = EXTENDED;
};

TASK TaskD {
        PRIORITY = 5;
        SCHEDULE = NON;
        ACTIVATION = 1;
        AUTOSTART = FALSE
        STACK = 1024;
        TYPE = EXTENDED;
};

TASK TaskE {
        PRIORITY = 6;
        SCHEDULE = NON;
        ACTIVATION = 1;
        AUTOSTART = FALSE
        STACK = 1024;
        TYPE = EXTENDED;
};

COUNTER HardwareCounter {
        MAXALLOWEDVALUE = 100000;
        TICKSPERBASE = 1000;
        MINCYCLE = 1;
        TYPE = HARDWARE;
        COUNTER = HWCOUNTER0;
};
};
```

B.1.2 Source Code

```
/** \brief main function
 **
 ** Project main function. This function is called after
 ** the c conformance initialisation. This function shall call
 ** StartOS ()
 **/
void PartitionMain
(
        void
)
{
        /* Start OSEK */
        StartOS(AppMode1);

        /* never reached: */
        return;
}

/** \brief Init Task
 **
 ** This task is called one time after every reset and takes care of
 ** the system initialization.
 **/
TASK(InitTask)
{
        int i;

        XM_write_console("Hello World!\n", 13);

        /* Terminate Init Task */
        ActivateTask(TaskA);
        TerminateTask();
}

TASK(TaskA)
{
        XM_write_console("taskA\n", 6);
        ChainTask(TaskB);
}
```

```
TASK(TaskB)
{
        XM_write_console("taskB\n", 6);
        ChainTask(TaskC);
}

TASK(TaskC)
{

        XM_write_console("taskC\n", 6);
        ChainTask(TaskD);
}

TASK(TaskD)
{
        XM_write_console("taskD\n", 6);
        ChainTask(TaskE);
}

TASK(TaskE)
{
        XM_write_console("taskE\n", 6);
        ChainTask(TaskA);
}
```

B.2 xm_timer

The second example demonstrates the usage of counters and timers in an OSEK compliant environment. The example consists of two simple tasks, and `InitTask` doing the initialization - in this example configuring the timer - and `TaskA` the task that is periodically triggered by the timer. Both of these tasks are configured in the configuration file. Furthermore the configuration file contains the configuration of the counters (`HardwareCounter` and `SoftwareCounter`), as well as the alarms (`IncrementSWCounter` and `ActivateTaskA`).

As stated previously, the sources for incrementing counters can be all sorts of things, ranging from timers to network cards to software events or explicit calls by the application. In this case the `HardwareCounter` is bound to the timer interface provided by XtratuM, if the timer goes off, the `HardwareCounter` is incremented. When it overflows the `IncrementSWCounter` Alarm is triggered, which in turn increments the `SoftwareCounter`. As soon as the `SoftwareCounter` overflows, the `ActivateTaskA` alarm is triggered, and this alarm activates TaskA which is executed once every time alarm `ActivateTask` expires.

B.2.1 Configuration File

```
OSEK OSEK {

OS ExampleOS {
STATUS = EXTENDED;
STARTUPHOOK = FALSE;
ERRORHOOK = FALSE;
SHUTDOWNHOOK = FALSE;
PRETASKHOOK = FALSE;
POSTTASKHOOK = FALSE;
MEMMAP = FALSE;
USERESSCHEDULER = FALSE;
};

TASK InitTask {
PRIORITY = 1;
SCHEDULE = NON;
ACTIVATION = 1;
AUTOSTART = TRUE {
APPMODE = AppMode1;
};
STACK = 1024;
TYPE = EXTENDED;
};

TASK TaskA {
PRIORITY = 2;
SCHEDULE = NON;
ACTIVATION = 1;
AUTOSTART = FALSE;
STACK = 1024;
TYPE = EXTENDED;
};

APPMODE AppMode1;

COUNTER HardwareCounter {
MAXALLOWEDVALUE = 100000;
TICKSPERBASE = 1000;
MINCYCLE = 1;
TYPE = HARDWARE;
COUNTER = HWCOUNTER0;
```

```
};

COUNTER SoftwareCounter {
MAXALLOWEDVALUE = 100000;
TICKSPERBASE = 100;
MINCYCLE = 1;
TYPE = SOFTWARE;
};

ALARM IncrementSWCounter {
COUNTER = HardwareCounter;
ACTION = INCREMENT {
COUNTER = SoftwareCounter;
};
AUTOSTART = TRUE {
APPMODE = AppMode1;
ALARMTIME = 1;
CYCLETIME = 1;
};
};

ALARM ActivateTaskA {
COUNTER = SoftwareCounter;
ACTION = ACTIVATETASK {
TASK = TaskA;
}
AUTOSTART = FALSE;
};
};
```

B.2.2 Source Code

```
/** \brief main function
 **
 ** Project main function. This function is called after
 ** the c conformance initialisation. This function shall call
 ** StartOS ()
 **/
void PartitionMain
(
        void
)
{
```

```
        /* Start OSEK */
        StartOS(AppMode1);

        /* never reached: */
        return;
}

/** \brief Init Task
 **
 ** This task is called one time after every reset and takes care of
 ** the system initialization.
 **/
TASK(InitTask)
{
        int i;

        SetRelAlarm(ActivateTaskA, 2689, 2689);

        /* Terminate Init Task */
        TerminateTask();
}

TASK(TaskA)
{
        XM_write_console("taskA\n", 6);
        TerminateTask();
}
```

APPENDIX C

Papers in the Context of this Thesis

The following is a list of papers that have been published during the course of this thesis.

OVERSEE - a generic FLOSS communication and application platform for vehicles. *A*ndreas Platschek, Nicholas Mc Guire and Georg Schiesser, The 12^{th} Real-Time Linux Workshop in Nairobi, Kenya, 2010

Linux as a Real-Time Hypervisor for the Automotive Industry *A*ndreas Platschek, Nicholas Mc Guire and Georg Schiesser, Embedded World Conference in Nürnberg, Germany, 2011

Migrating a OSEK run-time environment to the OVERSEE platform *A*ndreas Platschek and Georg Schiesser, The 13^{th} Real-Time Linux Workshop in Prague, Czech Republic, 2011

Design and Implementation of a Safety-Critical Application Targeting Modular Certification *A*ndreas Platschek and Nicholas Mc Guire, The 14^{th} Real-Time Linux Workshop in Chapel Hill, USA, 2012

Abbreviations

ABS Anti Blocking System

API Application Programming Interface

ARINC Aeronautical Radio Incorporated

AUTOSAR AUTomotive Open System ARchitecture

BCC Basic Conformance Class (OSEK/VDX)

BT Basic Task(s) (OSEK/VDX)

CC Common Criteria

CENELEC European Committee for Electrotechnical Standardization

CNC Computer Numeric Controlled

COTS Common Off the Shelf

CPU Central Processing Unit

DOD Departement of Defense

EAL Evaluation Assurance Level

ECU Error Containment Unit

ECC Extended Conformance Class (OSEK/VDX)

E/E/PE Electric/Electronic/Programmable Electronic

ET Extended Task(s) (OSEK/VDX)

FCU Fault Containment Unit

FLOSS Free/Libre Open-Source Software

FMEA Failure Mode and Effects Analysis

FSF Free Software Foundation

GPL General Public License

GNU GNU's not Unix

GSN Goal Structured Notation

HPET High Precision Event Timer

HAZOP Hazard and Operability Study

IMA Integrated Modular Avionics

IEC International Electrotechnical Commission

IPC Inter Process Communication

ISO International Organization for Standardization

ISR Interrupt Service Routine

NooM N out of M

OBD On-Board Diagnosis

OSEK Offene Schnittstellen für Elektronik in Kraftfahrzeugen

POSIX Portable Operating System Interface

PP Protection Profile

RTE Runtime Environment

RTSAD Real-Time Structured Analysis and Design

SFR Security Functional Requirements

SIL Safety Integrity Level

ST Security Target

STD State Transition Diagram

TCB Trusted Code Base

TMR Tripple Modular Redundancy

TOE Target of Evaluation

TPMS Tire Pressure Monitoring System

V2I Vehicle(s) to Infrastructure Communication

V2V Vehicle(s) to Vehicle(s) Communication

XEF Xtratum Executable Format

Bibliography

[Ada03] Charlotte Adams. A380 Innovations: A Balancing Act. *Avionics Magazine*, 1. March 2003.

[AGS08] Heinz Kantz Andreas Gerstinger and Christoph Scherrer. TAS Control Platform: A Platform for Safety-Critical Railway Applications . ERCIM News, 75 (2008), p. 49 - 50., October 2008.

[BHF+92] Alan C. Bomberger, Norman Hardy, A. Peri Frantz, A. Peri, William S. Frantz, Charles R. Landau, William S. Frantz, Jonathan S. Shapiro, and Ann C. Hardy. The KeyKOSreg; Nanokernel Architecture. In *Proc. of the USENIX Workshop on Microkernels and Other Kernel Architectures*, pages 95–112, 1992.

[CBW07] Randy Walter Christopher B. Watkins. Transitioning from Federated Avionics Architectures to Integrated Modular Avionics. *GE Aviation*, 2007.

[Com03] Airlines Electronic Engineering Commitee. ARINC653 – AVIONICS APPLICATION SOFTWARE STANDARD INTERFACE, October 2003.

[Con99a] OSEK/VDX Consortium. Binding Specification. `http://portal.osek-vdx.org/files/pdf/specs/binding142.pdf`, 1999.

[Con99b] OSEK/VDX Consortium. OS Test Plan. `http://portal.osek-vdx.org/files/pdf/modistarc/ostestplan20.pdf`, 1999.

[Con99c] OSEK/VDX Consortium. OSEK Implementation Language (OIL). `http://portal.osek-vdx.org/files/pdf/specs/oil25.pdf`, 1999.

[Con99d] OSEK/VDX Consortium. OSEK Network Management. `http://portal.osek-vdx.org/files/pdf/specs/nm253.pdf`, 1999.

[Con01] OSEK/VDX Consortium. OSEK Time Triggered Operating System. `http://portal.osek-vdx.org/files/pdf/specs/ttos10.pdf`, 2001.

[Con04] OSEK/VDX Consortium. OSEK Communication Specification 3.0.3. `http://portal.osek-vdx.org/files/pdf/specs/osekcom303.pdf`, 2004.

[Con05] OSEK/VDX Consortium. OSEK Operating System Specification 2.2.3. http://portal.osek-vdx.org/files/pdf/specs/os223.pdf, 2005.

[Con10] OVERSEE Consortium. Specification of Secure Communication. https://www.oversee-project.com/fileadmin/oversee/deliverables/D2-4_Specification_of_Secure_Communication_v1_5.pdf, 3. June 2010.

[Coo03] Jim Cooling. *Software Engineering for Real-Time Systems*. Addison Wesley, first edition edition, 2003.

[DD77] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.

[DeM81] Tom DeMarco. *Structured Analysis and System Specification*. Yourdon P.,U.S., april 1981 edition, 1981.

[DoD80] United States of America Departement of Defense. Military Standard 1629A, Procedures for Performing a Failure Mode, Effects and Criticallity Analysis. 24. November 1980.

[Eng98] Dawson R. Engler. The Exokernel Operating System Architecture. http://pdos.csail.mit.edu/exo/theses/engler/thesis.ps, 18. May 1998.

[GKH09] Amanda McPherson Greg Kroah-Hartman, Jonathan Corbet. Linux Kernel Development. http://www.linuxfoundation.org/publications/whowriteslinux.pdf, 1.August 2009.

[Goo01] Hassan Gooma. *Software Design Methods for Concurrent and Real-Time Systems*. Addison Wesley, fifth printing edition, 2001.

[Gre06] William S. Greenwell. A taxonomy of fallacies in system safety arguments. In *Proceedings of the 2006 International System Safety Conference*, 2006.

[Hat94] Les Hatton. *Safer C: Developing Software for High-Integrity and Safety-Critical Systems*. Mcgraw-Hill Professional, first edition edition, 1994.

[IEC10a] IEC. 61508-0, Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 0: Functional safety and IEC 61508. April 2010.

[IEC10b] IEC. 61508-1, Part 1: General requirements. April 2010.

[IEC10c] IEC. 61508-2, Part 2: Requirements for electrical/electronic/programmable electronic safety-related systems. April 2010.

[IEC10d] IEC. 61508-3, Part 3: Software requirements. April 2010.

[IEC10e] IEC. 61508-4, Part 4: Definitions and abbreviations. April 2010.

[IEC10f] IEC. 61508-5, Part 5: Examples of methods for the determination of safety integrity levels. April 2010.

[IEC10g] IEC. 61508-6, Part 6: Guidelines on the application of IEC 61508-2 and IEC 61508-3. April 2010.

[IEC10h] IEC. 61508-7, Part 7: Overview of techniques and measures. April 2010.

[Ism08] Baurzhan Ismagulov. Linux in Safety-Critical Systems. `https://www.osadl.org/fileadmin/dam/presentations/Linux-in-Safety-Critical-Systems/20080228-Siemens-Certification.pdf`, 2008.

[KCR+10] Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, and Stefan Savage. Experimental security analysis of a modern automobile. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 447–462, Washington, DC, USA, 2010. IEEE Computer Society.

[KM98] Tim Kelly and John McDermid. Safety Case Patterns - Reusing Successful Arguments. In *Proceedings of IEE colloquium on understanding patterns and their application to system engineering*, 1998.

[KM07] Niels Klußmann and Arnim Malik. *Lexikon der Luftfahrt*. Springer Berlin Heidelberg, May 2007.

[KS92] Marilyn Keller Ken Shumate. *Software Specification and Design - A Disciplined Approach for Real-Time Systems*. Wiley, first printing edition, 1992.

[Lie93] Jochen Liedtke. Improving IPC by Kernel Design. In *14th ACM Symposium on Operating System Prinicples (SOSP), Asheville North Carolina*, 1993.

[Lie94] Jochen Liedtke. On μ-Kernel Construction. In *15th ACM Symposium on Operating System Prinicples (SOSP), Copper Mountain Resort, Colorado*, 1994.

[Lie96] Jochen Liedtke. Toward Real Microkernels. In *Communications of the ACM Vol.39*, 1996.

[Mar06] George Marsh. Europe's Vision of Future Avionics. *Avionics Magazine*, 1. June 2006.

[Min00a] Ministry of Defense. HAZOP Studies on Systems Containing Programmable Electronics, Part 1 Issue 2. `tee.uq.edu.au/~elec7500/00-58%20part%201%20issue%202.pdf`, 1. May 2000.

[Min00b] Ministry of Defense. HAZOP Studies on Systems Containing Programmable Electronics, Part 2 Issue 2. `cs.anu.edu.au/student/comp4100/lectures/DEF-STAN-58.pdf`, 1. May 2000.

[MIS04] MISRA. MISRA-C:2004 Guidelines for the use of the C language in critical systems. October 2004.

[MK00] David L. Mills and Poul-Henning Kamp. The Nanokernel. http://www.eecis.udel.edu/~mills/database/papers/nano/nano2.pdf, 2000.

[MR11] Miguel Masmano and Ismael Ripoll. XtratuM Hypervisor for INTEL x86 - Volume 4: Reference Manual. June 2011.

[MRC11] Miguel Masmano, Ismael Ripoll, and Alfons Crespo. XtratuM Hypervisor for INTEL x86 Volume 2: User Manual. March 2011.

[OCYL11] on behalf of the Contributors Origin Consulting (York) Limited. GSN COMMUNITY STANDARD VERSION 1. http://www.goalstructuringnotation.info/documents/GSN_Standard.pdf, November 2011.

[PA08] et al Philippe Aigrain. 2020 FLOSS Roadmap. http://www.2020flossroadmap.org/docs/OWF_2020_Roadmap%20v2.18-3.pdf, 2008.

[PS08] Detlef John Peter Sieverding. Sicas ECC – Die Plattform für Siemens-ESTWs für den Nahverkehr. In *Signal + Draht 5/2008*, 2008.

[Ram07] James W. Ramsey. Integrated Modular Avionics: Less is More. *Avionics Magazine*, 1. February 2007.

[RMM+10] Ishtiaq Rouf, Rob Miller, Hossen Mustafa, Travis Taylor, Sangho Oh, Wenyuan Xu, Marco Gruteser, Wade Trappe, and Ivan Seskar. Security and privacy vulnerabilities of in-car wireless networks: a tire pressure monitoring system case study. In *Proceedings of the 19th USENIX conference on Security*, USENIX Security'10, pages 21–21, Berkeley, CA, USA, 2010. USENIX Association.

[Rus81] John Rushby. Design and Verification of Secure Systems. In *8th ACM Symposium on Operating Systems Principles*, 14. December 1981.

[Rus82] John Rushby. Proof of Seperability - A Verificaton Technique for a Class of Securtiy Kernels. In *Springer Verlag LNCS No. 137, pp. 352-367*, 1982.

[Rus99] John Rushby. Partitioning in Avionics Architectures: Requirements, Mechanisms and Assurance. In *SRI International*, 1. June 1999.

[Spi05] Cary Spitzer. Perspectives: Reusable Software in Integrated Avionics. *Avionics Magazine*, 1. April 2005.

[STA11a] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. ISO 26262: Road vehicles — Functional safety. November 2011.

[STA11b] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. ISO 26262: Road vehicles — Functional safety; Part 10: Guideline. November 2011.

[Sum96] Steve Summit. *C Programming FAQs: Frequently Asked Questions*. Addison Wesley Pub Co Inc, 1st revised edition edition, 1996.

[Wat06a] Christopher B. Watkins. Integrated Modular Avionics: Managing the Allocation of Shared Intersystem Resources. *Smiths Aerospace LLC*, 2006.

[Wat06b] Christopher B. Watkins. Modular Verification: Testing a subset of Interated Modular Avionics in Isolation. *Smiths Aerospace LLC*, 2006.

[WGA08] WGA11 of Subcommittee CENELEC SC9XA. EN 50128: Railway applications - Communications, signalling and processing systems - Software for railway control and protection systems. April 2008.

[Ye05] Fan Ye. Justifying the Use of COTS Components within Safety Critical Applications. In *Dissertation, University of York - Department of Computer Science*, September 2005.

Internet References

[1] Robert N. Charette. This Car Runs on Code. http://spectrum.ieee.org/green-tech/advanced-cars/this-car-runs-on-code/ [Acccessed: July 16th, 2013].

[2] Wikipedia. Google driverless car. http://en.wikipedia.org/wiki/Google_driverless_car [Acccessed: July 16th, 2013].

[3] OVERSEE consortium. *Homepage of the OVERSEE Project*. www.oversee-project.com [Acccessed: July 16th, 2013].

[4] NASA. NASA Open Source Software. http://opensource.arc.nasa.gov/ [Acccessed: July 16th, 2013].

[5] Linus Torvalds and Andrew S. Tanenbaum. Linus vs. Tanenbaum. http://groups.google.com/group/comp.os.minix/browse_thread/thread/c25870d7a41696d2/f447530d082cd95d?tvc=2 [Acccessed: July 16th, 2013].

[6] Wikipedia. Ring (CPU). http://de.wikipedia.org/wiki/Ring_%28CPU%29 [Acccessed: July 16th, 2013].

[7] Neil Brown. Object-oriented design patterns in the kernel, part 1. https://lwn.net/Articles/444910/ [Acccessed: July 16th, 2013].

[8] Neil Brown. Object-oriented design patterns in the kernel, part 2. https://lwn.net/Articles/446317/ [Acccessed: July 16th, 2013].

[9] *philipsu*. Broken Windows Theory. http://blogs.msdn.com/philipsu/archive/2006/06/14/631438.aspx [Acccessed: July 16th, 2013].

[10] Google. Android webpage. http://www.android.com [Acccessed: July 16th, 2013].

[11] MeeGo community. MeeGo webpage. http://www.meego.com [Acccessed: July 16th, 2013].

[12] Canonical. Ubuntu Webpage. http://www.ubuntu.com [Acccessed: July 16th, 2013].

[13] RedHat Inc. Redhat Webpage. http://www.redhat.com [Acccessed: July 16th, 2013].

[14] Top 500. Top 500 SuperComputers. http://www.top500.org [Acccessed: July 16th, 2013].

[15] Inc. Free Software Foundation. Hompage of GNU/HURD. http://www.gnu.org/software/hurd/ [Acccessed: July 16th, 2013].

[16] NICTA Australia's ICT Research Centre of Excellence. The L4.verified project - A Formally Correct Operating System Kernel. http://www.ertos.nicta.com.au/research/l4.verified [Acccessed: July 16th, 2013].

[17] Gernot Heiser. Microkernel, Nanokernel - what's the difference? http://www.ok-labs.com/blog/entry/microkernel-nanokernel-whats-the-difference/ [Acccessed: July 16th, 2013].

[18] wikipedia. CP/CMS - a time-sharing operating system of the late 60s and early 70s. http://en.wikipedia.org/wiki/CP/CMS [Acccessed: July 16th, 2013].

[19] Oracle. VirtualBox Homepage. http://www.virtualbox.org [Acccessed: July 16th, 2013].

[20] Fabrice Bellard. QEMU Homepage. http://www.qemu.org [Acccessed: July 16th, 2013].

[21] KVM. KVM - Kernel Based Virtual Machine. http://www.linux-kvm.org [Acccessed: July 16th, 2013].

[22] Xen. XEN Homepage. http://www.xen.org/ [Acccessed: July 16th, 2013].

[23] Universidad Politécnica de Valencia (Spain). XtratuM - Homepage. http://www.xtratum.org/ [Acccessed: July 16th, 2013].

[24] Amit Singh. A Taste of Computer Security. http://www.kernelthread.com/publications/security/sandboxing.html [Acccessed: July 16th, 2013].

[25] Julien Delange. POK - A Partitioned Operating System. http://pok.safety-critical.eu [Acccessed: July 16th, 2013].

[26] Windriver. VxWorks. http://www.windriver.com/products/vxworks/ [Acccessed: July 16th, 2013].

[27] Greenhill. Hompage Greenhill Integrity. http://www.ghs.com/products/safety_critical/integrity-do-178b.html [Acccessed: July 16th, 2013].

[28] AUTOSAR. AUTOSAR - Automotive Open System Architecture. http://autosar.org/ [Acccessed: July 16th, 2013].

[29] John Regher. A Guide to Undefined Behavior in C and C++. http://blog.regehr.org/archives/213 [Acccessed: July 16th, 2013].

[30] LLVM Blog. What Every C Programmer Should Know About Undefined Behavior. http://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html [Acccessed: July 16th, 2013].

[31] FSF Free Software Foundation. GNU make Homepage. http://www.gnu.org/software/make/ [Acccessed: July 16th, 2013].

[32] FSF Free Software Foundation. GNU Compiler Collection Homepage [acccessed: July 16th, 2013]. http://gcc.gnu.org/.

[33] Scott Chacon. GIT - The Fast Version Control System. http://git-scm.com/ [Accessed: July 16th, 2013].

[34] Doxygen. Doxygen Homepage. http://www.doxygen.org/ [Acccessed: July 16th, 2013].

[35] Julia Lawall. Coccinelle: A Program Matching and Transformation Tool for Systems Code. http://coccinelle.lip6.fr/ [Acccessed: July 16th, 2013].

[36] OSEK/VDX consortium. OSEK/VDX Homepage. http://osek-vdx.org/ [Accessed: July 16th, 2013].

[37] Yutaka Matsuno. D-Case Editor – A Typed Assurance Case Editor. http://www.il.is.s.u-tokyo.ac.jp/deos/dcase/ [Acccessed: July 16th, 2013].

[38] Mariano Cerdeiro. FreeOSEK Project Webpage. http://opensek.sourceforge.net/ [Acccessed: July 16th, 2013].