Die approbierte Originalversion dieser Diplom-/Masterarbeit ist in der Hauptbibliothek der Technischen Universität Wien aufgestellt und zugänglich.
http://www.ub.tuwien.ac.at

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology.
http://www.ub.tuwien.ac.at/eng

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Reliable devices for safe communication in networks

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Markus Klein
Matrikelnummer 0726101

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Wolfgang Kastner
Mitwirkung: Univ.Ass. Dipl.-Ing. Dr.techn. Lukas Krammer, Bakk.techn.

Wien, 29.09.2015

(Unterschrift Markus Klein) (Unterschrift Betreuung)

Erklärung zur Verfassung der Arbeit

Markus Klein
Spaunstraße 101, 4020 Linz

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 Markus Klein)

Danksagung

"Manch Ding braucht viel Weile" und so hat sich auch die vorliegende Master-Arbeit über einen viel zu langen Zeitraum erstreckt. Diese Zeit, geprägt von universitärer und außeruniversitärer Arbeit, in der die eigene Master-Arbeit nur allzu oft liegen blieb und das wissenschaftliche Arbeiten aus den täglichen Gedanken verdrängte, war von vielen Tiefen durchwachsen, wie keine andere Zeit in meinem bisherigen Leben. So manchen Selbstzweifel galt es zu überwinden und Motivation zu finden, war wahrlich nicht leicht. Trotz allem habe ich mein Ziel diese Arbeit zu einem Abschluss zu bringen, nie aufgegeben.
Dies alleine zu bewerkstelligen, wäre allerdings ein Ding der Unmöglichkeit gewesen.

Ich bedanke mich daher aufrichtigst bei meinen Eltern, die mich über die lange Zeit des zähen Werdens dieser Arbeit, aber auch im gesamten bisherigen Lebensweg, in allen Belangen immer unterstützt haben und mir, mit weisen Worten und aus eigener Erfahrung sprechend, immer wieder Mut gemacht haben und immer an mich geglaubt haben. Leider gelang es mir nicht immer die Hilfestellungen in jenem Maß anzunehmen, wie es notwendig gewesen wäre.
Danke, dass ihr mich meinen Weg gehen habt lassen und immer als Wegweiser zur Verfügung standet, wenn ich ihn brauchte.

Ich danke meiner lieben Katharina für das Durchhaltevermögen und das Verständnis für meine so zahlreichen Stunden und Abende, die ich vor dem Computer für meine Vielzahl an Tätigkeiten zubrachte. Danke, dass du mich trotzdem immer wieder in die Arme schließt und mit mir den Weg gehst.

Ein großes Dankeschön geht an meine Betreuer Lukas Krammer und Wolfgang Kastner, die trotz dieser langen Zeit und den vielen Pausen nicht müde geworden sind, meine Arbeit zu einem guten Ende zu begleiten.

Schlussendlich möchte ich meinen Freunden Johannes und Gerald für ihr Entgegenkommen danken. Den Freiraum und das Vertrauen zu bekommen, neben den gemeinsamen Projekten noch seine Ausbildung unterzubringen, ist nicht selbstverständlich. Aus unseren erfolgreichen Projekten konnte ich immer wieder positive Energie schöpfen.

Vielen Dank!
Markus

Abstract

Embedded systems are part of our daily life and they pervade almost all areas of living. Mobile phones, cars, debit cards, medical equipment, energy meters and the intelligent fridge are just a few examples. These embedded systems also come in a variety of designs, from tiny single chip applications to wide-spread distributed solutions.

All these systems have one thing in common - some may call it even a requirement - they have to be able to communicate. Communication is the key-factor, no matter if this communication happens within a system or with other connected systems or even with the environment. Integration is getting tighter and tighter in any case and demand for even more integration is never ending.

Another demand these systems share is the fact that those system are getting cheaper. Obviously, the manufacturers of these systems have to reduce costs, with the consequence that safety-margin in the production are reduced to save material. This results mostly in less lifetime of products and also error-proneness is rising. The overall reliability is therefore getting worse.

A small part of those networked embedded systems though plays a special role. These are those systems which actually care about human safety. Typically those safety systems are not very visible to the average person as they work rather in the background. They are to be found in cars, airplanes, the processing-industry or the fire-safety domain. Clearly, this special role implies some additional requirements those systems have to fulfil.

This thesis takes a closer look at communication and reliability of such safety-critical embedded systems, taking the fire-safety domain as a reference use case. System requirements within this domain combine all properties mentioned above in a special way: frequent communication over longer distances, low-priced hardware, but high reliability demands.

The thesis suggests a combined software- and hardware-framework, which improves reliability (and therefore also communication) of the devices in smaller embedded systems, which are connected in a larger network while keeping the necessary hardware costs low. The requirements in the fire-safety domain are regulated in various standards, which will be explained and used to describe the challenge of constructing such systems.

Kurzfassung

Embedded Systems sind aus unserem täglichen Leben nicht mehr wegzudenken. Sie durchdringen beinahe sämtliche Bereiche unseres Lebens. Mobiltelefon, Auto, Bankomatkarte, Herzschrittmacher, Smartmeter und der intelligente Kühlschrank sind nur einige Beispiele. Diese embedded Systems sind in einer Vielfalt an Ausprägungen zu finden, von kleinen Einzel-Chip Anwendungen bis zu weiträumig verteilten Lösungen.

Alle diese Systeme haben eine Sache gemeinsam - manche würden es sogar eine Voraussetzung nennen: sie müssen in der Lage sein zu kommunizieren. Kommunikation ist ein Schlüsselfaktor, unabhängig davon, ob diese Kommunikation innerhalb eines Systems stattfindet oder ob mit anderen Systemen oder sogar mit der Umwelt kommuniziert wird. Die Integration wird jedenfalls immer dichter und dichter, und die Nachfrage nach mehr Integration steigt stetig.

Eine Eigenheit, die diese Systeme teilen, ist die Tatsache, dass sie immer billiger werden sollen. Notwendigerweise müssen Hersteller solcher Systeme daher Kosten einsparen, mit der Konsequenz, dass der Sicherheitszuschlag in der Produktion zunehmend geringer wird, um Material zu sparen. Daraus resultiert meist eine geringere Lebenszeit dieser Produkte, aber auch die Fehleranfälligkeit steigt. Daher nimmt die Zuverlässigkeit ab.

Ein kleiner Teil dieser embedded Systems spielt eine spezielle Rolle. Das sind jene Systeme, die für die Sicherheit von Menschen sorgen. Typischerweise sind solche Systeme im Normalfall für den Durschnittsbürger nicht sonderlich sichtbar, da sie eher im Hintergrund arbeiten. Solche Systeme lassen sich in Autos, Flugzeugen, der Prozess-Industrie oder im Brandmelde-Bereich finden. Naheliegend ist daher auch, dass solche Systeme zusätzliche Anforderungen erfüllen müssen.

Die Arbeit fokussiert auf die Kommunikation und Zuverlässigkeit von Geräten in solchen sicherheitskritischen embedded Systems, wobei der Feuersicherheits-Bereich als Referenz-Beispiel herangezogen wird. Die Systemvoraussetzungen aus diesem Bereich kombinieren alle zuvorgenannten Eigenschaften auf spezielle Weise: häufige Kommunikation über längere Distanzen, günstige Hardware, aber hoher Zuverlässigkeitsbedarf.

Diese Arbeit stellt eine kombinierte Software- und Hardware-Grundstruktur vor, die die Zuverlässigkeit (und damit auch die Kommunikation) von Geräten in kleineren embedded Systems verbessert, die in einem größeren Netzwerk miteinander verbunden sind und dessen Hardwarekosten niedrig sind. Die Anforderungen im Feuersicherheits-Bereich sind in mehreren Normen geregelt, die vorgestellt und verwendet werden, um die Herausforderungen der Konstruktion solcher Systeme zu beschreiben.

Contents

CHAPTER 1

Introduction

The human need for safety and security shows its characteristics more than ever. Feeling safe and secure is a salving mood and tends to relieve many sorrows, although it is obvious that a 100% safety and security can never be achieved. Safety is also always coupled with trust. One has to trust a system for being and feeling safe, but nothing is worse than being lulled into a false sense of safety and security.

Safety can have a variety of forms, such as:

- The safety of a tether
- The safety of driving a well constructed vehicle
- The safety of a machine to not cause harmful accidents
- The safety of governmental protection through police or military

1.1 Definition of safety

The Oxford Dictionary defines safety as:

> The condition of being protected from or unlikely to cause danger, risk, or injury.

Wikipedia extends this definition and defines safety as:

> Safety is the state of being "safe" (from French *sauf*), the condition of being protected against physical, social, spiritual, financial, political, emotional, occupational, psychological, educational or other types or consequences of failure, damage, error, accidents, harm or any other event which could be considered non-desirable. Safety can also be defined to be the control of recognized hazards to

> achieve an acceptable level of risk. This can take the form of being protected from the event or from exposure to something that causes health or economical losses. It can include protection of people or of possessions. [4]

Naturally, institutions with focus on a certain area of expertise have a slightly more selective view on safety. The International Electrotechnical Commission (IEC), for instance, defines the term safety in a way that is clearly targeted for the Functional Safety Standard and its strong relationship to risk assessment:

> 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. [19]

Safety is therefore a relative term. It may even depend on the perception if somebody feels safe or not. Each of the definitions above clearly relaxes the strong term by adding phrases of "likeliness" and "risk".

Because safety is such a fundamental topic in human life, it is reasonable that today's technical applications have to be checked on their impact on safety. For many areas, there are regulations and standards that set a lower bound to what is required for an application in regards to functional safety. The standards are updated in rather short intervals and additional standards for more fields of application are released regularly.

Although the requirements are getting stricter over time and manufacturers are trying to achieve higher levels of safety, there are some limitations for manufactures that hinder them to reach the next level of safety with their application. These limitations may be of spatial, energetic or financial kind. For example, it might not be possible for a manufacturer to fund a desired SIL-3 for an application that only requires SIL-1 by standards (details on SIL are described in Chapter 2), as the required investments would overprice the product, which in turn would prevent a successful placement on the market. Another example for a combined spatial and energetic limitation are **embedded devices**, which are omnipresent in today's safety critical environments. These devices may not have enough space around them to be supplied by a bigger or redundant power supply or battery.

An important, yet unfortunate, aspect of safety - although a common aspect for lots of other examples in daily life as well - is that safety is always a **trade-off**. A trade-off between effort and safety gained by this effort, or - even more visible for a customer - a trade-off between convenience and safety. Well known examples for this trade-off in our daily life are safety belts, helmets and similar "tools". A helmet is a good safety tool, but it is usually not really convenient to wear or to carry around. A helmet represents the trade-off of reduced convenience but increased safety.

Functional safety

As already outlined above, it is our goal to get today's safety related systems as safe as possible. That is, the risk that a system's intended functionality is not present anymore is as low as possible or as reasonably achievable.

This goal is subsumed by the IEC under the term "Functional Safety", which is specified as:

> ...the part of the overall safety that depends on a system or equipment operating correctly in response to its inputs.
> Functional safety is the detection of a potentially dangerous condition resulting in the activation of a protective or corrective device or mechanism to prevent hazardous events arising or providing mitigation to reduce the fight consequence of the hazardous event. [19]

The definition already summarizes well what it means to care about functional safety. Still, on closer examination, the question arises what those “hazardous events” actually are, depending on the context, and what instruments are suitable to mitigate or even prevent such events. **In this point the whole complexity of this topic is hidden.** Determining what is generally to be considered a “dangerous condition” and to what extent it is reasonable to require a system to implement certain measures is the scope of the IEC 61508 standard [20], which will be described in Chapter 2.

1.2 Target application domain

Fire plays the major role in human evolution. Humans always loved and used fire, while at the same time fearing it for its destructive nature. The importance of fire is depicted in Wikipedia as:

> The control of fire by early humans was a turning point in the cultural aspect of human evolution that allowed humans to cook food and obtain warmth and protection. Making fire also allowed the expansion of human activity into the dark and colder hours of the night, and provided protection from predators and insects. [3]

With fire, and the fear of it, comes the need for protection against (unintended) fires.
As of today, people are used to the fact that many safety instruments to prevent and/or fight fires are established in our environment. Examples range from simple smoke detectors over fire extinguishers, via semi-automatic fire doors to fully automated sprinkler and other fire extinguishing systems. Putting special focus on fire safety within buildings, the law in most countries of the world regulates a minimum set of equipment that has to be available for fire safety, depending on the type and usage of a building.

More and more of these systems nowadays are fully automated. Consequently, all bigger functional buildings are therefore equipped with so called “Fire Detection and Fire Alarm Systems (FDASs)”. These systems may incorporate thousands of sensors and actuators and therefore need a certain degree of automation in order to be manageable. It should be clear to the reader by now that these FDASs are safety critical by nature as their **non-functionality** or wrong functionality will **increase the likeliness “to cause danger, risk, or injury”**.

This thesis targets safety in the FDASs domain with a specific focus on the hardware responsible for information exchange with an FDAS.

FDAS in building automation

In buildings, the FDASs are part of the overall building automation system. As depicted in Figure 1.1, an FDAS is situated in the lower layers of the classical "automation pyramid" used to describe automation in buildings, in particular the sensors (e.g. smoke detectors) and actuators (e.g. fire doors) being on the "field level" whereas Control and Indicating Equipment (CIE) being situated in the "controller level" or even above, depending on the actual functions performed by each device.

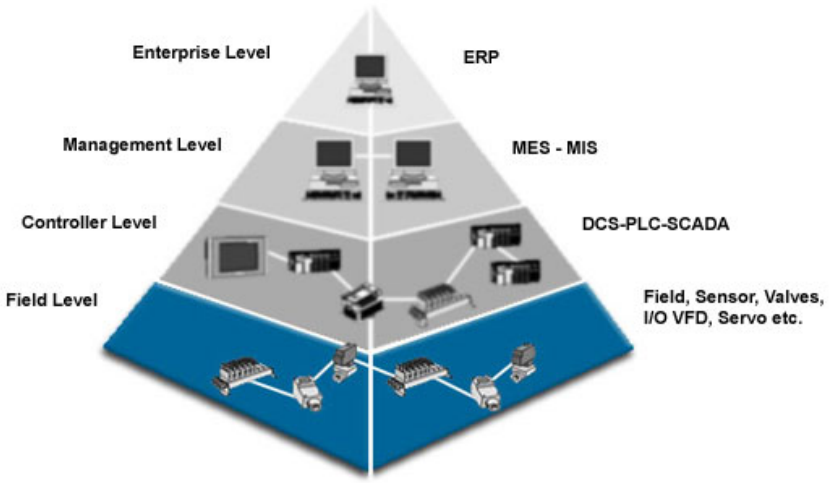

Figure 1.1: FDAS are located in the field and controller levels of the automation pyramid [14]

Before programmable controllers have been introduced in the fire alarm domain several years in the past, these systems where rather isolated systems with little to no interfaces to other systems. Today, where Microcontroller Units (MCUs) are omnipresent in any kind of automation system, also the interfaces between systems evolved and integration of systems in general got a boost. This trend of course does not skip FDASs, where features like connecting such a system to a shared network - e.g. the Internet - or connecting distributed subsystems via shared networks are commonly asked by customers.

Clearly this integration is feasible for gaining efficiency, but also allows for more advanced safety features. As example, one can consider the cooperation of an FDAS with a Heating, Ventilation and Air Conditioning (HVAC) system in case of smoke alarms. The FDAS can switch off all the air conditioners to prevent further dissemination of smoke.

Inevitably the growing number of interfaces and interactions within or among systems raises complexity and error-proneness, making it even more important to provide solid means of dangerous condition detection to prevent hazards.

1.3 Problem statement and motivation

Typically the fire alarm domain has direct impact on human life and health and needs to guarantee safety in everyday life. Fire safety equipment has to provide its services in rather tough environments, which tend to

- have equipment distributed in big and unstructured areas,
- be comprised of multiple diverse subsystems from different manufacturers,
- be susceptible to power outages and
- have a variety of interfaces to third party systems,

which are by far not optimal for easily achieving high safety levels.

Besides the challenges in deploying such a system, it is even more critical to distribute information correctly within such FDASs (e.g. fire alarm messages). Therefore, one of the critical paths in such projects is proper communication. As most of the systems at larger scale are networked in the one or the other way, it seems important that **fault tolerant communication mechanisms** are deployed. That comprises of network structure, network protocols, but of course also the devices connected by these networks. Consequently, it must be ensured that at least those devices, which are crucial for the overall system functionality, are fault tolerant as well, in a sense that overall communication is neither interrupted nor disturbed by a faulty device. The devices in this view shall of course not be limited to safe communication, but must of course provide safe data handling in general as well. Therefore, it does not suffice to guarantee data transmission is fault tolerant, but the devices being responsible for sending and receiving the data (data source and data sink) must be fault tolerant as well, otherwise the overall fault tolerance of the system would be limited by the availability of the non-fault tolerant devices. For instance, it does not make sense to have a perfect fault tolerant transmission channel, if the data sent through such a channel is already corrupted due to some error on the sending device.

As integration of FDASs into other systems is a major trend, the combination bears new safety risks as well. Combining non-safety critical applications now with safety critical applications also takes existing standards to their limits, as those well established documents were originally crafted with the assumption of strict separation of those applications. To put it differently, they assumed an encapsulated environment, where service disruptions could only stem from a limited, well known, set of sources. The new risks implied by this combination of systems therefore even more require thorough design of the safety-critical parts, to ensure functional safety.

Today, there is lots of knowledge on how to construct safety critical applications. Specifically in the domain of IT systems this knowledge covers all fields from hardware construction to software design. Some of the approaches are only theoretical, whilst others are practical. The basis of the research conducted in this area even reaches back to the 1980s.

Still, each field of application of this knowledge requires targeted solutions, whose implementation is highly depending on the environment set out by the application. The fire safety

domain is one field of application, providing its own challenges when it comes to implementing safe and interconnected systems.

As this thesis **focuses on the fire safety domain**, it tries to evaluate possible approaches for gaining more safety in FDASs, going beyond what is required by standards today and still respecting the restrictions imposed by the environment by improving the reliability of the most important devices of FDASs, the CIE, which are the crucial nodes within the communication infrastructure of an FDAS. Increasing fault tolerance within those devices should open the doors for higher levels of safety and finally strengthens the trust into these important, yet rather invisible, devices.

This thesis presents a concept on what can be done to make CIE - based on modern MCUs - fault tolerant, in order to help reach the overall goal of a bullet proof communication within FDASs, with communication being the critical path for the availability and reliability of the overall system.

Methodical approach

At first point an analysis of applicable standards for the fire alarm domain is given to define a basis of requirements requested by legislators. Furthermore, a literature study about state of the art safety concepts in hardware and software is presented.

In consequence, the presented state of the art concepts are analysed for applicability in the target domain of FDASs. Based on this analysis, a concept for CIE, specifically shaped for this application domain, is presented. A prototype implementation, based on modern hardware, shows the key features and offers the basis for an evaluation of the presented concept, and additionally serves to document the system's behaviour in several typical fault scenarios.

Finally, the limits of the chosen concept are identified and an outlook to further possibilities of improvement is given.

CHAPTER 2

Standard specifications

This chapter presents standards currently applicable to Fire Detection and Fire Alarm Systems (FDASs) and standards around Functional Safety in general. These standards set the minimum requirements faced when building FDASs. Albeit there might be further regulations by local laws or national standards, the focus is on international standards as those are generally reflected in the national regulations as well.

The *"Regulation (EU) No 305/2011 of the European Parliament and of the Council of 9 March 2011 laying down harmonised conditions for the marketing of construction products and repealing Council Directive 89/106/EEC Text with EEA relevance"* [1] requires all products in the European Union (EU) to conform with harmonized technical standards (where such exist) in order to qualify for a CE mark. The CE mark is the prerequisite, which allows free movement of the product within the markets of the EU.

The harmonized standard for FDASs is the **EN 54 Fire detection and fire alarm systems** standard. Any FDAS product needs to comply with the EN 54 standard in order to receive the CE mark of the EU. On top of that the EN 54 standard is widely recognized and therefore qualifies these products to be sold outside the EU as well.

Besides the EN 54 standard, an FDAS has to comply to several other standards mostly defined by national building regulations.

For Austria, those standards are:

- TRVB-123 00 "Fire Detection Systems" [8]
- TRVB-114 06 [7]

For Germany, those standards are:

- DIN 14675 "Fire detection and fire alarm systems - Design and operation" [16]

- VDMA 24200-1 “Gebäudeautomation Automatisierte Brandschutz- und Entrauchungssysteme - ABE” [36]
- DIN VDE 0833-2 “Alarm systems for fire, intrusion and hold up” [15]
- VdS 2095 “Guidelines for automatic fire detection and fire alarm systems” [37]

In terms of Functional Safety there is the international “mother of all” standard **IEC 61508**. It is an application independent - yet technology bound - standard, which builds the foundation of a variety of derived standards, each of them tailored towards a specific field of application.

A brief listing of standards related to Functional Safety - most of them being derived from IEC 61508 - includes:

- US RTCA DO-178B North American Avionics Software
- US RTCA DO-254 North American Avionics Hardware
- IEC 62304 - Medical Device Software
- IEC 62425 - Railway Signaling Systems
- IEC 61513 - Nuclear Systems
- ISO 26262 - Road Vehicles Functional Safety
- IEC 61511 - Functional safety – Safety instrumented systems for the process industry sector
- IEC 62061 - Safety of machinery - Functional safety of safety-related electrical, electronic and programmable electronic control systems
- ISO 13849-1, -2 Safety of machinery - Safety-related parts of control systems. Non-technology dependent standard for control system safety of machinery

As there exists no specific Functional Safety standard for FDASs, standards like VDMA 24200-1 provide guidelines on how to handle functional safety in these systems. It is worth noting that most guidelines are again based on the non-normative parts of IEC 61508, hence, concepts, methods and terms of IEC 61508 are used in these guidelines.

2.1 EN 54

The EN 54 standard comprises 25 parts in total[1], covering all components involved in FDAS. Each part describes one component or piece of equipment in detail.

The introduction to EN 54 is in part 1 [10] and describes the standard to specify amongst others:

[1] Part 6a and part 8 of EN 54 have been withdrawn.

- Requirements, test methods and performance criteria against which the effectiveness and reliability of the component parts of FDAS can be assessed, and
- Requirements and test methods against which the ability of components to be combined into an effective system can be assessed.

In the course of this thesis, the main focus is on the so called "Control and Indicating Equipment (CIE)". This equipment is specified to be **responsible** for monitoring the "correct functioning of the system" and "warning of any faults", which clearly points out that functional safety is the major topic of this very equipment. Therefore, a closer look will be taken at the regulations in EN 54 part 2 *"Fire detection and fire alarm systems - Control and indicating equipment"* [11], as those regulations define most of the important guidelines for building CIE.

EN 54 part 2

The standard EN 54 part 2 defines its scope as:

> This European Standard specifies requirements, methods of test, and performance criteria for control and indicating equipment for use in fire detection and fire alarm systems installed in buildings. [11]

The standard first defines general requirements how the state of the CIE has to be displayed and indicated. This covers audible indications as well as indications on displays and lamps.

The subsequent chapters describe the various functional conditions a system can be in. This is the list of recognized conditions by the European Standard:

- fire alarm condition
- fault warning condition
- disable condition
- test condition
- quiescent condition

The standard continues to elaborate on standardized input/output interface and design requirements. A separate chapter is dedicated to software controlled CIE and outlines more specific design requirements for this type. Finally, the last chapter deals with tests and extensively describes all sorts of mandatory tests to be performed with the product in order to achieve standard compliance.

The following paragraphs show a selection of requirements from the standard, which specifically influence functional safety and are therefore important for this thesis.

Section 5.1.1 in EN 54 part 2 demands that a CIE needs to be able to unambiguously indicate fire alarm and fault warning conditions. Moreover, the time limit to indicate such a fire alarm

condition may not exceed ten seconds. After a reset operation, indication of the current condition has to be re-established within 20 seconds.

Chapter 8 of EN 54 part 2 is of special interest, as it is listing the requirements for coping with various faults and how these have to be detected, indicated and resolved. In general, the CIE should enter the fault warning condition within 100 seconds from detecting a fault situation. Section 8.4 requires the equipment to emit an audible indication for at least one hour in case of a complete loss of the main power source and Section 8.5 stipulates how software faults have to be indicated.

In case faults can be indicated on some other equipment, Section 8.9 requires the FDAS to also indicate de-energized CIEs. Section 12.5 additionally states that a fault in any transmission path of the system shall not affect the correct functioning of the CIE or any other transmission path. The re-establishment of function of operational devices after an interruption may not exceed 300 seconds.

For software controlled CIE, Section 13.4 requires a separate monitoring device with a separate timebase to monitor software execution. Furthermore, the software has to enter a safe state if a system fault is detected. Section 13.5 deals with memory aspects for program and data. A requirement is that memory containing site specific data has to be continuously verified for correctness in an interval not exceeding one hour.

Since this thesis deals with software controlled CIE only, this packed list of facts will serve as a minimum basis for the further concept.

2.2 IEC 61508

The IEC 61508 standard plays a different role in the fire alarm domain. While it is not strictly required by EN 54's testing procedures or software requirements, it seems obvious that the ideas of a standard entitled *"Functional safety of electrical/electronic/programmable electronic safety-related systems"* shall be applied to state of the art CIEs being comprised almost entirely of programmable electronics with the sole purpose of providing safety. The German standard VDMA-24200-1 explicitly deals with safety requirements in Chapter 6 and refers to IEC 61508 in Section 6.2.

The origins of the standard reach back to 1984, where Hölscher and Rader described requirements for microcontrollers in safety related systems [18]. The standard evolved from the process control industry, yet it has to be pointed out, that the IEC 61508 standard is a generic one. Therefore, it requires interpretation in some areas, in order to match the actual application. For many industries derived standards have been released, which contain the interpretation of IEC 61508 and add further requirements specific to that industry.

The standard covers the complete safety life-cycle and is split into seven parts, where only the first three parts have normative character. Figure 2.1 shows which topics are addressed by the parts of the standard. The basic concepts dealt with in the standard are **risk assessment** and

safety functions. Safety functions may include electronic technology and are used to lower a certain risk that hazardous events cause severe consequences (such as death) by bringing the **equipment under control** into a safe state.

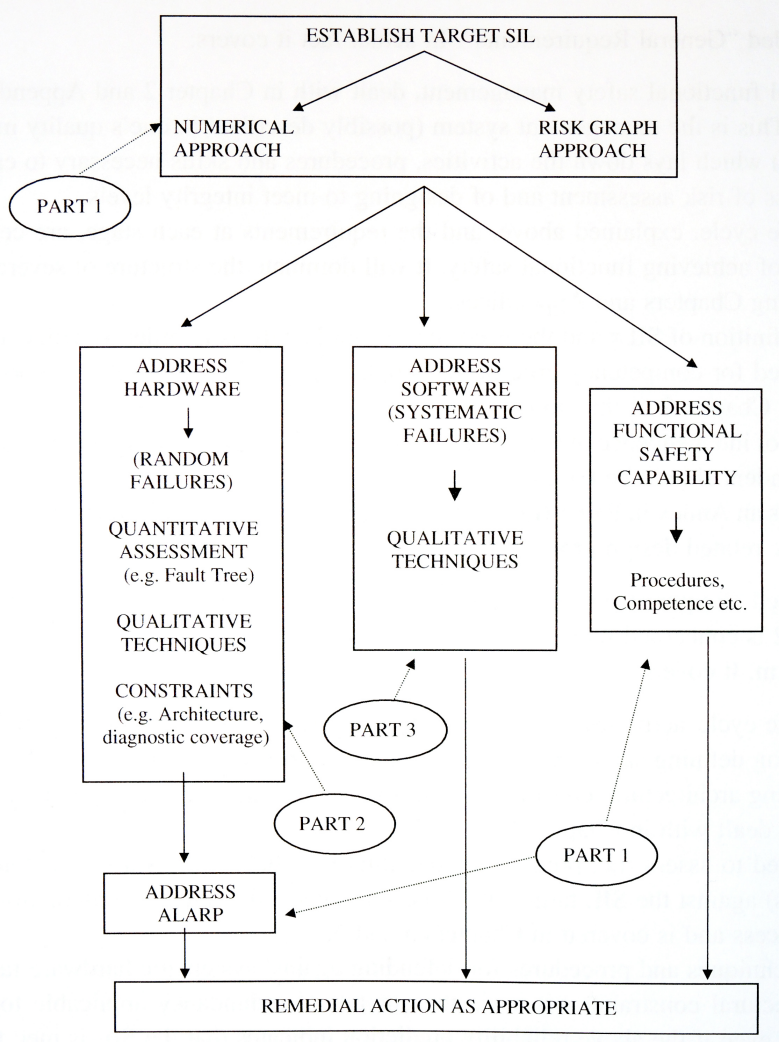

Figure 2.1: The Parts of IEC 61508 [33, P. 17]

The types of failures we are looking at are random hardware failures and systematic failures. The random hardware failures can be assessed in terms of failure rates, whereas the systematic failures can only be addressed by applying sufficient rigor throughout the design process. Safety has to be considered throughout the whole life-cycle of a product, for that reason the safety life-cycle covers the complete life-cycle.

A central goal of the standard is to ensure that non-tolerable risk - which is to be found out by risk assessment - is reduced As Low As Reasonably Practicable (**ALARP**).

One of the central terms in the IEC 61508 standard is the term **Safety Integrity Level (SIL)**. The "Safety Critical Systems Handbook" nicely describes how this term is to be understood:

> The maximum tolerable failure rate that we set, for each hazard, will lead us to an integrity target for each piece of equipment, depending upon its relative contribution to the hazard in question. These integrity targets, as well as providing a numerical target to meet, are also expressed as "safety-integrity levels" according to the severity of the numerical target. [33]

The safety-integrity levels are specified from SIL 4 to SIL 1. The "Safety Critical Systems Handbook" defines them as:

- **SIL 4** is the highest target and requires state of the art techniques to achieve it. This level is usually avoided as the effort involved to reach this target, is massive.
- **SIL 3** requires sophisticated design techniques.
- **SIL 2** only requires good design and operating practise on a level such as would be found in an ISO 9001 management system.
- **SIL 1** is the lowest target level, but still implies good design practise.
- Anything **below SIL 1** is considered to be not safety related as of IEC 61508.

The general process is to first conduct hazard and risk assessments, which yield a target SIL. The SIL then defines how the life-cycle and its processes have to be set up in order to be able to justify that the final system attains the target SIL.

For hazard and risk assessment, the standard suggests to use either qualitative or quantitative analysis techniques and provides examples. One of the qualitative approaches is a method, where a set of categories of likelihood of occurrence is set in relation to a set of four categories of consequences (catastrophic, critical, marginal, negligible). The resulting matrix provides information how tolerable a certain risk is and assigns a class for this in each cell of the matrix. The classes being from one (unacceptable) to 4 four (acceptable). For instance: Having occasional occurrence combined with catastrophic consequences is assigned to Class 1, unacceptable.
Contrary to the qualitative risk assessment method described above, the VDMA-24200-1 standard requires risk assessment based on risk graphs. The authors outline that "real risk assessments have shown that the majority of FDASs have to fulfil SIL 1" [36, S. 12], albeit some situations - e.g. when deviating from construction laws - might require higher levels as well.

An example risk graph, as suggested in IEC 61508 part 5, is in the appendix of VDMA-24200-1 (see Figure 2.2), which is specifically targeted for FDASs.
In order to define the requirements for each SIL, another distinction is made by the standard: The demand rate. The standard separates the demand rates into only two groups. The **high demand rate** applies to all systems that operate continuously or where the safety function is demanded more than once per year. The **low demand rate** is used for all other systems. For each demand rate the standard defines the maximum failure rate and the PFD as listed in Table 2.1.

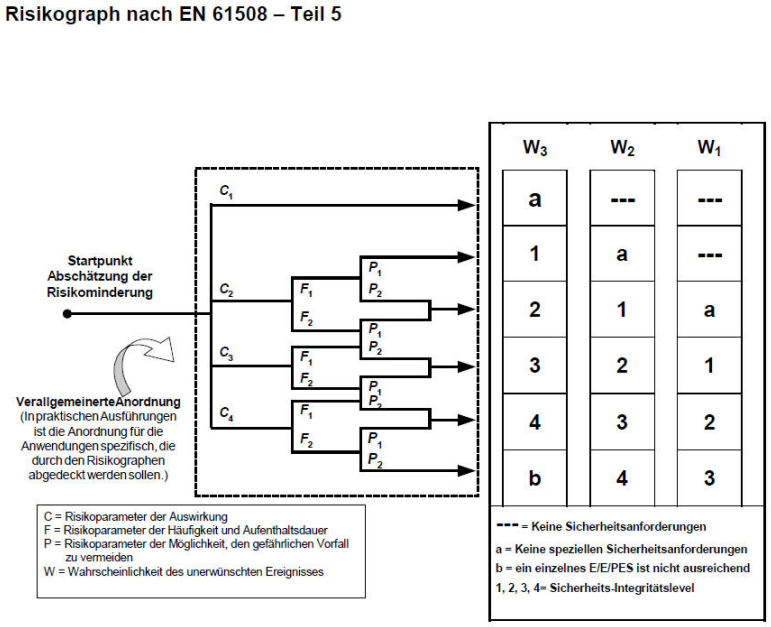

Figure 2.2: VDMA-24200-1: Example for a risk graph to assess a target SIL [36]

SIL	dangerous failures / hr (high demand rate)	probability of failure on demand (low demand rate)
4	$>= 10^{-9}$ to $< 10^{-8}$	$>= 10^{-5}$ to $< 10^{-4}$
3	$>= 10^{-8}$ to $< 10^{-7}$	$>= 10^{-4}$ to $< 10^{-3}$
2	$>= 10^{-7}$ to $< 10^{-6}$	$>= 10^{-3}$ to $< 10^{-2}$
1	$>= 10^{-6}$ to $< 10^{-5}$	$>= 10^{-2}$ to $< 10^{-1}$

Table 2.1: SIL specification for low and high demand rates according to IEC 61508

Probability of failure on demand (PFD) is the same as probability of being failed at a random chosen moment, which is the same as unavailability. The PFD is dimensionless and is given by:

$$\text{PFD} = \text{UNAVAILABILITY} = (\lambda\ \text{MDT})/(1 + \lambda\ \text{MDT}) \cong (\lambda\ \text{MDT})$$

λ is the failure rate and MDT is the mean down time, which includes the mean time to repair (MTTR), and usually λ MDT $<< 1$ holds true.

The safety function of FDAS is classified as low demand rate system by VDMA-24200-1 (Appendix F.2), due to the fact that the demand, namely a fire incident, will occur only rarely on average.[2]

[2]Obviously, the classification whether a low demand rate or a high demand rate is applicable is not very clear. For instance, SafetyLON, a network protocol for safety related applications within the building automation domain, was defined to be a high demand rate system [29], albeit its domain includes fire safety as well.

Section 7.4 of IEC 61508 part 2 outlines requirements for design and development. The following list is a summary of those requirements as denoted in [33].

- Use of "in-house" design standards.
- On manual or auto-detection of a failure of the safety system, the design should ensure system behaviour which maintains the overall safety targets. This implies additional monitoring or a shut down of the equipment under control.
- Sector specific requirements have to be observed.
- The system design should be structured and modular.
- Systematic failures caused by the design should be tackled by monitoring the functionality with extra circuitry. The complexity of this monitoring varies depending on the target SIL.
- Systematic failures caused by environmental stress: Components should be designed for the environment in question. Generally, components reaching a CE marking approval (or similar) would be expected to meet this requirement.
- Communications. The failure rate of the communications process has to be addressed.
- Synthesis of elements: The standard allows configurations involving parallel elements, where each of them demonstrates a particular SIL in respect to systematic failures, to claim an increment of one SIL. Prerequisite is that a common cause analysis has been conducted to demonstrate independence. An example is shown in Figure 2.3

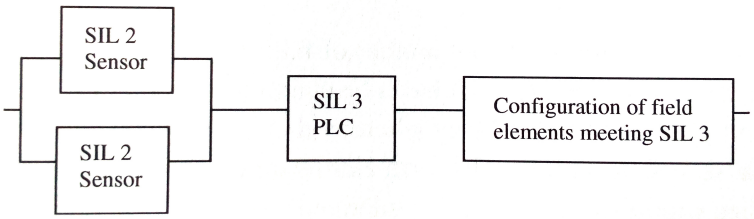

Figure 2.3: Two SIL 2 elements achieving a SIL 3 result. [33, P. 51]

A special focus is on the architecture of the design, the standard dedicates Section 7.4.4 to this topic.

Besides the reliability calculations established for hardware, the standard specifies minimum levels of redundancy together with levels of fault tolerance. This is subsumed by the term Safe Failure Fraction (SFF). The SFF is defined as:

$$\text{SFF} = \frac{\text{Total revealed hazardous failures} + \text{Total safe failures}}{\text{Total failures}}$$

Type A SFF	SIL for Simplex HFT 0	SIL for (m+1) HFT 1	SIL FOR (m+2) HFT 2
$< 60\%$	1	2	3
$60\% - 90\%$	2	3	4
$90\% - 99\%$	3	4	4
$> 99\%$	3	4	4
Type B SFF			
$< 60\%$	Not allowed	1	2
$60\% - 90\%$	1	2	3
$90\% - 99\%$	2	3	4
$> 99\%$	3	4	4

Table 2.2: Requirements for SFF

where "Total revealed hazardous failures" are potentially dangerous failures revealed by auto-test, "Total safe failures" being the number of failures resulting in a safe state and the "Total failures" comprising the aforementioned PLUS the unrevealed hazardous failures.

There is another distinction into two types of components. Type A applies to components where failure modes and behaviour under fault conditions are well defined and failure data is available. The remainder of components is classified as Type B. Depending on the type, Table 2.2 shows the maximum SIL which can be claimed depending on the SFF. The number m specifies the number of failures which lead to system failure, $(m + 1)$ designates redundancy, the number of elements.

Part of the overall safety is of course also operations. Depending on the target SIL, the systems are required to provide feedback or even correctional actions to operator activities. This is the place where the "human factor" has to be considered. An example here are the fly-by-wire systems of airplanes (typically SIL 3 to 4). These systems, for instance, prevent the pilots from stalling the plane.[3] Quantification of human error is not a requirement of IEC 61508, but the consideration of human error is required and mentioned in various places in normative parts one to three of the standard.

Planning and implementing the software should also include semi-formal methods as well as coding standards and structured programming. Dynamic objects, interrupts, pointers and recursion should be avoided as much as possible. The used programming language should be restricted to a subset to minimize unsafe usage. An example is the MISRA-C standard from the automotive sector that describes such a subset of the programming language C. For SIL 3 and 4, it is furthermore strongly recommended to use certified tools only.

The Life-cycle approach

The IEC 61508 standard is based on a safety life-cycle approach. The reason for that is that qualitative safety criteria, such as avoidance of systematic failures, have to be tackled mostly on

[3]To stall an airplane means to increase the angle of attack (raise the nose) to a point, where the air flow over/under the wings stalls, causing the plane to actually stop flying. This condition is deliberately used in aerobatics flight, but is to be avoided for normal flights like on airliners as it could cause the plane to crash.

the management level and throughout the design phase of a product. Therefore, this approach aims to control all phases of the life-cycle, which influence safety. The phases comprise of the process from safety specification and assurance throughout the development and operations to decommissioning. The standard describes a model for the life-cycle and identifies all activities throughout the life-cycle based on this model. This assures that decisions made in the planning and design phase are in the end fulfilling the target failure probabilities. A simple life-cycle is depicted in Figure 2.4. It is worth noting that the Functional Safety Assessments are to be carried out in all phases.

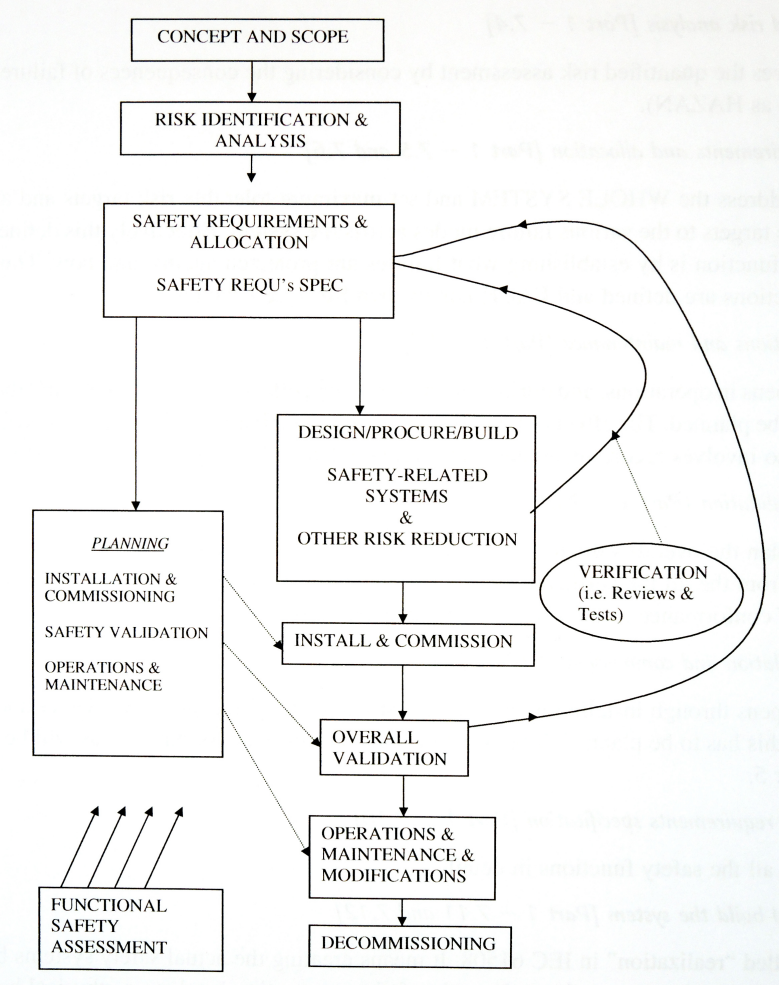

Figure 2.4: Safety life-cycle [33, P. 11]

The life-cycle has to be applied to all electrical and programmable aspects of the safety-related equipment. In case such equipment is embedded in another system, the life-cycle applies to the system as a whole. That involves also mechanical and pneumatic equipment. Keep in mind though, that each safety function has its own SIL target, so different management functions

might be necessary for each safety function.

All steps in the life-cycle are to be verified by functional safety assessments. Following this methodology helps to improve the quality of the product as well as it helps to detect mistakes (systematic failures) at an early stage. Depending on the intended SIL, the assessments have to be undertaken by dedicated safety managers or even external certification institutions.

Software

Any software involved in safety functions needs to comply to the requirements of IEC 61508, too. Part 3 of the standard is called "Software requirements" and deals with systematic failures. The level of software verification needed may vary enormously and might even reach to the adoption of formal methods, which is common for SIL 4 safety function for instance in airplanes[4]. For lower levels, common automated tests and unit-testing will already suffice and are generally accepted.

Tests

When striving towards the requirements of SIL 3, the complexity of the checking mechanisms and test methods significantly jump up and the used methods have to be extended to involve:

- Memory checks with checksums or parity bits
  This comprises of methods to detect or even tolerate memory corruption of all kinds. Checksums may be calculated on the content of the memory on fixed intervals allowing to detect unexpected corruption of any data. The checksum of course needs to be stored, so this yields the necessity for planning on extra memory depending on the granularity of the memory blocks to be checked. The bigger the area that is covered by a checksum, the less memory is required to store those checksums, but on the other hand, writing data is slowed down as the checksums have to be updated as well, which takes longer for bigger areas of memory. Parity bits may have the advantage here that those are easy to implement in hardware, hence the time to create those is not a problem.

- Signal checks
  Any data sent around on communication interfaces has to be checked for validity. Means for achieving this can be parity checks or reasoning about the adequateness of a signal in the given state of an application. So these checks can happen at the various levels of abstraction.

- I/O module tests
  All I/O modules shall have appropriate test circuitry which allows testing their functionality. For instance, SPI interfaces may provide internal loopback functionality (inside the circuit or even on pin level) to test the receiving and sending unit of the interface.

[4] A big European airplane manufacturer seeks help from German universities (Prof. Dr. Reinhard Wilhelm) in order to reach SIL 4 for the primary flight control system. The code for the primary flight control is said to be verified by formal methods to a large extent. [2]

- Sensor/Actuator tests
  As with I/O modules, connected sensor or actuator devices shall provide self-test capabilities, which allow periodic monitoring of their functionality.

- Fault injection tests
  These kind of tests involve active injection of faults into the system (which are expected to be covered by the fault hypothesis) and the verification whether the system under test reacts as specified. Assuming a redundant network connection a possible fault injection testing vector would be to forcefully disable a connection line in order to assert that the equipment really correctly performs the expected fail-over.

- and many more

Besides complex monitoring of the equipment under control, there are other means that help to reach the targeted SIL. On the architectural level, one important design method is modularization, albeit the IEC 61508 standard (Part 2) asks to keep the number of modules in a "limited size" for SIL 1 and 2. Moreover, these modules are required to have a certain documented field experience. Instead of packing all the intelligence of safety into the safety functions (monitoring), an even better approach would be to improve the equipment under control, such that the risk imposed by it is lowered. This has direct impact on the risk assessment and therefore lowers the (necessary) complexity of the safety functions. Note that this especially holds true for integrated systems, where equipment under control and safety function are not separated. e.g. a monitoring software is running on the same Central Processing Unit (CPU) as the control software.

2.3 Other standards

Apart from EN 54 part 2 and IEC 61508 some other standards are crucial for FDASs and their CIE. For a reliable system, "power management" is an important asset. Therefore, EN 54 part 4 [9] specifies Power Supply Equipment (PSE). It requires at least two power sources per CIE unit, where at least one of them has to be connected to the public grid and another one has to be a rechargeable battery (Section 4.2.4). Each of them must be capable of operating the CIE within its specification (Section 4.2.5).

Despite the existence of the mandatory European standard EN 54, the individual countries additionally established national standards for FDASs. Some of those standards just extend on the regulations of EN 54, whilst others include their own provisions independently.

CHAPTER 3

Functional safety - state of the art methods

The field of functional safety and the technologies used and developed to tackle the challenges linked with it, have undergone big improvements in the last decades, while the importance of safety and security grows every day as embedded systems conquer our everyday life and everyday needs. Specifically systems that require high reliability and therefore having targets of Safety Integrity Level (SIL) 3 to 4 face a very tough problem: “The system as a whole must be more reliable than any one of its components.” [23]

This chapter will present an overview of state of the art techniques and methods used in the context of functional safety. These methods specifically focus on design and implementation methods. The chapter does not cover aspects like material quality, which of course is crucial for safety of a product as well, but is not directly a factor of functional safety.

3.1 Definition of terms

As with every field of expertise, it is necessary to speak the same language in order to avoid confusion. In the safety area, a few terms are of special importance, which are described as follows:

- **Fault**
  “Condition that can cause an element or an item to fail.” [20]
- **Error**
  “Discrepancy between a computed, observed or measured value or condition, and the true, specified, or theoretically correct value or condition.” [20]
- **Failure**
  “Termination of the ability of an element, to perform a function as required.” [20]

- **Fault hypothesis**
  "An assumption about the types and numbers of faults that must be tolerated by the planned system." [22]

- **Fault-Containment Region**
  Fault-Containment Regions are a concept to define boundaries in a systems, where errors do not propagate over and therefore do not cause the next part in the system to fail as well.

The subsequent chapters will use these terms extensively to describe various aspects. The reader should put special focus on the subtle difference between fault and failure.

3.2 Hardware methods

To achieve a high level of safety, a variety of methods and guidelines exist on the hardware level that can be utilized to reduce the likelihood of random hardware failures - which occur throughout the runtime of a product - and to avoid systematic failures - which are shortcomings in the design and development phase. On ASIC/chip level, the following list provides a selection of those methods.

- Throughout the design phase: Validation of the whole code with simulation
  The designed chip layout is validated by simulating the design with accordant simulation tools for Hardware Definition Languages (HDLs) like Questa. For FPGA designs, ModelSim would be an example for a well known simulation tool. With these simulation tools, it is possible to detect critical paths in the design, which may be the first spot in the system to be the cause of a random hardware failure due to limit exhaustion.

- Design for testability [38] [25]
  The HDL design is created with testability in mind. The tests to be conducted are defined precisely and the design is built with account for ease of test execution. This may also include built-in testing structures, which are finally disabled within the shipped product.[1]

- Functional tests on module and top-level
  Designated testbeds are established for each module additionally to a testbed for the overall system. This allows detailed testing of the modules while the system test helps to rule out emergent effects caused by the combination of modules.

- Avoidance of asynchronous circuits
  For safety critical systems, only synchronous chip designs are recommended as it might be impossible to verify correctness with asynchronous constructs.

- Consideration of gate and wire delays
  The timing calculations for HDL designs as well as Printed Circuit Board (PCB) designs

[1] This is usually done via so called fuses, which provide the possibility to enable or disable parts of digital logic after a chip has been produced. Commonly a laser ray is used to burn such fuses. Popular applications of this technique range from "shaping a product to one's needs" to "removing test circuits".

should consider the delays imposed by wires and gates. HDL synthesis tools like Quartus usually take care of gate delays automatically for calculating the maximum clock frequency of a synchronous design.

- Separate physical blocks on the substratum for each channel and monitoring block
  This precaution tries to prevent negative effects like errors in the system influencing a monitoring circuit.

- Distance of physical blocks shall be large enough to avoid crosstalk
  This measure helps to avoid crosstalk effects and reduces the probability that two or more blocks are affected by a common cause fault.

- On-chip redundancy
  Lifetime of certain elements might not always fulfil the estimation. Deploying units redundantly on a chip allows to switch the operational unit or to validate results of those units for correctness. See Section 3.3 for details of redundancy concepts.

- Flipped orientation of redundant elements on the die
  The physical orientation of an element might be crucial when it comes to sporadic events like cosmic radiation. [40]
  Positioning redundant elements in different orientations on the die (flipping each by $90°$ for instance) will lower the chance that all elements are affected the same way and helps to prevent common cause failure.

- Avoidance of unsymmetrical wiring

- Verification of the complete ASIC
  Testing the complete ASIC against the specification.

Still, when using cheaper hardware components one may fail to reach the desired SIL.

> For achieving a defined level of integrity either highly reliable hardware with a low residual error probability has to be used or comprehensive online self tests have to be performed. The chosen standard microcontrollers must perform online self tests in order to detect as much hardware faults of the CPU internals as possible. [35]

These kinds of self tests can be implemented in hardware or software. Generally, hardware implementation is to be preferred, as those tests have access on a lower level and can be executed faster or even in parallel to normal CPU execution. Dedicated safety Microcontroller Units (MCUs) provide a big set of embedded self tests for many MCU components.

Besides testing the hardware and improving its reliability, it is of course also necessary to track proper execution of the overall functionality, which mostly is comprised by software. The following section will deal with approaches to detect faults on a higher level.

3.3 Approaches for fault detection

In order to detect or even tolerate faults in a system and make it more safe, lots of ideas and approaches have been proposed over the decades. This section will present some approaches and outlines their solutions and the problems they solve.

Fault detection and tolerance always requires some form of redundancy in a system. In the simplest incarnation the correct functionality of a system during runtime can be checked by employing monitoring. In hardware, this usually is a separate, distinct circuitry. The requirements imposed on such a monitoring circuitry differs for complexity depending on the desired SIL. In software, fault detection is usually solved by examining memory and calculating checksums for data. Code-wise the detection is a bit more involved, but there are pure software solutions for transient and permanent error detection as well. [30]

A simple monitoring system in hardware would be a watch-dog. A watch-dog is typically a separated logic that consists of a timer and a reset line at its most basic style. The monitored application has to reset the watch-dog's counter within a certain amount of time, otherwise the watch-dog logic initiates a reset of the application as corrective action. The concept is well known from railways where this is called "dead man's control". A train is stopped automatically if the locomotive driver does not trigger the watch-dog switch regularly. The corrective action taken by a watch-dog varies of course. For some applications the watch-dog may trigger some fail-safe circuitry and additionally starts a second watch-dog which monitors the the fail-safe function again. Besides that, a watch-dog may also do some recording of program state to ease fault detection or post-mortem analysis. Watch-dogs can also be combined with software services to allow for more fine-grained corrections if software faults can be detected by such services. The outer watch-dogs serves as last resort and monitors the software services only.
Yet the simple watch-dog leaves us with an issue, which was described by Lu in 1982:

> Major issues in the design of watchdog processors are the selection of operations to be checked for errors, the means of encoding and transmitting information from the main processor to the watchdog, and the programming of the watchdog. [27]

A slightly different watch-dog approach is suggested by Michel, T. [28] which describes a watch-dog processor that is specifically targeted to software flow control on an Intel 80386sx microprocessor, with the special goal to avoid performance issues. So essential the watch-dog concurrently monitors the program flow on the processor and solving the above issue by leveraging knowledge about the internals of the processor.

In terms of functional safety, a watch-dog is generally considered a good design practice and thus suffices for most applications with a SIL 1 target.

On the software side, a promising approach has been presented by Nahmsuk Oh [30] in 2000. The idea is to save cost for hardware in space applications and use "Commercial Off The Shelve (COTS)", cheap, non-shielded hardware only. As radiation is a major topic for space applications, transient errors - like bit-flips - have to be detected. Oh proposes three techniques to

detect hardware faults in software, based on signatures and code duplication, with one of the techniques reaching 98% fault coverage in injection tests.

A different kind of redundancy and to gain safety, is to actually use multiple instances of the actual equipment. The setup changes in a way that error-checking can now be done by comparison, where error-checking component - called voter - does not need to know about the internals of the system. This has the additional benefit that such a true redundancy covers all aspects of the equipment, hardware and software. Any fault in any component can potentially be detected, as long as the output shows sufficient discrepancy to the voter.

Redundant systems can be categorized by the number of faults they may tolerate. IEC 61508 uses the term XooY ("X out of Y") to categorize the systems. This means that X out of Y systems have to fail for the overall system to be inoperable.
Typical examples and their applications are:

- 1oo1: Single channel processing.
- 1oo2: If one of the two systems fails, the whole system is disabled (used for automatically guided vehicles).
- 2oo2: If both systems fail, the whole system is disabled (used for gas-turbines).
- 2oo3: At least two systems have to be operable (used for airplanes and chemical plants).
- 2oo4: Three out of four systems must be operable (typically required for space shuttles and nuclear power plants).

The following sections describe typical redundancy implementations.

Standby redundancy

A simple possibility of redundancy is the standby pattern. The general idea of the standby pattern is to have a replica of the active system available in spare, which can be activated whenever the primary system fails. However, to automatically switch between the primary and the spare component, some additional logic is required. Figure 3.1 shows the setup of such a system. The standby pattern is a dynamic redundancy concept, as it involves reconfiguration (switching the component) after a fault has been detected. According to IEC 61508, this is a 2oo2 system.
The distinction between hot and cold standby refers to the state the spare system is held in. For cold standby, the spare system is generally not active and needs to be initialized once it is activated. Contrary, the hot standby scheme refers to a system, where the spare system is active too, such that any operation performed by the primary system is also performed by the spare system. Both systems are fed with the same inputs at the same time. This ensures the spare system is up to date and a possible switchover can be made with minimal delay as no initialization is necessary.

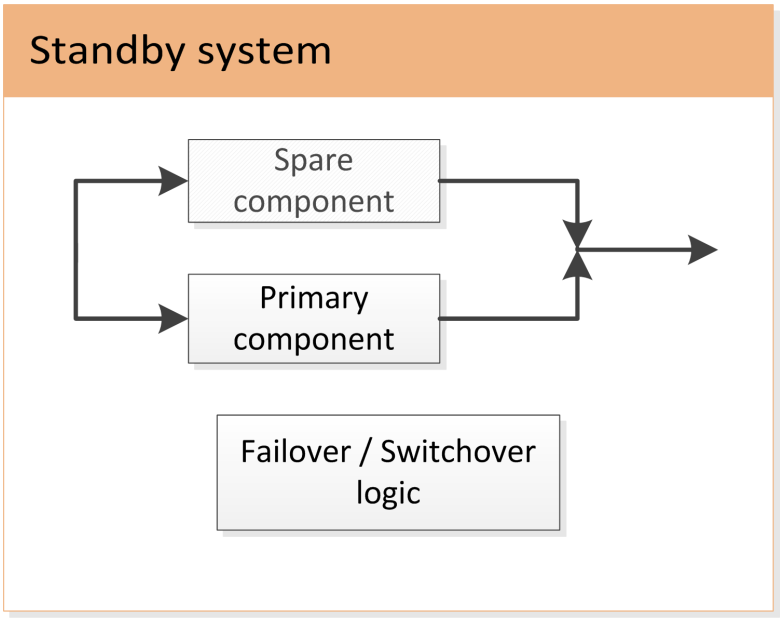

Figure 3.1: System setup for standby redundancy.

For hot standby systems, some literature suggests to include the switchover logic into the spare system, such that the external logic can be omitted. This kind of setup (hot standby with integrated switchover logic) is very similar to the Dual Modular Redundancy (DMR) scheme described below and hence the terms may be used interchangeably.

Dual and Triple Modular Redundancy

Modular redundancy refers to a system setup that has multiple instances of the same component working in parallel which are synchronized. A voter is responsible to determine the final output of the whole system. Popular setups are dual and triple modular redundancy. The N-modular redundancy concept is, contrary to the standby concept, a static redundancy concept as the fault is not propagated to the output.

DMR consists of two components working in parallel and is capable of **detecting** a fault, but is not able to correct for this error. The voter is not able to determine which component is wrong and which right. The difference between DMR and hot standby is rather subtle and mostly relates to the amount of synchronization performed between the components. As described in Section 3.3 on the preceding page, a hot standby system might be constructed to be similar to a DMR system.

While a DMR system is not capable of **tolerating** a fault in general, a Triple Modular Redundancy (TMR) system is able to actually tolerate a single fault by adding one more component. This is the accepted standard for fault tolerance in system architecture.
Figure 3.2 shows the setup of DMR and TMR. The TMR setup allows the voter to decide which component is faulty, as the non-faulty components' outputs match. It can furthermore switch to

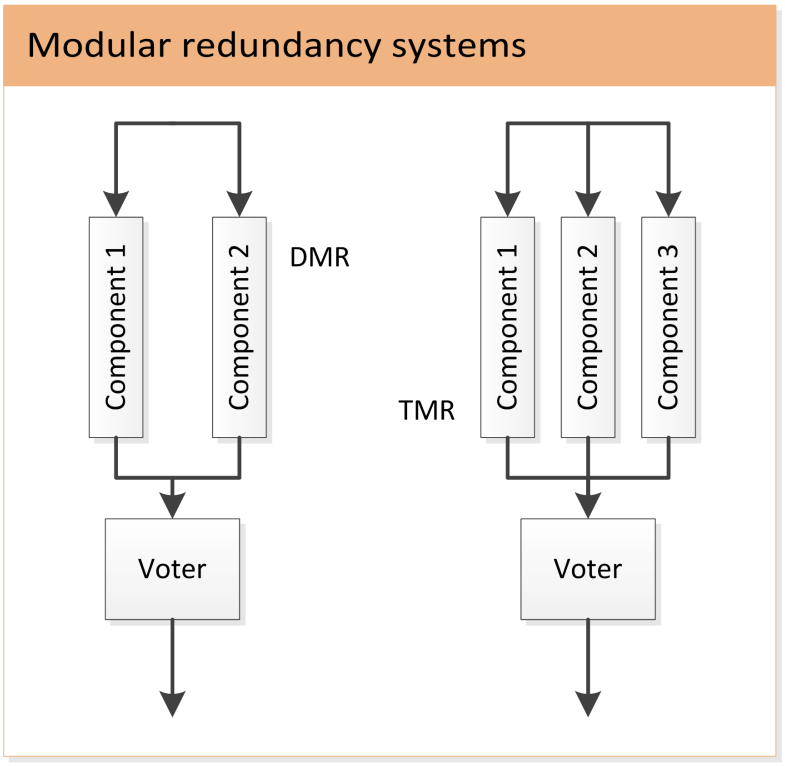

Figure 3.2: System setup for dual and triple modular redundancy

a DMR strategy until the faulty component has been repaired, so at least error detection is still possible in case one component has failed. The TMR scheme can be extended to achieve even higher levels of fault tolerance by adding more components.

H. Kopetz [23] defines the architectural services that are needed to implement TMR:

- Provision of an independent Fault-Containment Region for each one of the replicas
  This ensures that the replicas do not influence each other and assures true independence.
- Synchronization infrastructure
  The synchronization of the replicas is crucial as it is the basis for deterministic behaviour, i.e., the voter needs to know when the output of the replicas is ready to make a decision.
- Predictable multicast communication
  A one-to-many communication allowing each node in the system to talk to all nodes. This can be understood as a fully connected graph of the communication channels in the logical structure of a system.[2] The predictability is important as everything else would be an asynchronous communication pattern, which brings a lot of undesired properties.

[2]This does not imply that the actual communication system is really a fully connected network. The goal is to ensure that every node can list to all other nodes, regardless whether it is the recipient of the message or not.

- Replicated communication channels
  Of course the channels need to have redundancy as well, since a single communication channel would introduce yet another single point of failure, which would contradict the goals of TMR.

- Voting support

- Deterministic (which includes timely) operation
  The various steps of operation have to be deterministic in order to be able to guarantee fault tolerance. This excludes asynchronous communication and generally requires known worst-case execution times.

As can be concluded from this list of services, TMR or bigger redundancy implies a lot more hardware effort and involves significantly higher costs than standby redundancy. This is a main reason why lower levels of redundancy is generally preferred if the system and the indented functionality permits that.

3.4 Software architecture

While the methods above mostly focus on transient failures, hence physical issues, this section takes a look at possibilities to avoid systematic failures.

Software does not fail randomly, hence any fault is always a consequence of a wrong implementation, or even worse, a wrong specification. These systematic failures have to be tackled throughout the design phase of the life-cycle. There is only one exception, a random hardware fault might alter state, which causes the software to execute in an undesired manner.

Safety tactics

In [39], Wu and Kelly present a guidance on how to develop a basic safety strategy in software architecture design. They describe so called “safety tactics”, which connect software safety and software architecture. The safety tactics focus on the three aspects elicitation, organisation and documentation and can be organized as a hierarchy shown in Figure 3.3.

The derived safety tactics should then be used to create software patterns, which are subsequently used to design the software architecture.

In [34], from 1995, a number of industrial applications were inspected for used software patterns. The survey included a wide spectrum of applications in various sizes (in terms of code size) and important system characteristics. The introduction of that paper already states one important fact about architectures: “When poorly understood, these aspects (component interaction and interconnection) of design are major sources of errors.” The paper defines four categories of structures, where each of them can be described in different perspectives. The identified categories are: conceptual architecture, module interconnection architecture, execution architecture and code architecture.

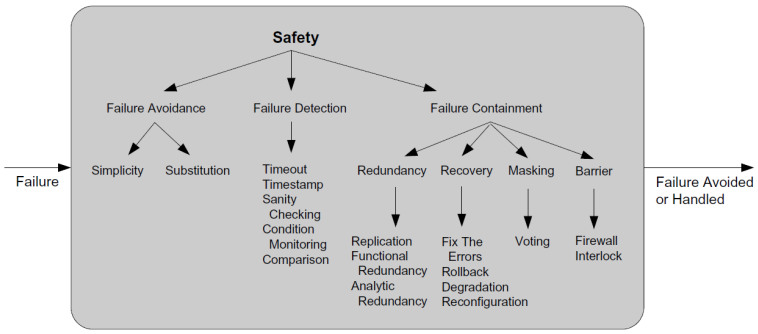

Figure 3.3: Hierarchy of safety tactics [39]

Twenty years later, the available technology vastly changed and these categories of architectures are integrated into frameworks and methodologies, which purport several aspects of these architecture descriptions into new concept.

Development process and patterns

Every software development process involves a methodology which defines the basic workflow how code is created. In the recent years a whole new class of such methodologies emerged, all of them named with a three letter code ending with the characters DD (standing for "-driven development"). The first character makes up the difference and the focus of each framework. Those are: BDD (Behaviour), FDD (Feature), DDD (Domain), MDD (Model) and TDD (Test)

Those methodologies of course mostly stem from the classic application software development industry, but still some approaches might be useful for safety related applications as well. For instance, test-driven development is an approach, which has one rule as essence: “Write the test first.” This assumes that a suitable testing framework is at hand, which allows to write the specification in the form of tests, before starting to implement the actual functionality. Since safety systems need to be tested thoroughly anyway and best practise, like unit-testing, should be applied to the development process according to IEC 61508, employing a methodology like TDD seems to be a good choice to ensure high software quality.

Besides the workflow used to create the software, IEC 61508 also suggests a clear structure of the software in modules as a good design practise for software. Good ways to structure software for various appliances are usually described by so called patterns.

On the top level, the architectural patterns describe the general scheme how the components of a software application should interact. The goal of those patterns is usually to address hardware, performance or availability limitations on a high level of abstraction. Popular and commonly known architectural patterns for instance are: Event-driven architecture, Model-View-Controller, Microservices, Service-oriented architecture, and so forth.

Design patterns are located on a lower abstraction level than architectural patterns and are used to either simplify the structure of code or to build code in a way that is easier to understand for

a developer used to the pattern. Lowering the complexity barrier is a major factor in avoiding systematic failures. It must be noted though, that a wrong usage of patterns may also increase complexity of software.

Design patterns are available for all possible levels of software engineering. Clearly, since object-oriented programming is the predominant design paradigm for application software, many patterns exist around this topic. To name a view popular: Factory method, Object pool, Singleton, Facade, Proxy, Template method, Scheduler, etc.

For safety systems it seems adjuvant to leverage such patterns as well. Specifically the concurrency design patterns - like Scheduler or Thread pool - are of interest for embedded systems as well. Mostly such patterns are already incorporated into existing libraries or even operating systems.

3.5 Other concepts

The introduction to this chapter already foretold the problem that targeted integrity levels may not be achievable by single systems, even though they leverage one or more of the above presented methods. In order to reach those higher integrity levels, the system needs to be decomposed into modules, which are connected together. The resulting system is called a distributed system, i.e., independent subsystems are connected to each other with a suitable communication medium.

With distributed systems and the communication channels involved, we face a whole new category of problems that arise in such systems. To name only a few of them: [6]

- Leader election
- Mutual exclusion
- Consensus
- Clock synchronization

A solution for these problems is again a prerequisite to enable the service required for TMR as listed above. It should be clear to the reader that none of these problems is easy to solve. A huge amount of research has been and is still being conducted in this problem area. [13] [31] [32]

3.6 Applications for fieldbuses

Several fieldbus systems are available on the market, which employ safety properties either in their core or as extension. A selection shall be outlined in the following.

SafetyLON

The Local Operating Network (LON) is a fieldbus system standardized in EN 14908. The SafetyLON project aims to extend LON with a so called “safe node” [29], which provides means to

achieve compatibility with IEC 61508 SIL 3. The focus of SafetyLON is on safe communication by extending the network protocol LonTalk and the network nodes with safety properties.

A first description explains that due to backwards compatibility any existing network must be extendable, hence establishment of a dedicated safety-network is not feasible. Therefore the prevalent hardware is extended. The components of a SafetyLON node are depicted in Figure 3.4.

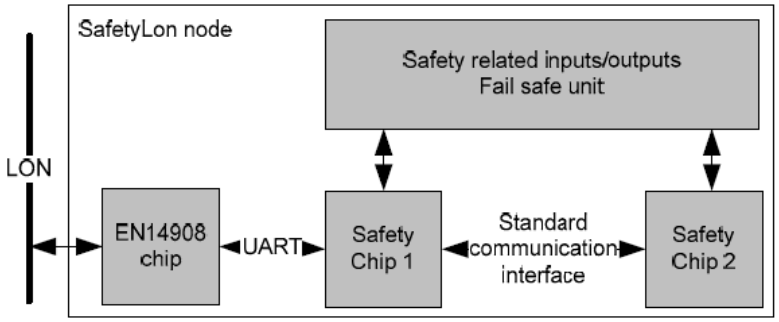

Figure 3.4: Components of a SafetyLON node [29]

SafetyLON employs a 1oo2 system in hardware by adding two more MCUs. The software stack is modified to comprise of three layers, one of them being the Safety Layer taking care of safe communication. For the application software, the safe communication is therefore fully transparent. The protocol for safe data transfer is embedded into the standard LonTalk (protocol tunnelling).

As the chosen hardware does not provide excessive means of self-test functionalities, the safety software has to provide all self-test routines. This of course comes with loss of performance.

CAN based solutions

Based on the CAN bus technology there are several solutions for safety related applications. Two of those are “CANopen safety”, standardized in EN 50325-5 [12], and “SafetyBUS p”.

Within a SafetyBUS p network only safety-related devices are used exclusively, which are already multi-channel internally. Parallel processors process data simultaneously and only if output signals do match an output is generated. This is a 1oo2 system.

CANopen safety employs functional safety communication based on CANopen (EN 50325-4). It employs the basic idea to transmit each safety-related message twice. The second message has bit-wise inverted data. On reception, the two messages are compared. The CANopen-Safety-Protocol (CiA 304) is used. The standard does not specify any safety guidelines or requirements for the devices themselves, but uses two CAN controllers internally for redundancy.

PROFIsafe

PROFIsafe is specified in IEC 61784-3 and is a pure software solution. It only adds various safety precautions on top of the existing protocol for better fault detection.

openSAFETY

“The first open and bus-independent safety standard for all Industrial Ethernet solutions.”
This is the slogan on the openSAFETY website. openSAFETY is a protocol stack available as open-source software, which is certified by TÜV Rheinland up to SIL 3 in accordance with IEC 61508:2010. The protocol stack is created by the Ethernet POWERLINK Standardization Group and designed to work on top of virtually any bus system. It claims to be the fastest IEC 61508 SIL 3 communication solution on the market. The protocol is furthermore standardized in IEC 61784-3 FSCP 13.

Figure 3.5: openSAFETY interoperability [open-safety.org, 20.9.2015]

As shown in Figure 3.5, openSAFETY is usable in many of the widely used bus systems. Yet it has to be acknowledged that the actual results depend on the used bus technology. Especially the real time capabilities vary a lot between the available bus systems.

The open safety protocol is designed to work in "Black Channel operation" and therefore only specifies the application layer 7 of the OSI model. The stack provides an extremely flexible telegram format and also allows to automatically redeploy a configuration into a new application, which has been stored in the safety controller beforehand.

An openSAFETY network may comprise of up to 1023 safety domains with up to 1023 nodes each. The unique feature is that each domain can span over different and inhomogeneous networks. A Safety Configuration Manager monitors all safety nodes within a domain.

A special focus is laid on cross-traffic transfers, which allow direct end-to-end communication between nodes, which is not routed over a central master node. The protocol encapsulates all safety data within a standard Ethernet frame and guarantees fault-free transmission by providing precautions like in-frame data redundancy, timestamps and unique frame identifiers.

CHAPTER 4

Concept

This chapter presents a concept to tackle the problems and challenges identified for Fire Detection and Fire Alarm Systems (FDASs) in the preceding chapters. After a short requirements analysis, a framework that matches the assessed demands is specified. Subsequently, the intended behaviour and the internal structure of the framework is described in detail.

4.1 Requirements analysis

As we have seen in the previous discussion of state-of-the-art functional safety methods (Chapter 3), lots of hardware, sophisticated software and thorough design processes are required to achieve high level safety goals.

In the context of domains like the fire safety domain on the other hand, we encounter rather soft limits and requirements regarding the safety (compare Chapter 2 on page 7), but stringent requirements regarding processing power, memory size and energy consumption are given. Additionally, one should not to forget about a competitive market limiting financial flexibility of manufacturers.

We may define a bottom line of requirements for the fire safety domain, derived from the applicable standards, as follows:

- The Control and Indicating Equipment (CIE) must be able to indicate alarm and fault conditions. The overall time limit for indicating an alarm condition must not exceed ten seconds. This includes the whole process from retrieving an alarm signal from a sensor, processing the signal and finally issuing the alarm.

- After a reset condition, e.g. after rebooting the device, the current condition and its according indication have to be established no longer than 20 seconds after the reset.

- The CIE must be able to detect faults. We distinguish between faults of devices connected to the CIE such as sensors and system faults of the CIE itself. After detecting a fault condition on the CIE, the condition has to be indicated within 100 seconds.
- The requirement for audible indication for at least one hour in case of complete loss of the main power source of the Power Supply Equipment (PSE) implies that - assuming only the battery is left as power source - the whole system has to be designed to survive at least one hour on battery.
- In order to indicate a de-energized CIE on another device, the CIE has to provide means for detecting such a situation. This might be a physical line which is directly connected to some indication device or a smart network communication solution, which can conclude the power loss (or any other crash failure) from previous known states of the CIE affected.

These minimum requirements are rather easy to achieve with an up to date embedded system. The time limits for alarm alerts and re-establishment of state after reset operations stem from earlier times many years ago, where powerful integrated circuits were not common. With the processing power of today's embedded hardware it seems that these actions should be possible in a fraction of the specified time. Of course the actual time needed strongly depends on where the alarm signalling has to happen. In case the signalling is distributed over a wide and slow network, this may of course take a bit longer.

Fault detection, if implemented independently in the various subsystems (e.g. sensor networks), is also possible in a quick manner. On chip level, many safety chips already include special hardware circuits to detect a wide range of possible hardware faults already in hardware, so time-consuming checks on software level can be reduced a lot. The communication overhead (for the network of CIE) is again subject to the structure of the FDAS. Still the limit of 100 seconds shall be far outside the scope of targeted limits for current systems.

The "survive one hour on battery" requirement can be tackled by using either a battery with big capacity or by reducing power consumption to a level, where a reasonable battery can be used. With modern hardware, the power consumption is in a range, where this should be easily possible as embedded systems tend to get smaller and smaller and the power needed by a chip decreases with its size. Mobile phones with really powerful processors can be operated for many hours under full load with rather small batteries. Since fire incidents, which cause higher load on the hardware, are rare cases, typical battery blocks used for uninterruptable power supplies should easily suffice to power the system for at least one hour.

In addition to the soft requirements presented above, this thesis tries to undermatch the requirements as much as possible and add additional ones as it seems practicable and adjuvant to have these for a modern FDAS. In addition to that, and although not strictly required by the EN 54 standard, any new functional safety related equipment built today should comply to the requirements retrieved from the analysis conducted as outlined in IEC 61508. Fulfilling these requirements is definitely a benefit regarding market positioning and insurance coverage.

Therefore, the "requirement" to achieve a system that mostly complies to Safety Integrity Level (SIL) 2 or even SIL 3 on a technical level are added on top. This thesis does not provide any

means of documentation which is asked for by these targets, but focuses mainly on improving reliability for the overall FDAS by increasing the reliability of the individual devices (CIE). The **main focus is therefore on the devices** and not on the network connecting those devices.

In terms of IEC 61508 the concept therefore aims for a system that has single fault tolerance and therefore is a Type B system with a target Safe Failure Fraction (SFF) of approximately 90%.

Fault hypothesis

The concept is based on a fault hypothesis which can be outlined as follows:

The first and most important definition to add to the fault hypothesis is to at least tolerate **a single fault in the whole FDAS**, where an FDAS comprises multiple CIE units connected to each other (see also Figure 4.1). This trivial sentence may sound ordinary, but actually has a huge impact on the whole system design as will be described in Section 4.2. The permitted failure modes are limited though. Specifically Byzantine failures [26] in general are excluded. In the temporal domain of faults the focus is on consistent crash failures (permanent faults) and transient faults.

Byzantine faults are omitted, as they would require a much more involved solution, particularly in terms of necessary hardware, and this thesis is specifically tailored to resource limited hardware. Literature shows many variants of problems with Byzantine faults and defines a lower bound for tolerating k Byzantine faults to a minimum of $3k + 1$ nodes with $2k + 1$ connectivity [17]. Of course those numbers may be different if prerequisites change such as the scheduling model, switching asynchronous or synchronous communication and the like. Nevertheless, even for a single Byzantine fault we would need four nodes, which is beyond the goal of this thesis.

Another goal of the concept is to let CIEs form independent fault containment regions. In particular, any fault occurring within a CIE, which results in a failure, must not be propagated to the outside of the CIE, hence the error must not lead to another fault in another device.

4.2 Proposed model

After having had a look at the requirements and into state-of-the-art solutions for well-known safety and reliability problems, we shall now propose a suitable concept to solve the problem described in Section 1.3, whilst fulfilling the requirements as outlined in Section 4.1.

The proposed concept provides a **framework consisting of hardware and software parts** building the necessary foundation for application software to be run on top of the framework. This system, comprising the hardware and the software, will be called a “CIE unit” throughout this thesis.

Multiple of these units are to be connected in a network to create the full-featured FDAS. As one unit is **not capable** of containing all possible single faults (refer to Chapter 5 on page 47) and as single faults on the network have to be covered as well, it is necessary to wisely choose a proper network connection, network structure and according network protocols in order to comply to

the single fault tolerance requirement for the overall FDAS. A solution for the network part, specifically tailored to the fire safety domain, is presented in [21], but the presented approaches for fieldbuses in Section 3.6 can be used as well. Those are SIL 3 compliant and are therefore a solid basis for an overall SIL 3 compliance.

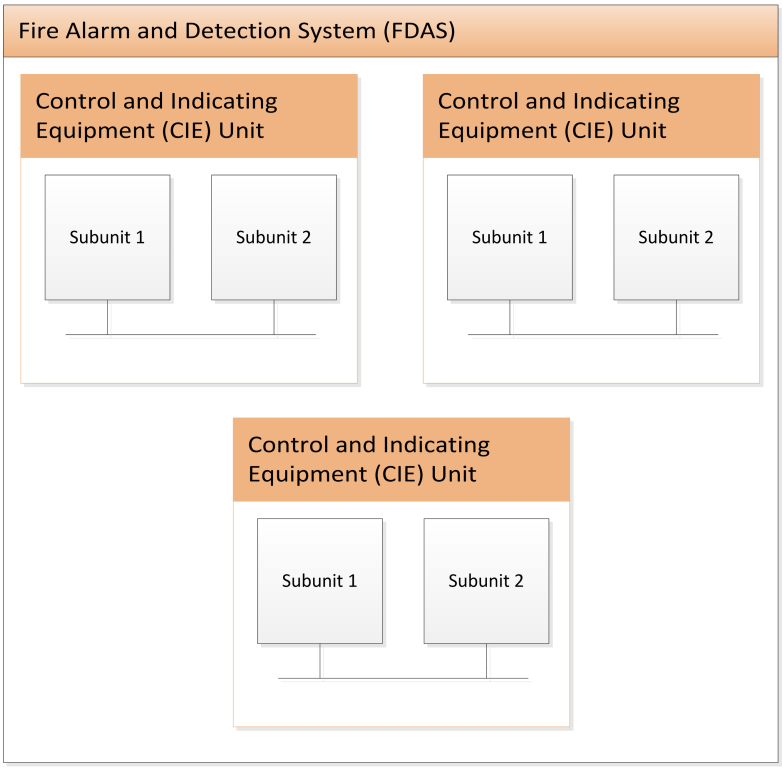

Figure 4.1: Basic structure of an FDAS not including supplementary equipment

Figure 4.1 shows an exemplary structure of an FDAS with its direct components, the CIE units. The figure does not show sensor and actuator networks or other equipment, which is likely present for network connectivity or other means.

Basic principle

The basic principle of the proposed hard- and software framework is based on the hot-standby pattern (see Section 3.3), with the special addition that both components include the switching logic. The main reason for choosing this specific pattern over the described Dual Modular Redundancy (DMR) or Triple Modular Redundancy (TMR) patterns is the simplicity of the setup and the saved costs due to saved hardware.

DMR is very similar to the hot-standby pattern, but the cost advantage over TMR is significant, as we save a complete subunit and the voting logic. Moreover, by replacing the voting

logic in the DMR pattern by a cross-checking mechanism, another single point of failure can be avoided.

Of course - and this should be clearly pointed out here again - neither the pure DMR approach nor the hot-standby pattern are generally able to tolerate all possible single faults of the system. This is acceptable for the intended solution proposed though, as further detection of errors over multiple units can be done on a higher level by other nodes of the network. Still, by modifying the hot-standby pattern we are able to detect several abnormalities on a node level and can furthermore even correct for some of these faults on the node level without bothering any other nodes on the network, which is already an important step towards improving system reliability. A node forms a fault containment region in this case. The **ability to tolerate some faults** is an extension of the classic hot-standby pattern, whose sole intention is to be fail-safe, but does not care for being (partially) fail-operational.

As implied by the hot-standby pattern, one CIE unit consists of **two identical subunits**. The subunits use the same hardware and the same software and are equivalent partners (except the active/passive state). Each of these subunits consists of a Microcontroller Unit (MCU) and a bus connection hardware including a bus-guard. A prerequisite for the MCU is the **availability of a general error pin**, which is controlled by internal hardware fault detection. This important prerequisite should be mentioned here as it is of special importance for the overall principle. Note that many hardware faults could be also detected by employing complicated software constructs (double execution of code, frequent self-tests, etc.), but the availability of MCUs with dedicated support for hardware fault detection eases the development process and provides a far superior detection coverage than any software solution. Therefore this concept requires hardware with self-test capabilities. All other requirements imposed on appropriate MCUs are specified in Section 4.2 on page 39.

The two subunits - as shown in Figure 4.2 - are connected with each other in four distinct and independent ways:

1. The first connection is via the normal network bus.
2. The second connection is a direct connection between the units via serial communication lines.
3. The third connection is a low-level hardware line, directly connected to the general error pin of the opposite MCU and to the bus-guard of the same unit. This pin has to be low-active - a zero level indicates an error - to ensure that power loss is detected as failure.
4. The fourth connection is a control line from one unit to the bus-guard of the other unit.

This setup allows mutual cross-checking of the subunits in order to detect faults on the other MCU and allows to react upon such a fault. The concept of mutual cross-checking is necessary for tolerating faults within the boundaries of a CIE unit and comprises hardware fault detection and software fault detection. Hardware fault detection is based on MCU **internal fault detection circuits**, whereas software fault detection is based on **software-state synchronization**. The monitoring responsibilities will be illustrated separately for hardware and software, as the complexity involved in the two cases is substantially diverging.

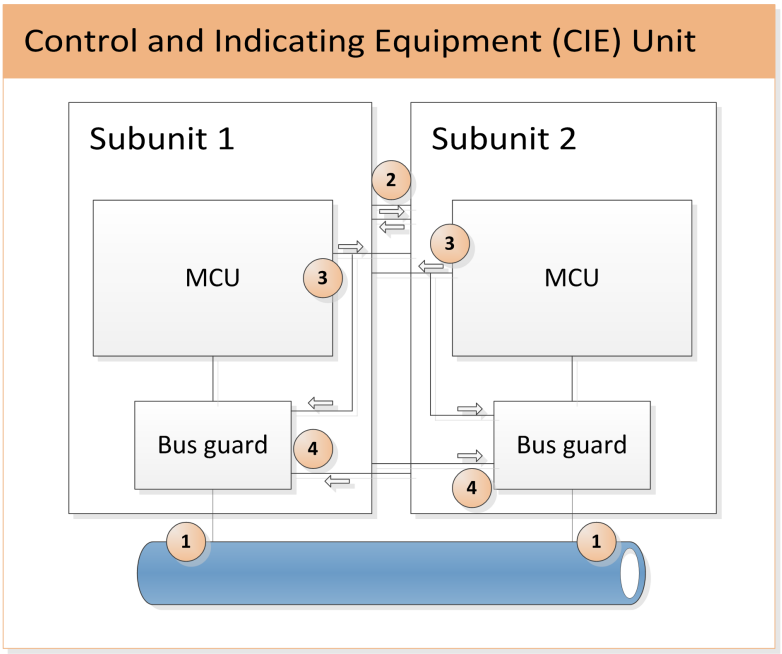

Figure 4.2: Hardware structure of a CIE unit

Information exchange between subunits

This section will further describe the usage of the connections mentioned in Basic Principle-Section 4.2 on page 36. As is the nature of a hot-standby system, it requires the definition of the "active" and "passive" subunit. While both subunits perform all calculations, receive the same input data and execute monitoring functions, only the active one may actually transmit data for other CIEs via the network bus to other units and via the fieldbuses to sensors and actuators.

The first connection, the network bus, has its main functionality in allowing the reception of network traffic on both subunits independently. Moreover, it allows the passive subunit to verify the outgoing traffic of the active subunit. Additionally this connection will be used for validation processes between the subunits as described in the scenarios below.

The monitoring of the hardware is kept simple and utilizes the third connection, which is a direct indicator for a hardware failure on the other subunit. Reliable detection can be achieved by monitoring the electrical state of this direct connection.

Looking at the software layer, monitoring of the software state is much more complex. For this purpose, the framework provides an abstract state-machine, whose concrete peculiarity is defined by the application software. Each state change, triggered by the application software, is synchronized between the two subunits. This state synchronization is performed by using the second connection, the serial communication lines. Those serial communication lines are required to be full-duplex and must allow independent initiating communication from both sides.

As this connection is time-critical for the overall application performance, it should be a fast connection. For this purpose we strongly suggest to use two Serial Peripheral Interface (SPI) connections.

Finally, the fourth connection line has its importance when it comes to disabling access of one subunit to the network bus. Take any situation where a subunit considers the other subunit to be not trustworthy anymore, at this point it is of utmost importance that the detecting subunit has the ability to cut off the faulty subunit from the system completely to avoid any kind of disturbance by the faulty subunit.

Hardware requirements

As already described in Section 4.2, the MCU is required to feature internal hardware fault detection. This is necessary to identify misbehaviour, which is crucial for achieving the desired SIL 3, which requires Central Processing Unit (CPU) tests and memory checks.

To enable detection of transient processor faults, we require the MCU to provide two independent, lock-stepped CPUs. Furthermore, all memory shall be secured with Error-Correcting Code (ECC) mechanisms to be able to detect memory errors and bit flips. Again, this could be done in software, but specialized hardware is a better approach. Additional built-in self test routines for all core components of the MCU should be present in order to gain an overview of the system's hardware health easily.

Possible faults have to be detectable by the software running on the MCU and by external hardware via dedicated connection pins.

The subunits use dedicated software tasks, which monitor the hardware state of the underlying hardware as well as the hardware of the other subunit. This task plays a very important role, because any kind of hardware error or failure is a severe system fault and according action has to be taken upon. The task is hence responsible to shut down the subunit, if the error occurred on the subunit currently running on, or to shut down the other subunit, if the error occurred there and the subunit itself did not detect the situation and shut down on its own in a reasonable time frame.

Bus-guards

The bus-guard used to control access of the MCU to the network bus should be as simple as possible, since any complexity involved would just provide more possible fault scenarios. Therefore bus-guards are required to consist of passive electronic parts only, such as relays. Relays allow physical separation of the MCU from the network bus.

More precisely, the bus-guard has to have a **safe state** in case of self-failure, which is to make the complete subunit **fail silent**, i.e. the subunit is cut off the network bus. To achieve this, mono-stabil relays are recommend, which will cut off the connection once the voltage level drops on the control pin.

Software

As hardware has its internal independent fault detection mechanisms, software needs an equivalent as well.

Software monitoring can be achieved by one of the following approaches, or can be a combination of these:

- A hardware watchdog. Considering a time-out based watchdog, the software tasks will have to repeatedly reset the timer of the watchdog. If the timer goes down to zero, the watchdog will initiate a hardware reset of the complete MCU ultimately forcing a clean restart of the system.
- A dedicated software task. This can be implemented with either of two paradigms:
  - In a similar manner like the watchdog paradigm above, where the software tasks have to send a signal to the monitoring tasks in order to indicate being alive.
  - The monitoring task permanently watches the state of the software tasks and determines progress by asserting specific time-outs for expected state changes.

A first approach would be to have a dedicated task on the same MCU that is responsible for monitoring the other software tasks. This method would have the advantage of easy implementation, but has some major drawbacks in what can be accomplished with it:

- It is hard to guarantee that no software bug may corrupt memory areas assigned to the dedicated monitoring task, hence, a bug in the monitored software task may potentially manipulate the monitoring task's memory and consequently the behaviour of the monitoring task.
- Any oddity in timing of tasks will also influence the monitoring task.
- If the monitoring task itself fails for some reason, there is no way to detect this failure.

Considering these drawback it seems obvious that it is not reasonable to have the monitoring task on the same MCU, as it is not possible to have a totally separated software task to verify software state on the same MCU. Still it does make sense to have a dedicated monitoring task, which checks software states, but for an entirely different purpose as described hereinafter.

Cross-checking

A much better approach is to let this dedicated monitoring task observe the software state of the MCU on the other subunit. To actually determine progress - which is usually measured as change over time - of the other subunit a comparison reference is needed. Two possible types of references for such a comparison are distinguished:

1. **Absolute reference:** Of course a globally precise solution would require the establishment of an absolute reference. Implementing a common time base within a CIE unit - or even a whole FDAS - requires sophisticated distributed algorithms for time synchronization in real-time systems. Adequate algorithms can be found in [24] and [5].

2. **Relative reference:** A relative reference has its scope only on a local domain, in this case the MCU. Progress is measured as change happening in comparison to this local reference.

In our use case it seems enough to go for a relative reference, that is a local clock or timer, running at a given frequency. To determine progress we need to define instants on the reference, where the current state is checked. Again for our use case, it seems reasonable to use the instant, where the local task commits a state change, as the same state change is expected to happen approximately around the same instant on the monitored subunit. We do allow a certain timespan the other subunit's state-change is allowed to be delayed. This bounded waiting time is required to avoid infinite waiting time on the results of the other subunit. If the limit is crossed, we may assume that the other subunit has failed. According actions will be taken by the subunit to shutdown the other subunit.

The chosen state-change commit instants, chosen by the application engineer, eventually define the synchronization points for the subunits.

By applying this scheme on both subunits, we establish a pattern, which shall be called a "lock-stepped state synchronization" with mutual cross-checking.

The proposed framework provides the necessary tools for setting up this pattern to control the execution of the predefined state flow on two subunits in a deterministic way. It is of course up to the actual application to define the intended state flow. As H. Kopetz describes in his introduction to fault tolerance, it "is absolutely necessary" to unreel end-to-end error detection on application level. Therefore, it is important to say that the **software using this framework is responsible** to define the granularity of the software states used to synchronize the subunits.

For instance, it may suffice in some cases that a state comprises the execution of a complete function of code, but for some calculations it may be necessary to define extra states for the if-branches of the validity check of the calculation result. This just depends on the criticality of performed actions.

See Listing 4.1 and Listing 4.2 for an abstract example of a coarse state definition and a detailed state definition.

Listing 4.1: Code with coarse state definition

```
void showResult() {
  newState(1);
  result = addNumbers(1,2,3,4)
  if (isValidResult(result)) {
    fputs("Successful␣result");
    ...
  } else {
    ...
  }
```

```
  newState(2);
}
```

Listing 4.2: Code with detailed state definition

```
void calculate() {
  newState(1);
  result = addNumbers(1,2,3,4)
  newState(2);
  if (isValidResult(result)) {
    newState(3);
    fputs("Successful result");
    ...
  } else {
    newState(4);
    ...
  }
  newState(5);
}
```

Synchronization messages

The protocol used to synchronize the state between the subunits is kept fairly simple. The only information transmitted are two types of messages.

1. State-Change messages. (type 0)
2. Command messages. (type 1) The command message may be one of the following
   - Subunit is shutting down. (CMD_DOWN)
   - Last transmission failed. (CMD_RETRY)

A message may not exceed the maximum message size the serial connection is able to send in one transfer, e.g. 16 bits.

The messages are secured by using odd-parity for the transmission (handled by SPI) and by using a message number, which is increased for each message.

Table 4.1 shows the message format. Note that the number of bits used for the message number is not critical as it is only used to identify if two succeeding message are distinct or a retransmission. It is suggested to encode all possible software states with an integer number.

0/1 - message type	number	content

Table 4.1: SPI message format

Network structure

As denoted already, multiple CIEs are connected in a network to comprise the full FDAS. This connection medium and its structure used for this network connection is generally not relevant for the concept presented here (except for one requirement), but of course it should guarantee some properties like a certain availability. This commonly involves redundant networks or ring-based structures. If a fieldbus is a feasable solution, one of the systems tailored for functional safety presented in Chapter 3 Section 3.6 might be a good solution. A clever network setup may already be able to tolerate a single fault of an arbitrary network component.

For this concept, there is one important requirement though: Both subunits must be able to **read all traffic** on the network connection target for the CIE, but must still be able to **address each subunit individually**.

As an example, we consider an IP-based network. In order to fulfil the requirement, one has to ensure that the subunits are in the same collision domain. Hence, connecting both subunits to a standard switch would violate the assumption as switches generally direct the traffic to the targeted device's port exclusively. An option would be to use manageable switches that allow to alter the configuration accordingly, or to place a hub in front of the CIE, which generally forwards incoming traffic to all ports. The subunits would then need to share an IP-address and would each have an address on their own, to make them addressable individually.

Be aware though that this setup yields a single point of failure, the hub or managed switch. A more intelligent setup would be required in such a case to circumvent this restriction.

The serial communication channel between the subunits, used for state synchronization, imposes two requirements:

1. The connection must be full-duplex, so each side of the channel can talk to the other side independently.

2. A hardware fault, e.g. a cable break-through, must be detectable by at least one side of the channel.

Implementation-wise the first requirement can be met by using two SPI channels, where each side is the master on one channel. The second requirements can be tackled by implementing a communication protocol and checksums that make data loss and corruption recognizable.

4.3 Behavior of the framework

For the subsequent description of behavior of a CIE, for each case it is assumed to let subunit 1 be the active subunit. Furthermore, for simplicity of describing the solution, each software task is assumed to be structured in a state-machine with three states, numbered 1 to 3.

The framework itself implements a state-machine as depicted in Figure 4.3.

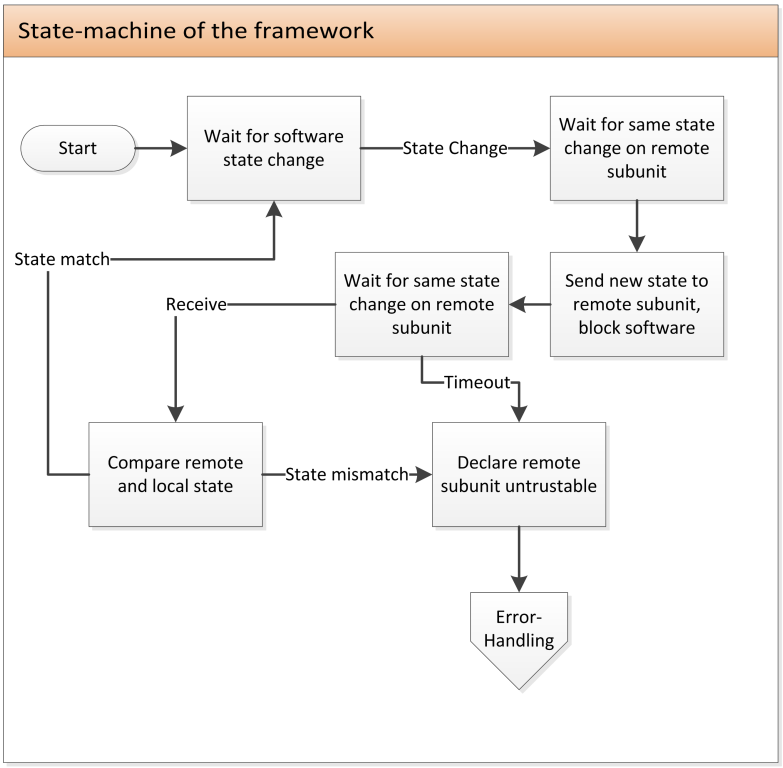

Figure 4.3: State-machine of the framework

Normal operation

Throughout normal operation, the basic interaction scheme works like outlined in Figure 4.4.

Step 1: Subunit 1 receives input from some field device and starts processing.

Step 2: Subunit 2 receives same input and starts processing as well.

Step 3: Assuming subunit 1 finishes the first part of its current task and therefore has a state change from state 1 to state 2. The state change is reported to the framework accordingly.

Step 4: The framework reports this state change to subunit 2 and now blocks the task until the framework running on subunit 2 reports the same state change being committed there. The reporting happens via the dedicated serial communication lines number 2 in Figure 4.2.

Step 5: Subunit 2 finishes the first part of the task, too.

Step 6: The framework communicates that to subunit 1 and lets the task continue as the successful state change on subunit 1 has already been received.

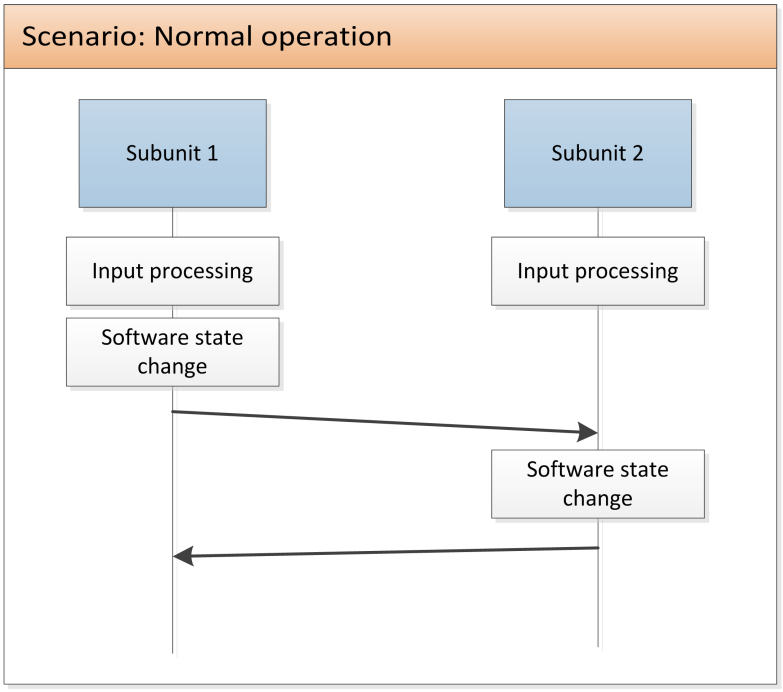

Figure 4.4: Typical subunit interaction for normal operation

Step 7: The framework on subunit 1 now unblocks the task and it continues.

This scheme repeats for every task the application software executes.

4.4 Booting the CIE

Bringing a CIE to life for the first time requires some initial actions. This timespan, from the un-powered state until the CIE is ready for operation, is called the boot-phase.

The boot-phase comprises the following steps:

1. Each subunit has to initialize its hardware modules.
2. If the hardware features built-in hardware self-tests (e.g. CPU test, RAM and flash ECC tests, memory tests of I/O modules), those tests are executed.
3. The software-part of the framework is loaded.
4. The framework is initialized.
5. The framework waits for a configured timespan and listens for possible network traffic.

6. If no traffic is observed the framework sets its role according to its configuration (active or passive), otherwise it chooses the passive role[1]. See below for more details.

7. The framework loads the application task.

Determining the active subunit

During the boot-phase a procedure is required to define which subunit will be the active subunit and which will be the passive subunit. This very problem is the minimum instance of the leader election problem in rings[2] already mentioned in Chapter 3. The problem, solutions for it, and proofs are extensively described in [6].

For the purpose of this framework though, it seems to be an overhead to run a full leader election process/algorithm for determining the active subunit. Therefore, it is proposed to define the initial state of a subunit in the subunit's configuration. This has the advantage of a faster boot process and guarantees a deterministic behaviour of the CIE unit.

4.5 Shut down of a subunit

In case of a failure being present within the CIE, it might be necessary to disable or shut down a subunit to ensure fail-silence of the subunit on the main network bus. A subunit can be shut down in multiple ways, depending on the situation. In order to assure fail-silence, the bus-guard (connection number 4 in Figure 4.2) is forced to cut off the associated subunit from the main communication bus (Number 1 in Figure 4.2).

In case of a software fault, the failure is detected by the opposite subunit, so the bus-guard is triggered by this subunit via communication line number 4 in Figure 4.2.

In case of a hardware fault, the error is detected by the on-board hardware fault detection logic and is therefore consequently triggering the bus-guard cut off by communication line number 3 in Figure 4.2. This cut off can be initiated by either of the two subunits, depending on which one detects the failure first. If the hardware fault is not too severe, the subunit itself might be able to gracefully shut down, otherwise the other subunit has to react.

[1] Although this case should never happen as both should subunits should always boot at the same time. This is a safety precaution to not interfere with a possibly active subunit.

[2] We truly have a ring-structure at hand, as the two subunits are connected by a full-duplex channel.

CHAPTER 5

Results

In this chapter an evaluation of the concept proposed in Chapter 4 is given. First, some thoughts on the concept itself are provided and how or to which degree it satisfies the requirements stipulated. After that, a prototype implementation of the proposed framework is outlined as a proof of concept.

5.1 Analysis of the concept

This section provides a comparison of the requirements defined in Section 4.1 and the applicable standards of Chapter 2 and the potential of the presented concept in Chapter 4.

Difference to existing solutions

The presented concept is fairly different to the solutions described in Section 3.6. Those solutions focus on safe communication on the network level exclusively and aim for high Safety Integrity Level (SIL) compliance on that level. For achieving a fully certified product though, it does not suffice to ensure safe network communication alone, but the actual devices sending and receiving data are crucial in the end-to-end communication as well. A safe network connection is only as good as the quality of data transferred over it. If the connected devices are corrupted, the overall safety of the system is affected, even though the network might be perfectly fine.

The outlined concept therefore focuses on the devices and improves their resilience against failures. Together with a safe network and an appropriate application software, the presented framework is one brick towards reaching the goals for functional safety on the overall Fire Detection and Fire Alarm System (FDAS).

Requirements met

First, we will revisit the requirements of EN 54.

"Fire alarms have to be indicated within ten seconds." This should be easily possible as the Control and Indicating Equipment (CIE) has close to no load throughout normal operation. Scanning the sensor bus every half a second and assuming the application's evaluation code takes another half second, then there is nine seconds left. Furthermore, assuming the evaluation has 100 synchronization points, the overhead of the synchronization between the subunits ($50Mbit/s$ - two bytes per synchronization) makes up less than a millisecond. The overhead is therefore negligible and there are still nine seconds left to send this information over the network bus to another CIE. This is achievable with almost any network bus system.

The 100 second time limit for entering a fault warning condition should be easy as well. If we only consider the CIE itself and leave the sensors aside, the detection of a failure should happen at the next state synchronization point for software failures and will be detected immediately for hardware problems. If we assume the application to have activity (in terms of state change towards the framework) at least every ten seconds, for sending keep-alive signals over the network bus for instance, then there is more than 90 seconds margin.

The re-establishment of function of operational devices after an interruption may not exceed 300 seconds. There are only a few conditions where a device would be considered still being operational after some interruption. In case of a power-outage (and draining of the backup batteries) the re-establishment is the same as resetting and booting the CIE. This takes for sure less than a second for the framework part, so the limiting factor is the application software in this case.

The requirements of Sections 13.4 and 13.5 - separate monitoring hardware and protected memory - are fulfilled by definition as we have two physically separated subunits monitoring each other, each with a separate timebase and with Error-Correcting Code (ECC) protected memory.

Looking at the IEC 61508 standard, a full analysis whether the FDAS made up of a bunch of CIE units reaches a certain SIL is not possible, as this would require to have the full-fledged application at hand and operations is part of the safety integrity as well. Both is beyond the scope of this thesis, therefore it will be outlined why the concept supports the application manufacturer to reach higher SIL.

SIL 2 requires "good design practise". This is clearly the case, as state-of-the-art methods are used, which are considered best practise. Moreover, the framework helps to modularize the application, which is an important criterion as well.

Towards SIL 3, the single fault tolerance is definitely a helpful property, which counts as "sophisticated design method". The technical challenge to reach a SIL 3 is for sure to reach the necessary failure rate of $< 10^{-7}$ dangerous failures per hour. Having a good redundancy concept of the overall application is the first step towards this goal, where the presented framework is a part of. In order to get a SIL 3 certification though, it is of course important that the underlying hardware and the used network bus are also developed according to the standards.

Overall the concept should therefore fit pretty well to what is asked by the standards and what can be considered useful for the application.

SPI communication

The communication protocol is a light-weight protocol. By definition a message shall not exceed the size of what can be transferred in a single Serial Peripheral Interface (SPI) transmission. For many hardware modules this is 16 bits. As described in the concept, the software states shall be encoded as unique integer numbers. This limits the possible number of software states to $2^14 = 16384$ if one bit is reserved for the message type and one bit is used for the message number[1].

In contrast to the main network bus connection, which can be freely chosen as long as the minimum requirements are met, the internal connection and its protocol is under full control of the framework. As outlined above, the use of SPI as technology is recommended. Table 5.1 provides an overview how typical communication errors as specified in IEC 61784-3 are being handled for this internal connection and which safety measures are used.

	Safety measures								
Communication errors	Sequence number	Time stamp	Time expectation	Connection authentication	Feedback message	Data integrity assurance	Redundancy with cross checking	Different data integrity assurance system
Corruption					✓			
Unintended repetition	✓							
Incorrect sequence				- not applicable -				
Loss				✓	✓			
Unacceptable delay			✓					
Insertion				- not applicable -				
Masquerade				- not applicable -				
Addressing				- not applicable -				

Table 5.1: SPI connection - errors and safety measures

Communication errors marked as “not applicable” are not considered for the SPI connection, as this is a one-to-one connection with a dedicated channel for each communication direction, therefore those errors cannot occur by construction.

[1] It suffices to use a single bit for the message number as only two subunits are connected and only one source for messages is present for each communication direction. Therefore, alternation of the single bit already allows to detect repeated transmissions.

Fault containment

The framework handles all, except one, of the described fault scenarios in a way which ensures that a failure does not cause a fault in another component outside of the CIE.[2] So the fault is contained inside the CIE.

The only exception, where a fault may influence the remaining components in a negative way, is the situation, when the internal serial communication channel for state synchronization fails completely (e.g. a full cable break-through). In this case, some communication has to happen via the network bus, which causes increased load on that bus. If this communication can not be kept local to the CIE due to the chosen network bus technology, the fault containment assumption is violated, as the increased traffic may interfere with regular traffic. In an extreme unlikely worst-case, all serial lines of all CIE are broken and the full traffic has to be handled by the network bus.

5.2 Proof of concept

As the proof of concept, a prototype implementation of the proposed framework was chosen. The main focus was on implementing the software state synchronization and on testing typical fault-scenarios to verify the robustness of the system.

Selected hardware

The prototype was implemented by using hardware parts that were specifically designed for safety applications by the manufacturer. This may sound expensive and counterproductive to what has been stated in the chapters before when talking about the low-cost requirement of the hardware, but actually those chips are fairly cheap (only a few dollars), so they really fit best for the purpose of this work.

The chosen products were the RM42 Launchpad boards from Texas Instruments. These boards carry an ARM Cortex-R4 32bit Safety-Microcontroller Unit (MCU) (RM42L432) with JTAG debugging interface and UART interface as well as power supply over USB right from the PC. It additionally provides two SPI interfaces, but also multiple Controller Area Network (CAN) and Local Interconnect Network (LIN) bus interfaces, next to a multitude of timers with advanced pulse-width modulation (PWM) functionality. These properties clearly make this chip targeted towards the automotive industry, which itself is of course heavily influenced by the safety standard ISO 26262 “Road vehicles – Functional safety”, which is based on the IEC 61508 standard.

The chosen RM42L432 chip has a clock-speed of only $100MHz$, but while writing this thesis Texas Instruments released a newer version of the Launchpad series, which now comes with the RM46 MCU providing twice the speed. Moreover the RM46 series also provides the ARM Cortex-R4**F** architecture, which includes a floating point unit. This comes in very handy for applications requiring heavy usage of floating point arithmetic. Albeit the clock-speeds

[2] Of course the unavailability of the CIE is an interruption of the normal state to other components as well, but it does not cause those components to fail as well.

of the safety-hardware will not let the masses exult, this has the big advantage of low power consumption.

In regards to safety-applications, some technical features of the platform have to be specifically outlined:

- ECC support for memories to detect and correct memory corruption.
- Memory protection of almost any configuration memory including vital areas like interrupt vector tables.
- Real time timer as independent separate hardware module, which is intended to be leveraged as a local timebase.

A few more advantages of the chosen board shall be named, which are considered beneficial for this prototype:

- The hardware fulfils industrial standards for safety applications and may therefore be certified for IEC 61508 and ISO 26262.
- The hardware is developed following IEC 61508 standards up to SIL 3.
- Low energy consumption is a central criterion for FDAS, which is perfectly covered by the selected hardware.
- The hardware provides enough interfaces and I/O pins to implement the proposed concept.
- The usage of Real-time Operating Systems (RTOSs), like FreeRTOS, is supported and software ports are available.[3]
- External fault injection is possible to a certain degree.
- A real safety-project would surely choose such a hardware to ensure easy certification.

Hardware implementation

In Chapter 4, Figure 4.2 shows the suggested hardware setup for a CIE unit. The described communication facilities are implemented or simulated using the I/O modules of the RM42 board and a terminal application on the PC.

Figures 5.1 and 5.2 show the setup of the prototype hardware with connected pins, but without the USB connection to the PC.

[3]For the prototype implementation, RTOS has not been used, but generally this seems to be a good idea to ease certification processes when using certified RTOSs.

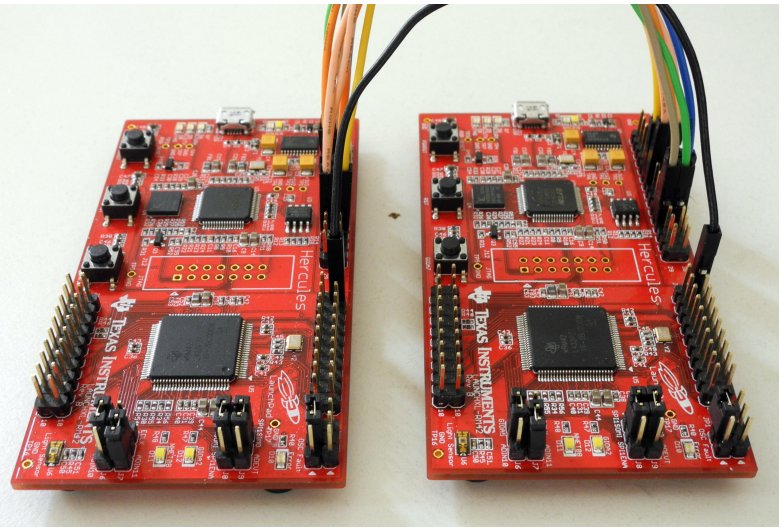

Figure 5.1: Two RM42 Launchpads

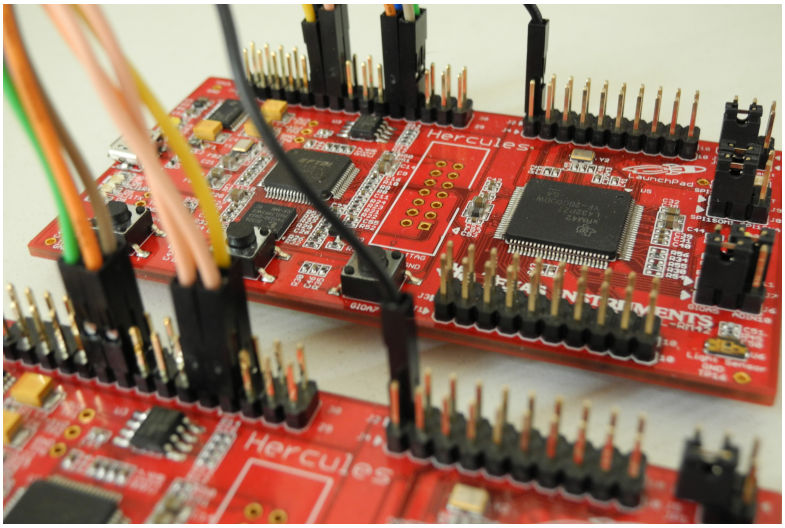

Figure 5.2: The serial connection is realized with SPI2 and SPI3 to achieve two independent full duplex connections.

Serial communication

The serial communication lines for software state synchronization are implemented as SPI connections as suggested in the concept. The advantage of SPI is a high bandwidth. The test hardware allows baudrates up to $50 Mbit/s$. To ensure independent full duplex communication of both subunits, two separate SPI connections are used, one for each direction, where each subunit is (clock) master of one connection. By construction, SPI is already full duplex by itself,

but with the limitation of having exactly one master chip, which controls the clock and therefore initiates every communication. The concept, however, requires each subunit to be able to initiate a data transfer independently. This is solved by using two SPI connections. The back-channel (SOMI; slave-out-master-in) of each connection is used to transfer a strictly increasing value, which allows to detect broken cables or other means of incomplete data transfers. In order to be able to detect broken SOMI or MOSI (master-out-slave-in) lines, the channels have to use a parity bit. More precisely odd-parity has to be used as this guarantees a parity-mismatch when only zeros are transmitted, i.e. when a line is broken.

Network bus

Since the hardware does not support a physical layer for Ethernet, the primary network bus is simulated by using the UART connection of the hardware board. The UART is tunnelled to the PC via the USB interface and enables simple data transfers via COM-port.

Hardware error pin

The hardware error pin of each board is connected to a general purpose I/O pin of the other board. These GPIO pins are configured to trigger hardware interrupts whenever a falling edge on such a pin is detected. This assures immediate reaction to possible hardware faults. Unfortunately the launchpads do not provide direct access to the hardware error pins as those have an LED connected for demonstration purposes. A workaround for this problem is to exploit the Error Signaling Module (ESM) (see Figure 5.3), which allows to react to a vast majority of hardware faults with software routines. This allows to trigger a separate error pin. The errors are grouped into three groups with group 3 being the most severe one.

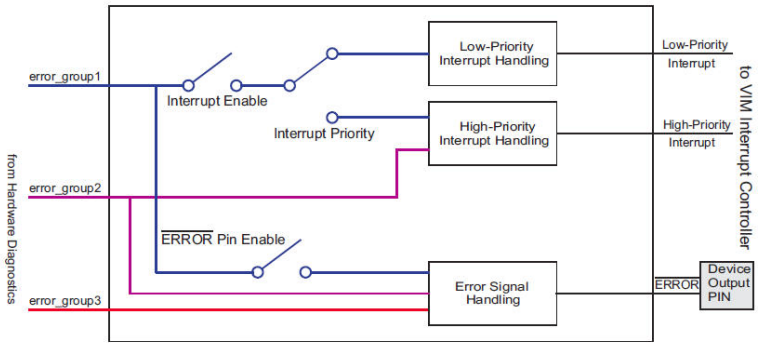

Figure 5.3: Error groups of the Error Signaling Module with interrupt mapping and error pin mapping.

As one can see from the diagram, it is not possible to react on errors in group 3 by interrupt handler, so this form of severe hardware failure could only be handled if the built-in LED would have been removed.

Purpose / Connection	Subunit 1	Subunit 2
SPI2 CLK	J10-24	J10-11
SPI2 SIMO	J10-23	J10-14
SPI2 SOMI	J10-22	J10-12
SPI2 CS	J10-21	J10-13
SPI3 CLK	J10-11	J10-24
SPI3 SIMO	J10-14	J10-23
SPI3 SOMI	J10-12	J10-22
SPI3 CS	J10-13	J10-21
GND	J2-1	J2-1
ERR OUT (GIOA0)	J2-3	J2-4
ERR IN (GIOA1)	J2-4	J2-3
J8 and J9	1=2	1=2
BUSGUARD local (GIOA7)	J2-09	J2-10
BUSGUARD remote (GIOA4)	J2-10	J2-09

Table 5.2: Pinout and jumper configuration used to connect the RM42 launchpads

The functionality of the error pin is easily testable with fault injection by writing a specified key value into a register. This forces the error pin to a low-level.

Bus-guard control line

Due to the lack of the network bus interface, building physical bus-guards does not seem useful. Therefore this functionality is controlled by the PC as well.

Pinout

Table 5.2 shows the detailed pinout and jumper configuration used to connect the devices.

Software implementation

Although the used hardware has a low price, the capabilities are overwhelming. Of course the complexity of using such a manifold number of modules rises enormously. For instance, it takes hours to grasp the concept and the details of interrupt handling in this chip. Luckily Texas Instruments provides a great tool named HalCoGen, which provides convenient possibilities to configure the various modules. The tool generates actual source code for initializing the configured modules. The code has marked gaps the engineer can fill then with the application's code. The resulting code, where the C-files alone, without header-files, make a total of more than 10,000 lines.

The code is structured in separate code-modules per hardware-module, a central interrupt handling module and the modules of the framework itself. The entire I/O communication uses interrupt-based transfer modes avoiding any kind of blocking the CIE by polling some hardware

state. The interrupt handling module allows for registration of callback functions helping to have a clear separation of concerns.

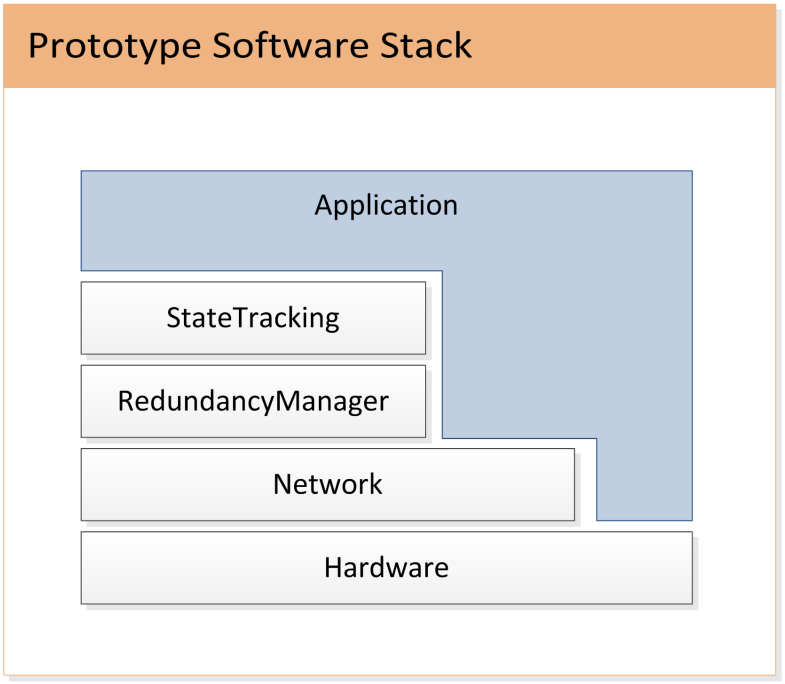

Figure 5.4: Software stack of the prototype implementation.

The framework implementation itself, as depicted in Figure 5.4, consists of three modules, each of them responsible for specific tasks. Generally, all modules provide an error-reporting interface towards the application.

Network module

This module is responsible for controlling the interaction with the network bus and the SPI and also controls the bus-guard. Additionally it holds the code to disable the bus-guard of the opposite subunit. For testing purposes and due to the missing physical bus-guard of the test-setup, it also contains code to simulate cutting off the local bus-guard.

The module listens to the complete traffic on the network bus and therefore provides the interfaces to send and receive data over the normal network bus. For the application software this means that the functions of this layer are used to send and receive data. Besides this functionality the module filters out any traffic which is not meant for the application software. Specifically this includes synchronization messages of the State Tracking module. While the subunit is in passive mode, the network module reads all data from the network bus and stores it internally until the matching output of the application software is captured or vice versa. Any mismatch or missing or corrupted data is reported to the Redundancy Manager module for further handling of the situation.

Towards the Redundancy Manager the module also provides the interface to cut off the network connection by controlling its own and the opposite subunit's bus-guard as well as the necessary functions to send and receive messages from the other subunit via SPI. Therefore the hardware pins associated with this functionality are assigned to the Network module and are configured and handled by it. The module wraps the low-level hardware interaction and reacts on any interrupt originating from those hardware modules.

For the purpose of testing the prototype, the module additionally provides dedicated filters for the data transmitted via the UART connection towards the PC. As the network bus is also simulated using the UART, the filter transparently adds commands and data targeted for the PC and removes commands targeted for the subunit from the PC from the data stream on the UART connection.

Redundancy Manager module

This module, sitting in the center of the framework, in between the other two modules, is the heart of the framework and is responsible for making most of the decisions when it comes to fault handling. Therefore, it keeps track of the role (active/passive subunit) the subunit currently has and contains the configuration, which defines the default role of the system when it is reset.

The module provides interfaces towards the Network and the State Tracking module, but does not communicate with the application except for the above mentioned error-reporting.

In case the Network module reports communication errors, either on SPI or the network bus, or the State Tracking module instructs to switch the communication path, this module is responsible for handling the error and initiates according actions like:

- Switching from SPI to the network bus in case the State Tracking module requests to do so.
- Shutting down the subunit if the current situation requires this action.
- Cutting off the opposite subunit from the network bus (by instructing the Network module to do so).
- Switching the role when necessary.

State Tracking module

This module provides the main functionality of keeping the subunits synchronized and provides also the main API towards the application software. It holds the necessary configuration options, like timeouts, which are essential to determine fault conditions.

The application software interacts with the module mainly by informing it about a new state. Whenever a new state is reported by the application, the module sends the new state to the other subunit via the interfaces provided by the Redundancy Manager module. Furthermore, it checks if the other subunit already reported reaching the given state, otherwise it waits - and therefore blocks the application process - until a new state is received. Once the new state is received, the states are compared and checked for validity and the application process is unblocked and operation can proceed.

At the level of this module, the correct reception of data is already guaranteed as any transmission error is already handled within the other modules. Still, two fault cases may occur that need to be dealt with in the State Tracking module. These are:

- No new state is received at all and the module runs into a timeout. In this case the other subunit has to be assumed crashed, as no other error has occurred, like a connection problem. The appropriate actions have to be taken, i.e. shutting down the crashed subunit, changing the own role to active, if not yet the case, and informing the application software about the problem.

- The correctly received state does not match the local, expected state. As transmission errors can again be ruled out, this means that some severe software error must have occurred, which caused the subunits to run a different code branch, leading to different states.

Debugging and testing

In order to be able to test certain scenarios it is necessary to debug the hardware. Examples of such scenarios are halting a Central Processing Unit (CPU) to provoke timeouts, altering variable contents and other actions that require direct access to the target. The JTAG interface with the USB connection has great integration with the Code Composer Studio (CCS) IDE used to write the software.

Things are more interesting though, when debugging two targets at the same time using only one PC. Albeit this feature is hopelessly under-documented in the IDE's documentation, we managed to convince CCS to start a debugging session for two targets at the same time.

This setup allows for comfortable debugging of two targets, being able to halt targets any time and in any order. The UART connection to the PC is additionally used as logging facility to transmit status messages.

5.3 Evaluation

The prototype was used to evaluate typical use cases and fault scenarios, which might occur throughout life-cycle of such systems.

This section describes those scenarios and the results when executed or simulated on the prototype. The described scenarios are valid without loss of generality.

Tests were conducted by injecting the various faults manually, either by disconnecting cable connections or simulating hardware failures through software by either using internal hardware fault injection mechanisms or by simply switching off certain hardware modules. For this purpose, a set of commands were established, which can be sent via the (debugging) UART-connection from the PC. Examples for such commands were "turn off SPI" or "send different state".

Each of the following sections will present a scenario with a description of the initial situation, the expected behaviour of the system and the test procedure with the actual reaction of the prototype implementation.

Complete power outage

Initial situation

The system is under normal operation. Abruptly the power supply for the whole CIE is not available anymore and the CIE is without power.

Expected behaviour

This case is the most simple case. The system has to comply with the “fail-silence” requirement. Due to the power outage the whole CIE should be physically cut off the network by construction, as the mono-stabil relays are not powered anymore. The expected recovery from this state is a normal bootup sequence.

Test case

As the prototype implementation does not provide physical relays, this scenario can’t be fully validated. The system runs a full bootup sequence, though, after power is restored.

Bootup one subunit

Initial situation

The initial situation is to have one subunit crashed and the other one running as active subunit. Specifically subunit 1 is assumed to be the crashed subunit, which is the default active subunit by configuration. Subunit 1 is rebooted by issuing a reset.

Expected behaviour

Assuming resetting subunit 1 solves the crash cause, it is expected that throughout the bootup of subunit 1 the already active subunit 2 will be recognized and subunit 1 will **not** try to get the active subunit as well.

Test case

This scenario is tested by only using subunit 1. While booting it, the PC simulates network bus traffic over the UART interface. The timeout for traffic detection on bootup was set to 5 seconds for the purpose of the test to ensure enough time to start the traffic simulation manually.

The debug output of the booted subunit shows that the traffic is recognized properly and that the subunit enters passive mode successfully.

Bus-guard fails

Initial situation

During normal operation a bus-guard abruptly fails.

Expected behaviour

In case a bus-guard fails, the subunit will be disconnected from the network bus automatically due to the design requirements of a bus-guard (see Section 4.2).

Assuming the bus-guard of subunit 1 fails, the situation is easily detectable by subunit 2, as expected network traffic - which is usually received on the network interface of subunit 2 - is not present, but the lock-stepped state synchronization is still working.

If the problem is shifted over to subunit 2 (being the passive unit), the situation is more difficult as the passive unit does not actively send data over the network bus. So subunit 1 has no way to detect a failing bus-guard of subunit 2. Still the lock-stepped state synchronization works normally and the impression for subunit 2 is that subunit 1 has a failed bus-guard, as it receives no data at all from subunit 1.

This situation clearly implies that the principle "always trust yourself"[4] is not applicable for this scenario. It is not reasonable to let the subunit detecting the fault be the ruling subunit, hence it may not be held responsible to switch the active subunit.

The solution to this scenario is therefore to only detect the missing bus activity **on** the passive subunit[5], but avoiding to take any overruling actions on the passive subunit. It is much cleverer to let the subunit, which detected its own bus-connection is disrupted due to missing traffic, to shut down on its own, i.e. signalling the remaining subunit explicitly that something is not OK. In case the active subunit is the affected subunit, it can actively retreat as active subunit and can switch the roles. This ensures that both subunits have correct perception of the state of the whole unit. If the passive subunit's bus-guard fails, the active subunit will notice that the passive one shut down and will in no case try to switch over to it.

This scenario does not require any specific recovery actions. If the bus-guard is getting repaired and connection to the network bus is re-established, the currently active subunit can stay active.

Test case

As outlined above, missing communication on the network bus must never lead to an action towards the opposite subunit.

Therefore, the tests conducted for this scenario were split into two approaches.

- The first approach was to hide network traffic for the passive subunit simulating a defective bus-guard for the passive subunit. As intended, the passive unit retreated and declared itself as shut down. The active subunit detected the missing SPI communication and acted as if it was a passive subunit crash, which is perfectly fine.

[4] This principle comes in very handy in many cases, where a component detects a problematic situation, but does not know the actual cause. In such a case the "always trust yourself"-principle suggests that the component should assume it is fault-free and it should act accordingly until other components prove the opposite.

[5] Detecting a missing bus activity **of** the passive unit is not possible, as there is no bus activity by definition.

- The second approach was to implement a dedicated signal of the network bus interface of each subunit to signalize a broken cable connection. Since the network bus is only simulated via UART this should mimic the behaviour of Ethernet, which would detect if a network cable is removed from its socket, or if the bus-guard cut the connection off. This assumption of being able to detect a broken cable on hardware level is of course only one way to achieve a distinction between a failing bus-guard and lack of active traffic on the bus.[6] The actual test was then to simulate the bus-guard cut off by sending the described signal to the active subunit. The active subunit expectedly sent the CMD_DOWN command via SPI after receiving the signal. The passive subunit took over the active role as intended.

Permanent hardware failure

Initial situation

Some hardware part fails permanently, without the option to repair it automatically.

Expected behaviour

Having some failing hardware part in a subunit will trigger the dedicated hardware failure pin, which indicates the problem.[7]

As by construction of the connection between the subunits and the bus-guard (refer to Figure 4.2), the affected subunit will be disconnected from the network bus by its bus-guard automatically (fail-safe state) when the error-pin is low. As the indication about the hardware error is also seen by the other subunit, the other subunit elevates to the active subunit if it wasn't already.

A permanent hardware failure involves manual intervention to replace the hardware. A recovery is therefore happening only by rebooting the system with the new hardware.

Test case

As the physical error pin on the prototype hardware has a fixed connection to an LED, the error pin has to be simulated. As denoted in Section 5.2, the additional error pin connection is used by simulation to indicate a hardware error. The test was conducted by letting the subunit, being assumed failed, send the error signal to the other subunit. This special action has again been triggered by a command from the PC.

Serial communication fails (hardware)

Initial situation

While the two subunits are in sync, one SPI channel is disrupted.

[6] In case the real network bus does not support this kind of feature, the introduction of confirmation messages or regular keep alive signals have to be employed on protocol or even application level to provoke network activity.

[7] One may implement hardware fault detection by software checks as described in Chapter 3, but this can never happen on a level, which is possible for hardware-based detection. Hence, certain hardware faults are simply not detectable by software.

Expected behaviour

Generally, we have to distinguish between a fault in the hardware, i.e. a cable break-through - remember we assume the sender/receiver hardware units are checked by the on-board hardware fault detection, so they are not considered here - or transient faults like electromagnetic radiation. But regardless of the exact fault both cases need the same treatment, so we can neglect the exact cause of the failure.

The state synchronization should detect this problem and should react appropriately by changing the active subunit if necessary. The connection problem will be detected by the slave unit of the disrupted channel as it will run into a timeout condition for the expected state change.

Whenever such a fault occurs, both subunits will notice the broken communication link. The subunits consequently try to reach (e.g. ping) the other subunit via the normal network bus to signalize being alive.

Once the physical connection is re-established the passive subunit has to re-synchronize with the active subunit. In case of a real cable break-through, this procedure is of theoretical nature obviously, because fixing a cable usually requires manual intervention and will require a reboot of the CIE anyway, so the re-synchronization happens implicitly through the reboot.

For this scenario we see a major service degradation. First, it requires to move some of the communication from the erroneous serial communication path to the main network bus. Second, the communication speed on the network bus is far slower than on the internal serial bus. Imagine the difference in bandwidth when using High Speed SPI (e.g. approximately 50 Mbit/s net) on the internal serial channel versus 10base2 (maximum 10 Mbit/s gross) on the network channel, which is not uncommon in the field of FDASs. Additionally, the load on the network bus increases even more, as the synchronization overhead is added to the normal traffic. This might also decrease the general bus performance outside the CIE, if the traffic can't be kept local to the CIE by the used bus technology.

Test case

The test was conducted by removing one cable of an SPI connection.

The application software was triggered by a signal via the UART to simulate a state change, which started the state information exchange in the StateTracking module. In the prototype implementation, a connection error is detected by the SPI hardware module. Depending on whether SIMO/SOMI, CE or CLK cable is getting interrupted, the detection differs slightly.

- In the first case, where SOMI or SIMO lines are cut, the transfer would work as expected, but the parity check would fail. Note that the concept requires odd-parity for the communication channel, so sending only zeros due to missing connection will trigger the parity error interrupt.

- The second case is when the CE line is disconnected. This would trigger a falling edge on the CE line causing the subunit being slave to initiate a new transfer. This is not a problem for the transmission of data in general, but the transmission would never be finished as

this would require the CE line to return to high-level. The subunit would therefore never receive the data.

- The third case is a missing CLK signal. This slave subunit would receive a falling edge CE signal and would wait for the transmission to start. The transmission would be finished from the master by pulling CE high. At this instant the slave unit would issue a data length error interrupt due to missing data bits.

As expected, the slave side of the broken connection ran into a timeout, signalized this condition and sent a CMD_RETRY to the other subunit via the network bus.

Application fails

Initial situation

We consider the same course as described in Section 4.3, but assume that in step 5 subunit 2 fails to finish the first part and crashes. The serial communication, including the software stack of the framework, is still fully functional as data is transferred successfully. Only the application software fails to indicate the new state change, which should be the same as the one on subunit 1. This might be caused by a very short transient error in the hardware, which causes only the application task to crash or run in an endless loop.

Expected behaviour

The consecutive process of operation has to change as depicted in Figure 5.5 and involves the following steps:

Step 6: The framework on subunit 1 detects a timeout while waiting for the response from subunit 2.

Step 7: The framework shuts down subunit 2 as described in Section 4.5.

Step 8: The framework classifies subunit 2 as failed and reports this to the application.

Step 9: The framework unblocks the waiting task.

The reaction of subunit 1 to shut down subunit 2 is correct as subunit 1 is the currently active subunit by definition of the process in Section 4.3. The system's functionality is preserved and therefore this single fault can be tolerated.

If the active subunit is the failing subunit, though, the passive subunit will be in a slightly different position. It faces the same symptoms as the active subunit would face, but **may not shut down** the other subunit immediately. It rather has to figure out whether the active subunit has really crashed by sending a CMD_RETRY command. Additionally, watching the activity on the network bus for a defined time span may also reveal liveliness of the active subunit. If there is no life-sign from the active subunit, the passive subunit has to become the active subunit and can safely shut down the crashed subunit.

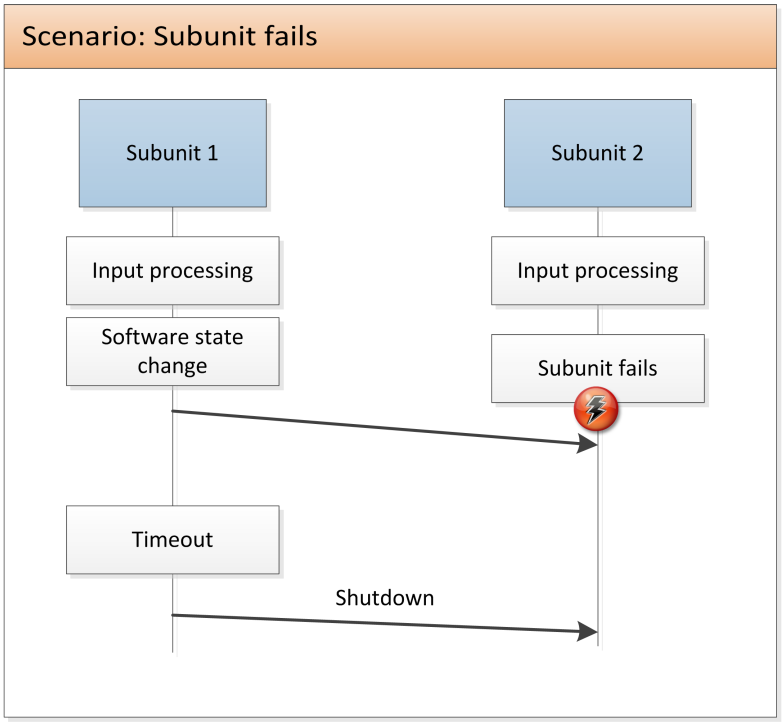

Figure 5.5: Subunit interaction with failure of one subunit

If the cause of the failure is only due to software reasons, a reset of the crashed subunit should suffice to bring it back to life. This will of course require re-synchronization with the active subunit. Unfortunately this re-synchronization of state is not limited to the framework itself as it does not store any state of the application software. Most likely re-synchronization involves heavy work by the application software as well. Since the actual application software is out of scope of this work, this form of recovery is not covered by this thesis.

Resetting the whole CIE unit will generally be the cleanest solution, which allows both subunits to gain up-to-date insight on the state of the overall system and the connected sensor buses.

Test case

Active subunit crash:
This case was simulated by omitting the response on any communication path by putting an endless loop into the demo application software just before output for the network bus would have been generated and before the new state would have been signalled to the StateTracking module.

The passive subunit successfully determined the failing active subunit due to timeout of the state synchronization and lack of traffic on the network bus and insufficient response to the

CMD_RETRY command. It successfully issued a shutdown command towards the bus-guard of the active subunit and took over the active role.

Passive subunit crash:
The simulation of this case was done like above by using an endless loop. Since it is usually not critical to shutdown passive subunit, the implementation does this, whenever state synchronization fails.

State mismatch

Initial situation

The information is received correctly via the serial communication line, but the received state does not match the expected local state. The state synchronization reveals a state or message number mismatch.

Expected behaviour

There are two possible reasons. Either a programming error caused the issue or the transmission was corrupted, which did not trigger fault detection (parity check). The latter might happen when an even number of identical bits have flipped.

In order to distinguish these two cases, the receiving subunit will reply a CMD_RETRY message. If the retry fails again, the connection is assumed to be broken and the system reacts as described in the scenario for failing serial communication by shifting communication to the network bus. Consequently, if the mismatch persists, the first reason has to be assumed, a programming error, and the whole CIE has to be shut down (safe state).

A recovery from this situation is not possible without manual intervention. A state mismatch is **severe issue** and needs careful investigation. Programmatic errors need to be fixed in design and a failing serial communication needs replacement of hardware.

Test case

This test case was simulated by employing an application software that simply runs a loop, which increments the state number. The invalid state injection was done by a UART command, which manipulated the state number, so a wrong state was sent to the other subunit in the next iteration.

The other subunit successfully detected the mismatch.

CHAPTER 6

Conclusion

Safety does not come for free, neither does it in the real world, nor does it in an embedded system. The methods for achieving higher levels of safety integrity presented in Chapter 3 come with a major drawback: significant increase in cost. Also flexibility in design is very limited by safety interests as flexibility requires freedom, but ultimate freedom and total safety (and also security) simply do not fit together and neither of those two extremes are desirable.

As the complexity of the proposed framework clearly shows, it is not trivial to create a system from resource-restricted hardware components[1] that is compatible with international functional safety standards, but it is certainly possible.

This thesis clearly shows that a well-chosen hardware and software combination in a hot-standby pattern with mutual cross-checking is capable of identifying an arbitrary single fault and is also capable of tolerating a number of single faults in the system. In order to guarantee single fault tolerance, though, it is necessary to tackle those situations, which are not solvable on the Control and Indicating Equipment (CIE) level, on the next higher abstraction level.

By using the proposed concept, a system can gain substantial benefits in terms of reliability and dependability. Increasing the reliability of the devices connected to a (already safe) network is one more step towards reaching higher Safety Integrity Levels (SILs). Detecting and dealing with faults earlier, on a lower level, before those result in a total system failure, also eases system design on the higher abstraction level as the devices of the system have less possible failure states. Using the framework ensures that those devices reach a safe state in many fault scenarios.

The critical reader may say that fault tolerance has to be present in the bigger system, the “Fire Detection and Fire Alarm System (FDAS)” in the fire safety domain, anyway, so why bother with individual units at all? We are convinced that tackling faults and failures at the lowest possible level is always beneficial for the overall system. Even more so, as a CIE unit can be

[1] Resources are interpreted in a wide definition ranging from slow hardware, over missing interfaces, all the way to costs. All of them might be restricted.

defined as a single fault containment region, freeing the upstream system from coping with the nasty details of those faults.

6.1 Lessons learned

One of the first learned lessons was the one about hardware selection for the prototype. It is by far not trivial to select the best fitting hardware today, as the market provides a lot of options, even for the specific field of safety. Since the manufacturers invest a fortune into this market segment, it is clear that the market for safety-appliances is growing faster than ever, which correlates with the perception of tighter and more comprehensive integration of today's embedded systems.

Another new insight was that microcontrollers designed for high safety levels play in a completely different league. The complexity involved to even get the basic bootup right is enormous. Albeit a really small version of the available safety controllers was used, the technical reference manual already ships with over 1000 pages and that does not include any safety manuals or similar documents.

The complexity of today's microchips is really daunting. Even the cheapest chip has capabilities, which cause severe difficulties for bigger safety-projects in regards to safety management. To give a simple example: The RM42 controller used for the prototype supports around 90 interrupt channels, which can be mapped to the interrupt routines dynamically, hence allowing to change priority of interrupts on the fly. Ensuring that there is no unsafe combination of interrupt execution that might cause system failure is a challenging task. In aid of safety it is even questionable whether certain features of those Microcontroller Units (MCUs) should be used at all.

"Projects targeted for safety critical applications need to plan a lot of budget and time for even the simplest tasks." That very saying is widely known, yet understanding the real impact, one has to work through such a project once to make this really comprehensible.

SPI can be nasty

A really tough problem, that had to be overcome, was the Serial Peripheral Interface (SPI) connection between the boards of the prototype implementation. As denoted in Section 5.2, two SPI interfaces are used to connect the subunits. On each of those interfaces one of the two subunits is the master, on the other one it is slave.

The major problem faced was that the subunit communication did only work in one direction. More precisely, only one SPI interface worked flawlessly, the other interface only issued corrupted data. After writing a dedicated SPI test software and doing many tests, it was figured out that it is crucial for a working communication that the master subunit of an SPI connection is always booted first, which is impossible for the proposed concept, where two CIE subunits are always equal. Hence, there is no defined order on the boot sequence. One test case also was to disconnect the two subunits from each other completely, let them boot independently and reconnect them afterwards. But it turned out that this did not solve the problem either.

The reason, why initialization order is so important, is that if the slave is initialized first, it might receive unintended signals on the CS or CLK lines - which might happen during the boot

up and initialization of the master unit - causing the slave to already start a transmission and shifting bits. As the master is of course not sending/receiving anything in the meanwhile this situation leads to a complete de-synchronization of master and slave. Even worse, a synchronized state can never be re-established!

Commonly SPI is used to communicate with sensors, displays, and other devices, which are really "slaves" also in the way that they are rather useless without a master controlling them. This implies that usually the master is initialized first and then the slaves are initialized. Hence, this sort of problem does not occur in those standard use cases.

In the case of the prototype, this initialization scheme is not true, as each of the subunits is master and slave of two SPI connections and each subunit has its functionality independent of the SPI "partner". This setup has a severe consequence, namely the initialization order of the devices can not be fixed as both devices are equal.

Knowing the right initialization order, it would have been easy to define a “synchronized” initialization order, which would always initialize the master of each connection first. But here we face the next problem: We are not able to assure we can always achieve this particular order, as we assume independent failure of any subunit at any time. So the only solution for fixing this problem actually was to get rid of those disturbing signals before the SPI interfaces are properly initialized.

The final solution for fixing the two-way communication via two SPI interfaces was to ensure that the output states of all involved pins (but especially the CS and CLK pins) are constant until the SPI interfaces are initialized. As the datasheet, of the MCUs revealed, all the SPI pins are preset to work as inputs with pull-up resistors activated upon power-up of the MCU. Consequently, the configuration of the SPI interface has to ensure that those pins, which are configured as outputs, will retain the high level of the pin when switched from input to output direction, otherwise a change of level would potentially disturb the SPI connection again. As matter of this prerequisite, the clock polarity for transmissions has to be changed to be inverted, such that it is low-active. Additionally, all pins configured as input should keep their pull-up resistors activated to avoid accidental level changes if the other subunit is disconnected for some reason (e.g. power loss, cable broken, and others).

Special considerations

As with every concept there are cons of the proposed concept too, but no solution comes at no cost. Therefore special consideration should be contemplated to minimize the impact of those challenges.

The biggest challenge is obviously the maximum achievable speed the application may run at. It is not primarily limited by the actual Central Processing Unit (CPU)’s clock-speed, but by the speed (more precisely the net bandwidth) of the communication lines used to synchronize software state. Therefore, it is highly recommended to use the fastest communication interface available on the hardware for this very purpose.

The prototyping hardware allows to run the SPI on a maximum clock-speed half of the system-clock. Depending on the density of synchronization points in the user application, the execution speed has therefore an upper bound of $\frac{1}{2}$ the system-clock speed reduced by the minimum delay the state synchronization via the serial communication line may cause. This tells

us that for applications with tough speed requirements it is of utmost importance to **choose** an MCU with **high clock frequency**, if the speed of the synchronization channel is off the system's clock-frequency by such a high ratio. In addition to that, the number and density of synchronization points in the application software as well is a major factor of performance. It has to be stated with utmost importance that the number of synchronization points have to be kept as low as reasonably possible.

When choosing a different transmission channel for the state synchronization, one has to keep in mind the net bandwidth of this channel. The net bandwidth is heavily depending on the overhead of the communication protocol used and on the communication mode. Certain bus systems might provide a very fast connection, but are using an asynchronous communication mode (e.g. Ethernet) and will consequently have indeterministic delays. This is especially valid for bus systems using bus arbitration modes that may cause unlimited delay. So a synchronous communication system, with deterministic transfer times, is really to be preferred, otherwise the designed protection mechanisms do not suffice.

Note on the inter-CIE usage of the network bus: As the network bus is also used in certain situations to communicate signals amongst the subunits, some additional traffic is imposed to the network. It has to be ensured that this traffic is not captured by other units or subunits on the network. In structured (switched) networks a properly configured collision domain will already help to keep this sort of traffic local to the belonging CIE. On bus networks like 10Base2 a proper network address structure has to be used to avoid wrong system behaviour.

6.2 Outlook

The current concept may not be satisfactory when it comes to handle multiple faults. In order to handle such use cases, the system may be scaled to multiple nodes. In this respect, the cross-checking provisions as currently designed, would require a fully connected network between all nodes. Such a concept might not scale very well, hence further possibilities for interconnecting those nodes have to be assessed. The downside of more nodes clearly is an increasing overhead for lock-stepped state synchronization, which further degrades performance of the overall system. Possible ways of adapting the framework towards this kind of applications are yet unexplored.

In general, the presented concept is not bound to the fire safety domain and may be reused for other appliances. Those appliances may impose less stringent requirements on the overall system, such that the concept fits "out of the box".

A possible field of usage would be the control systems of multicopter platforms (aka drones), which are getting more and more popular these days also for commercial flying. These systems face very limiting environments (size, battery power), but shall fulfil rather high levels of functional safety. In some aspects those requirements are not far off the requirements imposed on manned aircraft. Putting a fully equipped Triple Modular Redundancy (TMR) system in action on such platform might not be feasible, but the proposed concept already provides a good fault detection and partial tolerance and edge cases do always exist, in any application. Hence it

should suffice to reduce the likeliness that those edge cases cause e.g. a fatal crash to the lowest reasonable value following the ALARP principle.

This chapter describes the selected hardware used for the prototype. When choosing this hardware several aspects were left aside, which are relevant for productive systems and which deserve further assessment. For the sake of completeness, a listing of some of those criteria is provided at this place:

- The temperature ratings of the hardware parts have to be matched for the intended environment of deployment of the system.
- Safety systems tend to be used for a very long time in comparison to the ubiquitous electronic gadgets. For the product owner, it will be of great interest that guaranteed availability of the system parts for the intended lifetime of the product is warranted by the manufacturer.
- When it comes to integration questions of the hardware parts, choosing a manufacturer with good customer support might be a key criterion for choosing the hardware supplier.
- Last but not least, pricing of the hardware parts is a major decision factor.

Clearly, these criteria have more focus on the management and application life-cycle, but still this is an important aspect of project planning. Future hardware may - or rather will for sure - provide even more possibilities in the functional safety area, hence the concept presented in this thesis may need adoption. More important, the assumptions need to be re-evaluated if they still hold true.

An aspect that has not been mentioned up to this point is **security**. The presented concept does not consider security as an integral brick. One might say this is careless, but security is not considered a critical factor for the purpose of the framework. Important communication between subunits (serial communication lines) and on fieldbuses (to sensors) happens on connections that are exclusively used by the subunit's application, so the attack surface is rather small for those interfaces. Software "hacks" in general have to be tackled on the operating system level anyway. Hence, further research may suggest security measures for the proposed concept which aids the application software.

Another possible direction of research would be to integrate the software prototype into an Real-time Operating System (RTOS). Doing so implies that the state synchronization scheme will need improvements then as well, since the system may use multiple parallel running tasks.

Regarding a further improvement in reliability, the complete design could be modified to implement "triple modular redundancy". Furthermore, on the application software level, diversity aspects may be implemented such as:

- Design diversity: Use different implementations of the same task on the subunits. This of course would affect the possible granularity of the state synchronization mechanism.

- Temporal diversity: Calling the same task multiple times on both subunits (maybe synchronizing the state in between those repeated calls) to compensate transient hardware faults. Although it should be noted that such faults are unlikely to occur at the same time on both subunits and additionally not being detected by the state synchronization mechanism.

Acronyms

CAN Controller Area Network. 50

CIE Control and Indicating Equipment. 4, 6, 9, 10, 18, 33–38, 41, 43, 45, 46, 48, 50, 51, 54, 58, 61, 63–66, 68

COTS Commercial Off The Shelve. 22

CPU Central Processing Unit. 18, 39, 45, 57, 67

DMR Dual Modular Redundancy. 24, 36, 37

ECC Error-Correcting Code. 39, 48, 51

EU European Union. 7

FDAS Fire Detection and Fire Alarm System. 3–10, 12, 13, 18, 33–36, 41, 43, 47, 48, 51, 61, 65

HDL Hardware Definition Language. 20, 21

HVAC Heating, Ventilation and Air Conditioning. 4

IEC International Electrotechnical Commission. 2, 3

LIN Local Interconnect Network. 50

LON Local Operating Network. 28

MCU Microcontroller Unit. 4, 6, 21, 29, 37, 39–41, 50, 66–68

PCB Printed Circuit Board. 20

PSE Power Supply Equipment. 18, 34

PWM pulse-width modulation. 50

RTOS Real-time Operating System. 51, 69

SFF Safe Failure Fraction. 14, 15, 35

SIL Safety Integrity Level. 12–19, 21, 22, 34, 39, 47, 48, 51, 65

SPI Serial Peripheral Interface. 39, 42, 49, 50, 52, 53, 55, 56, 60, 61, 66, 67

TMR Triple Modular Redundancy. 24–26, 28, 36, 68

List of Figures

List of Tables

Bibliography

[1] Regulation (EU) No 305/2011 of the European Parliament and of the Council of 9 March 2011 laying down harmonised conditions for the marketing of construction products and repealing Council Directive 89/106/EEC Text with EEA relevance. http://eur-lex.europa.eu/legal-content/EN/TXT/?qid=1423479686092&uri=CELEX:32011R0305, 03 2001.

[2] *Validierung des Zeitverhaltens von kritischer Echtzeit-Software*, volume INFORMATIK 2003. GI, Gesellschaft für Informatik, Bonn, 2003.

[3] Control of fire by early humans. http://en.wikipedia.org/wiki/Control_of_fire_by_early_humans, April 2014.

[4] Definition of Safety. http://en.wikipedia.org/wiki/Safety, July 2014.

[5] Hagit Attiya and Jennifer Welch. *Fault-Tolerant Clock Synchronization*, pages 277–293. John Wiley & Sons, Inc., 2004.

[6] Hagit Attiya and Jennifer Welch. *Leader Election in Rings*, pages 31–58. John Wiley & Sons, Inc., 2004.

[7] Austrian Federal Association of Fire Brigades. Anschaltebedingungen von Brandmeldeanlagen an öffentliche Feuerwehren, 2006.

[8] Austrian Federal Association of Fire Brigades. Fire Detection Systems, 2011.

[9] Austrian Standards Institute. EN 54 - Fire detection and fire alarm systems, Part 4: Power supply equipment , October 1997/2006.

[10] Austrian Standards Institute. EN 54 - Fire detection and fire alarm systems, Part 1: Introduction, May 2011.

[11] Austrian Standards Institute. EN 54 - Fire detection and fire alarm systems, Part 2: Control and indicating equipment, November 2011.

[12] Austrian Standards Institute. Industrial communications subsystem based on ISO 11898 (CAN) for controller-device interfaces - Part 5: Functional safety communication based on EN 50325-4, May 2013.

[13] Bernadette Charron-Bost, Shlomi Dolev, Jo Ebergen, and Ulrich Schmid. *Fault-Tolerant Distributed Algorithms on VLSI Chips*. Dagstuhl Seminar Proceedings. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2009.

[14] DIN German Institute for Standardization. Building automation and control systems (BACS) - Part 2: Hardware (ISO 16484-2:2004); German version EN ISO 16484-2:2004, 2004.

[15] DIN German Institute for Standardization. DIN VDE 0833-2 Alarm systems for fire, intrusion and hold up - Part 2: Requirements for fire alarm systems, June 2009.

[16] DIN German Institute for Standardization. DIN 14675 Fire detection and fire alarm systems - Design and operation, April 2012.

[17] Danny Dolev. The byzantine generals strike again. *Journal of Algorithms*, 3(1):14 – 30, 1982.

[18] Holger Hölscher and Johann Rader. *Microcomputers in safety technique: an aid to orientation for developer and manufacturer; results of a research report supported by the German Federal Minister for Research and Technology*. TÜV Rheinland, 1986.

[19] IEC. Functional Safety Explained. `http://www.iec.ch/functionalsafety/explained/`, April 2014.

[20] International Electrotechnical Commission. IEC 61508 - Functional safety of electrical/-electronic/programmable electronic safety-related systems, 2010.

[21] Johannes Kasberger. *Reliable IP based communication for fire alarm systems*. 2013. Parallelt. [Übers. des Autors] Reliable IP based communication for Fire Alarm Systems; Wien, Techn. Univ., Dipl.-Arb., 2014.

[22] H. Kopetz. On the fault hypothesis for a safety-critical real-time system. In Manfred Broy, Ingolf H. Krüger, and Michael Meisinger, editors, *Automotive Software & Connected Services in Mobile Networks*, volume 4147 of *Lecture Notes in Computer Science*, pages 31–42. Springer Berlin Heidelberg, 2006.

[23] Hermann Kopetz. *Real-Time Systems - Design Principles for Distributed Embedded Applications*. Real-Time Systems Series. Springer, 2011.

[24] Hermann Kopetz and Wilhelm Ochsenreiter. Clock synchronization in distributed real-time systems. *Computers, IEEE Transactions on*, C-36(8):933–940, Aug 1987.

[25] Parag K Lala. *Fault tolerant and fault testable hardware design*. Prentice-Hall, 1985.

[26] J.C. Laprie. Dependability: Basic concepts and terminology. In J.C. Laprie, editor, *Dependability: Basic Concepts and Terminology*, volume 5 of *Dependable Computing and Fault-Tolerant Systems*, pages 3–245. Springer Vienna, 1992.

[27] D. J. Lu. Watchdog processors and structural integrity checking. *IEEE Trans. Comput.*, 31(7):681–685, July 1982.

[28] T. Michel, R. Leveugle, and G. Saucier. A new approach to control flow checking without program modification. In *Fault-Tolerant Computing, 1991. FTCS-21. Digest of Papers., Twenty-First International Symposium*, pages 334–341, June 1991.

[29] T. Novak and T. Tamandl. Architecture of a safe node for a fieldbus system. In *Industrial Informatics, 2007 5th IEEE International Conference on*, volume 1, pages 101–106, June 2007.

[30] Nahmsuk Oh. Software implemented hardware fault tolerance, 2000.

[31] Martin Perner. Self-stabilizing byzantine fault-tolerant clock distribution in grids. Master's thesis, Institut für technische Informatik, 2013.

[32] Manfred Schwarz. Solving k-set agreement in dynamic networks. Master's thesis, Institut für technische Informatik, 2013.

[33] D. Smith and K. Simpson. *Safety Critical Systems Handbook - A Straightforward Guide to Functional Safety, IEC 61508 (2010 Edition) and Related Standards*. 2010.

[34] Dilip Soni, R.L. Nord, and Christine Hofmeister. Software architecture in industrial applications. In *Software Engineering, 1995. ICSE 1995. 17th International Conference on*, pages 196–196, April 1995.

[35] T. Tamandl and P. Preininger. Online self tests for microcontrollers in safety related systems. In *Industrial Informatics, 2007 5th IEEE International Conference on*, volume 1, pages 137–142, June 2007.

[36] VDMA. VDMA 24200-1 - Gebäudeautomation Automatisierte Brandschutz- und Entrauchungssysteme - ABE, 03 2004.

[37] VdS Schadenverhütung GmbH. VdS 2095 Guidelines for automatic fire detection and fire alarm systems - Planning and Installation, May 2010.

[38] Harald Pe Vranken, Marc F Witteman, and Ronald C van WUIJTSWINKEL. Design for testability in hardware-software systems. *IEEE Design & Test of Computers*, 13(3):79–87, 1996.

[39] Weihang Wu and T. Kelly. Safety tactics for software architecture design. In *Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International*, pages 368 – 375 vol.1, sept. 2004.

[40] J.F. Ziegler and W.A. Lanford. The effect of sea level cosmic rays on electronic devices. *Journal of Applied Physics*, 52(6):4305–4312, Jun 1981.