Master's Thesis

ECU with Safety Features for a Formula Student Electric Race Car based on FreeRTOS

Friedrich Lobenstock

Institute for Technical Informatics
Graz University of Technology
Inffeldgasse 16, 8010 Graz, Austria
Head: O. Univ.-Prof. Dipl.-Inform. Dr.sc.ETH Kay Römer

Supervisor/Assessor:
Ao.Univ.-Prof. Dipl.-Ing. Dr.techn. Eugen Brenner

Graz, October 2013

Abstract

The Formula Student Electric (FSE) is an all-electric race car series, which was started in 2010 as part of the Formula Student and Formula SAE® series of worldwide competitions. Every year, the teams have to develop new prototype race cars. In prototype development, general-purpose Electronic Control Units (ECUs) are commonly used. For cars with internal combustion engines, numerous commercial off-the-shelf, general-purpose ECUs are already available. Not so for FSE cars, because of the specific requirements defined in the rules of this series.

In this work, an ECU for use in FSE race cars was developed according to the Safety Element out of Context (SEooC) concept of the functional safety standard ISO 26262. The functional safety concept and the technical safety concept were derived from an item definition, which is based on the Maxwheel race car of the TU Graz Racing Team. The outcome is a Safety ECU platform for FSE race cars. Furthermore, as a basis for future software development, FreeRTOS, a real-time operating system for embedded systems, was ported to a CPU, based on the C166 V2 core.

Keywords: ISO 26262, Functional Safety, Item Definition, Functional Safety Concept, Technical Safety Concept, Safety Element out of Context, SEooC, Formula Student Electric, Electronic Control Unit, ECU, FreeRTOS, C166

Kurzfassung

Die Formula Student Electric (FSE) ist eine Rennserie für vollelektrische Rennwägen, die im Jahr 2010, als Teil der Formula Student- und Formula SAE®-Serie weltweit ausgetragener Wettkämpfe, gestartet wurde. Jedes Jahr müssen die Teams neue Rennwagenprototypen entwickeln. In der Entwicklung von Prototypen werden häufig Universal-Steuergeräte eingesetzt. Für Fahrzeuge mit Verbrennungsmotoren ist bereits eine Vielzahl von Universal-Steuergeräten kommerziell erhältlich. Nicht so jedoch für FSE-Rennwägen, aufgrund der speziellen Anforderungen, die das Reglement dieser Serie stellt.

In der vorliegenden Arbeit wurde ein Steuergerät für den Einsatz in FSE-Rennwägen nach dem Konzept des Safety Element out of Context (SEooC) der funktionalen Sicherheitsnorm ISO 26262 entwickelt. Sowohl das funktionale als auch das technische Sicherheitkonzept wurden von einer Item Definition, welche auf dem Maxwheel-Rennwagen des TU Graz Racing Team basiert, abgeleitet. Das Ergebnis ist eine Sicherheits-Steuergeräteplattform für FSE-Rennwägen. Des Weiteren wurde als Grundlage für zukünftige Software-Entwicklung FreeRTOS, ein Echtzeit-Betriebssystem für eingebette Systeme, auf eine CPU mit C166 V2 Kern portiert.

Schlagwörter: ISO 26262, Funktionale Sicherheit, Item Definition, Funktionales Sicherheitskonzept, Technisches Sicherheitskonzept, Safety Element out of Context, SEooC, Formula Student Electric, Steuergerät, ECU, FreeRTOS, C166

Acknowledgement

Diese Diplomarbeit wurde im Studienjahr 2012/13 am Institut für Technische Informatik an der Technischen Universität Graz durchgeführt.

Besonderer Dank gilt den Mitgliedern des TU Graz E-Power Racing Teams, und in Folge den Mitgliedern des vereinten TU Graz Racing Teams, für die Möglichkeit diese Arbeit überhaupt durchführen zu können und für die unvergesslichen Erlebnisse der gemeinsamen Saisonen 2009 bis 2012.

Weiterer Dank gilt Herrn Professor Eugen Brenner für die Chance diese Arbeit auch wissenschaftlich beleuchten zu können, ebenso wie Herrn Professor Christian Kreiner für seinen Kurs zur Funktionalen Sicherheit. Dank gilt auch meinem Kollegen Georg Macher fürs Korrekturlesen und meinem Kollegen Lukas Raschendorfer für seine Grafiktipps. Frau Maj-Britt Macher danke ich für ihre professionellen Englischkorrekturen.

Graz, im Oktober 2013 Friedrich Lobenstock

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Background

Since 2004 the TU Graz Racing Team[1] has been taking part in the Formula Student and the Formula SAE® competitions. The Formula SAE® competition series was started in 1979 by the Society of Automotive Engineers (SAE) in the USA. In 1998, the Institution of Mechanical Engineers (IMechE) in the UK started the Formula Student competition series in Europe. The first Formula Student Germany took place in 2005. Up to 2010, the Formula Student and the Formula SAE series were all about internal combustion cars, but that year Formula Student Germany introduced the first Formula Student Electric (FSE) competition. Since then, the TU Graz Racing Team has been competing with its all-electric race car named Maxwheel. Figure 1.1 shows Maxwheel 2012 in motion.

Photo: Formula Student Austria/Manuel Schwarz

Figure 1.1: FSE race car Maxwheel 2012 in motion.

[1] http://racing.tugraz.at/

1.2 Motivation and Aim

In the racing seasons 2010 to 2012, the TU Graz Racing Team was using general-purpose ECUs as vehicle control units, such as the ones shown in Figure 1.2.

Photo: dSPACE

(a) dSPACE MicroAutoBox.

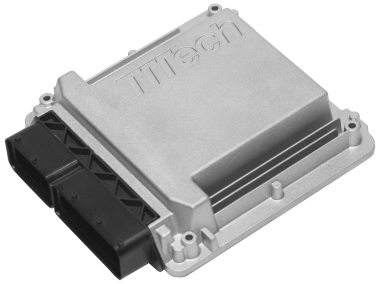


Photo: TTTech

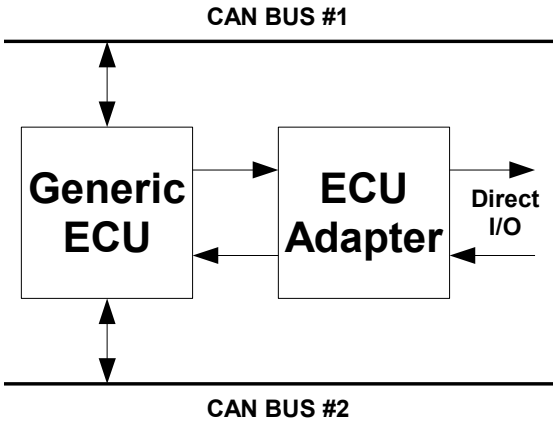
(b) TTTech HY-TTC 200.

Figure 1.2: General-purpose ECUs used by the TU Graz Racing Team.

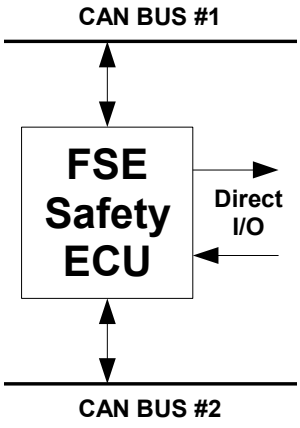
General-purpose ECUs are a great thing to start development projects with, as they usually come with support for, e.g., MATLAB Simulink® for easy control strategy design.

For use in Formula Student Electric (FSE) race cars, no readily available general-purpose ECU was found that supported the specific I/O interfaces needed to fulfill the rules of that series, see Appendix D on page 108.

The interlocks and the safety system required by the FSE rules at first resulted in an additional adapter box, referred to as "ECU Adapter" (see Figure 1.3a). To avoid the extra adapter box and to reduce system complexity we decided to develop a FSE Safety ECU, as shown in Figure 1.3b, which included the specific interfaces and could be thought of as the starting point for the development of a standard ECU for the series.



(a) General-purpose ECU.



(b) FSE Safety ECU.

Figure 1.3: Comparison of a general-purpose ECU and an assumed FSE Safety ECU.

Because the FSE organizers wrote a lot of safety-related rules into the book of rules, and the functional safety standard ISO 26262 had just been published, we decided to do the ECU development according to that standard. The SEooC concept presented in the standard was considered a suitable approach for the development of the planned FSE Safety ECU.

1.3 Organization of the Thesis

The rest of this thesis is organized as follows:

Chapter 2 contains an overview of automotive embedded systems, a brief history and overview of functional safety and the automotive functional safety standard ISO 26262, some background on the architecture of automotive hard- and software, an overview of automotive bus systems, basic information on real-time operating systems in general, and details about the FreeRTOS real-time operating system.

Chapter 3 goes into the details of the ECU design in accordance with the ISO 26262, i.e., applicability of the ISO 26262, description of the SEooC development process, the SEooC concept phase, the SEooC assumptions at item level, and the SEooC product development at the system and hardware level. Then it presents the final hardware, the Formula Student Electric Safety ECU Platform. A port of FreeRTOS, to the chosen hardware platform, rounds off the implementation part of this thesis.

Chapter 4 concludes the thesis and gives suggestions for future work.

Appendix A contains the used acronyms and abbreviations.

Appendix B contains additional ISO 26262 material, like the detailed overview figure, essential vocabulary and tables from ISO 26262 part 3.

Appendix C contains the tables that are too large to be presented in the SEooC sections of Chapter 3.

Appendix D lists the project-relevant rules of the Formula SAE (FSAE) Rules 2012 and Formula Student Electric (FSE) Rules 2012.

Appendix E contains relevant sections from the *C166S V2 User Manual*.

Appendix F contains relevant sections from the *TASKING VX-toolset for C166 v3.1 User Guide*.

Appendix G contains the source files of the FreeRTOS port to the Infineon C166S v2 Core.

Chapter 2

State of the Art

2.1 Automotive Embedded Systems

Nowadays embedded automotive systems build complex, networked, and distributed systems literally containing more than one hundred of so called ECUs [1, 2]. Basically an embedded system can be seen as "a computerized system dedicated to perform a specific set of real-world functions, rather than to provide a generalized computing environment." [3].

These systems are found at the heart of engine combustion, transmission, anti-lock braking, and many other control functionalities in today's vehicles. Figure 2.1 shows the relation of driver, vehicle, and environment in a block diagram as it is known from control systems design. The block named "Controller" can be implemented - or "mapped" -

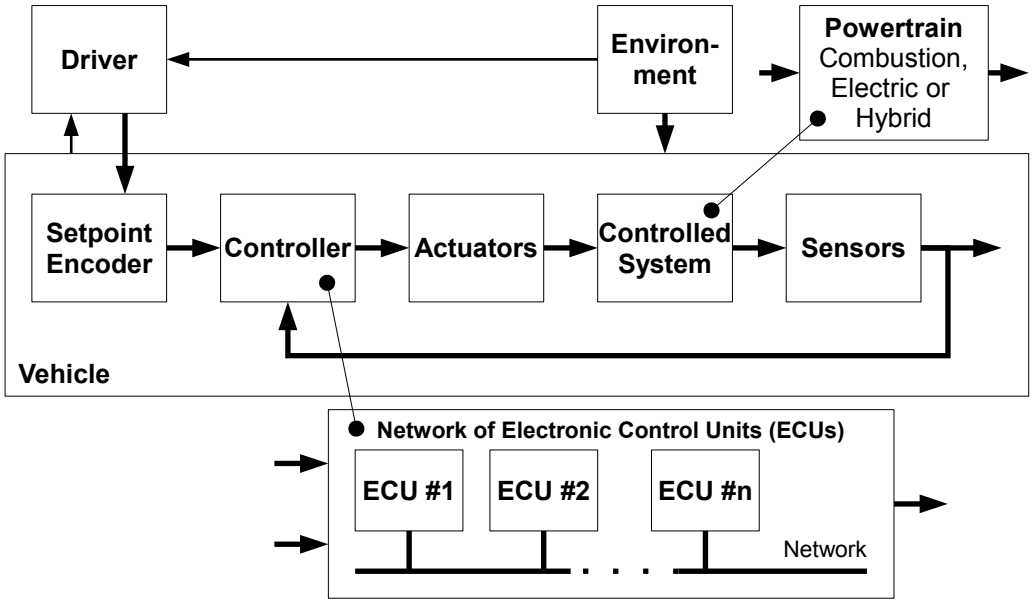


Figure 2.1: The driver-vehicle-environment relations with controller and controlled system mapping (adapted from [4]).

as one single unit, or its functionality can be spread over different controllers forming a distributed system, as shown in the figure. Systems without a direct user interface, but with rather indirect user input, e.g., for setpoint values, are called embedded systems [4]. When we talk about systems in this thesis, it is about embedded systems.

A distributed system needs a way of exchanging values and/or messages, which is accomplished by using a so called network or bus. A short overview of automotive bus systems is given in Section 2.6.

The aforementioned control functionalities generally restrict a system regarding its time budget for its responses, so that the system implementing them manifests itself as a real-time system. In Section 2.7 we are going to look at real-time systems and a specific implementation of a real-time operating system, i.e., FreeRTOS.

Another aspect of the control functionalities in automobiles is safety because they hold the risk of destabilizing the vehicle while driving at high speed on the motorway. Safety can usually only be defined as the absence of unacceptable risks. The following Section 2.2 will look into functional safety in general. An overview of the related industry standard for the automotive domain, the ISO 26262, will be given in Section 2.3.

The architecture of automotive hard- and software is then covered in Section 2.4. A brief overview of the E-Gas Architecture and Safety Concept will be given in Section 2.5.

2.2 Functional Safety

2.2.1 History of Functional Safety and the IEC 61508

In the 1960s safety became a concern in the process industry, but the definition of safety was not as broad as it is today. The VDI/VDE Guideline 2180 from 1966, for example, only considered safety in the context of a production site and had the produce as the only safety objective.

The major accidents in Flixborough (UK) in 1974 and Seveso (Italy) in 1976 then gave rise to the publication of novel loss-prevention actions like the Council Directive 82/501/EEC, the so-called Seveso Directive,[2] which was adopted in 1982. Germany enacted the so-called Störfallverordnung (StFV, English: Hazardous Incident Ordinance) in 1980. In the UK this led to the CIMAH (Control of Industrial Major Accident Hazards) regulations issued by the Health and Safety Executive (HSE) in 1984. Further events to come were the Bhopal (India) incident in 1984 and the Piper Alpha (UK) incident in 1989.

In 1984, the second edition of the VDI/VDE Guideline 2180 was published. It remarkably incorporated the concept of injury to persons, and a mandatory fault analysis was introduced. At that time also the concept of risk got more emphasis. The DIN V 19250, "Control technology; fundamental safety aspects to be considered for measurement and control equipment" emerged in 1989. The risk graph was used to estimate risks, and eight risk classes were defined, each being assigned a different level of risk posed by the process, and dedicated technical and organizational measures. The DIN V 19251, published in 1995, described those measures in depth.

Across the Atlantic in the US, catastrophic accidents in the chemical industry led the Occupational Safety and Health Administration (OSHA) to release the directive for

[2] http://ec.europa.eu/environment/seveso/

Process Safety Management (PSM) of Highly Hazardous Chemicals in 1992. Subsequently, the Instrument Society of America, now International Society of Automation, published the standard ISA S84.01, "Application of Safety Instrumented Systems for the Process Industries" in 1996.

Also in 1996, the Council Directive 96/82/EC on the control of major-accident hazards – the so-called Seveso II Directive – was ratified and replaced the original Seveso Directive. It added, e.g., new requirements related to safety management systems and emergency planning.

In the 1980s microprocessor-based systems (generally referred to as Programmable Logic Controllers (PLCs))) entered the safety control market, but risk assessment techniques at that time did not specifically include those. In 1985 the International Electrotechnical Commission (IEC)[3] set up a working group to develop a systems-based approach. This culminated in the publishing of the IEC 61508 in 1998, a safety standard not only including PLCs, but all types of electrotechnical technologies (electrical, electronic and programmable electronic systems).

Figure 2.2 shows the historic timeline leading up to the IEC 61508. For more historic details see references [5–10].

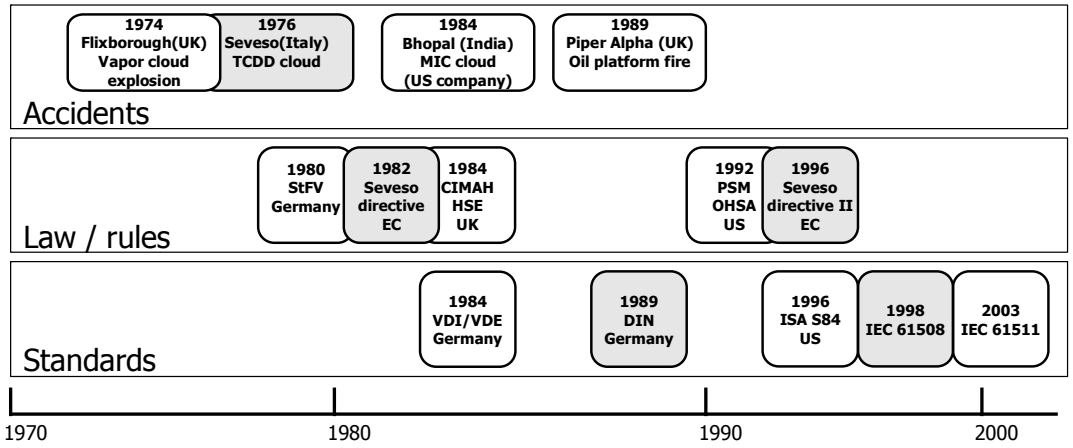


Figure 2.2: History of IEC 61508 (adapted from [10]).

2.2.2 Industry-/Application-specific Variants of the IEC 61508

In Europe safety standards can be classified according to the following three types - see also Figure 2.3:

- **Type-A** standards define the basic safety standards.
- **Type-B** standards define group- or area-specific safety standards.
- **Type-C** standards define additional, mandatory safety requirements for certain products.

[3] http://www.iec.ch/

Type-B and type-C standards are derived from type-A standards. The IEC 61508, or more specifically the EN IEC 61508, as the harmonized European standards are prefixed with "EN", is therefore a type-A standard.

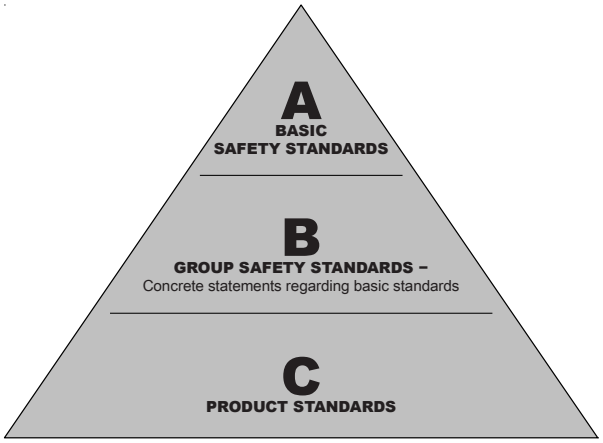


Figure 2.3: Hierarchy of harmonized European standards [11].

Figure 2.4 shows examples of type-B functional safety standards which are derived from the EN IEC 61508 for a specific industry or application.

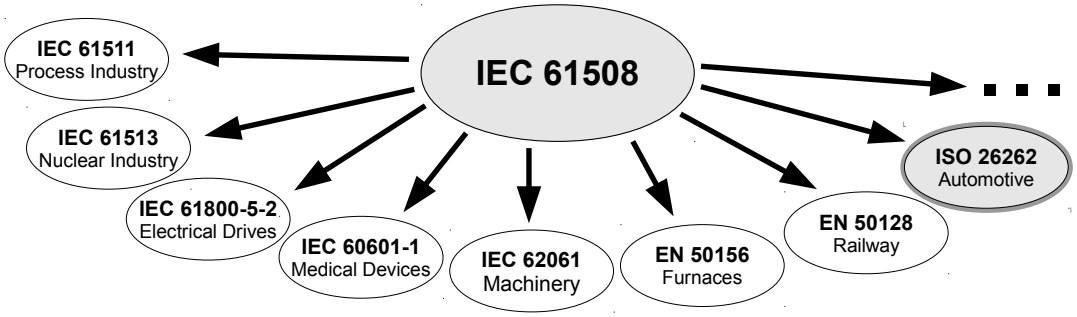


Figure 2.4: Functional safety standards (based on IEC 61508).

For road vehicles the relevant type-B standard, as shown, is the EN ISO 26262. More details of this standard will be given in the following section.

2.3 ISO 26262 Functional Safety for Road Vehicles

The ISO 26262 is an ISO standard which applies to safety-related systems of passenger road vehicles. It is the adaptation of the IEC 61508 to the automotive application sector and applies to automotive safety-related systems that include electrical and/or electronic (E/E) systems. Table 2.1 shows the relation of the two standards IEC 61508 and ISO 26262. In contrast to the ISO 26262, the human-factor concept of controllability is not considered in the IEC 61508. The Safety Integrity Level (SIL) 4 of the IEC 61508 has no corresponding Automotive Safety Integrity Level (ASIL) in the ISO 26262.

Table 2.1: Comparison of IEC 61508 and ISO 26262 (adapted from [12]).

Standard	**IEC 61508**	**ISO 26262**
Background	Chemical plants	Automotive
Relevance	Generic, E/E/PE systems	E/E systems in passenger vehicles <3.5 tons
Philosophy	Implement safety functions in separate subsystems	Implement safety functions in the same device that provides the safety-related functionality
Type of Production	Individual equipment	Mass/series production
Human Factor	Not considered	Concept of controllability
Safety Integrity Levels	SIL 1-3; Various ways to determine SIL	ASIL A-D; Hazard analysis and risk assessment
Focus	Development, start of operation, operation and maintenance, decommissioning	Concept phase, product development, production
Released	1st edition 1998, 2nd edition 2010	1st edition 2011, 2nd edn. expected 2016–2018

2.3.1 Functional Safety according to ISO 26262

Functional safety in the meaning of ISO 26262 means that hazards caused by potential malfunctioning behavior of electrical and/or electronic (E/E) systems do not cause unacceptable risks, or “unreasonable risks”, as the standard puts it. In other words, functional safety is simply the absence of unacceptable risks with respect to the desired behavior of E/E systems.

2.3.2 Overview of the ISO 26262

The ISO 26262 is divided into ten parts [13–22], of which parts 3 to 7 deal with the product lifecycle, whereas the other parts are lifecycle-independent. Figure 2.5 gives an overview of the parts of the ISO 26262. A more detailed overview is shown in Figure B.1 in Appendix B, page 83. The V’s in the figure denote that each development process, i.e., the

overall development process, the hardware, and the software development process, follows the V-model (or Vee Model) development process.[4]

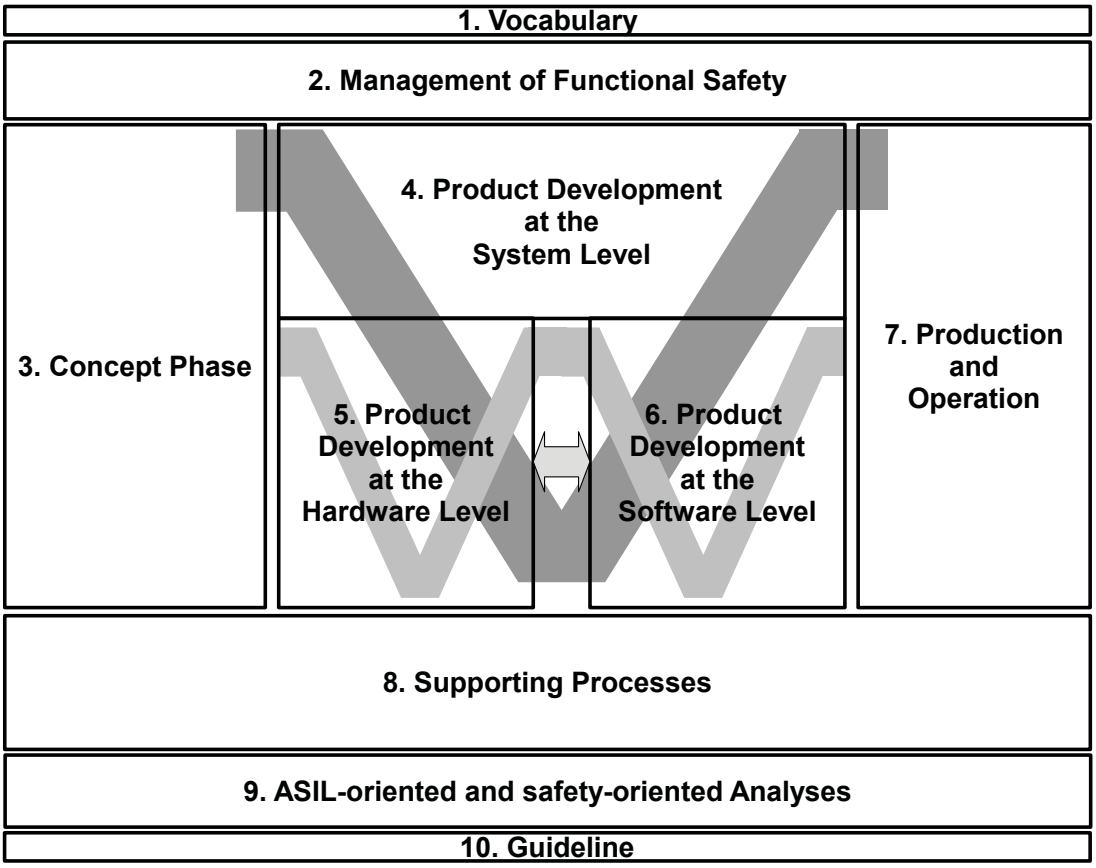


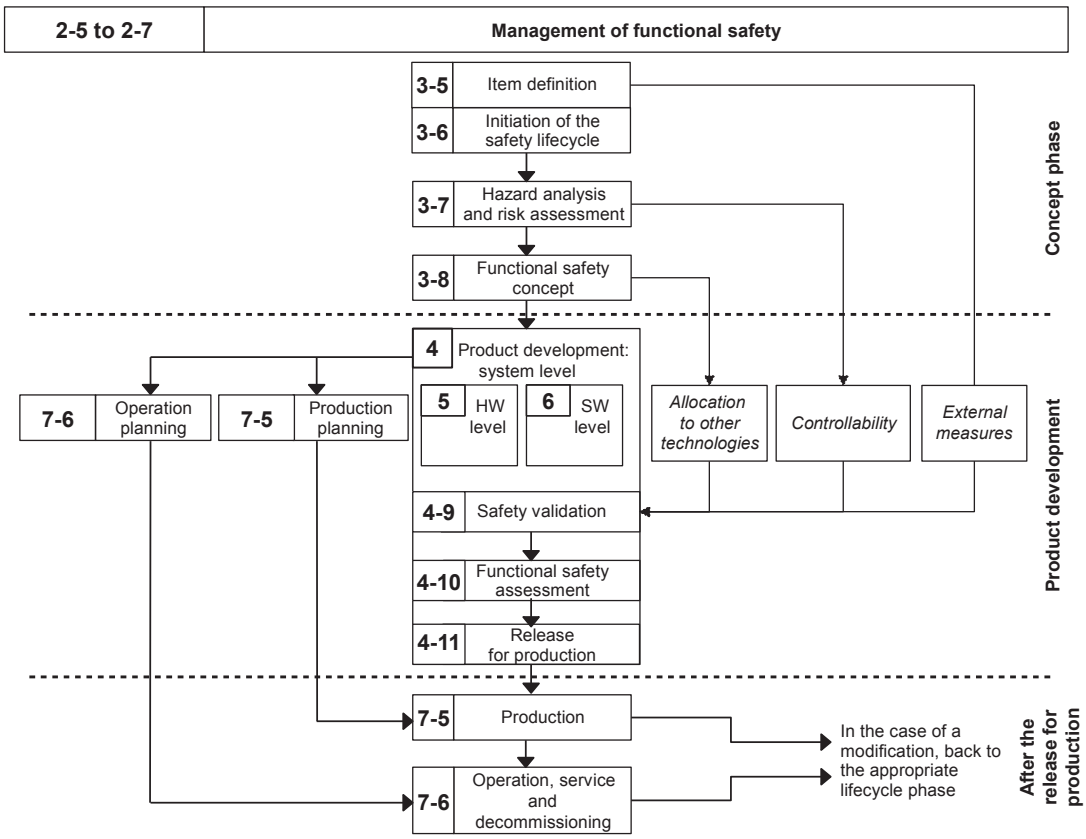
Figure 2.5: Overview of the ISO 26262 (adapted from [13, Figure 1]).

Part 1: Vocabulary defines the terms and definitions used in the context of ISO 26262. Some noteworthy examples are listed in Appendix B.2 on page 84.

Part 2: Management of functional safety describes the management of the functional safety processes. It defines the organization of the project and quality management over the whole product lifecycle. This includes a safety plan, project plan, safety case, functional safety assessment plan, and a confirmation measure report. It also specifies that the functional safety has to be monitored even after the product has been released for production. This safety lifecycle, as defined by the standard, is shown in Figure 2.6.

Part 3: Concept phase starts with the definition of the item - see definition in Appendix B.2 on page 84 - and initiates the safety lifecycle with an impact analysis and the refinement of the safety plan from the previous part.

[4] http://en.wikipedia.org/wiki/V-Model_(software_development)



Note: "m-n" denotes ISO 26262 part "m" clause "n"

Figure 2.6: Safety lifecycle according to ISO 26262 [14, Figure 2]).

The hazard analysis and risk assessment that follows analyzes (driving) situations, classifies the risks of those situations, and derives safety goals as shown in Figure 2.7. Each safety goal is assigned an ASIL according to a severity class, a probability class – also called exposure –, and a controllability class, as shown in Table 2.2. ASIL A is the lowest safety integrity level and ASIL D the highest one. The level QM denotes that only quality management is required and no further actions regarding safety need to be taken. Subsequently, a verification review by a third party has to show that the hazard analysis, the risk assessment, and the safety goals, are complete, compliant, and consistent.

The functional safety concept, as shown in Figure 2.7, derives functional safety requirements from the safety goals, which are further specified by, e.g., fault-tolerant time intervals and a safe state. These requirements are then allocated to elements of the preliminary architecture. Finally a verification step shows if the functional safety concept is consistent and compliant with the safety goals, and that it is able to mitigate or avoid hazardous events.

Part 4 to part 6: Product development at the system, hardware and software level cover the core processes of product development according to ISO 26262. The

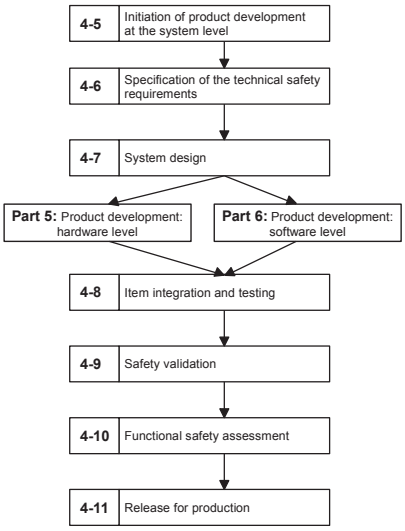
Note: "m-n" denotes ISO 26262 part "m" clause "n"

Figure 2.7: Concept phase (detail from [15, Figure 3]).

Table 2.2: ASIL determination [15, Table 4].

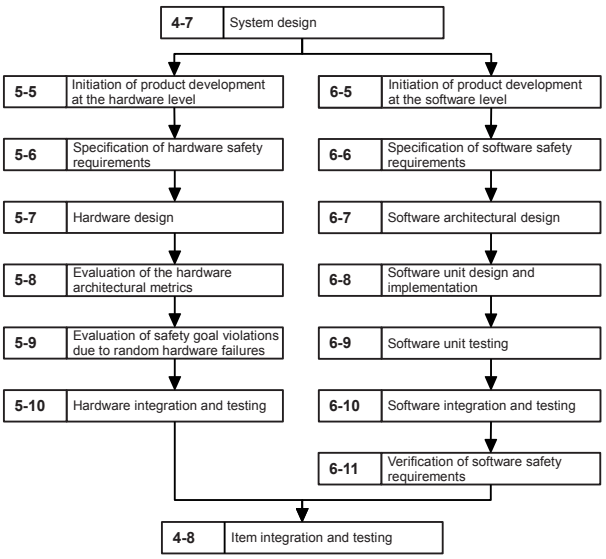
Severity class	Probability class	Controllability class		
		C1	C2	C3
S1	E1	QM	QM	QM
	E2	QM	QM	QM
	E3	QM	QM	A
	E4	QM	A	B
S2	E1	QM	QM	QM
	E2	QM	QM	A
	E3	QM	A	B
	E4	A	B	C
S3	E1	QM	QM	A
	E2	QM	A	B
	E3	A	B	C
	E4	B	C	D

development process at the system level is shown in Figure 2.8, that at the hardware and/or software level in Figure 2.9. The whole development process is structured according to



Note: "m-n" denotes ISO 26262 part "m" clause "n"

Figure 2.8: Overview of product development at the system level (adapted from [16, Figure 2]).



Note: "m-n" denotes ISO 26262 part "m" clause "n"

Figure 2.9: Overview of product development at the hardware and software level (adapted from [16, Figure B.1]).

the V-model,[5] as depicted before in the overview in Figure 2.5. Each development phase is divided into three subphases: planning of the activities, implementation, and then verification/validation [23].

At the system level, the functional safety concept created during the concept phase will be refined into the technical safety concept. The requirements for hardware and software will then be derived from the design, which is again derived from the technical safety concept. The link between hardware and software is specified in the so called Hardware-Software Interface (HSI) during system design.

Then the development process will split into development at hardware and at software level. When these steps are done, integration and testing will be followed by a validation and assessment phase, and finally the product will be released for production.

Part 7: Production and operation contains requirements for production planning, production, operation, service, and decommissioning of the product. For all phases, safety-related implications have to be considered. During production it is essential that "the correct embedded software and the associated calibration data are loaded into the ECUs as part of the production process" [19]. A field monitoring process for functional safety incidents shall also be implemented.

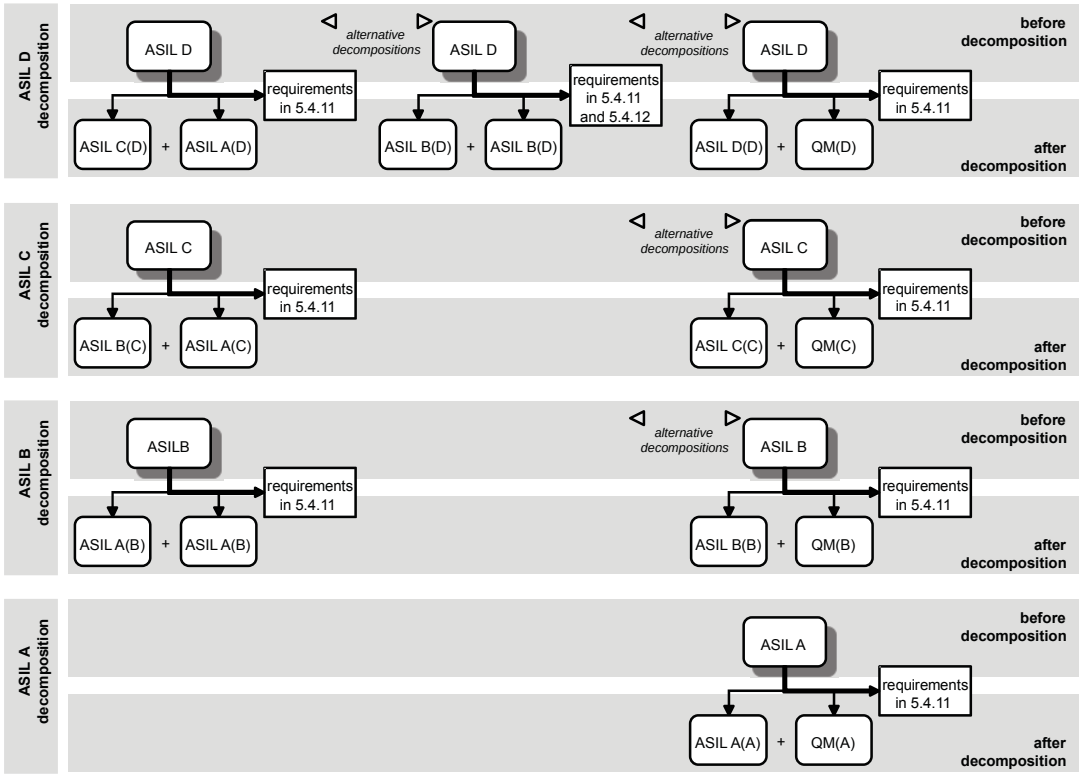
Part 8: Supporting processes describes supporting processes, such as a development interface agreement (DIA) in case subsystems are developed by third parties, configuration management, change management (i.e., traceability), verification, and documentation. The software tools used have to undergo an evaluation to provide evidence of suitability for use in safety-related development. Qualification reports for ready-made software and hardware used in the project need to be prepared as well. There is also the possibility to establish so called *Proven in Use* arguments for re-use of previously developed safety-related (sub-)systems.

Part 9: ASIL-oriented and safety-oriented analyses mainly deals with ASIL decomposition. All possible recursive decomposition schemes are shown in Figure 2.10. ASIL A(A) is the only one which can not be decomposed any further. During decomposition one has to provide sufficient evidence that the decomposed elements are independent of each other and that confirmation measures according to part 2 have been applied. Criteria for the coexistence of elements with different or no assigned ASIL are provided and requirements for safety analysis are specified. [24] gives some practical examples on ASIL decomposition.

Part 10: Guideline is the only part which is informative, in contrast to all other parts, which are all normative. It contains notes on topics regarding safety management, concept phase and system development, the safety process requirements structure, hardware development, proven-in-use argument(s), and ASIL decomposition.

It also introduces the concept of developing a "Safety Element out of Context (SEooC)" which is not developed in the context of a specific automobile. According to [25], "Concepts

[5] http://en.wikipedia.org/wiki/V-Model_(software_development)



Note: "ASIL X(Y)" denotes a component developed according to ASIL X but hardware architectural metrics and hardware failure target values are kept at ASIL Y, as the safety goal always keeps its ASIL level

Figure 2.10: ASIL decomposition schemes [21, Figure 2].

such as 'Safety Element out of Context' within the ISO DIS 26262 standard help to address the problem of component construction in a safety-related development process [...]".

2.4 Automotive Hard- and Software Architectures

In this sections we will look into automotive hardware and software architectures in general. The details of the so called E-Gas architecture, also known as the 3-layer safety monitoring concept, will be covered in the following section.

The architectures suggested in the literature [26–32] essentially boil down to the following three types:

- Asymmetric Processor Architecture
- Dual Processor Architecture
- Lock-step Processor Architecture

These architectures will be outlined below.

The Asymmetric Processor Architecture uses two different CPUs, one main and one asymmetric processor, whereof the latter can also be an Application Specific Integrated Circuit (ASIC), as shown in Figure 2.11. Both run different software and communicate via Serial Peripheral Interface (SPI) or some other (high-speed) serial link and cross-check each other. The asymmetric processor has much lower requirements on processing power and memory because it only executes self-checks and checks the integrity of the main processor. A common integrity check is the verification of the main processor's control flow [26], also called program flow monitoring [33]. If the inter-processor communication follows a "question and answer" schema, this architecture is also known as the "challenge-response" architecture [30].

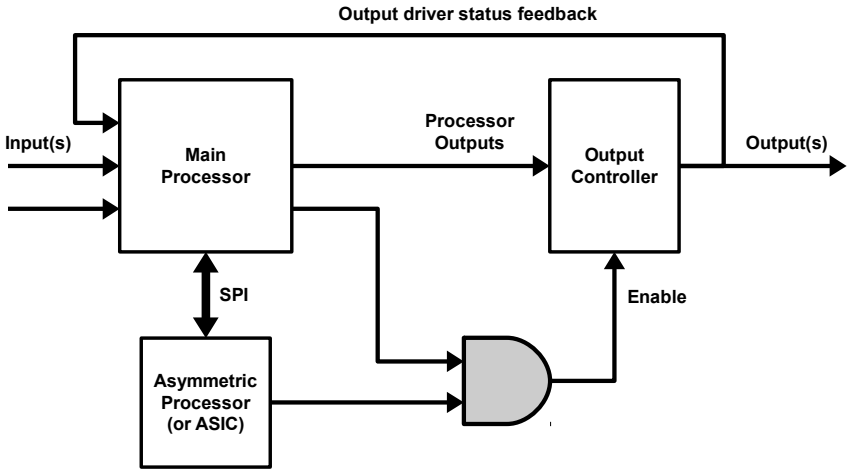


Figure 2.11: Asymmetric processor architecture (adapted from [29]).

The main processor runs the safety-related software and the monitoring task which communicates with the asymmetric processor. The software is implemented using redundant coding, i.e., data is stored in different representations in different memory locations, and certain self-tests are executed during start-up and during run-time [33]. The output

controller, as shown in Figure 2.11, represents the safety-related output driver stages of the main processor. Each processor can disable the enable signal to the output controller via the enable gate, in case a fault has been detected on the other processor.

An architecture in which the asymmetric processor is placed onto the same chip as the main processor, but is kept in form of an ASIC implementing an intelligent watchdog as Finite State Machine (FSM), is presented in [34].

This architecture is fail-silent and the achievable Automotive Safety Integrity Level is up to ASIL C/D [32].

The Dual Processor Architecture uses two identical CPUs running the same software, as shown in Figure 2.12. The primary and secondary sensor inputs are connected to the first and second CPUs, respectively. Both communicate via SPI or some other (high-speed) serial link and cross-check each other's software in near lock-step[6] with software synchronization. If the inter-processor communication follows a "question and answer" schema, this architecture is also known as the dual-core "challenge-response" architecture [30].

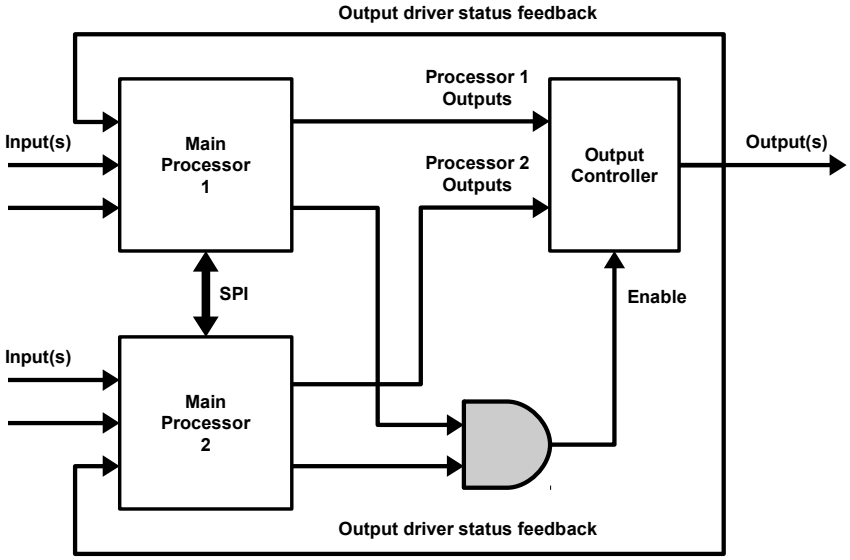


Figure 2.12: Dual processor architecture (adapted from [29]).

Both processors run the safety-related software and a monitoring task which communicates with the respective other processor's monitoring task. The software is implemented using redundant coding, i.e., data is stored in different representations in different memory locations, and certain self-tests are executed during start-up and during run-time [33]. The output controller represents the safety-related output driver stages of both processors. Each processor can disable the enable signal to the output controller via the enable gate, in case a fault has been detected on the other processor.

This architecture is fail-silent and the achievable Automotive Safety Integrity Level is presumably ASIL D (no relevant studies found in the literature).

[6] http://en.wikipedia.org/wiki/Lockstep_(computing)

The Lock-step Processor Architecture uses two identical CPUs, like the Dual Processor Architecture, but here the two processors are running in hardware lock-step executing the same software, as shown in Figure 2.13. All inputs are routed to both processors. There is no need for communication between the processors, synonymously named cores, because the address and data compare unit ensures that the software produces the same results on both cores. Because the same software is running on both cores, the software development efforts are considerably reduced as compared to the other architectures.

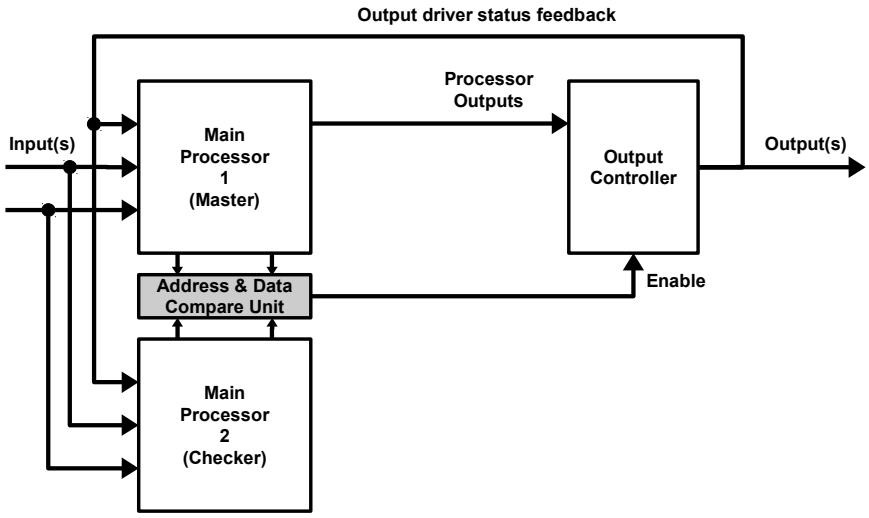


Figure 2.13: Lock-step processor architecture.

The software is as implemented using redundant coding, i.e., data is stored in different representations in different memory locations, and certain self-tests are executed during start-up and during run-time [33]. The output controller represents the safety-related output driver stages of the main processor. The results of the second processor are only used for comparison. The address and data compare unit can disable the enable signal to the output controller in case a discrepancy in the execution results is detected.

If more than two cores are used and the address and data compare unit is changed into a majority voting function, the architecture is called MooN (M out of N) or majority voting system [35]. An example is the Triple Modular Redundant (TMR) architecture for three cores [27]. In contrast to the aforementioned architectures which are fail-silent, this architecture is fault-tolerant because it can still keep up its work if at least two of the processors are fully functional.

The original lock-step processor architecture is fail-silent, optionally fault-tolerant as MooN (M out of N) system, and the achievable Automotive Safety Integrity Level is ASIL D [32].

2.5 The E-Gas Architecture and Safety Concept

Designing safety-critical systems in the context of ISO 26262 leads to the E-Gas or Electronic Throttle Control (ETC)[7] architecture and safety concept, one of the automotive industry's "best practice" approaches. It is also called the 3-level safety monitoring pattern [36] or the "challenge-response" architecture [30].

The E-Gas or ETC architecture and safety concept [37, 38] is shown in figure 2.14. It is patented in [39], probably based on the preceding patent [40], and extended by the patent [41]. A dual-core variant is patented in [42]. Other possibly related patents are [43–46].

The Asymmetric Processor Architecture, described in the previous section on page 14, forms the basis for this architecture. The function controller is the main processor and the monitoring controller is the asymmetric processor. The latter can either be a microcontroller or an ASIC. Both are running on independent clocks and monitor each other's power supply.

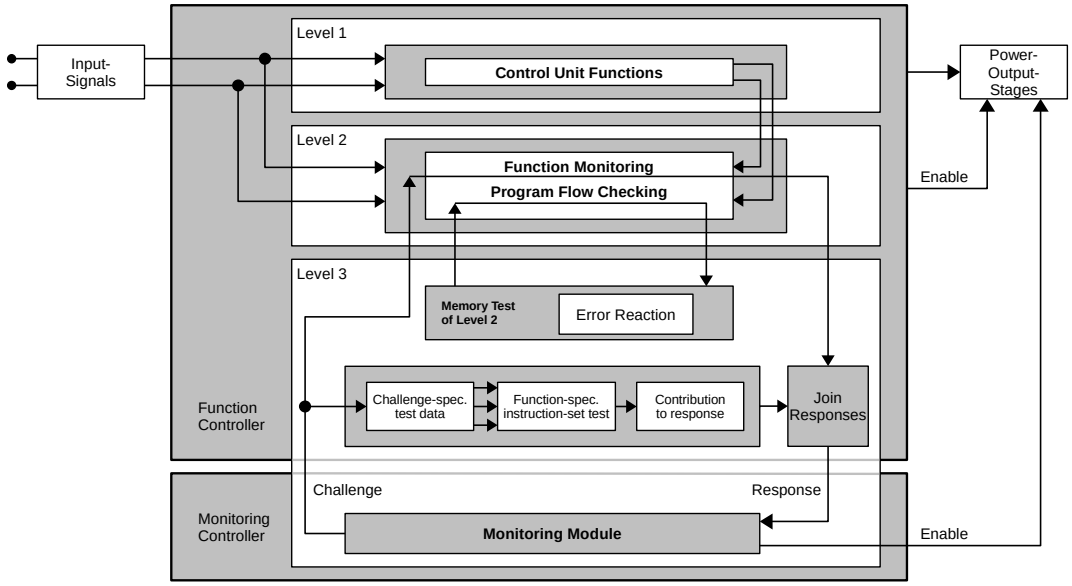


Figure 2.14: E-Gas architecture hard- and software concept (translated from [37]).

The software is structured as three levels implementing the following functionality:

Level 1 (Functional Level) implements all engine control functions, and diagnostics of input and output variables requiring monitoring.

Level 2 (Function Monitoring Level) implements the monitoring of functions that determine the performance in level 1, e.g., by torque or acceleration monitoring. The level 1 error reaction is monitored if level 2 cannot independently generate an error reaction. The memory area for variables is kept separate from that of level 1 and has to be monitored cyclically. The program flow of level 1 is checked as well.

[7] http://en.wikipedia.org/wiki/Electronic_throttle_control

Level 3 (Controller Monitoring Level) consists of a monitoring module running on the monitoring controller, and the monitoring software running on the function controller, these two communicating with each other in a challenge-response fashion. RAM/ROM storage is tested at least once per driving cycle. The status of level 2 program flow monitoring and the result of the RAM/ROM storage test are checked, and the function-specific processor instructions are tested as well.

2.6 Automotive Bus Systems Overview

A brief overview of contemporary automotive bus systems, based on data from [47–49], is shown in Table 2.3. A comparison of the costs per node of different bus systems is shown

Table 2.3: Classification of bus systems.

Class[a]	**Data rate [bit/s]**	**Motivation**	**Protocols**	**Application**
A	< 10k	Simple control data, Low-cost technology	ISO 9141 K-Line, LIN	Diagnosis, chassis electronics (door locking, climate control, ...)
B	10k...125k	Direct data exch. between ECUs	CAN (Low-Speed)	
C	125k...1000k	High-speed communication requirements, gateway between subsystems	CAN (High-Speed)	ABS, ADAS, engine control, electronic gear box, increasingly diagnosis
"D"	> 1M		FlexRay, CAN FD	Powertrain, steer- and brake-by-wire
"E"	> 10M		MOST, Ethernet	ADAS sensor fusion, multimedia (audio, video)

[a]only SAE classes A to C formally defined

in Figure 2.15. One can easily see why the CAN bus, despite its known limitations [1], is still the dominant bus system in the automotive world today.

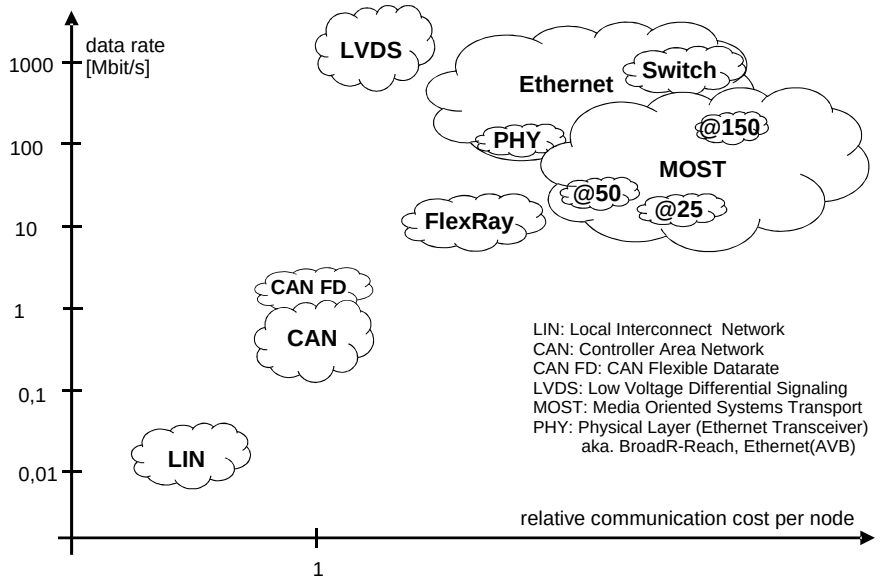


Figure 2.15: Communication cost per node (adapted from [50]).

Further reading on automotive bus systems can be found in [1, 47–51].

2.7 FreeRTOS Operating System

The FreeRTOS project website [52] describes FreeRTOS as follows [53]:

> FreeRTOS^{TM} is a market leading RTOS from Real Time Engineers Ltd. that supports 33 architectures and received 103000 downloads during 2012. It is professionally developed, strictly quality controlled, robust, supported, and free to embed in commercial products without any requirement to expose your proprietary source code.
>
> FreeRTOS has become the de facto standard RTOS for microcontrollers by removing common objections to using free software, and in so doing, providing a truly compelling free software model.

The following sections will cover General RTOS Fundamentals and the details of FreeRTOS Tasks and Scheduling, Communication and Synchronization, Software Timers, Memory Management, the FreeRTOS The Portable Layer and FreeRTOS Additional Features. For the sake of completeness, it has to be stated that all references to FreeRTOS in this document refer to FreeRTOS Version 7.3.0.

2.7.1 General RTOS Fundamentals

This section gives an overview of the architecture of a Real-Time Operating System (RTOS), of multitasking, and of what real-time is about. More in-depth information related to the subject can be found in [54, Chapter 3], [55, Chapter 2], and [56, 57].

Architecture

The general architecture of an RTOS is shown in Figure 2.16. The RTOS kernel is in control of the Central Processing Unit (CPU) and provides the following fundamental functionality of a task:[8] management (scheduling, dispatching), communication, and synchronization. Additional functionality like memory management and support for peripherals is either provided by the RTOS itself, additional drivers alongside the RTOS, or by a so called Hardware Abstraction Layer (HAL). The HAL would be added as an additional layer between the hardware and all other layers, the advantage being that a change in hardware does not affect the RTOS and the application(s).

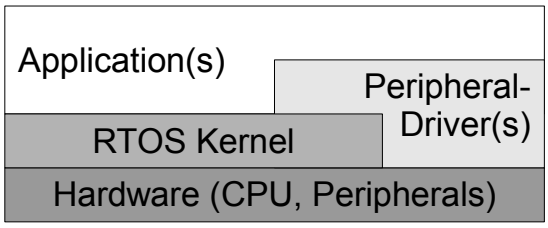


Figure 2.16: General RTOS architecture.

[8] Synonymously called "process."

Multitasking

Most people know multitasking from their desktop PC as the parallel execution of programs, although their PC, till a few years ago, did only possess one CPU. Therefore, this "parallel" execution will only be quasi-parallel because each program, or task has to share the same CPU with others. Multitasking can also be considered an "illusion of simultaneity" [54]. Figure 2.17 shows a comparison of sequential task execution and multitasking. With multitasking, the tasks are executed in a time-sliced manner, which creates the aforementioned illusion of simultaneity.

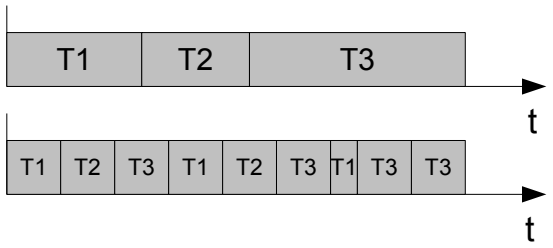


Figure 2.17: Sequential execution (top) vs. multitasking (bottom) [55].

Though multitasking shows that the tasks are executed in the same timespan as with sequential execution, there is overhead added by changing the order of execution from one task to the other. This is called a context switch, whereby sufficient information of a task is saved, so that it can be restored, or resumed, at a later time. In reality, the total "run to completion" time of the three tasks in the above example will be slightly longer, if multitasking is used instead of sequential execution.

The tasks' order of execution is determined by the task scheduler. If a task has to wait for, e.g., an I/O operation to complete or a timeout to happen, the task scheduler can relinquish control to another task in the meantime. An RTOS, or an operating system in general, usually provides an interface for letting a task wait for the occurrence of certain events. This leads to the actual definition of a task.

Task

As [54] puts it, "A task is an abstraction of a running program and is the logical unit or work scheduled by the operating system." As shown in the previous section in Figure 2.17, a task will not always be running, but could also be suspended or waiting while another task is being executed.

Figure 2.18 shows the general states of a task as a finite-state automaton with four states - the task state model. In the "ready" state a task can be run anytime, in the "active" state it is executed, and in the "blocked" state it is waiting for a resource to become available. In the "terminated" state the task is removed from the schedule. In an RTOS, the terminated state is usually not present as the system will typically be executing a fixed set of tasks for eternity, at least for as long as the system remains switched on.

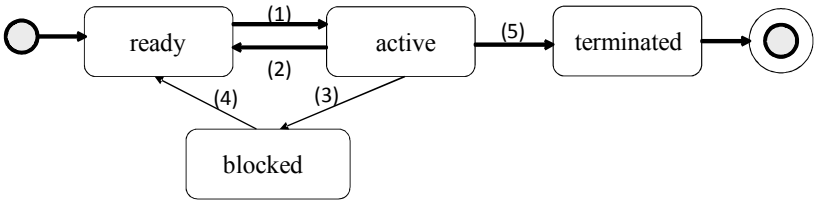


Figure 2.18: Task states as finite-state automaton [58].

The state transitions in Figure 2.18 can be described as follows [58]:

1. The Operation System (OS) selects the task to run (activation)
2. The OS selects another task (deactivation, preemption, interrupt)
3. The task blocks (e.g., waits on input or resource)
4. The task unblocks (e.g., resource becomes available)
5. The task ends or is terminated

Real-time

A common misconception about real-time systems is that these systems have to be fast, and therefore have to have a very short response time. Basically, it really depends on the application of the system; for example, the response time of a ticketing system will be dramatically different from a fuel injection system.

[54] gives the following definitions for a real-time system:

> **Definition:** A real time system is a system that must satisfy explicit (bounded) response-time constraints or risk severe consequences, including failure.

or more generally [54]:

> **Definition:** A real-time system is one whose logical correctness is based on both the correctness of the outputs and their timeliness.

With this in mind we can now differentiate between systems where not meeting a response-time constraint does not lead to a catastrophic failure [54]:

> **Definition:** A soft real-time system is one in which performance is degraded but not destroyed by failure to meet response-time constraints.

and those where it does [54]:

> **Definition:** A hard real-time system is one in which failure to meet a single deadline may lead to complete and catastrophic system failure.

A system in between soft and hard real-time, where a small number of missed deadlines can be tolerated, can be defined as follows [54]:

> **Definition:** A firm real-time system is one in which a few missed deadlines will not lead to total failure, but missing more than a few may lead to complete and catastrophic system failure.

Into which of the three categories an embedded system fits just depends on the definition of the response-time constraints. Relaxing those can change a firm real-time system into a soft one, or a hard one into a firm one.

2.7.2 Tasks and Scheduling

Task State Model

Figure 2.19 presents the FreeRTOS task state model, based on [59, Figure 7.], with the task termination transitions and the "terminating" state added to the finite-state automaton.

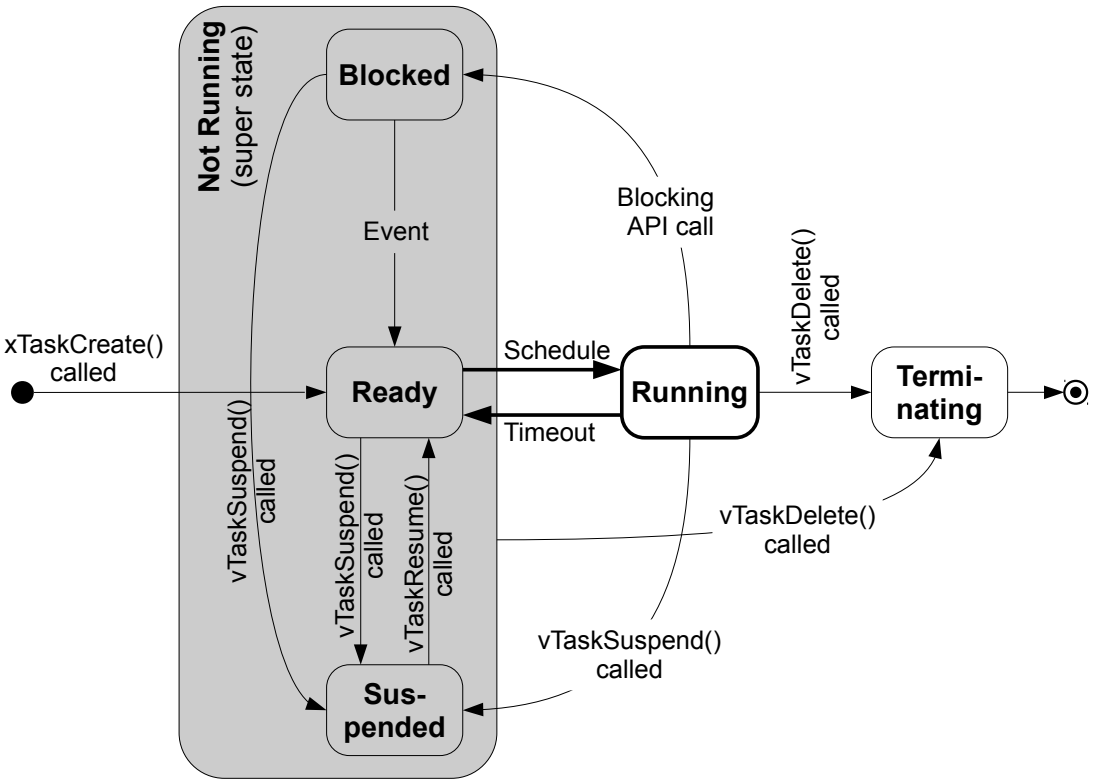


Figure 2.19: States of a task in FreeRTOS (extension of [59]).

The functions referenced in Figure 2.19 are API function calls which are documented in [60] or online at [61].

Task Management

FreeRTOS uses the Task Control Block (TCB) model [54] to represent and manage tasks in the system. The FreeRTOS TCB and its essential fields are shown in Figure 2.20.

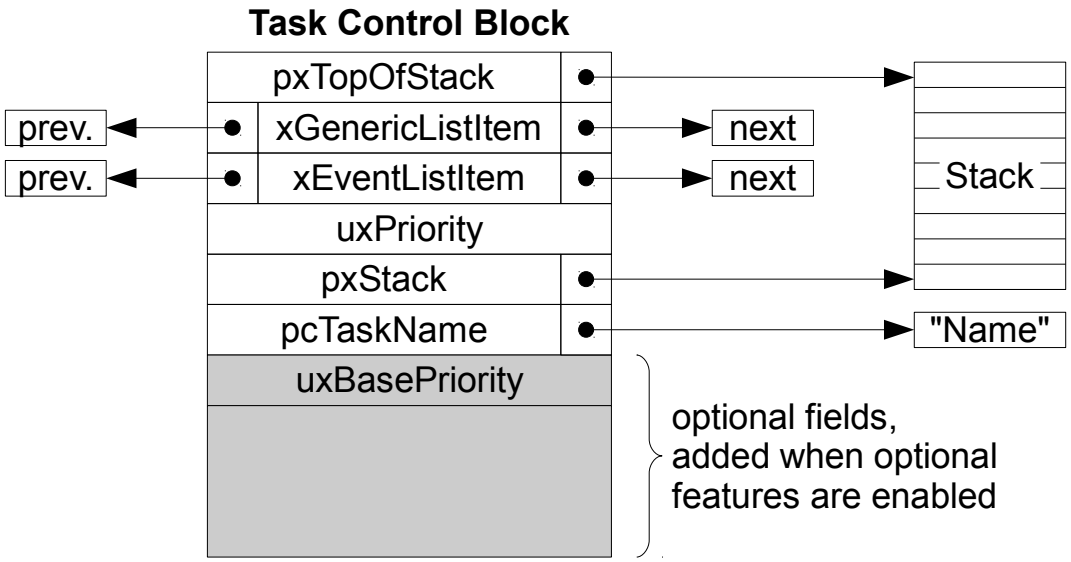


Figure 2.20: FreeRTOS task control block (TCB).

Each task gets allocated a separate task stack which is referenced in the TCB; in the literature it is called "Multiple-Stack Arrangement" [54]. In Figure 2.20, the stack shown grows from top to bottom. A stack that grows from bottom to top is supported as well on architectures that need such support. The pointers to the stack are optionally used to do some rudimentary stack-overflow checking during run-time. This feature is mainly enabled and used during the software development phase.

With the field *xGenericListItem* the TCBs are inserted into different double-linked lists, i.e., into lists for ready, blocked (a.k.a. delayed), suspended, or terminating tasks. Because a task is put into the respective list representing its state, there is no need to keep the task's state in the TCB itself. Thus, if a task changes its state, it is just moved from one list to another. The field *xEventListItem* is used to reference the TCBs from event lists, e.g., a reference to the task's *xEventListItem* is added to the event list of a queue, when the respective task is blocked on reading from a queue.

The field *uxPriority* is used to represent the task's execution priority. It is only assigned during task creation and never changed by the kernel itself. If enabled in the FreeRTOS configuration, tasks can change their priority during run-time. When mutexes are enabled, *uxPriority* is saved to *uxBasePriority* during a "priority inheritance" phase, and updated to the highest priority needed to avoid the so called priority-inversion [54, Section 3.3.10] problem.

Task Scheduler

The FreeRTOS scheduler, illustrated in Figure 2.21, can be characterized as follows:

- fixed-priority preemptive scheduling
- round-robin scheduling at the same priority level
- configurable number of priority levels, priority 0 being the lowest priority
- only mutexes include a basic 'priority inheritance' mechanism
- (optional feature) priorities changeable by tasks during run-time
- the per-task release or arrival time has to be set by explicit calls to *vTaskDelay()* or *vTaskDelayUntil()* - tasks are endless loops
- optionally, cooperative scheduling (a.k.a. coroutines) can be enabled

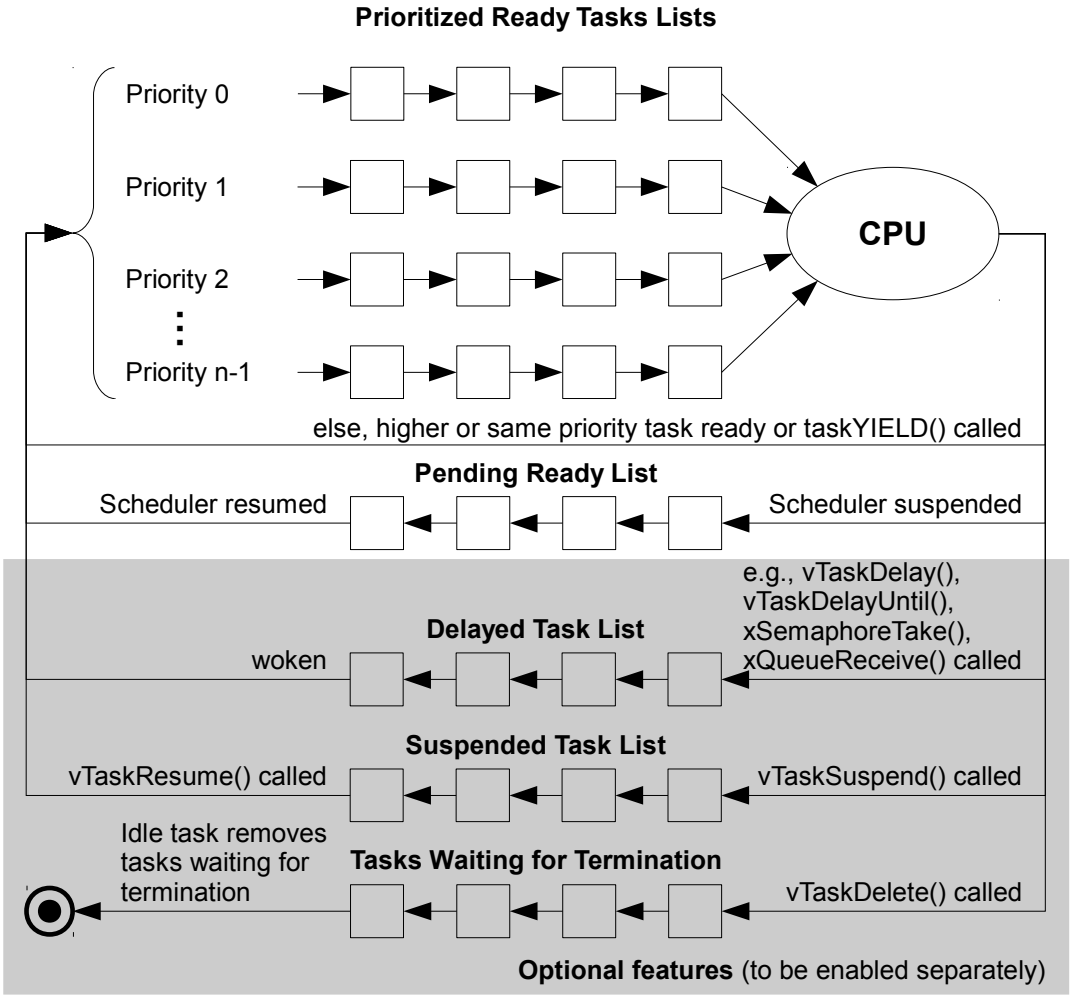


Figure 2.21: FreeRTOS scheduler using double-linked lists.

Tasks gain control of the CPU in accordance with their priorities and when their current state is "ready". At any given time the task with the highest priority, i.e., the current

task, is in control of the CPU. The prioritized, or fixed-priority preemptive scheduling policy is the one used by most multitasking RTOS products, according to [62]. This is due to the rate-monotonic (RM) theorem and the bound on the rate-monotonic algorithm (RMA), which can be stated as follows [54]:

> **Theorem (Rate-monotonic)** Given a set of periodic tasks and preemptive priority scheduling, then assigning priorities such that the tasks with shorter periods have higher priorities (rate-monotonic), yields an optimal scheduling algorithm.
>
> **Theorem (RMA Bound)** Any set of n periodic tasks is RM schedulable if the processor utilization, U, is no greater than $n \cdot (2^{1/n} - 1)$.

For big n, this converges rapidly to $ln(2) \approx 0.69$ as shown in Figure 2.22. [54] further states:

> Note that the RMA utilization bound is sufficient, but not necessary. That is, it is not uncommon in practice to construct a periodic task set with total processor utilization greater than the RMA bound but still RM-schedulable.

It is therefore up to the designer to choose the right schedule.

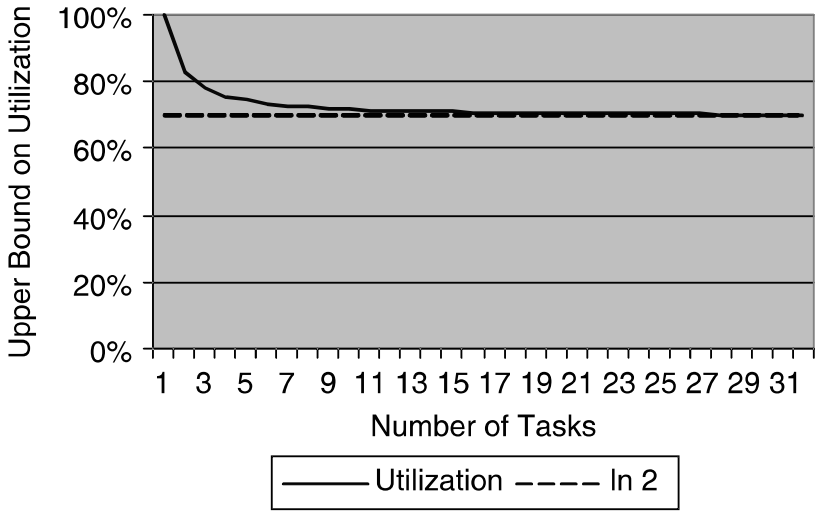


Figure 2.22: Rate-monotonic scheduling and utilization [54].

If support for task termination is enabled in FreeRTOS, tasks which are to be terminated are placed in a 'waiting for termination' list. The always existing *idle* background task will take care of freeing a task's TCB and the associated stack space. Any other memory area allocated to the task has to be freed explicitly before terminating the task.

The cooperative scheduling policy is not at all documented in the latest available printed FreeRTOS documentation [59, 60], and will hence not be discussed any further here.

More information on the FreeRTOS Task API can be found in the books [59] and [60], or online in [61]. [63] gives a short overview of the basic architecture of FreeRTOS. [64] is

another, if a little outdated, document available online, which provides information about the inner workings of FreeRTOS.

2.7.3 Communication and Synchronization

FreeRTOS offers queues [54, Section 3.3.5] for inter-task communication, and semaphores [54, Section 3.3.7] and mutexes [54, Section 3.5.2] for synchronization and resource management purposes. In [59] it is stated that "The 'queue' is the underlying primitive used by all FreeRTOS communication and synchronization mechanisms.".

Queues, as depicted in Figure 2.23, are used to convey information from one task to another. In FreeRTOS, queues consist of a fixed number of fixed sized elements and are

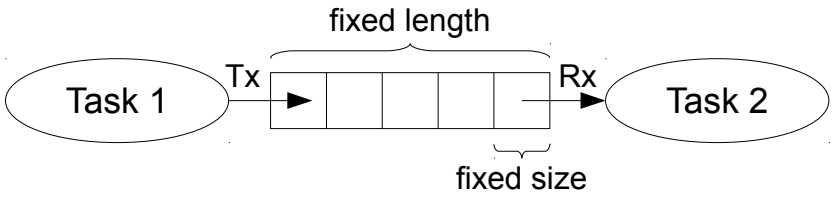


Figure 2.23: The FreeRTOS queue.

usually used as First In First Out (FIFO) style buffers. Data posted to a queue is copied into the queue and not just passed as a reference. It is therefore advisable to keep the data elements in the queue as small as possible.

Tasks reading from the queue can either block indefinitely, or specify a maximum 'block' time. The same holds true for tasks writing to the queue. It is possible to have multiple tasks writing to a queue and multiple tasks reading from it. Therefore, each queue has two associated event lists where tasks waiting to send or tasks waiting to receive are managed - see also *xEventListItem* on page 24.

From an Interrupt Service Routine (ISR), only functions with the suffix 'FromISR' are allowed to be called to transmit information, e.g., from the CAN receive ISR to a task or vice versa.

Semaphores and mutexes are essentially queues with elements of size zero. Accessing them from an ISR is also only allowed via functions with the 'FromISR' suffix.

More information on the FreeRTOS Queue and the Semaphore API can be found in the books [59] and [60], or online in [61].

2.7.4 Software Timers

In embedded systems the functionality of a timer is usually provided by a hardware timer unit. The function executed when the timer expires is called a timer ISR. Programming such a timer is hardware-dependent, hence it cannot be ported to other platforms.

Software timers, on the other hand, provide a convenient, hardware-independent way of calling software functions, so called callback functions, at a given time in the future. Hardware timers usually provide a better resolution and less jitter, but this is not always required.

FreeRTOS software timers allow tasks to have a callback function executed when a timer expires. Timers can be created on demand and can be of type 'auto-reload' or 'one-shot'. Changing the auto-reload period, resetting, starting, or stopping a timer during run-time is also supported.

Timers in FreeRTOS are handled by a separate process, the *timer service task*. Commands are sent to this task via a separate queue to control the timers. The callback functions are executed in the context of this task and are therefore not allowed to use blocking function calls. ISRs are only allowed to call timer API functions with the 'FromISR' suffix.

More information on the FreeRTOS Timer API can be found in the book [60], or online in [61].

2.7.5 Memory Management

FreeRTOS comes with the memory managers shown in Table 2.4. Most of the memory managers are built around a statically allocated array, which represents the heap. The memory manager "heap_1" probably covers most of the applications where FreeRTOS is used.

Table 2.4: FreeRTOS memory managers.

Memory Manager	**free()**	**Memory defragmentation**	**Heap pre-allocated**	**Deterministic**
heap_1	no	no	yes	yes
heap_2	yes	no	yes	no
heap_3	yes	dependent on C library	dependent on C library	no
heap_4	yes	yes	yes	no

More details can be found in the book [59] or online in [65].

2.7.6 The Portable Layer

The FreeRTOS portable layer has to provide the following hardware-dependent functionality:

- defining platform-dependent data types
- defining the stack growth
- function for initializing a task's context
- function for saving and restoring the task context and updating the task's TCB
- function for configuring the scheduler tick interrupt (hardware timer)
- function for starting the scheduler
- function for ending the scheduler
- scheduler tick interrupt function

- yield function
- critical section management functions

Usually the portable layer consists of just two files, namely `portmacro.h` and `port.c`, and is the only place where inline Assembler code is used. It is also the only hardware-dependent part of FreeRTOS; all other functionality is platform-independent C code.

The configuration of the tick interrupt in the portable layer defines the granularity of the timer ticker and thus the minimal time slice of the scheduler. The timer tick interval is inversely proportional to the overhead added by the saving and restoring of process contexts.

2.7.7 Additional Features

On some ports[9] there is also support for a Memory Protection Unit (MPU) [66], [67]. An MPU is a subset of a Memory Management Unit (MMU), which only provides protection of memory regions against unauthorized access, but no support for virtual memory.

A new feature in FreeRTOS is the so called "QueueSet". Instead of blocking on different queues, several queues can be put into a set and then a task can block on the whole set [68], [69].

Run-time statistics is another (optional) feature provided by FreeRTOS [70]. This feature collects and provides information about the absolute time and the time in percentage used by each task.

Tickless Idle Mode has recently been added to FreeRTOS, for better use of low-power modes supported by certain microcontrollers [71]. The tick interrupt usually wakes the CPU at an interval of, e.g., 1 ms. When no task needs attention, this interval is automatically extended to the time where the CPU really needs to wake up. This helps saving energy by only waking the CPU when there is work to be done.

Trace macros provide hooks to assist in debugging real-time applications [72]. This allows for simple instrumentation of the FreeRTOS kernel to, e.g., measure a task's activation time via CPU port pins.

[9] A "port" denotes FreeRTOS ported to a specific platform.

Chapter 3

Design and Implementation

The goal of this work is to design an ECU for a Formula Student Electric race car. First we will look into the design process according to the ISO 26262 standard, which is outlined in Section 2.3. The result is than an ECU platform for Formula Student Electric race cars, which is presented in Section 3.7. Finally we will discuss the porting of the FreeRTOS operating system, which is described in Section 2.7, to the chosen CPU platform.

3.1 Applicability of ISO 26262

Formally, the IEC 61508 is the functional safety standard which would be applied to the development of Electrical and/or Electronic and/or Programmable Electronic (E/E/PE) systems for Formula Student Electric race cars.

Formula Student Electric race cars - or Formula Student race cars in general - are designed to be produced in a quantity of 1000 units per year - see Appendix D on page 108, Sections D.1.1 and D.1.3. The IEC 61508 focuses on equipment safety with equipment produced in low volumes, whereas the ISO 26262, in contrast, focuses on vehicle safety provided by functionally safe control systems, and takes mass production into account [73].

The ISO 26262 could be imagined as the standard for other road vehicles as well, e.g., for race cars as in our case, as [23] explains (translated from German):

> However, the ISO 26262 does not prohibit the extension of its scope to other categories of vehicles. The ISO 26262, as the industry-specific derivative of the IEC 61508, represents a very good interpretation of the standard for all classes of road transport vehicles. In principle, the application of the ISO 26262 to road vehicles is thus possible and reasonable, provided that appropriate accompanying measures ensure that the current state of the art in science and technology is achieved.

One example is MAN Truck & Bus AG, which applies the ISO 26262 to trucks and buses, see [74] or [75].

In comparison to the IEC 61508, we considered the ISO 26262 a better representation of the current state of the art in the Automotive sector. Therefore we decided to align the development of Electrical and/or Electronic (E/E) systems for Formula Student Electric race cars with the ISO 26262.

3.2 Development as Safety Element out of Context (SEooC)

The ISO 262626 part 10 [22], despite only being informative, introduces in clause 9 the concept of developing a "Safety Element out of Context (SEooC)" which is not developed in the context of a specific automobile.

According to [25], "Concepts such as 'Safety Element out of Context' within the ISO DIS 26262 standard help to address the problem of component construction in a safety-related development process [...]".

[76] defines a SEooC as follows: "A Safety Element out of Context (SEooC) is a safety element for which an item[10] does not exist at the time of the development. A SEooC can either be a subsystem, a software component, or a hardware component."

The development of a (sub-)system, hardware, or software component as SEooC requires the developer to make assumptions on the design and assume requirements, as shown in Figure 3.1.

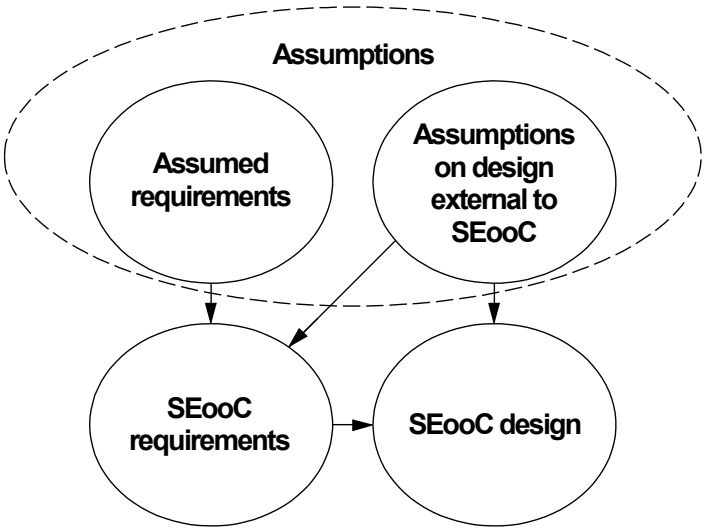
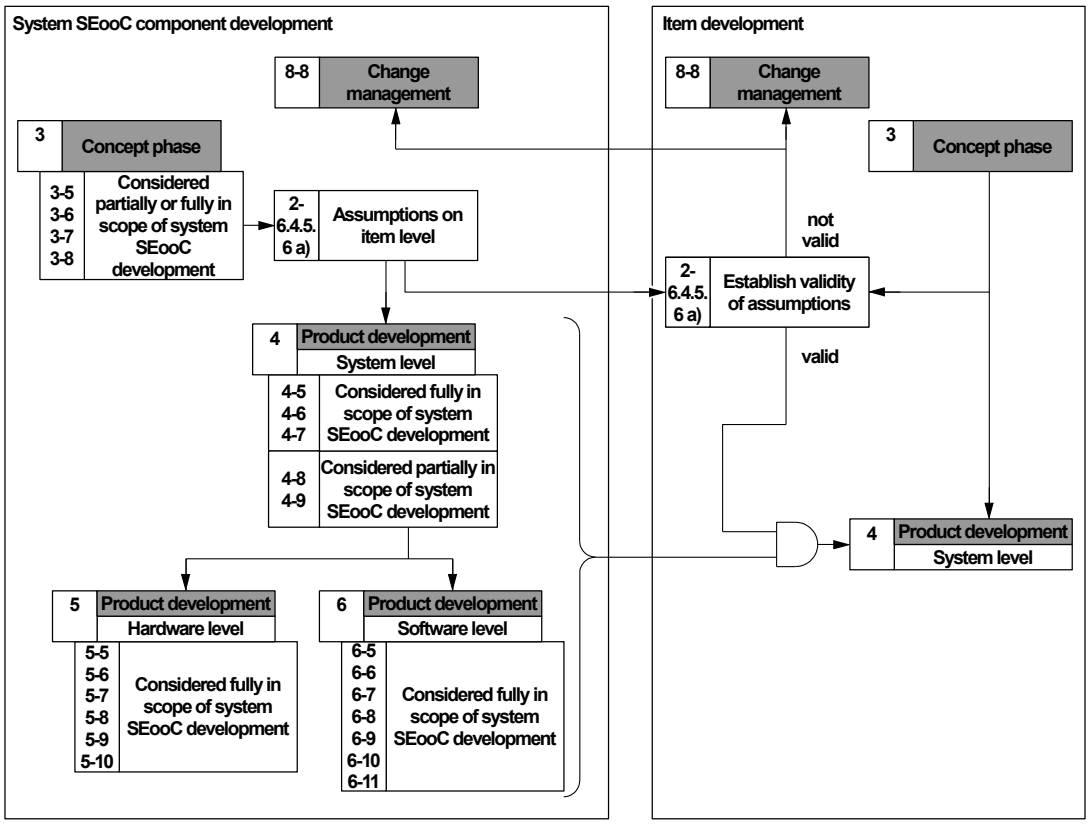


Figure 3.1: Relationship between assumptions and SEooC development [22, Figure 18].

The standard defines the SEooC (sub-)system development as depicted in Figure 3.2. For a new design, first the concept phase (ISO 262626-3 [15]) is executed and then assumptions on item level are derived. If a previously developed SEooC is being reused, the results from the concept phase just need to be updated. The assumed functional safety requirements then form the input for the design at system level (ISO 262626-4 [16]), hardware level (ISO 262626-5 [17]), and/or software level (ISO 262626-6 [18]).

All assumptions made during the SEooC development process, and all work products produced, are to be documented in a so called "Safety Manual". The verification of the correct implementation of the assumed safety requirements is done during the SEooC development process as well.

[10] For the ISO 26262 definition of 'item', see Section B.2 on page 84



Note 1: "m-n" denotes ISO 26262 part "m" clause "n"

Note 2: Some additional tailoring of the requirements can be necessary depending on the exact nature of the SEooC.

Note 3: Depending on the exact nature of the SEooC, some requirements of parts 3 and 4 cannot be applicable, and therefore only partial consideration is made.

Note 4: Although all clauses are not shown, this does not imply that they are not applicable.

Figure 3.2: SEooC system development [22, Figure 19].

When the SEooC is then used in the development of an item, the functional safety requirements of the item are matched against the functional safety requirements assumed for the SEooC. In case of a mismatch, a change management process has to be triggered, which includes an impact analysis. The assumed safety requirements are validated during item development, not during the SEooC development process.

For further information regarding the development of hardware and software components as SEooCs, please refer to the ISO 26262-10:2012 standard [22] and [77–79].

3.3 SEooC Concept Phase

Because there is no prior knowledge available about the functional safety requirements for an ECU developed in the domain of Formula Student Electric race cars, we have to build

up that knowledge first, to be able to make assumptions on functional safety requirements later on.

As described in the previous section, "Development as Safety Element out of Context (SEooC)", if we want to develop our new ECU as a SEooC, we have to complete the concept phase of ISO 262626-3 [15] first - see Figure 3.1. The sub-phases "Item Definition", "Hazard Analysis and Risk Assessment (HARA)", and "Functional Safety Concept (FSC)" are described in the following subsections.

3.3.1 Item Definition

The aims of the item definition according to ISO 262626-3 clause 5 [15] are the following:

> The first objective is to define and describe the item, its dependencies on, and interaction with, the environment and other items.
>
> The second objective is to support an adequate understanding of the item so that the activities in subsequent phases can be performed.

From 2010 to 2012, the TU Graz Racing Team has been taking part in the Formula Student Electric design and race events. The knowledge obtained during those seasons is now used to make assumptions on dependencies and interactions, and to define the item in whose context the ECU should be developed.

The item definition of a Formula Student Electric race car deploying two separately driven rear wheels, which is the electric powertrain architecture chosen by the TU Graz Racing Team, is shown in Figure 3.3. All general requirements are listed in the Formula SAE rules [80] and the Formula Student Electric rules [81], as well as in Appendix D.

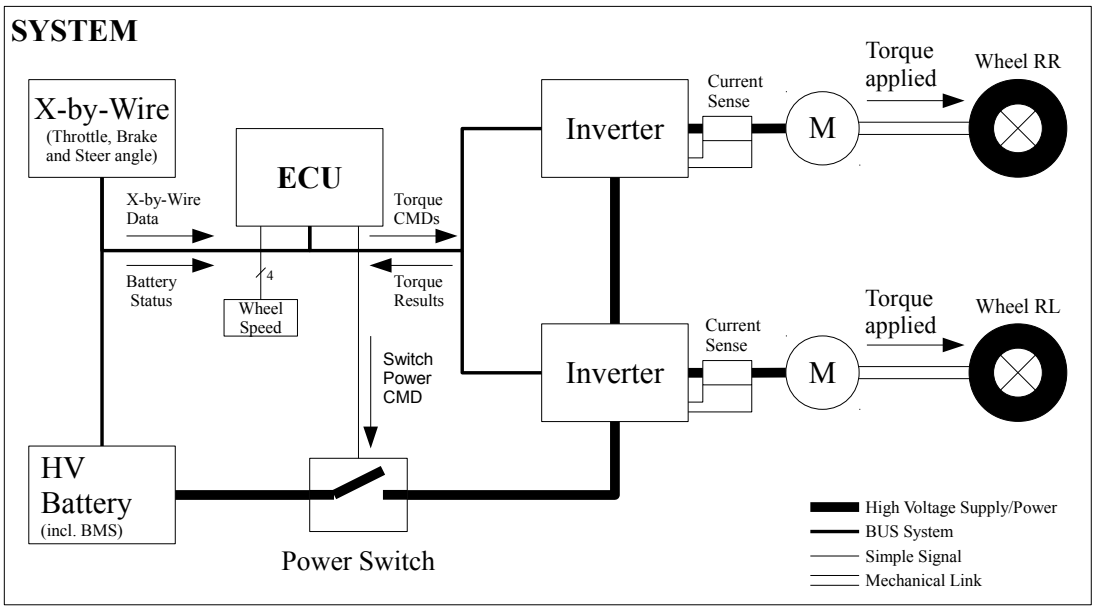


Figure 3.3: Item definition of a Formula Student Electric race car.

The ECU in the center of the item and the other subsystems interact as follows:

- The X-by-Wire subsystem contains redundant throttle, brake, and steering angle sensors, and sends this data to the ECU.
- The ECU takes the data from the X-by-Wire subsystem and the locally attached wheel speed sensors, and calculates the torque to be requested from each of the two inverters. It also controls the HV contactors ("Power Switch").
- Each Inverter takes the commanded torque, calculates the corresponding Alternating Current (AC) current, and delivers it to the attached motor. It also measures the resulting currents, calculates the corresponding torque, and sends that result to the ECU.
- The HV battery contains rechargeable Lithium Ion (LION) battery cells and a Battery Management System (BMS)[11] which sends battery health data to the ECU. If that data shows that at least one of the cells is out of its safe operating area, the ECU switches off the HV power supply of the inverters.

3.3.2 Situation Analysis

As a part of the hazard analysis and risk assessment (see next section), a situation analysis according to ISO 262626-3 clause 7.4.2.1 [15] has to be made.

The situation analysis for the Formula Student Electric Germany 2012 race event is shown in Table 3.1. The full table can be found in Appendix C in Table C.1 on page 89. Here we analyze the driving situations which can occur during a Formula Student Electric race event. This data will then be used in the next section, when we look at possible malfunctions in certain driving situations. The value "Percentage of total time" influences the exposure value for a certain hazard.

As can be seen from Table 3.1, the autocross and endurance race events are where the major portion of time is spent during a Formula Student Electric event. The acceleration race event is almost negligible with regard to the time spent, and will therefore have a very low exposure value.

[11] http://en.wikipedia.org/wiki/Battery_management_system

Table 3.1: Situation analysis of the Formula Student Electric Germany 2012.

Situation	**Percentage of total time**	**Average duration [sec]**	**Approx. track length [m]**	**Top speed [km/h]**	**Average speed [km/h]**
Acceleration race event	0.72%	17.70	300	110	-
Skid-pad race event	4.07%	100.43	920	-	33
Autocross race event	14.33%	353.99	4800	90	50
Endurance race event	66.71%	1647.81	22000	100	50
Approaching start line (from queue)	5.67%	140.00	140	10	-
Standstill (engaged, waiting to start)	8.50%	210.00	-	-	-

Data Source: Formula Student Electric Germany 2012 event results, speeds estimated based on logged data.

3.3.3 Hazard Analysis and Risk Assessment (HARA)

The aim of the hazard analysis and risk assessment according to ISO 262626-3 clause 7 [15] is as follows:

> The objective of the hazard analysis and risk assessment is to identify and to categorise the hazards that malfunctions in the item can trigger and to formulate the safety goals related to the prevention or mitigation of the hazardous events, in order to avoid unreasonable risk.

The hazard identification and classification for the item depicted in Subsection 3.3.1 are shown in Table 3.2. The full table, which also contains the arguments for classification, can be found in Appendix C in Table C.2 on page 90.

As per ISO 262626-3 clause 7.4.3, each hazard is classified according to severity (S), probability of exposure (E), and controllability (C). To classify the probability of exposure of a situation, the information from the preceding Subsection 3.3.2, “Situation Analysis”, is used. Expert judgment is used to determine the class of severity and controllability. Finally, each hazard is assigned an ASIL, as well as a safety goal.

Tables for classes of severity, probability of exposure, and controllability, as well as a table for the ASIL determination can be found in Appendix B in Section B.3, “Tables from ISO 26262 Part 3”, on page 85.

Table 3.2: Hazard identification and classification for a Formula Student Electric race car.

Hazard			Classification				Safety Goal	
ID	**Possible Malfunction**	**Situation**	**S**	**E**	**C**	**ASIL**	**ID**	**Description**
HZ01	Unintended acceleration (at zero speed)	Standstill; marshal crossing 1 m in front of race car	2	3	2	A	SG01	The vehicle shall not accelerate without a valid throttle demand (above the throttle function threshold).
HZ02	Unintended acceleration (at low speed)	Approaching Start; marshal on the side of the start position	2	3	2	A	SG02	Unintended acceleration shall be prevented.
HZ03	Unintended acceleration (at medium speed)	Autocross or Endurance race event; medium speed (30-50 km/h)	2	4	2	B	SG02	Unintended acceleration shall be prevented.
HZ04	Unintended acceleration (at medium speed)	Autocross or Endurance race event; another car close behind or marshal close to track; medium speed (30-50 km/h)	3	3	2	B	SG02	Unintended acceleration shall be prevented.
HZ05	Unintended acceleration (at high speed)	Autocross or Endurance race event; high speed ($\geq$70 km/h)	2	4	2	B	SG02	Unintended acceleration shall be prevented.

(continued)

Table 3.2: (continued)

Hazard			Classification				Safety Goal	
ID	**Possible Malfunction**	**Situation**	**S**	**E**	**C**	**ASIL**	**ID**	**Description**
HZ06	Unintended acceleration (at high speed)	Autocross or Endurance race event; another car close behind or marshal close to track; high speed (≥70 km/h)	3	3	2	B	SG02	Unintended acceleration shall be prevented.
HZ07	Unintended acceleration forward one motor, backward the other motor; yawing moment (at medium or high speed)	Cornering; medium or high speed (30-50 km/h or ≥70 km/h)	2	4	2	B	SG02	Unintended acceleration shall be prevented.
HZ08	Unintended generative braking, i.e., acceleration backwards (at zero speed)	Standstill; queue of waiting cars 5-10 m behind	2	3	2	A	SG03	The vehicle shall not do generative braking or accelerate backwards (below the electric braking speed threshold).
HZ09	Unintended generative braking, i.e., acceleration backwards (at low speed)	Approaching Start; queue of waiting cars 5-10 m behind	2	3	2	A	SG03	The vehicle shall not do generative braking or accelerate backwards (below the electric braking speed threshold).

(continued)

Table 3.2: (continued)

Hazard			Classification				Safety Goal	
ID	**Possible Malfunction**	**Situation**	**S**	**E**	**C**	**ASIL**	**ID**	**Description**
HZ10	Unintended generative braking, i.e., acceleration backwards (at medium speed)	Autocross or Endurance race event; medium speed (30-50 km/h)	2	4	2	B	SG04	Unintended generative braking or acceleration backwards shall be prevented.
HZ11	Unintended generative braking, i.e., acceleration backwards (at medium speed)	Autocross or Endurance race event; another car close behind or marshal close to track; medium speed (30-50 km/h)	3	3	2	B	SG04	Unintended generative braking or acceleration backwards shall be prevented.
HZ12	Unintended generative braking, i.e., acceleration backwards (at high speed)	Endurance race event; high speed (≥70 km/h)	2	4	2	B	SG04	Unintended generative braking or acceleration backwards shall be prevented.
HZ13	Unintended generative braking, i.e., acceleration backwards (at high speed)	Endurance race event; another car close behind or marshal close to track, high speed (≥70 km/h)	3	3	2	B	SG04	Unintended generative braking or acceleration backwards shall be prevented.

(continued)

Table 3.2: (continued)

Hazard			Classification				Safety Goal	
ID	**Possible Malfunction**	**Situation**	**S**	**E**	**C**	**ASIL**	**ID**	**Description**
HZ14	Battery parameters out of range	Any driving situation	2	4	2	B	SG05	Battery parameters shall be kept within defined safe operating area.
HZ15	Missing acceleration	Any driving situation	1	4	1	QM	SG06	Missing acceleration shall be prevented.
HZ16	Unintended deceleration without generative braking	Any driving situation.	1	4	1	QM	SG07	Unintended deceleration without generative braking shall be prevented.
HZ17	Missing deceleration	Any driving situation	1	4	1	QM	SG08	Missing deceleration shall be prevented.

Information about injury risks in frontal impacts of Formula Student cars can be found in [82], and about the impact attenuator used in those cars in [83, 84].

3.3.4 Safety Goals

ISO 262626-3 clause 7.4.4.3 [15] defines safety goals as follows:

> Safety goals are top-level safety requirements for the item. They lead to the functional safety requirements needed to avoid an unreasonable risk for each hazardous event. Safety goals are not expressed in terms of technological solutions, but in terms of functional objectives.

The safety goals that were determined in the previous section are shown in Table 3.3. Each safety goal which can only be achieved by transitioning to, or maintaining, a safe state, is assigned such a safe state.

In the next subsection, the functional safety concept will be derived from these safety goals.

Table 3.3: Safety goals for a Formula Student Electric race car.

Safety Goal			
ID	**Description**	**Safe State**	**ASIL**
SG01	The vehicle shall not accelerate without a valid throttle demand (above the throttle function threshold).	Switch off power to the motors (demand zero torque).	A
SG02	Unintended acceleration shall be prevented.	Switch off power to the motors (demand zero torque).	B
SG03	The vehicle shall not do generative braking or accelerate backwards (below the electric braking speed threshold).	Switch off power to the motors (demand zero torque).	A
SG04	Unintended generative braking or acceleration backwards shall be prevented.	Switch off power to the motors (demand zero torque).	B
SG05	Battery parameters shall be kept within defined safe operating area.	Switch off power to the inverters.	B
SG06	Missing acceleration shall be prevented.	-	QM
SG07	Unintended deceleration without generative braking shall be prevented.	-	QM
SG08	Missing deceleration shall be prevented.	-	QM

SG01 gleaned from [85].

3.3.5 Functional Safety Concept (FSC)

ISO 262626-3 clause 8 [15] defines the objective of the functional safety concept as follows:

> The objective of the functional safety concept is to derive the functional safety requirements, from the safety goals, and to allocate them to the preliminary architectural elements of the item, or to external measures.

It is essential that the "concrete implementation should be hidden from [the] FSC" [86].

The functional safety concept we derived for our item is shown in Table 3.4. The full table, which also contains specifications like, e.g., fault-tolerant time interval, for each functional safety requirement, can be found in Appendix C in Table C.4 on page 100.

Table 3.4: Functional safety concept for a Formula Student Electric race car.

Functional Safety Requirement				**Safety Goal**
ID	**Description**	**ASIL**	**Allocated to Element**	**ID**
FSR01	The system shall not send throttle data which causes the vehicle to accelerate without a driver demand.	A	X-by-Wire	SG01
FSR02	The system shall not send a torque command which causes the vehicle to accelerate without a driver demand.	A	ECU	
FSR03	Accurate throttle, brake and steering angle signals shall be generated.	B	X-by-Wire	SG02, SG03, SG04
FSR04	The throttle, brake, and steering angle signals shall be received and verified.	B	ECU	
FSR05	Accurate front and rear vehicle speed signals shall be generated.	B	ECU	
FSR06	Accurate torque command shall be generated.	B	ECU	
FSR07	The torque command shall be received and verified.	B	Inverter	
FSR08	Accurate calculation of the resulting torque by means of phase current measurement.	B	Inverter	
FSR09	The torque result shall be received and verified.	B	ECU	
FSR10	The torque result shall be validated.	B	ECU	
FSR11	An accurate battery status shall be provided.	B	HV-Battery	SG05
FSR12	The battery status shall be received, verified, and and reacted upon in case it is out of the safe operating area.	B	ECU	

FSR01 gleaned from [85].

3.4 SEooC Assumptions on Item Level

As our ECU will be developed anew as a SEooC, we now have to make assumptions regarding its "intended functionality and use context which includes external interfaces" [22].

The validity of these assumptions will need to be established during the integration of the ECU into an actual item. In case discrepancies between the assumptions and the actual requirements are discovered during the integration, change management activities, starting with an impact analysis, will need to be done.

3.4.1 Intended Functionality

The intended functionality of the ECU is to control a Formula Student Electric race car as defined by the Formula SAE (FSAE) Rules 2012 [80] and the Formula Student Electric (FSE) Rules 2012 [81].

3.4.2 Safety Goals and Functional Safety Concept

As the ECU will be developed anew as a SEooC, the safety goals from Subsection 3.3.4, "Safety Goals", and the functional safety requirements from Subsection 3.3.5, "Functional Safety Concept (FSC)", are now used as assumptions on item level.

Full tables can be found in Appendix C as Table C.3, "Safety Goals", on page 99 and Table C.4, "Functional Safety Concept", on page 100, respectively.

3.4.3 Bus System

The chosen bus system is the Controller Area Network (CAN) bus [87, 88]. Despite the fact that the CAN bus has not been designed for safety-related communication, it can still be used in safety-related projects if appropriate measures are taken at the software level.

The industry best practice measures to provide reliable end-to-end communication, e.g., as used in the AUTOSAR project "SW-C End-to-End Communication Protection Library" [89], are listed below:

- Adding a counter (alive counter or sequence number) to each message sent.
- Adding a Cyclic Redundancy Check (CRC) checksum to each message sent.
- Using timeouts when receiving messages.
- Using a data ID to calculate the CRC checksum.

The failure modes covered by these mechanisms are listed in Table 3.5.

Table 3.5: E2E mechanisms vs. failure modes [89, Table 7-3].

Mechanism	**Detected failure modes**
Counter	Repetition, deletion, insertion, incorrect sequence
Timeout	Deletion, delay
Data ID	Insertion, addressing faults
CRC	Corruption

The system's bus configuration is shown in Figure 3.4 and each CAN bus configuration is listed in Table 3.6.

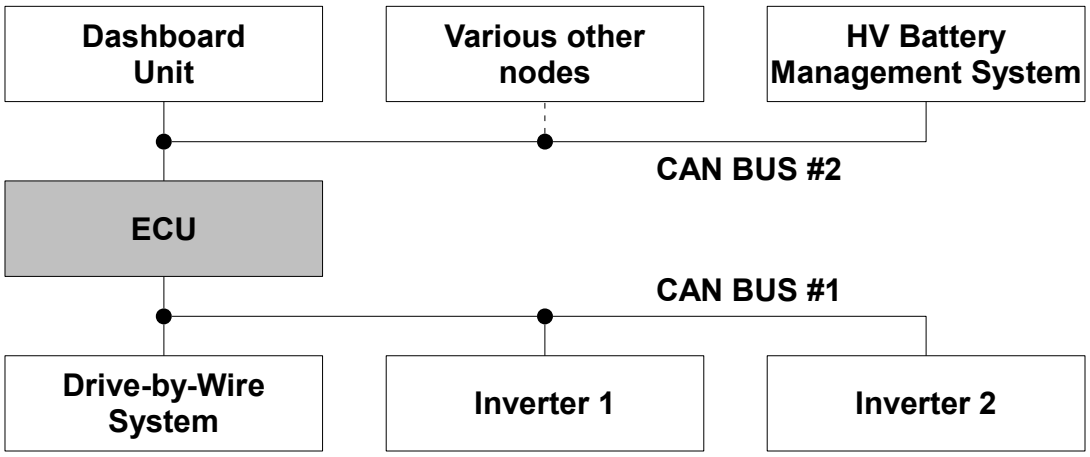


Figure 3.4: Assumed system's bus configuration.

Table 3.6: Configurations of the assumed CAN buses.

BUS #	Designation	Bitrate [kbps]	Adressing	Termination
1	Powertrain CAN	1000	11 Bit	Split termination
2	Feature CAN	500	11 Bit	Split termination

3.4.4 Communication with other devices

Because this work only deals with the development of the ECU, we have to make the following additional assumptions regarding the CAN communication (the CAN message numbers in square brackets refer to Table 3.7):

- The **Drive-by-Wire** subsystem transmits a CAN message containing the throttle signal, the inverse throttle signal, an alive counter, and a CRC checksum [*CAN Message #1*].
- The **Drive-by-Wire** subsystem transmits a CAN message containing the brake signal, the inverse brake signal, an alive counter, and a CRC checksum [*CAN Message #2*].
- The **Drive-by-Wire** subsystem transmits a CAN message containing the steering angle signal, the inverse steering angle signal, an alive counter, and a CRC checksum [*CAN Message #3*].
- Each **Inverter** expects to receive a CAN message containing the torque command, the inverse torque command, an alive counter, and a CRC checksum [*CAN Message #4*].
- Each **Inverter** transmits a CAN message containing the torque result, the inverse torque result, an alive counter, and a CRC checksum [*CAN Message #5*].

- The **Battery Management System (BMS)** transmits a CAN message containing the state of charge[12], the inverse state of charge, the overall BMS status, the inverse overall BMS status, an alive counter, and a CRC checksum [*CAN Message #6*].

The CRC checksum could use, e.g., the SAE J1850 CRC-8 polynomial as required by AUTOSAR E2E [89], or some other CRC-8 polynomial.[13]

Table 3.7: Assumptions on CAN messages (in compliance with AUTOSAR E2E [89]).

Msg#	ID	Int.[a] [ms]	DLC[b]	Byte							
				0	1	2	3	4	5	6	7
1	tbd.	10	6	CRC	Alive	Throttle Signal		$\overline{Throttle\ Signal}$		n/a	n/a
2	tbd.	10	6	CRC	Alive	Brake Signal		$\overline{Brake\ Signal}$		n/a	n/a
3	tbd.	10	6	CRC	Alive	Steering Angle Signal		$\overline{SteeringAngle\ Signal}$		n/a	n/a
4	tbd.	10	6	CRC	Alive	Torque Command		$\overline{Torque\ Command}$		n/a	n/a
5	tbd.	10	6	CRC	Alive	Torque Result		$\overline{Torque\ Result}$		n/a	n/a
6	tbd.	100	8	CRC	Alive	State of Charge		$\overline{State\ of\ Charge}$		Status	$\overline{Status}$

[a] Message transmission interval on the CAN bus.
[b] Data Length Code, indicates how many bytes a message contains.

3.4.5 Interlocks and Emergency Stop Function

According to the Formula Student Electric rules, "Rule 7.17 Safety Circuit" [81], hardware interlocks are demanded for the control of the HV contactors, as shown in Figure D.1 in Appendix D, on page 113.

Hardware Interlocks are well established in the industry. For example, [90] documents the use of interlocks already more than 65 years ago. How interlocks should be wired, as defined in [90], is shown in Figure 3.5. The reason for doing so is to safeguard the interlock circuit against faults against earth. Should a fault nonetheless happen, it will not jeopardize the safety system.

[12] http://en.wikipedia.org/wiki/State_of_charge
[13] http://en.wikipedia.org/wiki/Polynomial_representations_of_cyclic_redundancy_checks

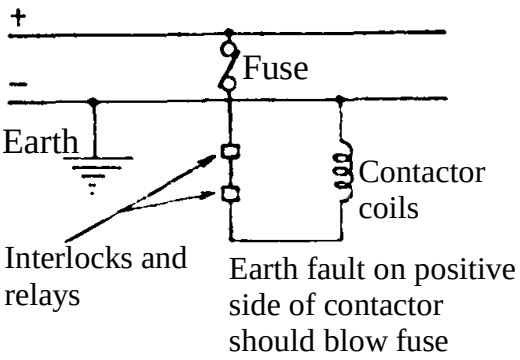


Figure 3.5: Wiring of interlock circuits [90, Fig. 3].

The emergency stop function demanded by the Formula Student Electric rules [81] is covered by ISO 13850, "Safety of machinery – Emergency stop – Principles for design", [91]. The emergency stop shall function according to *stop category 1*. This means that the emergency stop shall first deactivate the inverters, by demanding zero torque, and then shortly after remove the HV power. The emergency stop shall be executed within a defined Fault-Tolerant Time Interval (FTTI) of 100 ms.

3.5 SEooC Product Development at System Level

Now that the functional safety concept and the assumptions on item level have been established, development can continue at product level. In this section, first the technical safety requirements for the ECU will be derived from the functional safety concept, and then the ECU system design will be presented.

3.5.1 Technical Safety Requirements

ISO 262626-4 clause 6 [16] defines the objectives of the technical safety requirements as follows:

> The technical safety requirements specification refines the functional safety concept, considering both the functional concept and the preliminary architectural assumptions (see ISO 26262-3).

This is the step in the design, where we see that ASIL decomposition can and should be done, because after this step the requirements are assigned to hardware and software. A decomposition of a requirement into intended functionality and associated safety mechanism, according to ISO 262626-9 clause 5 [21], can then intuitively be mapped to the corresponding hardware and software requirements in the following design steps.

The technical safety requirements for the ECU, which were derived from the functional safety concept, are shown in Table 3.8. The full table, which also contains the specifications like, e.g., fault-tolerant time interval, for each technical safety requirement, can be found in Appendix C in Table C.5 on page 101.

Table 3.8: Technical safety requirements.

Technical Safety Requirement				**Func. Safety Req.**
ID	**Description**	**ASIL**	**Allocated to Element**	**ID**
TSR01	A throttle sensor signal shall be received from the CAN bus.	B	ECU	FSR02, FSR04
TSR01.1	The throttle sensor signal and its inverse representation shall be received from the CAN bus. Check the signal against its inverse representation and do a range check; assume 0, if failure.	QM(B)		

(continued)

Table 3.8: (continued)

Technical Safety Requirement				Func. Safety Req.
ID	**Description**	**ASIL**	**Allocated to Element**	**ID**
TSR01.2	Plausibility-check the throttle signal, e.g., gradient. Check the alive-counter for correct sequence and the checksum for message integrity. Force 0, if failure.	B(B)		
TSR02	A Brake sensor signal shall be received from the CAN bus.	B	ECU	FSR04
TSR02.1	The Brake sensor signal and its inverse representation shall be received from the CAN bus. Check the signal against its inverse representation and do a range check; assume 0, if failure.	QM(B)		
TSR02.2	Plausibility-check the brake sensor signal, e.g., gradient. Check the alive-counter for correct sequence and the checksum for message integrity. Force 0, if failure.	B(B)		
TSR03	A Steering angle sensor signal shall be received from the CAN bus.	B	ECU	FSR04
TSR03.1	The Steering angle sensor signal and its inverse representation shall be received from the CAN bus. Check the signal against its inverse representation and do a range check; assume 0, if failure.	QM(B)		
TSR03.2	Plausibility-check the steering angle sensor signal, e.g., gradient. Check the alive-counter for correct sequence and the checksum for message integrity. Force 0, if failure.	B(B)		
TSR04	An accurate vehicle speed front signal shall be generated.	B	ECU	FSR05

(continued)

Table 3.8: (continued)

Technical Safety Requirement				**Func. Safety Req.**
ID	**Description**	**ASIL**	**Allocated to Element**	**ID**
TSR04.1	Generate vehicle speed front signal by averaging two sensor signals.	QM(B)		
TSR04.1.1	There shall be a vehicle speed front sensor1.	QM(B)		
TSR04.1.2	There shall be a vehicle speed front sensor2.	QM(B)		
TSR04.2	The vehicle speed front sensors shall be checked for plausibility, e.g., gradient. Set "vehicle speed front failure" flag if failure, else reset failure flag if correct signals for 10 consecutive times.	B(B)		
TSR05	An accurate vehicle speed rear signal shall be generated.	B	ECU	FSR05
TSR05.1	Generate vehicle speed rear signal by averaging two sensor signals.	QM(B)		
TSR05.1.1	There shall be a vehicle speed rear sensor1.	QM(B)		
TSR05.1.2	There shall be a vehicle speed rear sensor2.	QM(B)		
TSR05.2	The vehicle speed rear sensors shall be checked for plausibility, e.g., gradient. Set "vehicle speed rear failure" flag if failure, else reset failure flag if correct signals for 10 consecutive times.	B(B)		
TSR06	An accurate torque command shall be generated based on throttle, brake, vehicle speed, and battery health status.	B	ECU	FSR06, FSR10

(continued)

Table 3.8: (continued)

Technical Safety Requirement				**Func. Safety Req.**
ID	**Description**	**ASIL**	**Allocated to Element**	**ID**
TSR06.1	Generate torque command. Take "vehicle speed failure" flags into account. Limit gradient to TORQUE_GRADIENT_LIMIT. Range-check output signal. Send torque command via CAN.	QM(B)		
TSR06.2	Torque command shall be plausibility-checked against torque result. Force 0, if failure.	B(B)		
TSR07	A torque result signal shall be received from the CAN bus.	B	ECU	FSR09
TSR07.1	The torque result signal and its inverse representation shall be received from the CAN bus. Check the signal against its inverse representation and do a range check.	QM(B)		
TSR07.2	Plausibility-check the torque result signal. Check the alive-counter for message freshness and the checksum for message integrity. Set "torque result failure" flag, if failure.	B(B)		
TSR08	A battery status shall be received from the CAN bus and reacted upon in case it is out of the safe operating area.	B	ECU	FSR12
TSR08.1	The battery status shall be received from the CAN bus and reacted upon in case it is out of the safe operating area.	QM(B)		

(continued)

Table 3.8: (continued)

Technical Safety Requirement				**Func. Safety Req.**
ID	**Description**	**ASIL**	**Allocated to Element**	**ID**
TSR08.1.1	The battery status and its inverse representation shall be received from the CAN bus. Check the signal against its inverse representation and do a range check, assume battery status is out of safe operating area if checks fail.	QM(B)		
TSR08.1.2	If battery status indicates "out of safe operating area", command zero torque from inverters within TIME_FTTI_THROTTLE (100ms) and switch off power to the inverters within TIME_FTTI_BATT (1s).	QM(B)		
TSR08.2	Plausibility-check the battery status. Check the alive-counter for correct sequence and the checksum for message integrity.	B(B)		
TSR09	There shall be a reliable HV contactor control.	B	ECU	FSR12
TSR09.1	There shall be a HV contactor control for 3 contactors.	QM(B)		
TSR09.1.1	There shall be a contactor control for HV plus contactor.	QM(B)		
TSR09.1.2	There shall be a contactor control for HV negative contactor.	QM(B)		
TSR09.1.3	There shall be a contactor control for HV pre-charge contactor.	QM(B)		

(continued)

Table 3.8: (continued)

Technical Safety Requirement				**Func. Safety Req.**
ID	**Description**	**ASIL**	**Allocated to Element**	**ID**
TSR09.2	Plausibility-check the contactor output stages. Compare voltage and current status feedback against contactor control status. Switch off all contactors in case of detected discrepancy.	B(B)		
TSR10	Control flow monitoring. In case of failure, command zero torque from inverters within TIME_FTTI_THROTTLE (100ms) and switch off power to the inverters within TIME_FTTI_BATT (1s).	B	ECU	FSR02, FSR04, FSR05, FSR06, FSR09, FSR10, FSR12
TSR11	External monitoring facility. In case of failure, command zero torque from inverters within TIME_FTTI_THROTTLE (100ms) and switch off power to the inverters within TIME_FTTI_BATT (1s).	B	ECU	FSR02, FSR04, FSR05, FSR06, FSR09, FSR10, FSR12
TSR12	Memory Check. In case of failure command zero torque from inverters within TIME_FTTI_THROTTLE (100ms) and switch off power to the inverters within TIME_FTTI_BATT (1s).	B	ECU	FSR02, FSR04, FSR05, FSR06, FSR09, FSR10, FSR12
TSR13	CPU Check. In case of failure command zero torque from inverters within TIME_FTTI_THROTTLE (100ms) and switch off power to the inverters within TIME_FTTI_BATT (1s).	B	ECU	FSR02, FSR04, FSR05, FSR06, FSR09, FSR10, FSR12

(continued)

Table 3.8: (continued)

Technical Safety Requirement				Func. Safety Req.
ID	**Description**	**ASIL**	**Allocated to Element**	**ID**
TSR14	Voltage Checks. In case of failure command zero torque from inverters within TIME_FTTI_THROTTLE (100ms) and switch off power to the inverters within TIME_FTTI_BATT (1s).	B	ECU	FSR02, FSR04, FSR05, FSR06, FSR09, FSR10, FSR12

3.5.2 System Architecture Design

From the technical safety requirements in Table 3.8, it can be seen that these requirements can be mapped to the three layers of the "E-Gas Architecture and Safety Concept", briefly described in Section 2.5. The chosen system architecture is shown in Figure 3.6.

For example, the technical safety requirement TSR01.1 maps to level 1 (Functional Level), TSR01.2 to level 2 (Function Monitoring Level) and TSR10 to level 3 (Controller Monitoring Level), and so on.

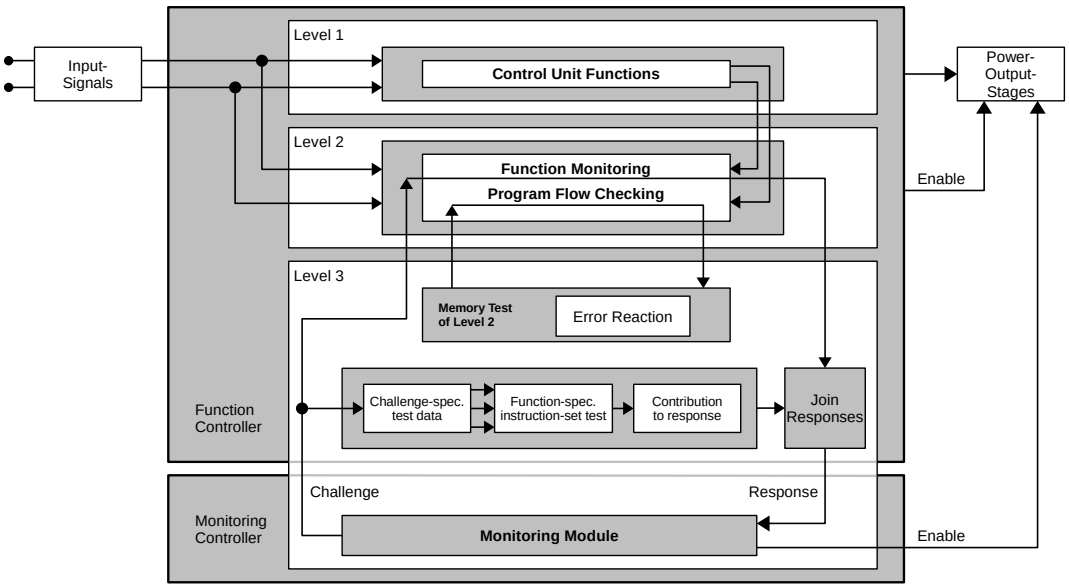


Figure 3.6: Chosen system architecture (translated from [37]).

3.6 SEooC Product Development at Hardware Level

3.6.1 Hardware Architectural Design

At the system level, in Subsection 3.5.2, the "E-Gas Architecture and Safety Concept" was chosen. It builds upon the "Asymmetric Processor Architecture" hardware architecture, described in Section 2.4 and depicted in Figure 3.7.

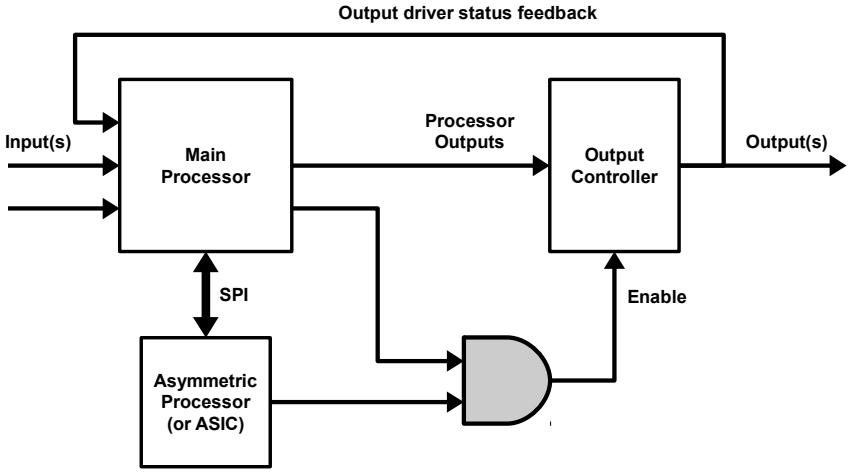


Figure 3.7: The chosen asymmetric processor architecture.

Now a main and an asymmetric processor need to be selected and the output controller needs to be designed according to the requirements established in the previous sections.

3.6.2 Main Processor and Asymmetric Processor

During an earlier evaluation it was found that, in terms of performance, our ECU should be comparable to the TTTech HY-TTC 90,[14] which uses a 16-bit microcontroller running at 80 MHz. It was therefore decided to use a similar microcontroller. Infineon Technologies AG was selected as the supplier for automotive-grade integrated circuits.

For safety-critical systems using a 16-bit microcontroller, Infineon offers a solution which consists of a microcontroller from the XC2000 safety train (XC2300 series),[15] the safety companion chip CIC61508,[16] and supporting safety software.

A brief overview is given in [92], and the presentations [93] and [94] give some more details about the solutions offered by Infineon. [95] presents possible hardware configurations of 16- and 32-bit microcontrollers in combination with the safety companion chip CIC61508 for Automotive Safety Integrity Levels (ASILs) up to ASIL-B(D).

[14] http://www.tttech.com/products/automotive/electronic-control-units/general-purpose-ecus/hy-ttc-90/

[15] http://www.infineon.com/xc2300

[16] http://www.infineon.com/cic61508

Main Processor

As the main processor, we chose the XC2387A microcontroller from the Infineon XC2000 safety train. Its main safety-related features, according to [96], are:

- High-performance CPU with five-stage pipeline and MPU (up to 80MHz)
- Hardware CRC-checker with programmable polynomial to supervise on-chip memory areas
- On-chip memory modules: SRAM, DPRAM, Flash with memory content protection through Error Correction Code (ECC)
- Programmable watchdog timer and oscillator watchdog

The multi-voltage processor power supply TLE6368G2 was chosen for the main processor and all peripherals, including the external sensors. Its most important features related to safety are [97]:

- Power-on reset functionality
- SPI-triggered window watchdog
- Six independent voltage trackers (followers)
- Tracker control and diagnosis by SPI

The window watchdog feature covers the technical safety requirement TSR11, listed in Subsection 3.5.1, “Technical Safety Requirements” on page 46.

Asymmetric Processor

The Infineon solution uses the CIC61508 as a safety supervisor companion chip. Its main safety-related features, according to [98], are:

- Power supply monitor for over- and under-voltage
- Sequencer
- Task monitor
- Data comparison and verification functions
- SPI communication monitor
- Safety path control (enable/disable)
- Configurable wake-up timer

The power supply of the CIC61508 is provided by an extra linear voltage regulator and is independent from the main processor’s power supply, to avoid Common Cause Failures (CCFs). The TLE4274DV33 has been chosen as voltage regulator for the CIC61508.

For greater accuracy of the power supply monitoring, the precision 2.5V bandgap voltage reference REF192 is used as the ADC voltage reference. The CIC61508 monitors the +5V (VDD PA), the +3.3V (VDD PB), and the +1.5V (VDD Core) supply rails of the main processor.

3.6.3 Output Controller

As per the asymmetric processor architecture, see Figure 3.7, both processors need to assert their respective enable signals for the safety-related output driver stages to be eventually enabled. To adhere to the FSE's "Rule 7.17 Safety Circuit" [81], see Appendix D on page 112, there will be a third enable signal from the interlock circuit. To fulfill the technical safety requirement TSR09.2, each contactor driver stage is equipped with a voltage and current status feedback signal.

To be able to argue for a reliable HV system shutdown, in anticipation of future FSE rule changes, a two-failure safe design is employed in designing the contactor drivers' shutdown paths. The two-failure safe design stems from the requirements set forth in the European Directive 94/9/EC.[17]

The final design of the output controller is shown in Figure 3.8.

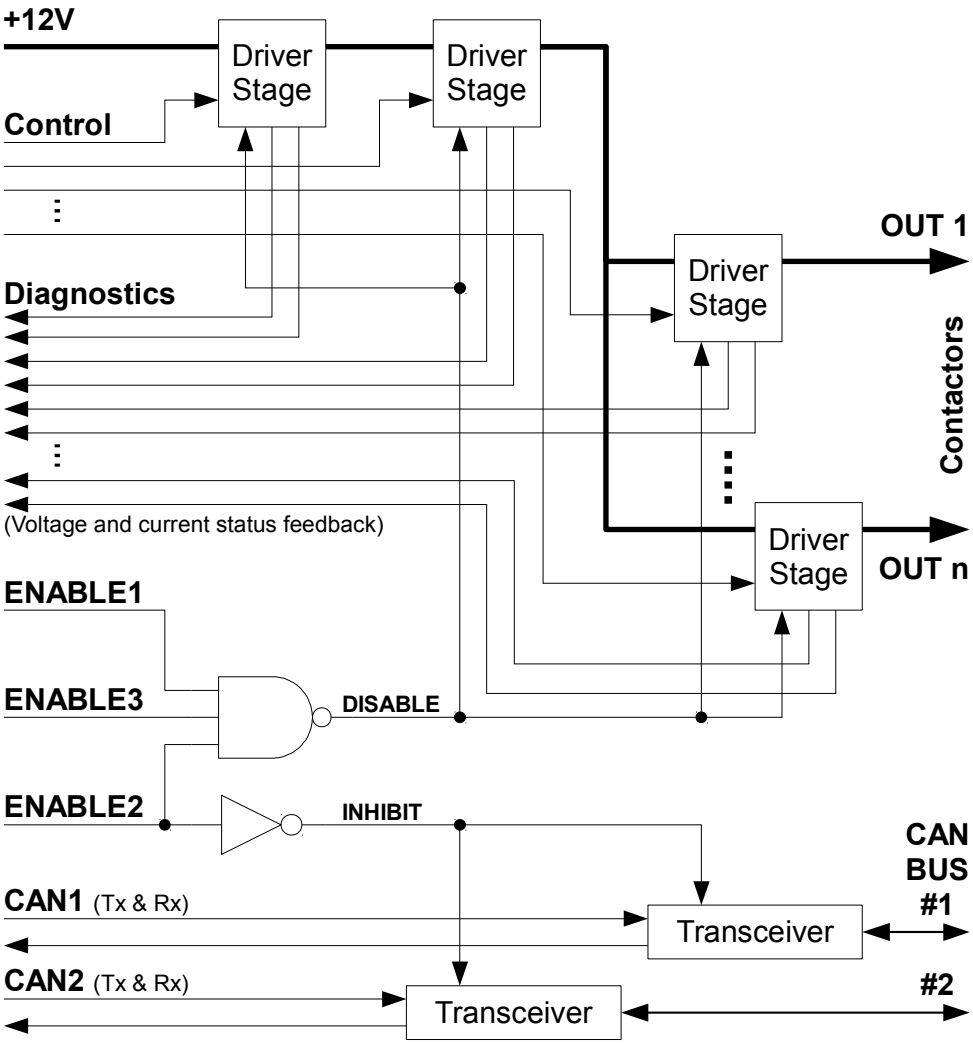


Figure 3.8: Safety output controller with diagnostics.

[17] http://ec.europa.eu/enterprise/sectors/mechanical/documents/legislation/atex/

The origins of the three enable signals shown in the figure are as follows:

- ENABLE1 is the enable signal from the main processor.
- ENABLE2 is the enable signal from the asymmetric processor.
- ENABLE3 is the enable signal from the interlock circuit.

A single contactor driver stage using the smart high-side power switch BTS6163D [99], is shown in Figure 3.9. Its diagnostic feedback feature is used to give feedback on the actual load current situation. A resistive divider is used to give voltage feedback used to diagnose the switching behavior of the high-side switch. Both features are necessary to fulfill the technical safety requirement TSR09.2 and both are connected to the main processors ADC.

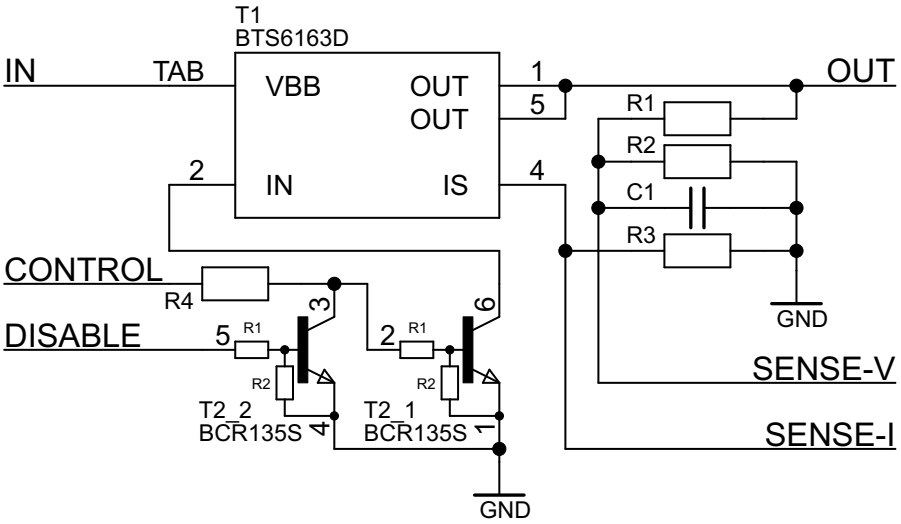


Figure 3.9: One contactor driver stage with diagnostics.

The chosen CAN transceiver is the Infineon TLE6250GV33 [100]. For termination purposes, a CAN split termination can optionally be populated for each CAN bus on the final PCB.

3.7 A Formula Student Electric Safety ECU Platform

The ECU electrical requirements are summarized in Table 3.9. These are a collection of the requirements of the Formula SAE [80] and the FSE rules [81], see Appendix D, and the general requirements of the setup chosen by the TU Graz Racing Team. This setup forms the basis for the SEooC development process outlined in Section 3.3 to Section 3.6.

Table 3.9: ECU electrical requirements.

Feature	**Qty**	**Specification**
Power Supplies	3	ECU, Contactors, AUX
Supply Voltage Range		$9...16V$
Over-/Reverse-Voltage Protection		$\pm 30V$
High-Side Drive HV Contactors	6	each max. 2A/pulse 5A
High-Side Drive Discharge Disable	1	each max. 0.2A/pulse 1A
High-Side Drive Auxiliary, high power, PWM	4	each max. 10A/pulse 20A
High-Side Drive Auxiliary, low power	2	each max. 2A/pulse 5A
Interlock Inputs	2	source from +supply
BMS-OK Input from BMS	1	$\overline{BMS-OK}$ latched
IMD-OK Input from IMD	1	$\overline{IMD-OK}$ latched
Reset Button	1	resets BMS/IMD-OK latches
Reset Button LED	1	40mA current source
Wheel Speed Sensor Inputs	4	digital/PWM
Wheel Speed Sensor Supplies, w/ diagnostics	4	each 5V/16 mA
Wheel Travel Sensor Inputs	4	analog ratiometric
Wheel Travel Sensor Supplies, w/ diagnostics	2	each 5V/16 mA
High-Speed CAN Interfaces	2–3	max. 1000kbps, term.
(opt.) LIN Interfaces	2	max. 20kbps, term.
(opt.) WIFI Extension	1	differential SPI
(opt.) MicroSD Card Slot (data logging)	1	MicroSD in SPI mode
(opt.) Real-Time Clock (RTC)	1	for logging timestamps

With the support for up to six contactors, up to two accumulator containers can be supported, see FSE "Rule 7.23 Accumulator Insulation Relay(s) (AIR)", see Appendix D on page 114. This provides a higher flexibility in the design of the energy storage of the race car.

The first interlock input will usually encompass the interlock demanded by FSE "Rule 7.17 Safety Circuit", see Appendix D on page 112. The additional interlock input allows the partitioning of the interlock system. It can be used to, e.g., guard additional covers

of the HV system, which are not covered by the rules. The BMS-OK signal allows the Battery Management System (BMS), enclosed in the accumulator container, to directly shut down the HV system.

Two high-speed Controller Area Network (CAN) bus interfaces allow for communication with external devices, such as dashboard unit, drive-by-wire system, inverters, BMS, etc. Optionally, a third high-speed CAN bus interface or two Local Interconnect Network (LIN) bus interfaces can be populated to support additional bus configurations.

An Insulation Monitoring Device (IMD) of type IR155-3203 [101] provides the signals IMD-OK and IMD-M. The IMD-OK signal directly shuts down the HV system, should the IMD trip at the factory-set trip point. Via the pulse-width-modulated signal IMD-M, the ECU is able to read the measured insulation resistance and set an internal trip point above the IMD factory-set trip point.

An optional differential Serial Peripheral Interface (SPI) allows for the connection of, e.g., a WiFi/WLAN extension module. An optional MicroSD card slot and a Real-Time Clock (RTC) allow for the implementation of data logging.

Figure 3.10 shows the inputs and outputs of the resulting ECU, and Figure 3.11 the corresponding block diagram.

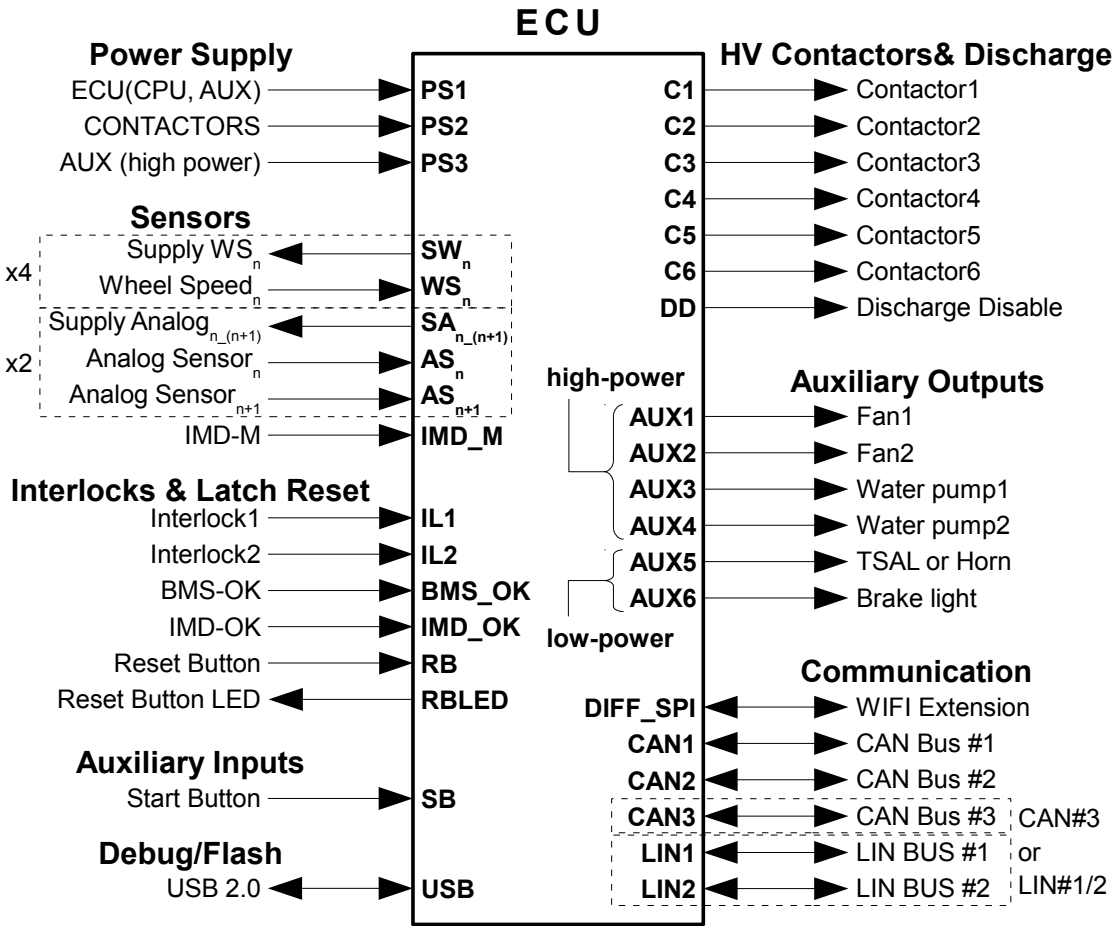


Figure 3.10: Inputs and outputs of the FSE Safety ECU Platform.

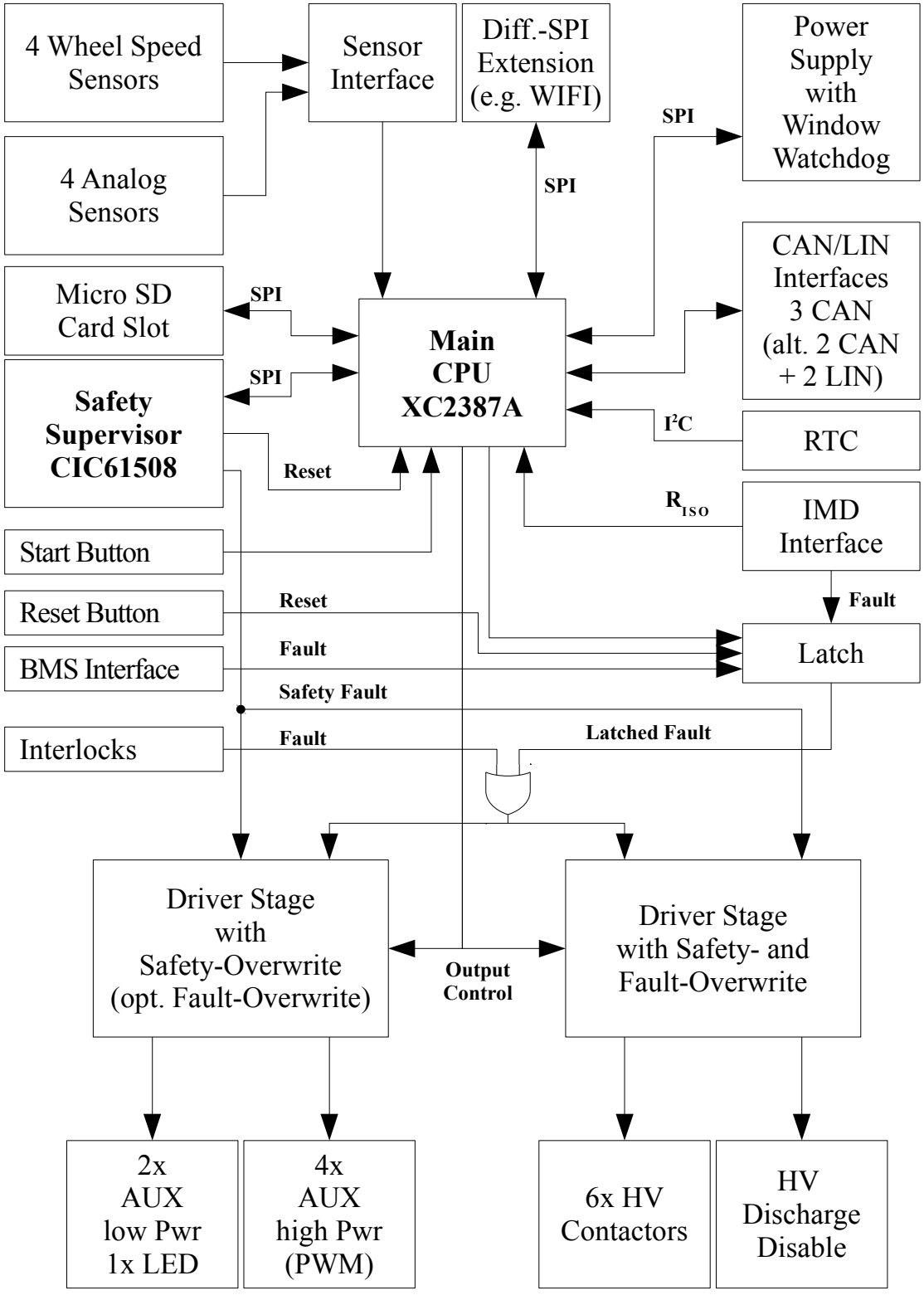


Figure 3.11: Block diagram of the FSE Safety ECU Platform.

3.8 Porting FreeRTOS to the Infineon C166S v2 Core

The microcontroller selected during design is the Infineon XC2387A microcontroller [96]. It is based on the Infineon C166S v2 Core, hence we will discuss the details of porting FreeRTOS to this core in the following sections.

Before porting FreeRTOS to the Infineon C166S v2 Core, one should be familiar with the following documents:

- C166S v2 CPU Core User Manual [102], see Appendix E on page 116
- Tasking C-Compiler User Guide [103], see Appendix F on page 120
- XC2300A Derivatives User's Manual [104]
- FreeRTOS Porting Guide [105]

3.8.1 Using the C166S V2 Architecture

In this subsection we will look at how the stack on the C166S V2 architecture is used by the compiler, which stack pointers are used, and how the stack is placed in the controller's memory. We will also look at the memory model used, how parameters are passed, and what is and is not part of the context saved by the invocation of an ISR.

Stack Usage The Tasking C-Compiler suite actually uses two stacks on the C166 architecture: One where the "usual" call return addresses and registers are pushed, and another one where local variables are stored.

This stems from the fact that the C166 architecture does not support system stack operations other than *PUSH* and *POP*. Therefore, a compiler has to work around this deficiency of the C166 CPU core. The first stack is called the *system stack* and the other one the *user stack*, both growing from top to bottom.

The core's system stack is documented in [102] and in Appendix E.2 on page 116. The Tasking C-Compiler suite's stack usage is documented in [103] and in Appendix F.2.2 on page 122.

Stack Pointers Because the compiler uses two stacks, but the CPU core only supports one *system stack pointer*, the compiler uses the general purpose register *R15* as *user stack pointer*.

This also means that *near addressing* is employed. Near addressing uses one of the four Data Page Pointers (DPPs), see [102, Chapter 2.5.2 "Long and Indirect Addressing Modes"]. Which one is actually used for user stack addressing, is determined at linking time.

The system stack is addressed by the Stack Pointer (SP) and the Stack Pointer Segment Register (SPSG).

Currently, FreeRTOS only handles the system stack pointer in the task's Task Control Block (TCB). As a direct consequence, the FreeRTOS core has to be taught about the second user stack. More on that in Subsection 3.8.2.

System Stack The system stack's placement in memory is configured in the *project.lsl* file (full Listing G.1 on page 125). Line number 4 in Listing 3.1 shows the relevant code line where the system stack is placed into the DPRAM memory.

```
1 // Define the system stack
2 section_layout ::shuge (direction = high_to_low)
3 {
4     group ( run_addr = [0xF600..0xFC00], ordered ) stack
          "system_stack" ( size = 256 );
5 }
```

Listing 3.1: Configuring the System Stack in *project.lsl*.

Data Page Pointers The Data Page Pointers (DPPs) DPP0 to DPP3 are also configured in the *project.lsl* file (full Listing G.1 on page 125). The relevant code lines which configure the DPP registers are shown in Listing 3.2. The DPP register configurations are kept at their default values as created by the Tasking VX IDE when executing *File* → *New* → *Linker Script File (LSL)*.

```
1 #define __DPP0_ADDR 0xC00000    /* [0xC00000..0xC03FFF] FLASH0 (Vector
      Table) */
2 #define __DPP1_ADDR 0xE00000    /* [0xE00000..0x003FFF] PSRAM */
3 #define __DPP2_ADDR 0x008000    /* [0x008000..0x00BFFF] DSRAM */
4 #define __DPP3_ADDR 0x00C000    /* [0x00C000..0x00FFFF] DSRAM, XSFR,
      ESFR, DPRAM, SFR */
```

Listing 3.2: Configuring the DPPs in *project.lsl*.

User Stack To force the compiler to use DPP1 for *user stack* addressing, the user stack is placed into the address range covered by DPP1 in the *project.lsl* file - see line number 4 in Listing 3.3. This placement has to be aligned at 16-bit boundaries.

```
1 // Define the user stack (force linker to use DPP1 for user stack)
2 section_layout ::near
3 {
4     group( run_addr = [0xE00002..0xE00100], ordered ) stack
          "user_stack" ( size = 254 );
5 }
```

Listing 3.3: Configuring the user stack in *project.lsl*.

Note: The user stack is not placed at the very beginning of that range because the Tasking User Guide [103, Section 8.7.4.] states that, "By default the near space is 'paged' in pages of 16 kB. The first byte in each space is reserved to avoid NULL pointer comparison problems with objects allocated at the beginning of the page."

Compiler Memory Model From the supported memory models, see Appendix F.1 on page 120, the *Huge Memory Model* needs to be used in order to get consistent pointers for the whole project. In this memory model, pointers are 32 bits wide.

Parameter Passing FreeRTOS specifies that each task must conform to the following task prototype - see Listing 3.4.

```
1 #define portTASK_FUNCTION( vFunction, pvParameters ) void vFunction(
    void *pvParameters )
```

Listing 3.4: FreeRTOS task prototype.

The task's parameter **pvParameters* needs to be stored in the task's initial context, so that it gets restored in the correct registers upon first activation. For details about the Tasking C-Compiler calling conventions for parameter passing see Appendix F.2.1 on page 121.

When passing a pointer, e.g., **pvParameters*, as the first parameter, we can see from Table F.2, that it is passed using the registers *R2* and *R3*. During initialization of the task's context on the stack later on, this fact will need to be considered.

Context saved by an ISR In FreeRTOS a context switch usually occurs during the so called tick ISR. When the CPU invokes an ISR in segmented mode, which is the default, it saves the Program Status Word (PSW), the Code Segment Pointer (CSP), and the Instruction Pointer (IP), in that very order, on the *system stack* - see Appendix E.6 on page 119.

3.8.2 The Task Stack(s)

Extending FreeRTOS to handle two Stacks for each Task As the compiler operates with two stacks, FreeRTOS had to be extended to handle the additional second stack. All the kernel API functions were supplemented with support for two stacks, like the function *xTaskGenericCreate()* shown in Listing 3.5.

The API was made dependent on the value of *portSTACK2*: To enable a two-stack enabled API, like in this port of FreeRTOS to the Infineon C166S v2 Core, *portSTACK2* has to be defined to *1* in *portmacro.h* - see Listing G.2 in Appendix G on page 126.

The Task Control Block (TCB) had to be extended by the essential fields for handling the second stack as well. Those additional fields were also made dependent on the value of *portSTACK2*.

Listing 3.5: Example of a modification to the FreeRTOS API to support a second stack.

```
1 #if ( portSTACK2 == 1 )
2 signed portBASE_TYPE xTaskGenericCreate( pdTASK_CODE pxTaskCode, const
    signed char * const pcName, unsigned short usStackDepth, unsigned
    short usStackDepth2, void *pvParameters, unsigned portBASE_TYPE
    uxPriority, xTaskHandle *pxCreatedTask, portSTACK_TYPE
    *puxStackBuffer, portSTACK_TYPE *puxStackBuffer2, const
    xMemoryRegion * const xRegions )
3 #else
4 signed portBASE_TYPE xTaskGenericCreate( pdTASK_CODE pxTaskCode, const
    signed char * const pcName, unsigned short usStackDepth, void
    *pvParameters, unsigned portBASE_TYPE uxPriority, xTaskHandle
    *pxCreatedTask, portSTACK_TYPE *puxStackBuffer, const
    xMemoryRegion * const xRegions )
5 #endif
```

Listing 3.5: Example of a modification to the FreeRTOS API to support a second stack.

Note: Further work would be necessary on a more generic handling of multiple stacks without changing the current API of FreeRTOS.

Task Context, Stack Layout and Stack Creation A task's context, saved and restored on the system stack, is depicted as stack layout in Figure 3.12. It consists of

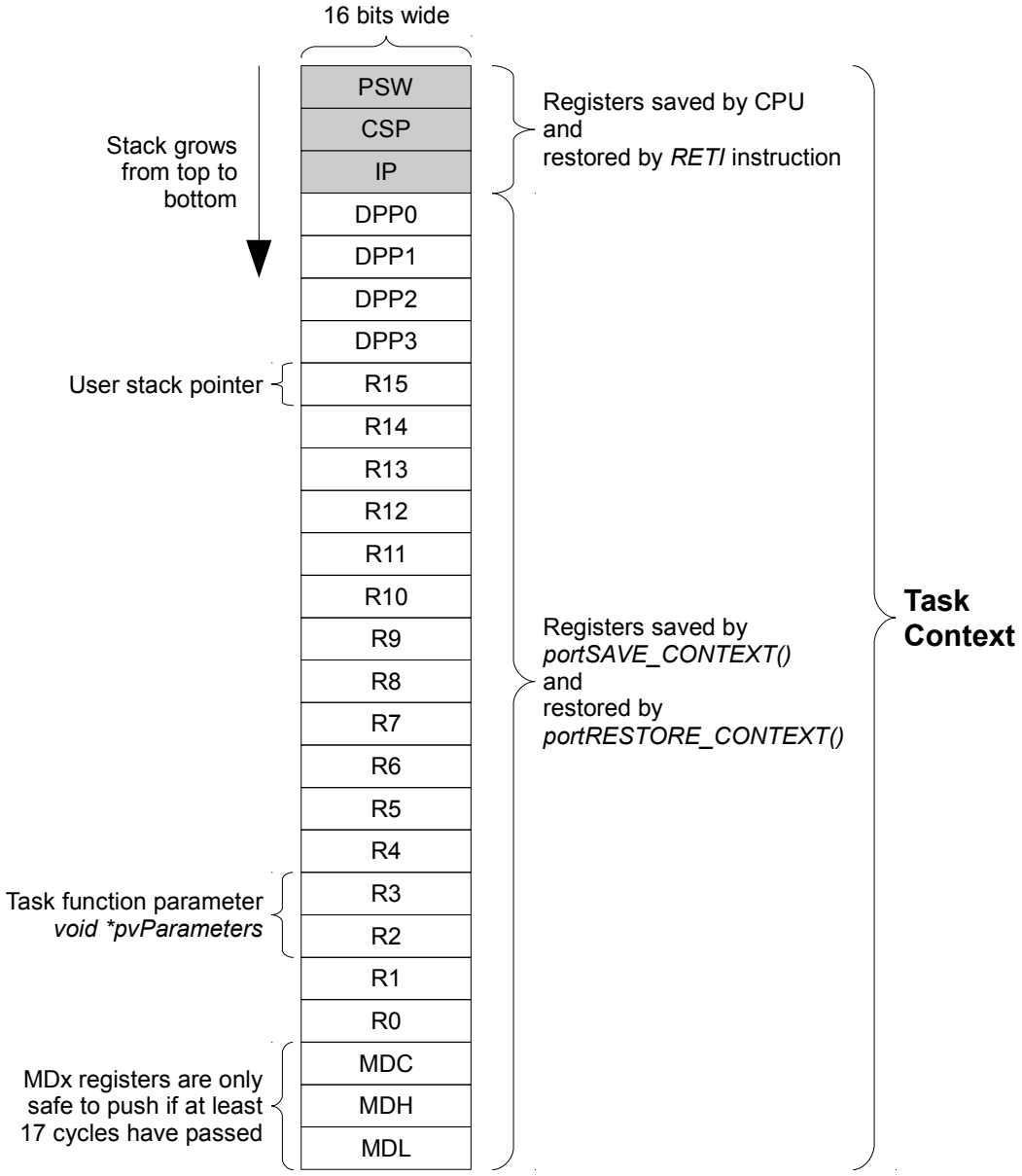


Figure 3.12: The stack layout for FreeRTOS on the Infineon C166S V2 architecture.

the Program Status Word (PSW), the Code Segment Pointer (CSP), the Instruction Pointer (IP), the DPP registers, the general purpose registers *R0* to *R15*, and the multiply divide unit registers *MDL*, *MDH*, and *MDC*. To not stall the CPU execution, the registers of the multiply divide unit are only safe to store when at least 17 CPU cycles have passed - see Appendix E.3 on page 117.

Note, that the state of the additional MAC unit, accessed via the new CoXXX arithmetic instructions [103, Chapter 1.12.5. "Intrinsic Functions"], is not part of a task's context. As a direct consequence, only one single task is allowed to make use of these instructions in this port of FreeRTOS.

The initialization of a task's context on the stack is done in the function *pxPortInitialiseStack()* in the file *port.c*. The relevant code lines are shown in Listing 3.6.

Listing 3.6: The initial stack layout is created by *pxPortInitialiseStack()*

```
1  portSTACK_TYPE *pxPortInitialiseStack( portSTACK_TYPE *pxTopOfStack,
       portSTACK_TYPE *pxTopOfStack2, pdTASK_CODE pxCode, void
       *pvParameters )
2  {
3  __PSW_type initialPSW;
4
5      // PSW - processor status word
6      initialPSW.U = 0;       // reset initialization value
7      initialPSW.B.ien = 1;   // enable interrupts
8      portPushToStack(pxTopOfStack, initialPSW.U);
9      // CSP - code segment pointer
10     portPushToStack(pxTopOfStack, __seg(pxCode));
11     // IP - instruction pointer
12     portPushToStack(pxTopOfStack, __sof(pxCode));
13
14     // initialize registers DPP0 to DPP3
15     // DPP1 is used to access the user stack (stack2)
16     // see the project LSL file how this is accomplished
17     portPushToStack(pxTopOfStack, DPP0);
18     portPushToStack(pxTopOfStack, __pag(pxTopOfStack2));
19     portPushToStack(pxTopOfStack, DPP2);
20     portPushToStack(pxTopOfStack, DPP3);
21
22     // initialize general purpose register R15
23     // as user stack pointer (stack2), per TASKING
24     // C-compiler calling convention, use DPP1
25     portPushToStack(pxTopOfStack, __dpof(1, pxTopOfStack2));
26
27     // initialize general purpose registers R14 to R4
28     portPushToStackCnt(pxTopOfStack, 0x00, 11);
29
30     // initialize general purpose registers R3 and R2 as
31     // input parameter pvParameters
32     // ATTENTION: we assume the huge memory model here!
33     portPushToStack(pxTopOfStack, __seg(pvParameters));
34     portPushToStack(pxTopOfStack, __sof(pvParameters));
35
```

```
36      // initialize registers R1 and R0
37      portPushToStackCnt(pxTopOfStack, 0x00, 2);
38
39      // initialize multiply/divide unit registers MDC, MDH and MDL
40      portPushToStackCnt(pxTopOfStack, 0x00, 3);
41
42      return pxTopOfStack;
43  }
```

Listing 3.6: The initial stack layout is created by *pxPortInitialiseStack()*

Note: The macro *portPushToStack()* is defined in the file *portmacro.h* - see Listing G.2 in Appendix G on page 126.

3.8.3 Task Context Switching Primitives

The definition of a task context requires context switching to be done inside an Interrupt Service Routine (ISR). The saving of the suspended task's CPU status is done by the CPU when calling this ISR. The assembler primitive *RETI* at the end of ISRs takes care of restoring the respective CPU status.

The primitives *portSAVE_CONTEXT()* and *portRESTORE_CONTEXT()* only need to take care of saving and restoring the general purpose registers *R0* to *R15*, the Data Page Pointers (DPPs) *DPP0* to *DPP3*, and the multiply divide unit registers *MDL*, *MDH*, and *MDC*. The stack pointers in the task's TCB and the CPU system stack pointer have to be updated or saved as well.

Listing 3.7 and Listing 3.8 show the respective primitives. A full listing of *port.c* is provided in Appendix G.2.2 on page 131.

Listing 3.7: The task context switching primitive *portSAVE_CONTEXT()*.

```
1  void __always_inline__ portSAVE_CONTEXT( void )
2  {
3  register portSTACK_TYPE * pxTopOfStack;
4      __asm ( "\n"
5              " push DPP0 \n"
6              " push DPP1 \n"
7              " push DPP2 \n"
8              " push DPP3 \n"
9              " push r15 \n"
10             " push r14 \n"
11             " push r13 \n"
12             " push r12 \n"
13             " push r11 \n"
14             " push r10 \n"
15             " push r9 \n"
16             " push r8 \n"
17             " push r7 \n"
18             " push r6 \n"
19             " push r5 \n"
20             " push r4 \n"
21             " push r3 \n"
```

```
22              " push r2 \n"
23              " push r1 \n"
24              " push r0 \n"
25              " push MDC \n"      // MDx registers are only safe to push
26              " push MDH \n"      // if at least 17 cycles have passed
27              " push MDL \n" );
28
29      // system stack pointer
30      pxTopOfStack = __mkhp(SP, SPSEG);
31      pxCurrentTCB->pxTopOfStack  = pxTopOfStack;
32      // user stack pointer
33      pxTopOfStack = __mkfp((unsigned int)__getsp(), DPP1);
34      pxCurrentTCB->pxTopOfStack2 = pxTopOfStack;
35  }
```

Listing 3.7: The task context switching primitive *portSAVE_CONTEXT()*.

Listing 3.8: The task context switching primitive *portRESTORE_CONTEXT()*.

```
1  void __always_inline__ portRESTORE_CONTEXT( void )
2  {
3  register portSTACK_TYPE * pxTopOfStack;
4
5       pxTopOfStack = (portSTACK_TYPE *)pxCurrentTCB->pxTopOfStack;
6
7       // let CPU stack pointer point to stack of task to be restored
            (write atomically)
8       __atomic(2);
9       SP    = __sof(pxTopOfStack);
10      SPSEG = __seg(pxTopOfStack);
11      __endatomic();              // used as fence for the compiler
12
13      // restore registers saved by portSAVE_CONTEXT
14      __asm ( "\n"
15              " pop MDL \n"
16              " pop MDH \n"
17              " pop MDC \n"
18              " pop r0 \n"
19              " pop r1 \n"
20              " pop r2 \n"
21              " pop r3 \n"
22              " pop r4 \n"
23              " pop r5 \n"
24              " pop r6 \n"
25              " pop r7 \n"
26              " pop r8 \n"
27              " pop r9 \n"
28              " pop r10 \n"
29              " pop r11 \n"
30              " pop r12 \n"
31              " pop r13 \n"
32              " pop r14 \n"
33              " atomic #4 \n" // atomically restore user stack pointer
```

```
34                "  pop r15 \n"
35                "  pop DPP3 \n"
36                "  pop DPP2 \n"
37                "  pop DPP1 \n"
38                "  pop DPP0 \n"
39                "\n" );
40  }
```

Listing 3.8: The task context switching primitive *portRESTORE_CONTEXT()*.

Note: As these primitives are only used in *port.c*, they are defined as inline functions there, and not as macros in *portmacro.h*. This facilitates debugging of those functions.

Note: Both functions are not allowed to use local variables without the *register* qualifier! This is because these functions cannot use the user stack, where local variables would be stored, as it is just being changed by them.

3.8.4 Interrupts, Interrupt Nesting, and Critical Section Management

Interrupts and Interrupt Nesting Interrupts with a priority level equal to or below the *Max. SysCall Priority* are allowed to use a only limited set of FreeRTOS API functions, and they can nest, i.e., a higher priority interrupt will suspend the execution of a lower priority interrupt. Interrupts at priority levels above *Max. SysCall Priority* can handle critical hardware interrupts, but are not allowed to use the FreeRTOS API.

The interrupt priority levels on the C166S V2 architecture, and their relation to FreeRTOS, are shown in Figure 3.13. *Max. SysCall Priority* has been set to a level of 11 to allow

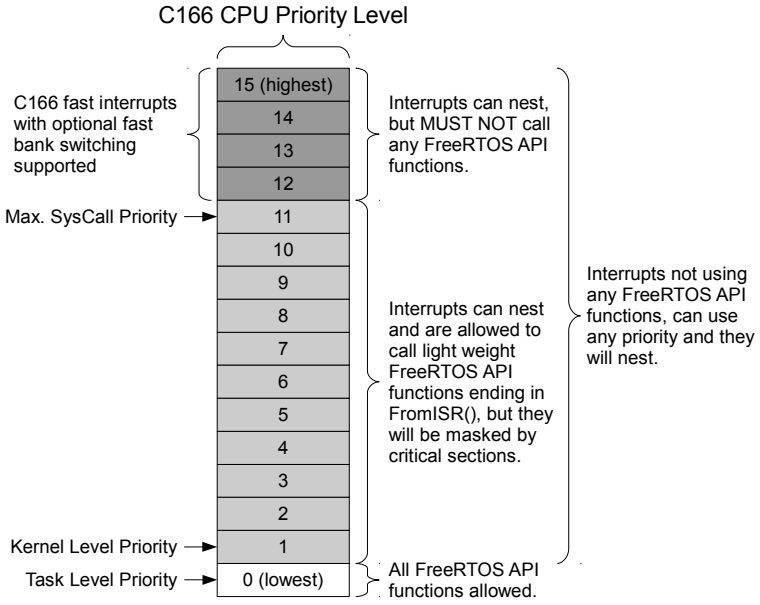


Figure 3.13: FreeRTOS interrupt nesting on the Infineon C166S V2 architecture.

critical hardware interrupts to utilize the fast interrupt and/or bank switching feature of the C166S V2 architecture when using an interrupt level of 12 and upward. The levels are defined in *FreeRTOSConfig.h* - see Listing G.4 in Appendix G on page 138 - and can be adapted for a given project, if need be.

Critical Section Management When a a critical section of of user code is entered by calling *taskENTER_CRITICAL()*, all interrupts up to the *Max. SysCall Priority* are disabled until the section is exited again by calling *portEXIT_CRITICAL()*.

As FreeRTOS supports the nesting of critical sections, this means that interrupts are only re-enabled again, when the last nested critical section is exited by calling *portEXIT_CRITICAL()*.

```
1 #define portENTER_CRITICAL()        vPortEnterCritical()
2 #define portEXIT_CRITICAL()         vPortExitCritical()
```

Listing 3.9: Critical section management macros.

Note: Interrupts above *Max. SysCall Priority* are not disabled. If it is important to disable those as well, *portDISABLE_INTERRUPTS()* and *portENABLE_INTERRUPTS()* have to be called.

Note: An interrupt-safe alternative for use in ISRs is to call *portSET_INTERRUPT_MASK_FROM_ISR()* and *portCLEAR_INTERRUPT_MASK_FROM_ISR()*.

The implementation of *taskENTER_CRITICAL()* as inline function *vPortEnterCritical()* is shown in Listing 3.10.

```
1  inline void vPortEnterCritical( void )
2  {
3      if (uxCriticalNesting == 0)
4      {
5          /* Disable all interrupts under RTOS control */
6          portSET_INTERRUPT_MASK_FROM_ISR();
7      }
8
9      /*
10      * Now interrupts are disabled ulCriticalNesting can be accessed
11      * directly.  Increment ulCriticalNesting to keep a count of how
            many
12      * times portENTER_CRITICAL() has been called.
13      */
14     uxCriticalNesting++;
15 }
```

Listing 3.10: The critical section management function *vPortEnterCritical()*.

The implementation of *portEXIT_CRITICAL()* as inline function *vPortExitCritical()* is shown in Listing 3.11.

Listing 3.11: The critical section management function *vPortExitCritical()*.

```
1 inline void vPortExitCritical( void )
```

```
{
    if (uxCriticalNesting > 0)
    {
        /* Decrement the nesting count as we are leaving a critical
           section. */
        uxCriticalNesting--;

        /*
         * If the nesting level has reached zero then interrupts
           should be
         * re-enabled.
         */
        if( uxCriticalNesting == 0 )
        {
            portCLEAR_INTERRUPT_MASK_FROM_ISR(
                configTASK_LEVEL_INTERRUPT_PRIORITY);
        }
    }
}
```

Listing 3.11: The critical section management function *vPortExitCritical()*.

3.8.5 Yield Function and System Timer Interrupt

Yield Function To force a context switch, also known as *yield*, two macros are provided: *portYIELD()* and *portYIELD_FROM_ISR()*. The first is to be used within tasks, mostly used when the cooperative scheduler is selected. The second one can be called at the end of ISRs to force a context switch, if resources have become available to a higher-priority task than the one currently suspended. Both macros are shown in Listing 3.12.

```
#if configUSE_PREEMPTION == 0
#define portYIELD()                 __int166( 127 )
#else
#define portYIELD()                 STM_vSTM1Trap()
#endif

#define portYIELD_FROM_ISR(SwitchRequired) \
    { if (SwitchRequired) portYIELD();  }
```

Listing 3.12: The task context switching primitive *portYIELD()*.

In case the non-preemptive or cooperative scheduler is used, the function *portYIELD()* is defined as a software trap function. This way *vPortYield()*, shown in Listing 3.13, is invoked as an interrupt function. The latter is important to get a correct stack layout, as a normal function call would not save the PSW on the task's stack.

In case the preemptive scheduler is used, the function *portYIELD()* is defined as a macro using the low-level STM module driver trap, which essentially sets a hardware flag to force an STM1 interrupt.

Listing 3.13: The yield function *vPortYield()*.

```
#if configUSE_PREEMPTION == 0
```

```
2     __interrupt(TRAP127_VECT) __frame() void vPortYield( void )
3     {
4         /* save current task's context */
5         portSAVE_CONTEXT();
6
7         /* do the context switch */
8         vTaskSwitchContext();
9
10        /* restore new task's context */
11        portRESTORE_CONTEXT();
12    }
13 #endif /* configUSE_PREEMPTION == 0 */
```

Listing 3.13: The yield function *vPortYield()*.

System Timer Interrupt The STM1 interrupt function, shown in Listing 3.14, is called the *FreeRTOS system tick interrupt*, which also acts as the yield function if the preemptive scheduler is used.

FreeRTOS tracks the system time via this interrupt. Task sleep delays and timeouts of blocking API function calls are all based on system ticks. On every tick the scheduler selects the next task with the same or a higher priority level which is in the *ready* state.

```
1  __interrupt(STM1_VECT) __frame() void STM_viSTM1I( void )
2  {
3  register unsigned portBASE_TYPE hw_tick;
4
5      /* save current task's context */
6      portSAVE_CONTEXT();
7
8      /* if set, this is a hardware tick, otherwise a software yield */
9      hw_tick = SCU_DMPMIT_STM1;
10
11     /* Clear interrupt status */
12     STM_vSTM1Clr(hw_tick);
13
14     /* hardware tick, otherwise a software yield */
15     if ( hw_tick )
16     {
17         // increment Tick
18         vTaskIncrementTick();
19     }
20
21     #if configUSE_PREEMPTION == 1
22     vTaskSwitchContext();
23     #endif /* configUSE_PREEMPTION == 1 */
24
25     /* restore new task's context */
26     portRESTORE_CONTEXT();
27 } //  End of function STM_viSTM1I
```

Listing 3.14: The system timer interrupt function *STM_viSTM1I()*.

3.8.6 Starting/Stopping the OS

From the main routine, the FreeRTOS scheduler is started by a call to *vTaskStartScheduler()*, which then calls the hardware-dependent setup function *xPortStartScheduler()*, shown in Listing 3.15. This function in turn calls the setup function of the FreeRTOS tick timer, restores and activates the first task. From this point, the kernel is said to be running.

```
portBASE_TYPE xPortStartScheduler( void )
{
    /*
     * Start the timer that generates the tick ISR at the kernel
     * interrupts priority. Interrupts are disabled here already.
     */
    prvSetupTimerInterrupt(configKERNEL_INTERRUPT_PRIORITY, 0);

    /* Start first task. */
    portSTART_FIRST_TASK();

    /* Should never reach here. */
    return pdFALSE;
}
```

Listing 3.15: The scheduler Start function *xPortStartScheduler()*.

The FreeRTOS system tick interrupt is configured and started by the function *prvSetupTimerInterrupt()*, shown in Listing 3.16. It enables the interrupts from the System Timer Module (STM) and then starts the system timer.

```
void prvSetupTimerInterrupt(unsigned short usILVL, unsigned short
    usXGLVL)
{
    /* STM_vInit() must have been called during hardware setup */

    /* enable STM1 interrupt node */
    STM_vNodeEnable(STM1_NODE, usILVL, usXGLVL);

    /*  enable STM1 interrupt */
    STM_vEnableSTM1();

    /* start the System Timer Module (STM) */
    STM_vStartSTM();
}
```

Listing 3.16: The system timer setup function *prvSetupTimerInterrupt()*.

The function *portSTART_FIRST_TASK()*, shown in Listing 3.17, is defined as a software interrupt, essentially calling *vPortStartFirstTask()*, shown in Listing 3.18, as an interrupt function. This is the only way we can get the compiler to produce a *RETI*[18] instruction. This is important because a normal function call return via *RET* would not restore the PSW from the task's stack.

[18]RETI = return from interrupt

```
#define portSTART_FIRST_TASK()      __int166( 126 )
```

Listing 3.17: The *portSTART_FIRST_TASK()* macro.

The function *vPortStartFirstTask()* saves essential information to be able to stop the FreeRTOS scheduler again later on. It also sets up the C166S V2 core's local register bank 1 to reuse the user stack used during system setup for interrupt functions.

This local register bank can then be used by, e.g., up to two fast interrupts utilizing fast bank switching at the same interrupt level. How this local register bank can be used in application-specific code is shown in Section 3.8.7, "Interrupt Handling".

Because this function is defined as an Interrupt Service Routine (ISR), the compiler generates a RETI instruction at its end. This instruction ensures that interrupts are enabled again, as all tasks' PSWs saved in their respective task context on the stack have interrupts enabled - see Listing 3.6 on page 64.

```
__interrupt(TRAP126_VECT) __frame() void vPortStartFirstTask( void )
{
    /* interrupts are still disabled */

    /* save TopOfStack for system and user stack */
    pxTopOfSystemStack  = __mkhp(SP, SPSEG);
    pxTopOfUserStack    = __mkfp((unsigned int)__getsp(), DPP1);

    /* switch to local register bank 1 */
    __switchregbank(1);

    /* set user stack pointer of local register bank (using DPP1) */
    __setsp(__mknp(1, pxTopOfUserStack));

    /* switch to global register bank */
    __switchregbank(0);

    /* restore context of first task */
    portRESTORE_CONTEXT();

    /* interrupts are enabled by returning to the first task */
}
```

Listing 3.18: The scheduler start function for the first task, *vPortStartFirstTask()*.

When the FreeRTOS kernel is being stopped by a call to *vTaskEndScheduler()*, subsequently the function *vPortEndScheduler()*, shown in Listing 3.19, is called. This function restores the system to its state before the call to *xPortStartScheduler()*.

Listing 3.19: The scheduler stop function *vPortEndScheduler()*.

```
void vPortEndScheduler( void )
{
register unsigned portSHORT reg;

    /* restore system stack pointer */
    SP    = __sof(pxTopOfSystemStack);
```

```
7       SPSEG = __seg(pxTopOfSystemStack);
8
9       /* restore user stack pointer */
10      DPP1  = __pag(pxTopOfUserStack);
11      __setsp(__mknp(1, pxTopOfUserStack));
12
13      /* stop RTOS timer tick */
14      STM_vStopSTM();
15
16      /* portSTART_FIRST_TASK() placed IP and CSP on the stack, remove
            them */
17      portPOP(reg);
18      portPOP(reg);
19
20      /* simulate return value of xPortStartScheduler() */
21      reg = pdPASS;
22      __asm ( " movw r2,%0 \n"        // set function return value
23              :: "w" (reg) );
24
25      /* portSTART_FIRST_TASK() placed PSW on the stack, restore it */
26      portPOP(PSW);
27  }
```

Listing 3.19: The scheduler stop function *vPortEndScheduler()*.

3.8.7 Interrupt Handling

Classical Interrupt Service Routine FreeRTOS requires that Interrupt Service Routines (ISRs) are defined using the *portTASK_FUNCTION()* macro, as shown in Listing 3.20. The macro *portTASK_FUNCTION_PROTO()* has to be used to create a function prototype, e.g., in header files. If, as in this example, the ISR sends data to a queue or uses a semaphore/mutex to communicate with a user task, it can wake the blocked task by calling *portYIELD_FROM_ISR()* as the very last statement.

Classical ISRs store their return address and the registers used by the function on the currently suspended task's system stack, and local variables on the task's user stack. Each of the two stacks has to be big enough to accommodate the maximal number of nested interrupts.

Listing 3.20: Example of a classical ISR.

```
1  portTASK_FUNCTION_PROTO( vMyISR1, MODULE_VECT );
2
3  portTASK_FUNCTION( vMyISR1, MODULE_VECT )
4  {
5  portCHAR cData;
6  portBASE_TYPE xHigherPriorityTaskWoken = pdFALSE;
7
8      /* read hardware registers */
9      cData = ....
10
11     /* post result to queue */
```

```
12      xQueueSendFromISR( xMyQueue, &cData, &xHigherPriorityTaskWoken );
13
14      /* switch context if necessary */
15      portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
16  }
```

Listing 3.20: Example of a classical ISR.

Interrupt Service Routine using Local Register Bank ISRs can optionally be defined using a local register bank via the *portINTERRUPT_HANDLER_BANKSEL()* macro, as shown in Listing 3.21. Currently only one of the two local register banks, namely local register bank 1, can be used. To use local register bank 2, its user stack pointer would need to be initialized first, but this is not done in this port of FreeRTOS.

An ISR using a local register bank stores only its return address on the currently suspended task's system stack, and local variables on the task's user stack. This way the space usage on the system stack is reduced as compared to the classical ISR. Another advantage is that a context switch to a local register bank is much faster than a switch to another global register bank.

However, still each of the two stacks has to be big enough to accommodate the maximum number of nested interrupts.

```
1  portINTERRUPT_HANDLER_BANKSEL_PROTO( vMyISR2, MODULE_VECT, 1 );
2
3  portINTERRUPT_HANDLER_BANKSEL( vMyISR2, MODULE_VECT, 1 )
4  {
5      /* function body same as vMyISR1 above */
6  }
```

Listing 3.21: Example of an ISR using a local register bank.

Note: As the compiler does not save the CPU registers on the stack when a local register bank is selected, ISRs defined using the same register bank need to have the same interrupt priority level, i.e., those ISRs cannot nest.

Note: ISRs defined with the same interrupt priority level (ILVL) need to have different interrupt group levels (XGLVL) assigned, see [102, Chapter 5.1.2, "Interrupt Arbitration"]!

Interrupt Service Routine using Private Global Register Bank ISRs can optionally be defined using a private global memory-mapped register bank via the *portINTERRUPT_HANDLER_BANKSEL()* macro, as shown in Listing 3.22. Instead of a register bank number, a memory-mapped register bank is assigned by name.

An ISR using a private global register bank stores only its return address on the currently suspended task's system stack, and local variables on the task's user stack. This way the space usage on the system stack is reduced as compared to the classical ISR.

However, still each of the two stacks has to be big enough to accommodate the maximum number of nested interrupts. It has to be noted that the context switch to another

global register bank adds additional latency to the ISR because the register bank has to be validated by the CPU first.

```
1 portINTERRUPT_HANDLER_BANKSEL_PROTO( vMyISR3, MODULE_VECT, "MY_bank" );
2
3 portINTERRUPT_HANDLER_BANKSEL( vMyISR3, MODULE_VECT, "MY_bank" )
4 {
5     /* function body same as vMyISR1 above */
6 }
```

Listing 3.22: Example of an ISR using a private global register bank.

Note: As the compiler does not save the CPU registers on the stack when another than the default global register bank is selected, ISRs defined using the same register bank need to have the same interrupt priority level, i.e., those ISRs cannot nest.

Note: ISRs defined with the same interrupt priority level (ILVL) need to have different interrupt group levels (XGLVL) assigned, see [102, Chapter 5.1.2, "Interrupt Arbitration"]!

3.8.8 Demo Application

A demo application is provided with this port of FreeRTOS to test and to demonstrate its functionality. The demo runs the tasks listed in Table 3.10. All tasks are either standard demo tasks or FreeRTOS core tasks, except "Reg1" and "Reg2", which have been modified to test the registers of the C166S V2 architecture.

Table 3.10: Demo application tasks and priorities.

Task Name	Priority	Task Description
BTest1	2	Test scenarios ensuring that tasks do not exit queue send or receive functions prematurely.
BTest2	1	
IntMath	0	Repeatedly performs a 32-bit calculation, checking the result of the calculation against the expected result.
QConsB1	2	Six tasks that operate on three queues, with each consumer task being followed by its accompanying producer task. Scenarios with different and identical priorities are tested.
QProdB2	0	
QConsB3	0	
QProdB4	2	
QProdB5	0	
QConsB6	0	
Reg1	0	Initializes processor registers with known values and checks if they remain unchanged during context switch.
Reg2	0	
Stats	0	Prints process statistics on the serial console. See Figure 3.14 for an example.
Tmr Svc	4	FreeRTOS Timer Service Task - executes timer callbacks.
IDLE	0	FreeRTOS IDLE Task.

RUN TIME STATISTICS

Task Name	1/10 Ticks	Percentage
IDLE	24828	<1%
IntMath	27909887	33%
QProdB2	88840	<1%
QConsB3	94174	<1%
QProdB5	1431541	1%
QConsB6	1447645	1%
Reg1	19473930	23%
Stats	8326	<1%
Reg2	27934058	33%
BTest2	64616	<1%
QConsB1	2738151	3%
QProdB4	2724660	3%
Tmr Svc	2801	<1%
BTest1	56573	<1%
TOTAL TICKS	8400000	
STATS EXE TICKS	7	
PRINT EXE TICKS	2301	

Figure 3.14: Output of the statistics task on the serial console.

Chapter 4

Conclusions and Outlook

4.1 Conclusions

This work focused on the development of a Safety Electronic Control Unit (ECU) for a FSE race car. Such an ECU has, to our knowledge, never before been developed according to the ISO 26262 [13–22].

Despite the fact, that this standard to date only covers road-going vehicles of less than 3.5 tons, it was applied to the development of electronics for a race car of the FSE series. As [106] notes, "There are demands from [the] commercial vehicle sector for extending the ISO 26262 for commercial vehicles and motor cycles.", we see a broadening of this standard in the future. [107] already hints that the German standards working group "VDA Arbeitskreis 16" is heading in this direction. As the global review process of this standard has not even started yet, it is too early to speculate about the ISO 26262 being expanded to cover other types of vehicles.

The development of electronics as Safety Element out of Context (SEooC), heavily used in this work, shows to be a very practicable method. We therefore anticipate that the next release of the ISO 26262 might evolve this concept further. This work also shows that the SEooC development process, according to the ISO 26262, is applicable to the development of electronics for FSE race cars.

The developed ECU is versatile and integrates most of the interfaces specific to FSE race cars. Furthermore, it simplifies wiring harness design and reduces weight. General-purpose ECUs, commonly used for rapid prototyping and pre-series prototypes, lack the necessary interfaces which are common on race cars of this series. The interlocks and the safety system required in this series are not common outside, and therefore need to be implemented as a separate device. This adds to the complexity and to the weight of the wiring harness. Two devices, instead of one, also take up more space in the interior, which is precious in race cars of this size. Additionally, with the port of FreeRTOS [52], the foundation for the software stack has been laid. We see that the ECU presented in this work could very well be established as a standard device on this series.

4.2 Outlook

As the foundation for the software has been laid with the port of an RTOS, the design and implementation of a driver library or some kind of CPU abstraction layer would be a logical next step. Further work would also involve the development of the software safety requirements and a corresponding software design, which would then be implemented and tested on a future race car.

Another possible direction for subsequent projects would also be an evaluation of other setups used within the series, in order to extend the current functional safety concept to also cover those.

The development of a parameterizable ECU software and accompanying PC configuration software for easier adaption within the series would also be desirable.

A totally different research path would be the realization of a port of an AUTOSAR[19] OS, e.g., Arctic Core.[20] As the Arctic Core Microcontroller Abstraction Layer (MCAL) package currently does not support the Infineon XC2300 series CPU, either a MCAL could be written from scratch, or the Infineon MCAL [108] could be evaluated regarding its compatibility with the Arctic Core OS.

Yet another project could as well take the functional safety concept developed in this work, use it to derive the technical safety concept for the drive-by-wire system developed in [109], and then refine the system as a SEooC.

[19] http://www.autosar.org/
[20] http://www.arccore.com/products/arctic-core/

Appendix A

Acronyms and Abbreviations

AC	Alternating Current
ABS	Antilock Break System
ADAS	Advanced Driver Assistance System
ADC	Analog to Digital Converter
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
ASIL	Automotive Safety Integrity Level
AUTOSAR	AUTomotive Open System ARchitecture
BMS	Battery Management System
CAN	Controller Area Network
CAN FD	CAN Flexible Data-rate
CCF	Common Cause Failure
CPLD	Complex Programmable Logic Device
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
CSP	Code Segment Pointer
DAP	Device Access Port, Infineon proprietary debug interface
DC	Direct Current
DPP	Data Page Pointer
DPRAM	Dual Port RAM

E/E	Electrical and/or Electronic
E/E/PE	Electrical and/or Electronic and/or Programmable Electronic
E-Gas	Elektronisches Gaspedal
ECC	Error Correction Code
ECU	Electronic Control Unit
ETC	Electronic Throttle Control
FIFO	First In First Out
FSC	Functional Safety Concept
FSE	Formula Student Electric
FSM	Finite State Machine
FTTI	Fault-Tolerant Time Interval
HAL	Hardware Abstraction Layer
HSI	Hardware-Software Interface
HV	High Voltage, in the automotive sector, includes voltages >60V and ≤1500V DC, and >25V and ≤1000V AC RMS
IP	Instruction Pointer
IMD	Insulation Monitoring Device
ISR	Interrupt Service Routine
LAN	Local Area Network
LIN	Local Interconnect Network
LION	Lithium Ion
LV	Low Voltage, in the automotive sector, includes voltages ≤60V DC, and ≤25V AC RMS, the voltage level of Protective Extra Low Voltage (PELV) according to presumably according to IEC 60364-4-41
MCAL	Microcontroller Abstraction Layer
MicroSD	Micro Secure Digital
MMU	Memory Management Unit
MOSFET	Metal Oxide Semiconductor Field Effect Transistor
MPU	Memory Protection Unit

OS	Operation System
PLC	Programmable Logic Controller
PSW	Program Status Word
PWM	Pulse Width Modulation
RAM	Random Access Memory
RMS	Root Mean Square
RTC	Real-Time Clock
RTOS	Real-Time Operating System
SEooC	Safety Element out of Context
SIL	Safety Integrity Level
SPI	Serial Peripheral Interface
SRAM	Static RAM
TCB	Task Control Block
TMR	Triple Modular Redundant
TSAL	Tractive System Active Light
WiFi	Wireless Fidelity
WLAN	Wireless LAN

Appendix B

ISO 26262

B.1 Detailed Overview of ISO 26262

Figure B.1 shows the detailed overview of ISO 26262 as defined in part 1 of ISO 26262 [13].

The specific clauses of the standard are indicated in the following manner: "m-n", where "m" represents the number of the particular part and "n" indicates the number of the clause within that part.

EXAMPLE "5-7" represents Clause 7 of ISO 26262-5.

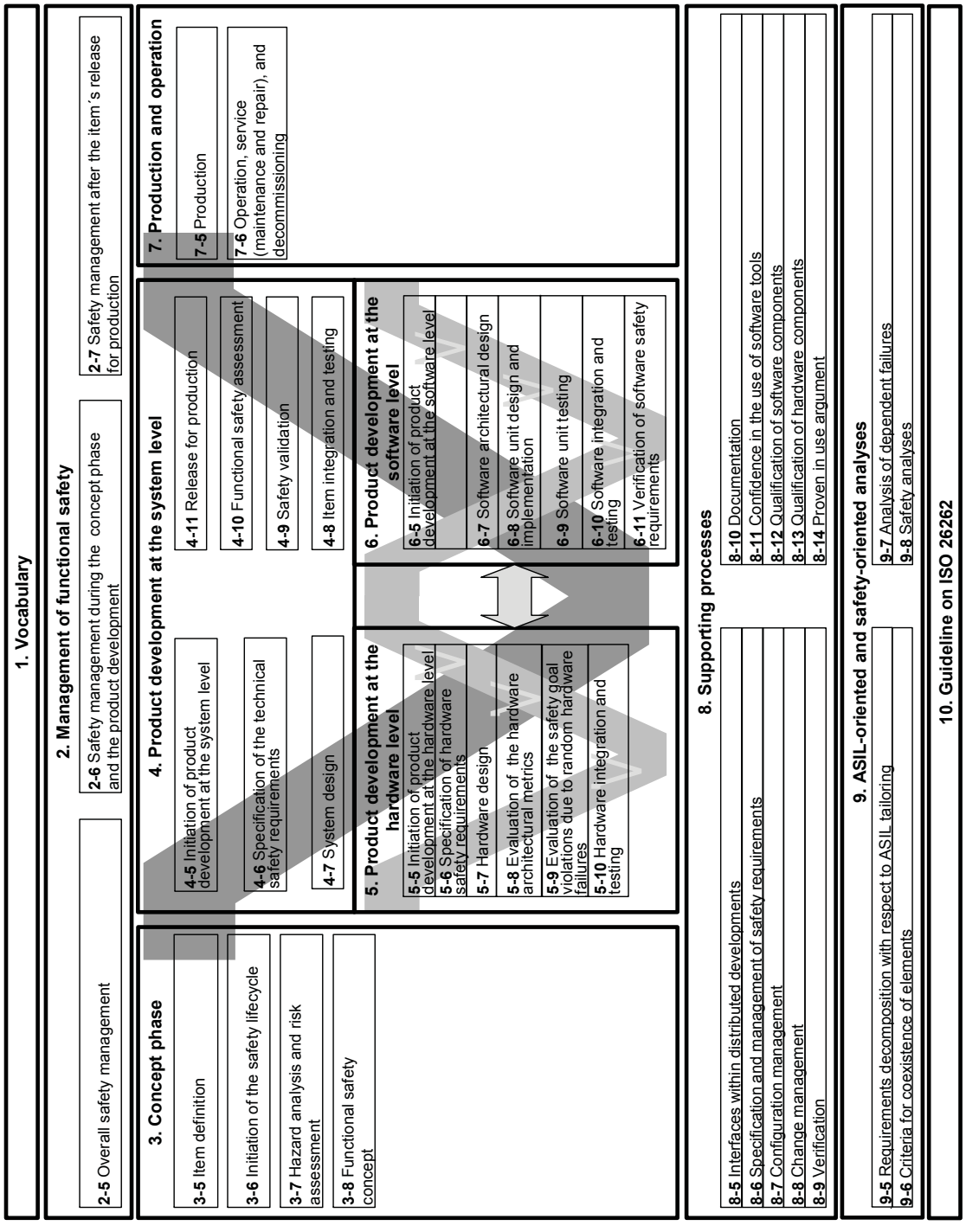


Figure B.1: Detailed overview of ISO 26262 [13, Figure 1]

B.2 Essential vocabulary from ISO 26262 Part 1

controllability
ability to avoid a specified **harm** or damage through the timely reactions of the persons involved, possibly with support from external measures

electrical and/or electronic system (E/E system)
system that consists of electrical and/or electronic **elements**, including programmable electronic elements

failure rate
probability density of failure divided by probability of survival for a hardware element

fault-tolerant time interval
time-span in which a fault or faults can be present in a system before a hazardous event occurs

functional concept
specification of the intended functions and their interactions necessary to achieve the desired behaviour

functional safety
absence of **unreasonable risk** due to **hazards** caused by **malfunctioning behaviour** of E/E systems

harm
physical injury or damage to the health of persons

hazard
potential source of **harm** caused by malfunctioning behaviour of the **item**

item
system or array of systems to implement a function at the vehicle level, to which ISO 26262 is applied

malfunctioning behaviour
failure or unintended behaviour of an **item** with respect to its design intent

risk combination of the probability of occurrence of **harm** and the **severity** of that harm

safe state
operating mode of an **item** without an unreasonable level of **risk**

safety
absence of unreasonable risk

severity
estimate of the extent of **harm** to one or more individuals that can occur in a potentially hazardous situation

unreasonable risk
risk judged to be unacceptable in a certain context according to valid societal moral concepts

B.3 Tables from ISO 26262 Part 3

B.3.1 Classes of severity

Table B.1: Examples of severity classification [15]

	Class of severity			
	S0	**S1**	**S2**	**S3**
Description	No injuries	Light and moderate injuries	Severe and life-threatening injuries (survival probable)	Life-threatening injuries (survival uncertain), fatal injuries
Reference for single injuries (from AIS scale)	- AIS 0 and less than 10 % probability of AIS 1-6 - Damage that cannot be classified safety-related	More than 10 % probability of AIS 1-6 (and not S2 or S3)	More than 10 % probability of AIS 3-6 (and not S3)	More than 10 % probability of AIS 5-6

Note 1: This table is a merger of ISO26262-3:2011 table 1 and table B.1

Note 2: For AIS see `http://en.wikipedia.org/wiki/Abbreviated_Injury_Scale#Severity`

B.3.2 Classes of probability of exposure

Table B.2: Classes of probability of exposure regarding duration [15]

	Class of probability of exposure in operational situations			
	E1	**E2**	**E3**	**E4**
Description	Very low probability	Low probability	Medium probability	High probability
Duration (% of average operating time)	Not specified	<1 % of average operating time	1 % to 10 % of average operating time	>10 % of average operating time

Note: This table is a merger of ISO26262-3:2011 table 2 and table B.2

Table B.3: Classes of probability of exposure regarding frequency [15]

	Class of probability of exposure in operational situations			
	E1	**E2**	**E3**	**E4**
Description	Very low probability	Low probability	Medium probability	High probability
Frequency of situation	Occurs less often than once a year for the great majority of drivers	Occurs a few times a year for the great majority of drivers	Occurs once a month or more often for an average driver	Occurs during almost every drive on average

Note: This table is a merger of ISO26262-3:2011 table 2 and table B.3

B.3.3 Classes of controllability

Table B.4: Classes of controllability [15]

	Class of controllability			
	C0	**C1**	**C2**	**C3**
Description	Controllable in general	Simply controllable	Normally controllable	Difficult to control or uncontrollable
Driving factors and scenarios	Controllable in general	99 % or more of all drivers or other traffic participants are usually able to avoid harm	90 % or more of all drivers or other traffic participants are usually able to avoid harm	Less than 90 % of all drivers or other traffic participants are usually able, or barely able, to avoid harm

Note: This table is a merger of ISO26262-3:2011 table 3 and table B.4

B.3.4 ASIL determination

Table B.5: ASIL determination [15, Table 4]

Severity class	**Probability class**	**Controllability class**		
		C1	**C2**	**C3**
S1	**E1**	QM	QM	QM
	E2	QM	QM	QM
	E3	QM	QM	A
	E4	QM	A	B
S2	**E1**	QM	QM	QM
	E2	QM	QM	A
	E3	QM	A	B
	E4	A	B	C
S3	**E1**	QM	QM	A
	E2	QM	A	B
	E3	A	B	C
	E4	B	C	D

Appendix C

Safety Element out of Context Tables

This appendix contains the tables that are too large to be presented in the SEooC sections of the main part.

C.1 Situation Analysis

Table C.1: Situation analysis of the Formula Student Electric Germany 2012.

Situation	**Percentage of total time**	**Average duration [sec]**	**Number of drivers**	**Number of attempts per driver**	**Approx. track length [m]**	**Top speed [km/h]**	**Average speed [km/h]**	**Median time per run or turn [sec]**	**Remarks**
Acceleration race event	0.72%	17.70	2	2	75	110	-	4.42	-
Skid-pad race event	4.07%	100.43	2	2	115	-	33	25.11	track wet
Autocross race event	14.33%	353.99	2	2	1200	90	50	88.50	-
Endurance race event	66.71%	1647.81	2	1	11000	100	50	823.90	-
Approaching start line (from queue)	5.67%	140.00	8	1.75	10	10	-	10.00	estimated
Standstill (engaged, waiting to start)	8.50%	210.00	8	1.75	-	-	-	15.00	estimated

Data source: Formula Student Electric Germany 2012 event results; speeds estimated based on logged data.

C.2 Hazard Identification and Classification

Table C.2: Hazard identification and classification for Formula Student Electric race car with arguments.

Hazard			**Classification of Hazardous Event (Severity, Exposure, Controllability)**							**Safety Goal**
ID	**Possible Malfunction**	**Situation**	**S**	**Argument**	**E**	**Argument**	**C**	**Argument**	**ASIL**	**ID**
HZ01	Unintended acceleration (at zero speed)	Standstill; marshal crossing 1 m in front of race car	2	A crash is possible because of the person crossing just 1 m before the car. The traffic participant could be badly injured.	3	According to the situation analysis, standstill accounts for less than 10% of the total operating time.	2	Driver can steer to side to avoid accident, can use the mechanically operating braking system to slow/stop the vehicle, and can additionally push the emergency-off button to shut down power to the motors.	A	SG01
HZ02	Unintended acceleration (at low speed)	Approaching start; marshal on the side of the start position	2	A crash is possible because of the person beside the start line. The traffic participant could be badly injured.	3	According to the situation analysis, approaching start accounts for less than 10% of the total operating time.	2	Driver can maintain driving path or steer to side to avoid accident, can use the mechanically operating braking system to slow/stop the vehicle, and can additionally push the emergency-off button to shut down power to the motors. Marshal can jump aside if he sees the car approaching beside the usual starting position.	A	SG02

(continued)

Table C.2: (continued)

Hazard			Classification of Hazardous Event (Severity, Exposure, Controllability)							Safety Goal
ID	**Possible Malfunction**	**Situation**	**S**	**Argument**	**E**	**Argument**	**C**	**Argument**	**ASIL**	**ID**
HZ03	Unintended acceleration (at medium speed)	Autocross or Endurance race event; medium speed (30-50 km/h)	2	A crash is possible and the driver could be badly injured.	4	Medium speed accounts for more than 10% of the total operating time.	2	Driver can maintain driving path or steer to side to avoid accident, can use the mechanically operating braking system to slow/stop the vehicle, and can additionally push the emergency-off button to shut down power to the motors. Marshal can jump aside if he sees the car approaching.	B	SG02
HZ04	Unintended acceleration (at medium speed)	Autocross or Endurance race event; another car close behind or marshal close to track; medium speed (30-50 km/h)	3	A crash is possible and the traffic participants could be badly injured.	3	Because of how the event is organized, one car running up to another accounts for less than 10% of the average operating time during one race event. Marshals are trained by the organizer to keep clear of the vicinity of the track.	2	Driver can maintain driving path or steer to side to avoid accident, can use the mechanically operating braking system to slow/stop the vehicle, and can additionally push the emergency-off button to shut down power to the motors. Marshal can jump aside if he sees the car approaching.	B	SG02

(continued)

Table C.2: (continued)

Hazard			Classification of Hazardous Event (Severity, Exposure, Controllability)							Safety Goal
ID	**Possible Malfunction**	**Situation**	**S**	**Argument**	**E**	**Argument**	**C**	**Argument**	**ASIL**	**ID**
HZ05	Unintended acceleration (at high speed)	Autocross or Endurance race event; high speed (≥70 km/h)	2	A crash is possible and the driver could be badly injured.	4	High speed accounts for more than 10% of the total operating time.	2	Driver can maintain driving path or steer to side to avoid accident, can use the mechanically operating braking system to slow/stop the vehicle, and can additionally push the emergency-off button to shut down power to the motors. Marshal can jump aside if he sees the car approaching.	B	SG02
HZ06	Unintended acceleration (at high speed)	Autocross or Endurance race event; another car close behind or marshal close to track; high speed (≥70 km/h)	3	A crash is possible and the traffic participants could be badly injured or killed.	3	Because of how the event is organized, one car running up to another accounts for less than 10% of the average operating time during one race event. Marshals are trained by the organizer to keep clear of the vicinity of the track.	2	Driver can maintain driving path or steer to side to avoid accident, can use the mechanically operating braking system to slow/stop the vehicle, and can additionally push the emergency-off button to shut down power to the motors. Marshal can jump aside if he sees the car approaching.	B	SG02

(continued)

Table C.2: (continued)

Hazard			Classification of Hazardous Event (Severity, Exposure, Controllability)							Safety Goal
ID	**Possible Malfunction**	**Situation**	**S**	**Argument**	**E**	**Argument**	**C**	**Argument**	**ASIL**	**ID**
HZ07	Unintended acceleration forward one motor, backward the other motor; yawing moment (at medium or high speed)	Cornering; medium or high speed (30-50 km/h or ≥70 km/h)	2	A crash is possible and the driver could be badly injured.	4	On average, cornering occurs during almost every race.	2	Driver can maintain driving path or steer to side to avoid accident, can use the mechanically operating braking system to slow/stop the vehicle, and can additionally push the emergency-off button to force the shutdown of power to the motors.	B	SG02
HZ08	Unintended generative braking, i.e., acceleration backwards (at zero speed)	Standstill; queue of waiting cars 5-10 m behind	2	A crash is possible because of the queue of cars waiting behind – front collision for the other car. The driver could be badly injured.	3	According to the situation analysis, standstill accounts for less than 10% of the total operating time.	2	Driver can use the mechanically operating braking system to slow/stop the vehicle, and can additionally push the emergency-off button to force the shutdown of power to the motors.	A	SG03

(continued)

Table C.2: (continued)

Hazard			Classification of Hazardous Event (Severity, Exposure, Controllability)							Safety Goal
ID	**Possible Malfunction**	**Situation**	**S**	**Argument**	**E**	**Argument**	**C**	**Argument**	**ASIL**	**ID**
HZ09	Unintended generative braking, i.e., acceleration backwards (at low speed)	Approaching start; queue of waiting cars 5-10 m behind	2	A crash is possible because of the queue of waiting cars behind – front collision for the other car. The driver could be badly injured.	3	According to the situation analysis, approaching start accounts for less than 10% of the total operating time.	2	Driver can maintain driving path or steer to side to avoid accident, use the mechanically operating braking system to slow/stop the vehicle, and can additionally push the emergency-off button to force the shutdown of power to the motors. Marshal can jump aside if he sees the car approaching.	A	SG03
HZ10	Unintended generative braking, i.e., acceleration backwards (at medium speed)	Autocross or Endurance race event; medium speed (30-50 km/h)	2	A crash is possible and the driver could be badly injured.	4	Medium speed accounts for more than 10% of the total operating time.	2	Driver can maintain driving path or steer to side to avoid accident, can use the mechanical operating braking system to slow/stop the vehicle, and can additionally push the emergency-off button to force the shutdown of power to the motors. Marshal can jump aside if he sees the car approaching.	B	SG04

(continued)

Table C.2: (continued)

Hazard			Classification of Hazardous Event (Severity, Exposure, Controllability)							Safety Goal
ID	**Possible Malfunction**	**Situation**	**S**	**Argument**	**E**	**Argument**	**C**	**Argument**	**ASIL**	**ID**
HZ11	Unintended generative braking, i.e., acceleration backwards (at medium speed)	Autocross or Endurance race Event; another car close behind or marshal close to track; medium speed (30-50 km/h)	3	A crash is possible and the traffic participants could be badly injured or killed.	3	Because of how the event is organized, one car running up to another accounts for less than 10% of the average operating time during one race event. Marshals are trained by the organizer to keep clear of the vicinity of the track.	2	Driver can maintain driving path or steer to side to avoid accident, can use the mechanically operating braking system to slow/stop the vehicle, and can additionally push the emergency-off button to force the shutdown of power to the motors. Marshal can jump aside if he sees the car approaching.	B	SG04

(continued)

Table C.2: (continued)

Hazard			Classification of Hazardous Event (Severity, Exposure, Controllability)							Safety Goal
ID	**Possible Malfunction**	**Situation**	**S**	**Argument**	**E**	**Argument**	**C**	**Argument**	**ASIL**	**ID**
HZ12	Unintended generative braking, i.e., acceleration backwards (at high speed)	Endurance race event; high speed ($\geq$70 km/h)	2	A crash is possible and the driver could be badly injured.	4	High speed accounts for more than 10% of the total operating time.	2	Driver can maintain driving path or steer to side to avoid accident, can use the mechanically operating braking system to slow/stop the vehicle, and can additionally push the emergency-off button to force the shutdown of power to the motors. Marshal can jump aside if he sees the car approaching.	B	SG04

(continued)

Table C.2: (continued)

Hazard			Classification of Hazardous Event (Severity, Exposure, Controllability)							Safety Goal
ID	**Possible Malfunction**	**Situation**	**S**	**Argument**	**E**	**Argument**	**C**	**Argument**	**ASIL**	**ID**
HZ13	Unintended generative braking, i.e., acceleration backwards (at high speed)	Endurance race event; another car close behind or marshal close to track, high speed (≥70 km/h)	3	A crash is possible and the traffic participants could be badly injured or killed.	3	Because of how the event is organized, one car running up to another accounts for less than 10% of the average operating time during one race event. Marshals are trained by the organizer to keep clear of the vicinity of the track.	2	Driver use the mechanically operating braking system to slow/stop the vehicle and can additionally push the emergency-off button to force the shutdown of power to the motors. Marshal can jump aside if he sees the car approaching.	B	SG04
HZ14	Battery parameters out of range	Any driving situation	2	The driver could be badly injured.	4	Always	2	Driver can maintain driving path and use the mechanically operating braking system to slow/stop the vehicle and get out if the battery, e.g., starts burning.	B	SG05
HZ15	Missing acceleration	Any driving situation	1	A crash is possible and the driver could be lightly injured.	4	Always	1	99% of the drivers are usually able to avoid harm.	QM	SG06

(continued)

Table C.2: (continued)

Hazard			**Classification of Hazardous Event (Severity, Exposure, Controllability)**							**Safety Goal**
ID	**Possible Malfunction**	**Situation**	**S**	**Argument**	**E**	**Argument**	**C**	**Argument**	**ASIL**	**ID**
HZ16	Unintended deceleration without generative braking	Any driving situation	1	A crash is possible and the driver could be lightly injured.	4	Always	1	99% of the drivers are usually able to avoid harm.	QM	SG07
HZ17	Missing deceleration	Any driving situation	1	A crash is possible and the driver could be lightly injured.	4	Always	1	99% of the drivers are usually able to avoid harm.	QM	SG08

C.3 Safety Goals

Table C.3: Safety goals.

Safety Goal			
ID	**Description**	**Safe State**	**ASIL**
SG01	The vehicle shall not accelerate without a valid throttle demand (above the throttle function threshold).	Switch off power to the motors (demand zero torque).	A
SG02	Unintended acceleration shall be prevented.	Switch off power to the motors (demand zero torque).	B
SG03	The vehicle shall not do generative braking or accelerate backwards (below the electric braking speed threshold).	Switch off power to the motors (demand zero torque).	A
SG04	Unintended generative braking or acceleration backwards shall be prevented.	Switch off power to the motors (demand zero torque).	B
SG05	Battery parameters shall be kept within defined safe operating area.	Switch off power to the inverters.	B
SG06	Missing acceleration shall be prevented.	-	QM
SG07	Unintended deceleration without generative braking shall be prevented.	-	QM
SG08	Missing deceleration shall be prevented.	-	QM

SG01 gleaned from [85].

C.4 Functional Safety Concept

Table C.4: Functional safety concept.

Functional Safety Requirement				**Safety Goal**	**Specification**		
ID	**Description**	**ASIL**	**Allocated to Element**	**ID**	**Operating Modes**	**Fault-tolerant Time**	**Safe State**
FSR01	The system shall not send throttle data which causes the vehicle to accelerate without a driver demand.	A	X-by-Wire	SG01	Vehicle powered on	100 ms	Switch off power to the motors (demand zero torque).
FSR02	The system shall not send a torque command which causes the vehicle to accelerate without a driver demand.	A	ECU				
FSR03	Accurate throttle, brake, and steering angle signals shall be generated.	B	X-by-Wire	SG02, SG03, SG04	Vehicle powered on	100 ms	Switch off power to the motors (demand zero torque).
FSR04	The throttle, brake, and steering angle signals shall be received and verified.	B	ECU				
FSR05	Accurate front and rear vehicle speed signals shall be generated.	B	ECU				
FSR06	Accurate torque command shall be generated.	B	ECU				
FSR07	The torque command shall be received and verified.	B	Inverter				
FSR08	Accurate calculation of the resulting torque by means of phase current measurement.	B	Inverter				
FSR09	The torque result shall be received and verified.	B	ECU				
FSR10	The torque result shall be validated.	B	ECU				
FSR11	An accurate battery status shall be provided.	B	HV-Battery	SG05	Vehicle powered on	1 s	Switch off power to the inverters.
FSR12	The battery status shall be received, verified, and reacted upon in case it is out of the safe operating area.	B	ECU				

FSR01 gleaned from [85].

C.5 Technical Safety Concept for ECU as SEooC

Table C.5: Technical safety requirements for the ECU as SEooC.

ECU Technical Safety Requirement			**Func. Safety Req.**	**Specification**			
ID	**Description**	**ASIL**	**ID**	**Operating Modes**	**FTTI**	**Safe State**	**Maintain Safe State**
TSR01	A throttle sensor signal shall be received from the CAN bus.	B	FSR02, FSR04	Powered on	100 ms	Switch off power to the motors (demand zero torque).	Keep until at least 10 consecutive valid zero values have been received, or keep till power-off.
TSR01.1	The throttle sensor signal and its inverse representation shall be received from the CAN bus. Check the signal against its inverse representation and do a range check; assume 0, if failure.	QM(B)					
TSR01.2	Plausibility-check the throttle signal, e.g., gradient. Check the alive-counter for correct sequence and the checksum for message integrity. Force 0, if failure.	B(B)					
TSR02	A brake sensor signal shall be received from the CAN bus.	B	FSR04	Powered on	100 ms	Switch off power to the motors (demand zero torque).	Keep until at least 10 consecutive valid values have been received, or keep till power-off.
TSR02.1	The brake sensor signal and its inverse representation shall be received from the CAN bus. Check the signal against its inverse representation and do a range check; assume 0, if failure.	QM(B)					
TSR02.2	Plausibility-check the brake sensor signal, e.g., gradient. Check the alive-counter for correct sequence and the checksum for message integrity. Force 0, if failure.	B(B)					

(continued)

Table C.5: (continued)

ECU Technical Safety Requirement			Func. Safety Req.	Specification			
ID	**Description**	**ASIL**	**ID**	**Operating Modes**	**FTTI**	**Safe State**	**Maintain Safe State**
TSR03	A steering angle sensor signal shall be received from the CAN bus.	B	FSR04	Powered on	100 ms	Switch off power to the motors (demand zero torque).	Keep until at least 10 consecutive valid values have been received, or keep till power-off.
TSR03.1	The steering angle sensor signal and inverse representation shall be received from the CAN bus. Check the signal against its inverse representation and do a range check; assume 0, if failure.	QM(B)					
TSR03.2	Plausibility-check the steering angle sensor signal, e.g., gradient. Check the alive-counter for correct sequence and the checksum for message integrity. Force 0, if failure.	B(B)					
TSR04	An accurate vehicle speed front signal shall be generated.	B	FSR05	Powered on	-	-	-
TSR04.1	Generate vehicle speed front signal by averaging two sensor signals.	QM(B)					
TSR04.1.1	There shall be a vehicle speed front sensor1.	QM(B)					
TSR04.1.2	There shall be a vehicle speed front sensor2.	QM(B)					
TSR04.2	The vehicle speed front sensors shall be checked for plausibility, e.g., gradient. Set “vehicle speed front failure” flag if failure, else reset failure flag if valid signals for 10 consecutive times.	B(B)					

(continued)

Table C.5: (continued)

ECU Technical Safety Requirement			Func. Safety Req.	Specification			
ID	**Description**	**ASIL**	**ID**	**Operating Modes**	**FTTI**	**Safe State**	**Maintain Safe State**
TSR05	An accurate vehicle speed rear signal shall be generated.	B	FSR05	Powered on	-	-	-
TSR05.1	Generate vehicle speed rear signal by averaging two sensor signals.	QM(B)					
TSR05.1.1	There shall be a vehicle speed rear sensor1.	QM(B)					
TSR05.1.2	There shall be a vehicle speed rear sensor2.	QM(B)					
TSR05.2	The vehicle speed rear sensors shall be checked for plausibility, e.g., gradient. Set "vehicle speed rear failure" flag if failure, else reset failure flag if valid signals for 10 consecutive times.	B(B)					
TSR06	An accurate torque command shall be generated based on throttle, brake, vehicle speed, and battery health status.	B	FSR06, FSR10	Powered on	100 ms	Switch off power to the motors (demand zero torque).	The number of fault reactions shall be stored in NVRAM. If fault reactions > TCMD_FAIL_-COUNT, then only reset in garage shall be possible, else keep until power-off.
TSR06.1	Generate torque command. Take "vehicle speed failure" flags into account. Limit gradient to TORQUE_GRADIENT_LIMIT. Range-check output signal. Send torque command via CAN.	QM(B)					
TSR06.2	Torque command shall be plausibility-checked against torque result. Force 0, if failure.	B(B)					

(continued)

Table C.5: (continued)

ECU Technical Safety Requirement			Func. Safety Req.	Specification			
ID	**Description**	**ASIL**	**ID**	**Operating Modes**	**FTTI**	**Safe State**	**Maintain Safe State**
TSR07	A torque result signal shall be received from the CAN bus.	B	FSR09	Powered on	100 ms	Switch off power to the motors (demand zero torque).	Keep until at least 10 consecutive valid values have been received, or keep till power-off.
TSR07.1	The torque result signal and its inverse representation shall be received from the CAN bus. Check the signal against its inverse representation and do a range check.	QM(B)					
TSR07.2	Plausibility-check the torque result signal. Check the alive-counter for correct sequence and the checksum for message integrity. Set "torque result failure" flag, if failure.	B(B)					
TSR08	A battery status shall be received from the CAN bus and reacted upon in case it is out of the safe operating area.	B	FSR12	Powered on	1 s	Switch off power to the inverters.	The number of fault reactions shall be stored in NVRAM. If fault reactions > BATT_FAIL_-COUNT, then only reset in garage shall be possible, else keep until power-off.
TSR08.1	The battery status shall be received from the CAN bus and reacted upon in case it is out of the safe operating area.	QM(B)					
TSR08.1.1	The battery status and its inverse representation shall be received from the CAN bus. Check the signal against its inverse representation and do a range check; assume battery status is out of safe operating area if checks fail.	QM(B)					
TSR08.1.2	If battery status indicates "out of safe operating area", command zero torque from inverters within TIME_FTTI_THROTTLE (100ms) and switch off power to the inverters within TIME_FTTI_BATT (1s).	QM(B)					

(continued)

Table C.5: (continued)

ECU Technical Safety Requirement			Func. Safety Req.	Specification			
ID	**Description**	**ASIL**	**ID**	**Operating Modes**	**FTTI**	**Safe State**	**Maintain Safe State**
TSR08.2	Plausibility-check the battery status. Check the alive-counter for correct sequence and the checksum for message integrity.	B(B)					
TSR09	There shall be a reliable HV contactor control.	B	FSR12	Powered on	1 s	Switch off power to the inverters.	The number of fault reactions shall be stored in NVRAM. If fault reactions > CONT_FAIL_-COUNT, then only reset in garage shall be possible, else keep until power-off.
TSR09.1	There shall be a HV contactor control for 3 contactors.	QM(B)					
TSR09.1.1	There shall be a contactor control for HV plus contactor.	QM(B)					
TSR09.1.2	There shall be a contactor control for HV negative contactor.	QM(B)					
TSR09.1.3	There shall be a contactor control for HV pre-charge contactor.	QM(B)					
TSR09.2	Plausibility-check the contactor output stages. Compare voltage and current status feedback against contactor control status. Switch off all contactors in case of detected discrepancy.	B(B)					
TSR10	Control flow monitoring. In case of failure, command zero torque from inverters within TIME_FTTI_THROTTLE (100ms) and switch off power to the inverters within TIME_FTTI_BATT (1s).	B	FSR02, FSR04, FSR05, FSR06, FSR09, FSR10, FSR12	Powered on	1 s	Switch off power to the inverters.	The number of fault reactions shall be stored in NVRAM. If fault reactions > CNTR_FAIL_-COUNT, then only reset in garage shall be possible, else keep until power-off.

(continued)

Table C.5: (continued)

ECU Technical Safety Requirement			Func. Safety Req.	Specification			
ID	**Description**	**ASIL**	**ID**	**Operating Modes**	**FTTI**	**Safe State**	**Maintain Safe State**
TSR11	External monitoring facility. In case of failure, command zero torque from inverters within TIME_FTTI_THROTTLE (100ms) and switch off power to the inverters within TIME_FTTI_BATT (1s).	B	FSR02, FSR04, FSR05, FSR06, FSR09, FSR10, FSR12	Powered on	1 s	Switch off power to the inverters.	The number of fault reactions shall be stored in NVRAM. If fault reactions > MON_FAIL_-COUNT, then only reset in garage possible, else keep until power-off.
TSR12	Memory Check. In case of failure command zero torque from inverters within TIME_FTTI_THROTTLE (100ms) and switch off power to the inverters within TIME_FTTI_BATT (1s).	B	FSR02, FSR04, FSR05, FSR06, FSR09, FSR10, FSR12	Power-up, powered on	1 s	Switch off power to the inverters.	The number of fault reactions shall be stored in NVRAM. If fault reactions > MEM_FAIL_-COUNT, then only reset in garage shall be possible, else keep until power-off.
TSR13	CPU Check. In case of failure command zero torque from inverters within TIME_FTTI_THROTTLE (100ms) and switch off power to the inverters within TIME_FTTI_BATT (1s).	B	FSR02, FSR04, FSR05, FSR06, FSR09, FSR10, FSR12	Power-up, powered on	1 s	Switch off power to the inverters.	The number of fault reactions shall be stored in NVRAM. If fault reactions > CPU_FAIL_COUNT, then only reset in garage shall be possible, else keep until power-off.

(continued)

Table C.5: (continued)

ECU Technical Safety Requirement			**Func. Safety Req.**	**Specification**			
ID	**Description**	**ASIL**	**ID**	**Operating Modes**	**FTTI**	**Safe State**	**Maintain Safe State**
TSR14	Voltage Checks. In case of failure command zero torque from inverters within TIME_FTTI_THROTTLE (100ms) and switch off power to the inverters within TIME_FTTI_BATT (1s).	B	FSR02, FSR04, FSR05, FSR06, FSR09, FSR10, FSR12	Power-up, powered on	1 s	Switch off power to the inverters.	The number of fault reactions shall be stored in NVRAM. If fault reactions > VLT_FAIL_COUNT, then only reset in garage shall be possible, else keep until power-off.

Appendix D

Rules of the Formula Student/FSAE Series

D.1 Formula SAE® Rules

This chapter lists the project-relevant rules of the Formula SAE (FSAE) Rules 2012 [80]. They are included here for easier referencing and in case the original documents are no longer available at a later date.

D.1.1 Rule A1.2 Vehicle Design Objectives

For the purpose of the Formula SAE competition, teams are to assume that they work for a design firm that is designing, fabricating, testing and demonstrating a prototype vehicle for the non-professional, weekend, competition market.

D.1.2 Rule B11.3.1 The cockpit-mounted master switch

a. Must be located to provide easy actuation by the driver in an emergency or panic situation.

b. Must be located within easy reach of the belted-in driver, alongside the steering wheel, and unobstructed by the steering wheel or any other part of the car. It is suggested that it be placed on the same side of the steering wheel as the shifter mechanism.

c. Must be a push/pull Emergency switch. The switch must be installed such that:

   i. From the ON position, pushing on the switch will disable power to the ignition and all fuel pumps, and

   ii. From the OFF position, pulling on the switch will enable power to the ignition and fuel pump(s). Switches that require a twist or twist and pull to enable power are acceptable.

d. May act through a relay.

D.1.3 Rule C3.6.1.a General Requirements

[...] designed to reflect a hypothetical car built for production at the annual volume of 1000 units per year.

D.1.4 Rule D3.1 Operating Conditions

The following operating conditions will be recognized at Formula SAE:

D3.1.1 Dry – Overall the track surface is dry.

D3.1.2 Damp – Significant sections of the track surface are damp.

D3.1.3 Wet – The entire track surface is wet and there may be puddles of water.

D.1.5 Rule D7.2.2 Autocross Course Specifications & Speeds

The length of each run will be approximately 0.805 km (1/2 mile) and the driver will complete a specified number of runs.

D.1.6 Rule D8.6.1 Endurance Course Specifications & Speeds

Course speeds can be estimated by the following standard course specifications. Average speed should be 48 km/hr (29.8 mph) to 57 km/hr (35.4 mph) with top speeds of approximately 105 km/hr (65.2 mph).

D.1.7 Rule D8.7 Endurance General Procedure

D8.7.1 The event will be run as a single heat approximately 22 km (13.66 miles) long.

D8.7.3 A driver change must be made during a three (3) minute period at the midpoint of the heat.

D.2 Formula Student Electric Rules

This chapter lists the project-relevant rules of the Formula Student Electric (FSE) Rules 2012 [81].

D.2.1 Rule 4.4.4 Brake Over-Travel Switch Function(Specific FSE change of Formula SAE® 2012 Rule B7.3.1)

Instead of switching off the ignition and fuel pumps the brake pedal over-travel switch must shut down the Tractive System by opening the safety circuit, see also 7.17 (Chapter D.2.10).

D.2.2 Rule 4.12.4 Torque Encoder (throttle pedal position sensor)

Drive by wire is permitted.
The torque encoder must be actuated by a foot pedal.
The foot pedal must return to its original position when not actuated.
The foot pedal must have a positive stop preventing the mounted sensors from being damaged or overstressed.
At least two separate sensors have to be used as torque encoder. Separate is defined as not sharing supply or signal lines.
If an implausibility occurs between the values of these two sensors the power to the motor(s) has to be immediately shut down completely. It is not necessary to completely deactivate the Tractive System, the motor controller(s) shutting down the power to the motor(s) is sufficient.
Implausibility is defined as a deviation of more than 10% pedal travel between the sensors. If three sensors are used at least two sensors have to be within 10% pedal travel. Each sensor has to have a separate detachable connector that enables a check of these functions by unplugging it during E-Scrutineering.

D.2.3 Rule 4.12.5 Torque Encoder Plausibility Check

The power to the motors has to be immediately shut down completely, if the brake pedal is actuated and the torque encoder signals more than 25% pedal travel at the same time. The motor power shut down has to remain active until the torque encoder signals less than 5% pedal travel, no matter whether the brake pedal is still actuated or not.

D.2.4 Rule 7.2 Failure Modes and Effects Analysis (FMEA)

Teams must submit a complete failure modes and effects analysis (FMEA) of the tractive system prior to the event.
A template including required failures to be described will be made available on the FSG website at `http://www.formulastudent.de/fse/2012/rules/`
Do not change the format of the template. Pictures, schematics and data sheets to be referenced in the FMEA have to be included in the ESF.

D.2.5 Rule 7.7 Insulation Monitoring Device (IMD)

Every car must have an insulation monitoring device (IMD) installed in the tractive system. For information regarding FSE approved IMD(s) please refer to the corresponding document in the “Rules & Important Documents” section of the FSG website.
The response value of the IMD needs to be set to 500 Ohm / Volt, related to the maximum tractive system operation voltage.
In case of an insulation failure or an IMD failure, the IMD must break the holding current flow of the accumulator insulation relay(s) to shut down the tractive system.
This has to be done without the influence of any logic e.g. a micro-controller. See also 7.17 (Chapter D.2.10) regarding the re-activation of the tractive-system after an insulation fault.
The status of the IMD has to be shown to the driver by a red indicator light in the cockpit

that is easily visible even in bright sunlight. This indicator has to light up, if the IMD detects an insulation failure or if the IMD detects a failure in its own operation e.g. when it looses reference ground. The IMD indicator light has to be clearly marked with the lettering “IMD” or “GFD” (Ground Fault Detector).

D.2.6 Rule 7.13 Tractive-system-active light (TSAL)

It must be clearly visible when the tractive system is set to active. The car is defined as active whenever the accumulator insulation relay is closed or the voltage outside the accumulator containers exceeds 40V DC or 25V AC RMS. For this the car must be equipped with a light mounted under the highest point of the main roll hoop which lights if the car’s tractive system is active and which is off when the tractive system is not active, see definition above.
The TSAL must be red.
The TSAL has to flash continuously with a frequency between 2Hz and 5Hz.
The voltage being present within the tractive system must directly control the TSAL using hard wired electronics (no software control is permitted).
It must not be possible for the driver’s helmet to contact the TSAL.
The TSAL has to be clearly visible from every horizontal direction, except small angles which are covered by the main roll hoop, even in very bright sunlight.
NOTE: If any offical e.g. track marshal, scrutineer, etc. considers the TSAL to not be easily visible during track operations the team may not be allowed to compete in any dynamic event before the problem is solved.
It is prohibited to mount other lights in proximity to the TSAL.

D.2.7 Rule 7.14 Shut Down Buttons

A system of three shut-down buttons must be installed on the vehicle.
Pressing one of the shut-down buttons must separate the tractive system from the accumulator block by opening the accumulator insulation relays, AIRs, see also Rule 7.17 (Chapter D.2.10).
After separating the system, the voltage in the tractive system must drop to under 40V DC or 25V AC RMS in less than **five** seconds.
Each shut-down button must be a push-pull or push-rotate emergency switch where pushing the button opens the circuit of the holding current of the accumulator insulation relays.
The shut-down buttons must not act through logic, e.g. a microcontroller.
One button must be located on each side of the vehicle behind the driver’s compartment at approximately the level of the driver’s head. The minimum allowed diameter of the shut down buttons on both sides of the car is 40 mm.
One shut-down button is equivalent to the cockpit-mounted Master Switch and must be easily accessible by the driver in any steering wheel position. The minimum allowed diameter of the shut down button in the cockpit is 24 mm.
The shutdown buttons are not allowed to be easily removable, e.g. mounted onto removable body work.

D.2.8 Rule 7.15 Master Switches

Each vehicle has to have two Master Switches, the Control System Master Switch, CSMS, and the Tractive System Master Switch, TSMS.
The CSMS must completely disable power to the Control System and must be direct acting, i.e. it cannot act through a relay or logic.
The CSMS must be located on the right side of the vehicle, in proximity to the Main Hoop, at shoulder height and be easily actuated from outside the car.
The TSMS must be located next to the CSMS and break the current flow holding the accumulator insulation relays. The TSMS must be direct acting, i.e. it cannot act through a relay or logic.
After separating the system, the voltage in the tractive system must drop to under 40V DC or 25V AC RMS in less than **five** seconds, see also Rule 7.17 (Chapter D.2.10).
Both master switches have to be of the rotary type, with a red, removable key, similar to the one shown in Figure 4 (see Figure D.1).
The master switches are not allowed to be easily removable, e.g. mounted onto removable body work.
The function of both switches must be clearly marked with "LV" and "HV". A sticker with a red or black lightning bolt on a yellow background or red lightning bolt on a white background must additionally mark the Tractive System Master Switch. The "ON" position of both switches must be in the horizontal position.

D.2.9 Rule 7.16 Inertia Switch

All vehicles must be equipped with an inertia switch. This must be a Sensata Resettable Crash Sensor or equivalent approved by FSE.
The inertia switch must be part of the Safety Circuit and must be wired in series with the shutdown buttons such that an impact will result in the Safety Circuit being opened.
The inertia switch must latch until manually reset.
The device must trigger due to an impact load which decelerates the vehicle at between 6g and 11g depending on the duration of the deceleration (see spec sheet of the Sensata device).
This may be reset by the driver from within the driver's cell.
It must be possible to demount the device so that its functionality can be tested by shaking it.

D.2.10 Rule 7.17 Safety Circuit

Setting any of the 2 master switches or of the 3 shut-down buttons to the "Off"- Position, activating the brake-over-travel-switch, an insulation failure detected by the IMD, a tripped inertia switch or critical values of the accumulators detected by the battery management system, BMS, must open all accumulator insulation relay(s) and the voltage in the tractive system must drop to under 40V DC or 25V AC RMS in less than **five** seconds after such event.
An exemplary schematic of the required safety circuit, excluding possibly needed interlock circuitry, is shown in Figure D.1.
If the tractive system is shut down by the BMS or the IMD the tractive system must

remain disabled until being manually reset by a person directly at the car which is not driver.
It must not be possible for the driver to re-activate the tractive system from within the car in case of an BMS or IMD fault.
For example: Applying an IMD test resistor between HV+ and control system ground must deactivate the system. Disconnecting the test resistor must not re-activate the system. The tractive system must remain inactive until it was manually reset. All circuits that are part of the safety circuit have to be designed in a way, that in de-energized state they are open with respect to the current controlling the AIRs.
If the tractive system is de-activated while driving, the motor(s) has/have to spin free e.g. no brake torque must be applied to the motor(s).

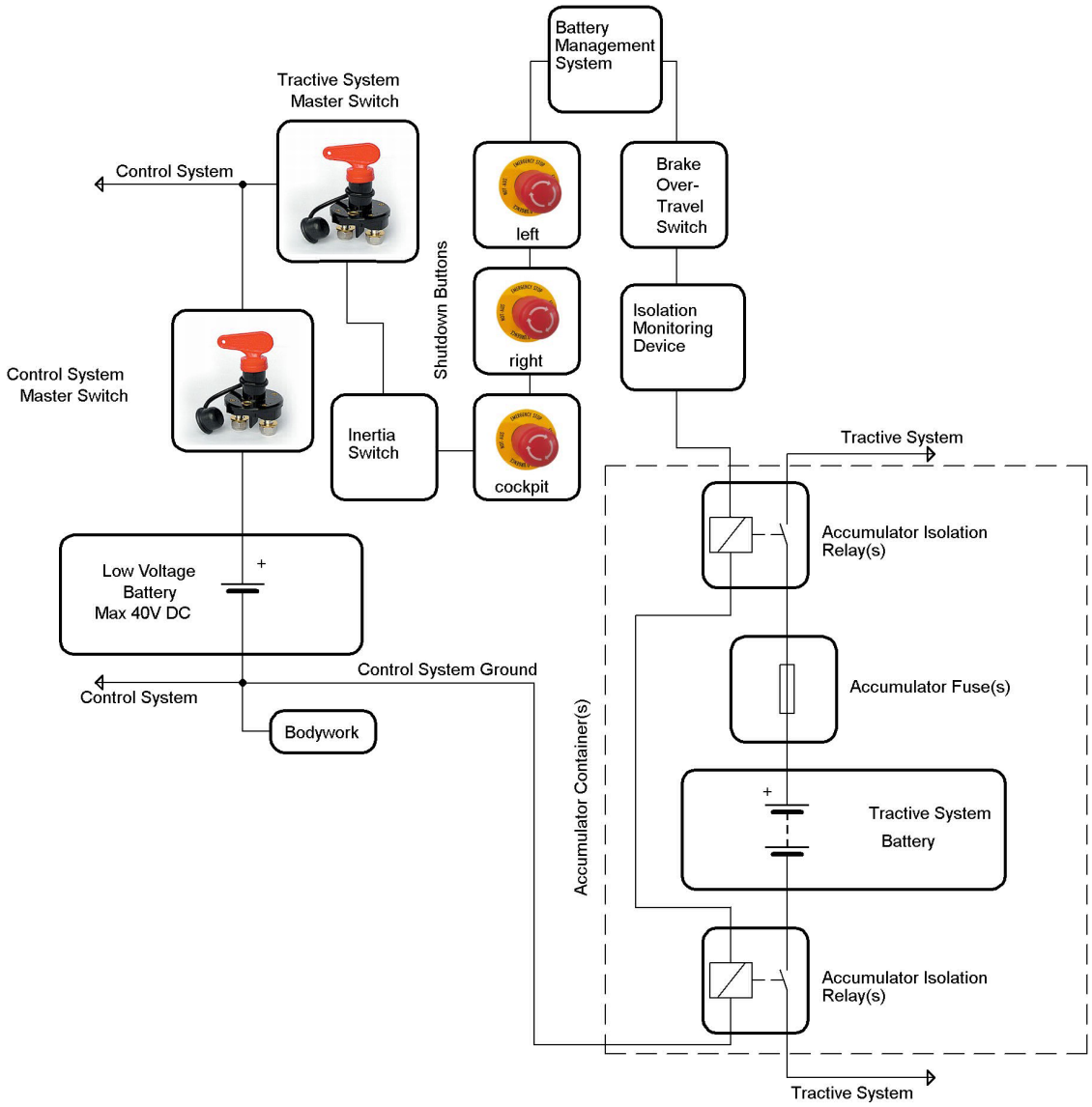


Figure D.1: Figure 4: Schematic overview of the car's Safety Circuit [81]

D.2.11 Rule 7.18 Activating the Tractive System

The driver has to be able to (re-)activate or reset the Tractive System from within the cockpit without the assistance of any other person except for situations in which the BMS or IMD have shut down the tractive system, see 7.17 (Chapter D.2.10).
Resetting or re-activating the Tractive System by operating controls which cannot be reached by the driver is considered as working on the car.
Only closing the Safety Circuit/AIRs must not set the car to ready-to-drive mode. The car is ready to drive as soon as the motor(s) will respond to the input of the torque encoder / acceleration pedal. Therefore additional actions are required by the driver to set the car to ready-to-drive-mode e.g. pressing a dedicated start button, after the tractive system has been activated.

D.2.12 Rule 7.23 Accumulator Insulation Relay(s) (AIR)

In every accumulator container at least two insulation relays must be installed. The accumulator insulation relays must cut both(!) poles of the accumulator.
If these relays are open, no HV may be present outside of the accumulator container.
The insulation relays must be of a "normally open" type. The maximum switch-off-current of the used accumulator insulation relay must be higher than the used accumulator fuse value.

D.2.13 Rule 7.24 Pre-Charge and Discharge Circuits

A circuit that is able to pre-charge the intermediate circuit to at least 90% of the current accumulator voltage before closing the second AIR has to be implemented.
This circuit has to be blocked by a de-activated safety circuit, see rule 7.17 (Chapter D.2.10).
Therefore the pre-charge circuit must not be able to pre-charge the system, if the safety circuit is open.
It is allowed to pre-charge the intermediate circuit for a conservatively calculated time, before closing the second AIR. A feedback via measuring the current intermediate circuit voltage is not required.
If a discharge circuit is needed to meet the "below five seconds under 40VDC"- bound, it has to be designed to handle the maximum discharge current for at least 15 seconds. The calculation proving this has to be part of the ESF.
The discharge circuit has to be wired in a way that it is always active whenever the safety circuit is open. Furthermore the discharge circuit has to be fail-safe.

D.2.14 Rule 7.26 Battery Management System (BMS)

Each accumulator must be monitored by a battery management system whenever the tractive system is active or the accumulator is connected to a charger.
The BMS must continuously measure the cell voltage of every cell in order to keep the cells inside the allowed minimum and maximum cell voltage bound stated in the cell data sheet. If single cells are directly connected in parallel, only one voltage measurement is needed.

The BMS must continuously measure the temperatures of critical points of the accumulator to keep the cells below the allowed maximum cell temperature bound stated in the cell data sheet.
The temperature of at least 30% of the cells has to be monitored by the BMS, if the used accumulator cells are not intrinsically safe, which has to be proven by corresponding documentation in the ESF. The monitored cells have to be equally distributed over the accumulator container(s).
Cells are only considered intrinsically safe, if their cell chemistry is based on LiFePO4.
The BMS must be capable of shutting down the tractive system, if critical values are detected.
FSE recommends to monitor every cell voltage and every cell temperature.

Appendix E

C166S V2 Core

All sections in this appendix are verbatim copies from [102], the *C166S V2 User Manual*. This information is included here for easier referencing and in case the original document is no longer available at a later date.

E.1 Section 2.5.2.1 Addressing via Data Page Pointer DPP

...

After reset, the DPP registers select data pages 3...0 within segment 0. If the user does not want to use any data paging, no further action is required.

...

E.2 Section 2.5.5 The System Stack

The C166S V2 CPU supports a system stack of 64 kBytes. The stack can be located internally in one of the on-chip memories or externally. The 16-bit Stack Pointer (SP) register addresses the stack within a 64 kByte segment. The Stack Pointer Segment Register (SPSG) selects the segment in which the stack is located. A virtual stack (usually bigger then 64 kBytes) can be implemented by software. This mechanism is supported by registers STKOV and STKUN (see descriptions below).

The Stack Pointer Register SP

The non-bit addressable Stack Pointer SP register is used to point to the top of the system stack (TOS). The SP register is pre-decremented whenever data is to be pushed onto the stack, and it is post-incremented whenever data is to be popped from the stack. Therefore, the system stack grows from higher toward lower memory locations. The SP register can be updated via any instruction capable of modifying an 16-bit SFR.

Note: Due to the internal instruction pipeline, a stack pointer initialization stalls the instruction flow until the operation is finished. A POP and RETURN instruction can immediately follow an instruction updating the SP.

...

E.3 Section 2.6.5 Multiply and Divide Unit

The C166S V2 CPU multiply and divide unit has two separated parts. One is the fast 16x16-bit multiplier that executes a multiplication in one CPU cycle. The other one is a division sub-unit which performs the division algorithm in 21 CPU cycles maximum. According to the data and division types, the division length varies between 18 and 21 cycles. The divide instruction requires four CPU cycles to be executed. For performance reasons, the rest of the division algorithm runs in the background during the following seventeen CPU cycles, while further instructions are executed in parallel. If another instruction tries to use the unit while a division is still running, the execution of this new instruction is stalled until the division is finished. Interrupt tasks can also be started and executed immediately without any delay. The previous division will be finished in the background. If an instruction of the interrupt task uses the multiply and divide unit before the previous division process is finished, the instruction flow will be stalled as well. To avoid these stalls, the multiply and division unit should not be used during the first fourteen CPU cycles of the interrupt tasks. This requires up to fourteen one-cycle instructions to be executed between the interrupt entry and the first instruction which uses the multiply and divide unit again (worst case).

...

E.4 Section 3.3 DPRAM, Internal SRAM, and SFR Areas

The C166S V2 CPU differentiates between various internal memory types and internal peripheral areas. These data memories and the IO/SFR areas are located within data page 3 and provide fast accesses using one dedicated Data Page Pointer (see Figure E.1).

Note: Code access is not possible from the DPRAM, the Internal RAM, or the IO/SFR areas.

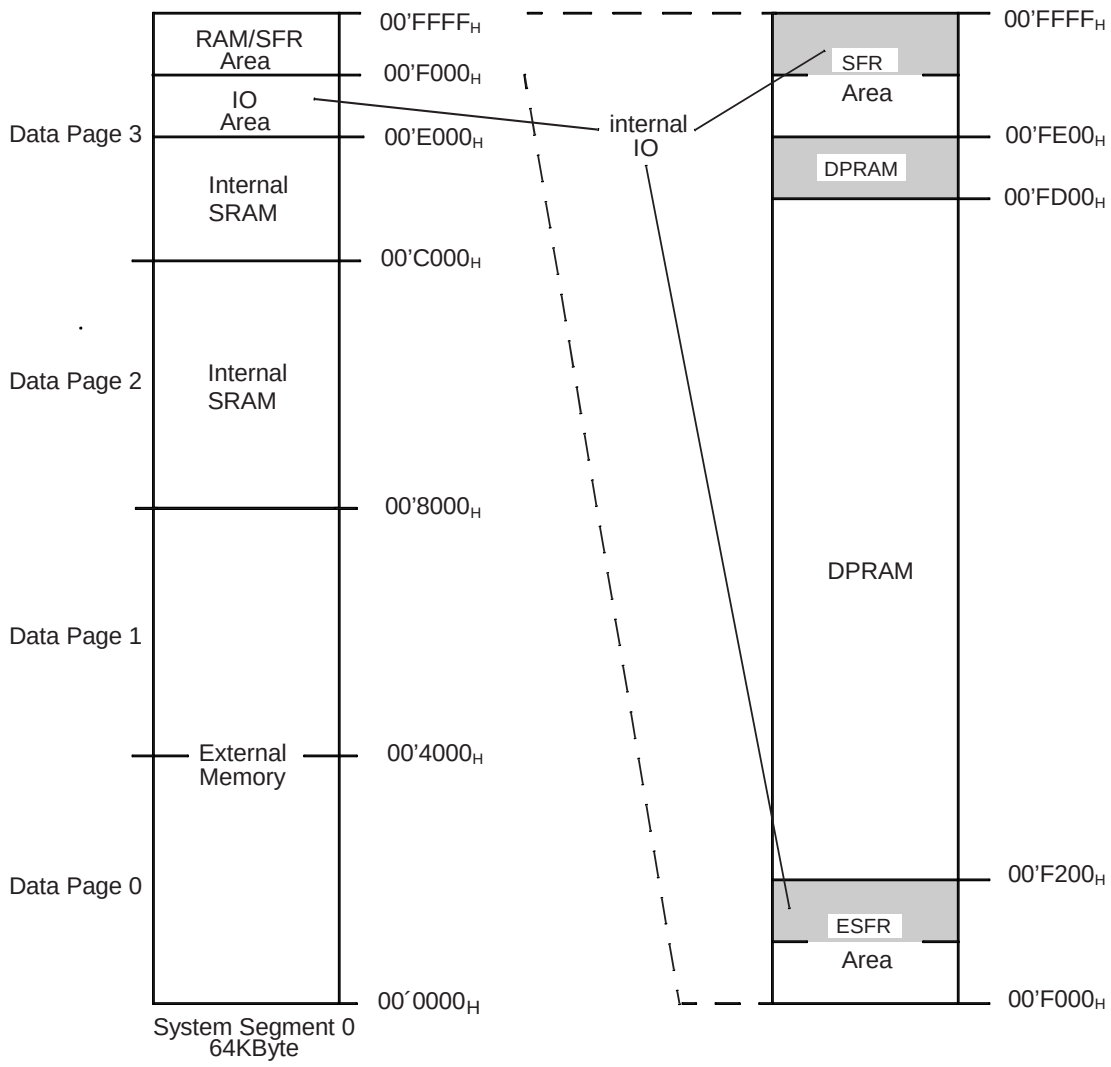


Figure E.1: Figure 3-3 RAM and SFR Areas [102]

E.5 Section 3.5 Crossing Memory Boundaries

. . .

Data Pages are contiguous blocks of 16 KBytes each. They are referenced via the data page pointers DPP3...0 and via an explicit data page number for data accesses overriding the standard DPP scheme. Each DPP register can select one of the possible 1024 data pages. The DPP register that is used for the current access is selected via the two upper bits of the 16-bit data address. Subsequent 16-bit data addresses that cross the 16 KByte data page boundaries will use different data page pointers, while the physical locations need not be subsequent within memory.

E.6 Section 5.2.2 Saving the Status during Interrupt Service

Before an operating system or ITC[21] can actually service a task switch request or interrupt, the CPU must save the current task status. The C166S V2 CPU saves the CPU status (PSW) along with the return address in the system stack. The return address defines the point at which the execution of the interrupted task is to be resumed after returning from the service routine. This return address is specified by the Instruction Pointer (IP) and, in the case of a segmented memory model, also by the Code Segment Pointer (CSP).

...

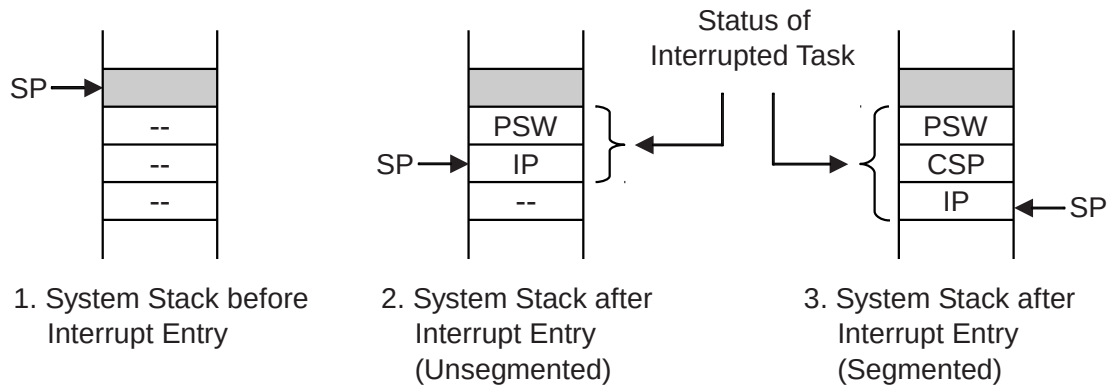


Figure E.2: Figure 5-3 Task Status Saved on the System Stack [102]

...

When the CPU returns from the interrupt service routine (RETI is executed), the status information is popped from the system stack in reverse order. The status information contents depend on the SGTDIS bit value (see Figure E.2).

[21] Interrupt Controller

Appendix F

Tasking VX-toolset for C166 v3.1

All sections in this appendix are verbatim copies from [103], the *TASKING VX-toolset for C166 v3.1 User Guide*. This information is included here for easier referencing and in case the original document is is no longer available at a later date.

F.1 Section 1.3. Accessing Memory

The TASKING VX-toolset for C166 internally knows the following address types:

- 32-bit linear, 'huge' addresses. The address notation is in bytes, starts at 0 and ends at 16M.
- 32-bit paged, 'far' addresses. In the address notation the high word contains the 10-bit page number and the low word contains the 14-bit offset within the 16 kB page.
- 16-bit, 'near' addresses. The high 2 bits contain the DPP number and the low 14 bits are the offset within the 16 kB page.
- 12-bit bit-addressable addresses. This embodies an 8-bit word offset in the bit-addressable space and a 4-bit bit number.
- 8-bit SFR addresses. This is an offset within the SFR space or within the extended SFR space.

The TASKING VX-toolset for C166 toolset has several keywords you can use in your C source to specify memory locations. This is explained in the sub-sections that follow.

F.1.1 Section 1.3.2. Memory Models

The C compiler supports four data memory models, listed in the following table.

Table F.1: Tasking C-Compiler supported memory models [103]

Memory model	Letter	Default data memory type
Near	n	__near
Far	f	__far
Segmented Huge	s	__shuge
Huge	h	__huge

Each memory model defines a default memory type for objects that do not have a memory type qualifier specified. By default, the C166 compiler uses the near memory model. With this memory model the most efficient code is generated. With the *C compiler option –model* you can specify another memory model.

For information on the memory types, see [103, Section 1.3.1], *Memory Type Qualifiers.*

F.2 Section 1.12.1 Calling Convention

F.2.1 Parameter passing

A lot of execution time of an application is spent transferring parameters between functions. The fastest parameter transport is via registers. Therefore, function parameters are first passed via registers. If no more registers are available for a parameter, the compiler pushes parameters on the stack.

The following conventions are used when passing parameters to functions.

Registers available for parameter passing are USR0, R2, R3, R4 R5, R11, R12, R13 and R14. Parameters <= 64 bit are passed in registers except for 64-bit structures:

Table F.2: Tasking C-Compiler parameter passing [103]

Parameter Type	Registers used for parameters
1 bit	USR0, R2.0..15, R3.0..15, R4.0..15, R5.0..15
8 bit	RL2, RH2, RL3, RH3, RL4, RH4, RL5, RH5
16 bit	R2, R3, R4, R5, R11, R12, R13, R14
32 bit	R2R3, R4R5, R11R12, R13R14
64 bit	R2R3R4R5, R11R12R13R14

The parameters are processed from left to right. The first not used and fitting register is used. Registers are searched for in the order listed above. When a parameter is > 64 bit, or all registers are used, parameter passing continues on the stack. The stack grows from higher towards lower address, each parameter on the stack is stored in little-endian. The first parameter is pushed at the lowest stack address. The alignment on the stack depends on the data type as listed in [103, Section 1.1], Data Types.

Example with three arguments:

```
func1( int a, long b, int * c )
```
a (first parameter) is passed in registers R2.
b (second parameter) is passed in registers R4R5.
c (third parameter) is passed in registers R3.

F.2.2 Stack usage

The stack on the C166 consists of a system stack and a user stack. The system stack is used for the return addresses and for data explicitly pushed with the PUSH instruction. The compiler usually does not push anything on the system stack, with exception to interrupt functions. The user stack is used for parameter passing, allocation of automatics and temporary storage. The compiler uses R15 as user stack pointer. The data on the stack is aligned depending on the data type as listed in [103, Section 1.1], Data Types. The stack pointer itself is always aligned at 16-bit. In the Super10/XC16x a user stack is allocated for each local bank. The user stack grows from high to low. The user stack is always located in near memory, the maximum size depends on the chosen memory model. The DPP register used for the user stack is determined at link time.

The stack pointer always refers to the last occupied slot. Meaning that the stack pointer first has to be decreased before data can be stored. A typical stack frame is outlined in the following picture:

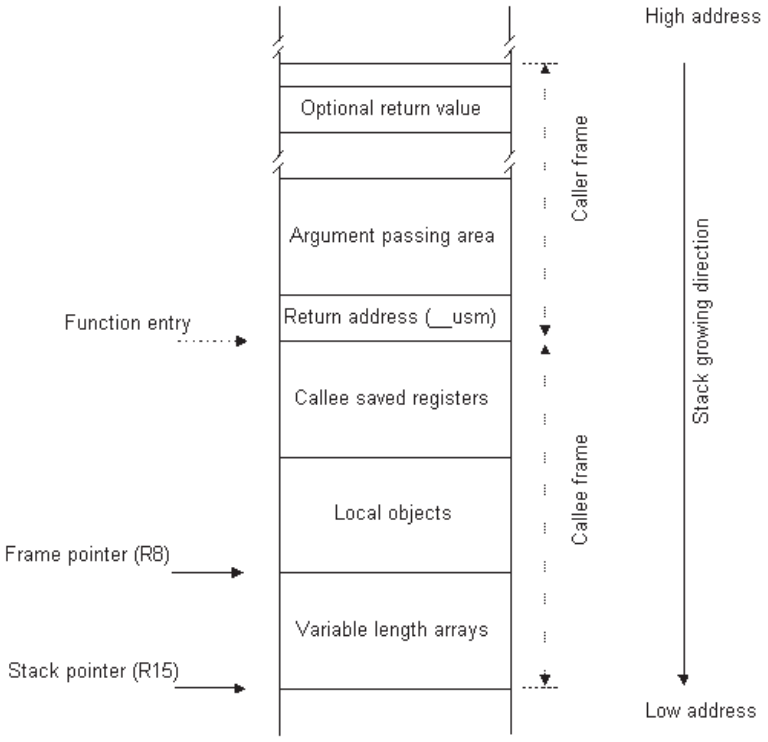


Figure F.1: Tasking VX-toolset for C166 user stack frame [103]

Before a function call, the caller pushes the required parameters on the stack. This area is called the argument passing area. For user stack functions the return address is saved on the user stack. After the call has been made, the callee will save the used callee-saved registers in the "callee saved" area. Next, the space for the local objects is allocated. After this, variable length arrays (VLAs) can be allocated. If VLAs are used in a function, register R8 is used to access the local objects and stack parameters. If no VLAs are used, R8 is available for other purposes. When the called function returns an object > 64 bit on the stack, the caller must reserve a stack area to hold the return value. After the function call, the caller must release this stack area. This also applies to the argument passing area. After the stack frame has been built, the stack pointer points to the argument passing area.

F.3 Section 1.12.2 Register Usage

The C compiler uses the general purpose registers according to the convention given in the following table.

Table F.3: Tasking C-Compiler calling convention - register usage [103]

Register	**Class**	**Purpose**
USR0	caller saves	Parameter passing and return values
R0, RL0, RH0	callee saves	Automatic variables
R1, RL1, RH1	callee saves	Automatic variables
R2, RL2, RH2	caller saves	Parameter passing and return values
R3, RL3, RH3	caller saves	Parameter passing and return values
R4, RL4, RH4	caller saves	Parameter passing and return values
R5, RL5, RH5	caller saves	Parameter passing and return values
R6, RL6, RH6	callee saves	Automatic variables
R7, RL7, RH7	callee saves	Automatic variables
R8	callee saves	Automatic variables, User stack frame pointer
R9	callee saves	Automatic variables
R10	callee saves	Automatic variables
R11	caller saves	Parameter passing
R12	caller saves	Parameter passing
R13	caller saves	Parameter passing
R14	caller saves	Parameter passing, return buffer pointer
R15	dedicated	User stack pointer

The registers are classified: caller saves, callee saves and dedicated.

caller saves These registers are allowed to be changed by a function without saving the contents. Therefore, the calling function must save these registers when necessary prior to a function call.

callee saves These registers must be saved by the called function, i.e. the caller expects them not to be changed after the function call.

dedicated The user stack pointer register R15 is dedicated.

The user stack frame pointer register R8 is used for functions containing variable length arrays.

Registers R0, R1, R2 and R3 can be used directly in an arithmetic instruction like:
`ADD Rx, [R0]`

Appendix G

FreeRTOS Port Files

This appendix contains the source files of the FreeRTOS port to the Infineon C166S v2 Core.

G.1 Linker Script Language File project.lsl

```
//
//  Linker script file for the VX-toolset for C166
//

// Define the near page addresses. Each DPP will point to a near page.
// __DPP3_ADDR is fixed at 0x00C000 (not being overwritten in cstart.c)
#define __DPP0_ADDR 0xC00000    /* [0xC00000..0xC03FFF] FLASH0 (Vector Table) */
#define __DPP1_ADDR 0xE00000    /* [0xE00000..0x003FFF] PSRAM */
#define __DPP2_ADDR 0x008000    /* [0x008000..0x00BFFF] DSRAM */
#define __DPP3_ADDR 0x00C000    /* [0x00C000..0x00FFFF] DSRAM, XSFR, ESFR, DPRAM, SFR */

#include <cpu.lsl>

// Define interrupt vector table
section_setup ::code
{
    vector_table "vector_table" ( vector_size = 4, size = 128, run_addr = 0xC00000,
                template="__vector_template", template_symbol="__lc_vector_target",
                vector_prefix=".vector.", fill = loop)
    {
        vector (id=0, fill="__cstart");
    }
}

// Define the system stack
section_layout ::shuge (direction = high_to_low)
{
    /*
     * Locate it in the DPRAM memory area, as small as possible, just used during
     * startup of FreeRTOS. Enable configCHECK_FOR_SYS_STACK_OVERFLOW to guard
     * against system stack overflows during startup.
     */
    group ( run_addr = [0xF600..0xFC00], ordered ) stack "system_stack" ( size = 256 );
}

// Define the user stack (force linker to use DPP1 for user stack)
section_layout ::near
{
    /*
     * Locate it in the PSRAM memory area. Used for ISRs using local
     * register bank 1. Enable configCHECK_FOR_SYS_STACK_OVERFLOW to guard
     * against user stack overflows during startup.
     *
     * NOTE: do not place at start of page, see TASKING VX-toolset for C166 User Guide,
     * section "The Architecture Definition", subsection "Address spaces"
     *
     * Approx. 250 bytes free for use by nested ISRs
     */
    group( run_addr = [0xE00002..0xE00100], ordered ) stack "user_stack" ( size = 254 );
}
```

Listing G.1: Linker Script Language File *project.lsl*

G.2 Portable Layer Files

G.2.1 portmacro.h

Listing G.2: Portable Layer File *portmacro.h*

```
/*
    FreeRTOS V7.3.0 - Copyright (C) 2012 Real Time Engineers Ltd.

    FEATURES AND PORTS ARE ADDED TO FREERTOS ALL THE TIME.  PLEASE VISIT
    http://www.FreeRTOS.org TO ENSURE YOU ARE USING THE LATEST VERSION.

    ***************************************************************************
     *                                                                         *
     *    FreeRTOS tutorial books are available in pdf and paperback.          *
     *    Complete, revised, and edited pdf reference manuals are also         *
     *    available.                                                           *
     *                                                                         *
     *    Purchasing FreeRTOS documentation will not only help you, by         *
     *    ensuring you get running as quickly as possible and with an          *
     *    in-depth knowledge of how to use FreeRTOS, it will also help         *
     *    the FreeRTOS project to continue with its mission of providing       *
     *    professional grade, cross platform, de facto standard solutions      *
     *    for microcontrollers - completely free of charge!                    *
     *                                                                         *
     *    >>> See http://www.FreeRTOS.org/Documentation for details. <<<       *
     *                                                                         *
     *    Thank you for using FreeRTOS, and thank you for your support!        *
     *                                                                         *
    ***************************************************************************


    This file is part of the FreeRTOS distribution.

    FreeRTOS is free software; you can redistribute it and/or modify it under
    the terms of the GNU General Public License (version 2) as published by the
    Free Software Foundation AND MODIFIED BY the FreeRTOS exception.
    >>>NOTE<<< The modification to the GPL is included to allow you to
    distribute a combined work that includes FreeRTOS without being obliged to
    provide the source code for proprietary components outside of the FreeRTOS
    kernel.  FreeRTOS is distributed in the hope that it will be useful, but
    WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY
    or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License for
    more details. You should have received a copy of the GNU General Public
    License and the FreeRTOS license exception along with FreeRTOS; if not it
    can be viewed here: http://www.freertos.org/a00114.html and also obtained
    by writing to Richard Barry, contact details for whom are available on the
    FreeRTOS WEB site.

    1 tab == 4 spaces!

    ***************************************************************************
     *                                                                         *
     *    Having a problem?  Start by reading the FAQ "My application does     *
     *    not run, what could be wrong?"                                       *
     *                                                                         *
     *    http://www.FreeRTOS.org/FAQHelp.html                                 *
     *                                                                         *
    ***************************************************************************


    http://www.FreeRTOS.org - Documentation, training, latest versions, license
    and contact details.

    http://www.FreeRTOS.org/plus - A selection of FreeRTOS ecosystem products,
    including FreeRTOS+Trace - an indispensable productivity tool.

    Real Time Engineers ltd license FreeRTOS to High Integrity Systems, who sell
    the code with commercial support, indemnification, and middleware, under
    the OpenRTOS brand: http://www.OpenRTOS.com.  High Integrity Systems also
    provide a safety engineered and independently SIL3 certified version under
    the SafeRTOS brand: http://www.SafeRTOS.com.
*/


#ifndef PORTMACRO_H
#define PORTMACRO_H

#ifdef __cplusplus
extern "C" {
#endif

/*-----------------------------------------------------------
 * Port specific definitions.
 *
```

```
80  * The settings in this file configure FreeRTOS correctly for the
81  * given hardware and compiler.
82  *
83  * These settings should not be altered.
84  *-----------------------------------------------------------
85  */
86
87 #if !defined(__TASKING__) || (__C166__ != 1) || !defined(__CORE_XC16X__) || !defined(__VX__)
88 #error Wrong compiler and/or wrong architecture
89 #endif
90
91 #if (__MODEL__ != 'h')
92 #error Wrong memory model selected. MUST use the huge model!
93 #endif
94
95 /* include SFR file for selected CPU */
96 #ifndef __TASKING_SFR__
97 # ifdef __CPU__
98 #  include __SFRFILE__(__CPU__)
99 # else
100 #  error Tasking SFRs are not included and CPU is not defined!
101 # endif
102 #endif
103 #include "STM/STM.h"
104
105 /* Type definitions. */
106 #define portCHAR        char
107 #define portFLOAT       float
108 #define portDOUBLE      double
109 #define portLONG        long
110 #define portSHORT       short
111 #define portSTACK_TYPE  unsigned portSHORT
112 #define portBASE_TYPE   portSHORT
113
114 #if( configUSE_16_BIT_TICKS == 1 )
115     typedef unsigned portSHORT portTickType;
116     #define portMAX_DELAY ( portTickType ) 0xffff
117 #else
118     typedef unsigned portLONG portTickType;
119     #define portMAX_DELAY ( portTickType ) 0xffffffff
120 #endif
121
122 #if ( configGENERATE_RUN_TIME_STATS == 1 )
123     typedef unsigned portLONG portStatsTickType;
124 #endif
125 /*-----------------------------------------------------------*/
126
127 /* Architecture specifics. */
128 #define portSTACK_GROWTH            ( -1 )
129 #define portTICK_RATE_MS            ( ( portTickType ) 1000 / configTICK_RATE_HZ )
130 #define portBYTE_ALIGNMENT          2
131
132 #define portHEAP_ABSOLUTE_ADDR      configHEAP_ABSOLUTE_ADDR
133
134 #define portSTACK2                  1
135
136 #if (__USER_STACK__ == 1)
137 #error Option "Use user stack for return addresses" is not supported!
138 #endif
139
140 #if !defined(configMINIMAL_STACK_SIZE) || !defined(configMINIMAL_STACK2_SIZE) || !defined(configHEAP_ABSOLUTE_ADDR)
141 #error This port needs 2 stacks. Define configMINIMAL_STACK_SIZE, configMINIMAL_STACK2_SIZE and
        configHEAP_ABSOLUTE_ADDR accrodingly!
142 #endif
143
144 /*-----------------------------------------------------------*/
145
146 /* Scheduler utilities. */
147
148 /* Task utilities. */
149
150 /*
151  * - in non-preempted mode, define yield as software trap
152  *   to correctly handle PSW, CSP and IP upon context switch
153  * - in preempted mode, define yield as a simulated tick isr
154  *   to correctly handle PSW, CSP and IP upon context switch
155  */
156 #if configUSE_PREEMPTION == 0
157 #define portYIELD()                 __int166( 127 )
158 #else
159 #define portYIELD()                 STM_vSTM1Trap()
160 #endif
161
162 #define portYIELD_FROM_ISR(SwitchRequired) \
163     { if (SwitchRequired) portYIELD();  }
164
165 /*
166  * define startFirstTask() as software trap to correctly handle
```

```
167  * PSW, CSP and IP upon context switch
168  */
169 #define portSTART_FIRST_TASK()      __int166( 126 )
170
171 /*-----------------------------------------------------------*/
172
173 /*
174  * fence macro needed to tell Tasking Compiler not to move
175  * code around
176  */
177 #define __fence()                   __asm ( "" )
178
179 /* Critical section management. */
180 inline void vPortEnterCritical( void );
181 inline void vPortExitCritical( void );
182 #define portENTER_CRITICAL()        vPortEnterCritical()
183 #define portEXIT_CRITICAL()         vPortExitCritical()
184
185 inline void vPortDisableInterrupts( void );
186 inline void vPortEnableInterrupts( void );
187 #define portDISABLE_INTERRUPTS()    vPortDisableInterrupts()
188 #define portENABLE_INTERRUPTS()     vPortEnableInterrupts()
189
190 inline unsigned portBASE_TYPE xPortSetInterruptMaskFromISR( unsigned portBASE_TYPE uxNewIntLevelValue );
191 inline void vPortClearInterruptMaskFromISR( unsigned portBASE_TYPE uxSavedIntLevelValue );
192 #define portSET_INTERRUPT_MASK_FROM_ISR() \
193     xPortSetInterruptMaskFromISR(configMAX_SYSCALL_INTERRUPT_PRIORITY)
194 #define portCLEAR_INTERRUPT_MASK_FROM_ISR( uxSavedStatusValue ) \
195     vPortClearInterruptMaskFromISR( uxSavedStatusValue )
196 /*-----------------------------------------------------------*/
197
198 extern volatile unsigned portBASE_TYPE uxCriticalNesting;
199
200 inline void vPortEnterCritical( void )
201 {
202     __fence();  /* fence compiler */
203
204     if (uxCriticalNesting == 0)
205     {
206         /* Disable all interrupts under RTOS control */
207         portSET_INTERRUPT_MASK_FROM_ISR();
208     }
209
210     /*
211      * Now interrupts are disabled ulCriticalNesting can be accessed
212      * directly.  Increment ulCriticalNesting to keep a count of how many
213      * times portENTER_CRITICAL() has been called.
214      */
215     uxCriticalNesting++;
216
217     __fence();  /* fence compiler */
218 }
219
220 inline void vPortExitCritical( void )
221 {
222     __fence();  /* fence compiler */
223
224     if (uxCriticalNesting > 0)
225     {
226         /* Decrement the nesting count as we are leaving a critical section. */
227         uxCriticalNesting--;
228
229         /*
230          * If the nesting level has reached zero then interrupts should be
231          * re-enabled.
232          */
233         if( uxCriticalNesting == 0 )
234         {
235             portCLEAR_INTERRUPT_MASK_FROM_ISR(
236                 configTASK_LEVEL_INTERRUPT_PRIORITY);
237         }
238     }
239
240     __fence();  /* fence compiler */
241 }
242 /*-----------------------------------------------------------*/
243
244 inline void vPortDisableInterrupts( void )
245 {
246     __fence();  /* fence compiler */
247
248     PSW_IEN = 0;
249
250     __fence();  /* fence compiler */
251 }
252 /*-----------------------------------------------------------*/
253
254 inline void vPortEnableInterrupts( void )
```

```
255 {
256     __fence();  /* fence compiler */
257
258     PSW_IEN = 1;
259
260     __fence();  /* fence compiler */
261 }
262 /*-----------------------------------------------------------*/
263
264 inline unsigned portBASE_TYPE xPortSetInterruptMaskFromISR( unsigned portBASE_TYPE uxNewIntLevelValue )
265 {
266 register unsigned portBASE_TYPE uxSavedIntLevelValue;
267
268     __fence();  /* fence compiler */
269
270     // store current interrupt level
271     uxSavedIntLevelValue = PSW_ILVL;
272
273     // set to highest system call level
274     PSW_ILVL = uxNewIntLevelValue;
275
276     __fence();  /* fence compiler */
277
278     return uxSavedIntLevelValue;
279 }
280 /*-----------------------------------------------------------*/
281
282 inline void vPortClearInterruptMaskFromISR( unsigned portBASE_TYPE uxSavedIntLevelValue )
283 {
284     __fence();  /* fence compiler */
285
286     // restore interrupt level
287     PSW_ILVL = uxSavedIntLevelValue;
288
289     __fence();  /* fence compiler */
290 }
291 /*-----------------------------------------------------------*/
292
293 /* Task function macros as described on the FreeRTOS.org WEB site. */
294 #define portTASK_FUNCTION_PROTO( vFunction, pvParameters ) void vFunction( void *pvParameters )
295 #define portTASK_FUNCTION( vFunction, pvParameters ) void vFunction( void *pvParameters )
296
297 /* ISR function macros - see INT/INT.h for vectors defined */
298 #define portINTERRUPT_HANDLER_PROTO( vIsrFunction, vector ) __interrupt(vector) void vIsrFunction ( void )
299 #define portINTERRUPT_HANDLER( vIsrFunction, vector ) __interrupt(vector) void vIsrFunction ( void )
300
301 #define portINTERRUPT_HANDLER_BANKSEL_PROTO( vIsrFunction, vector, bank ) __registerbank(bank) __interrupt(vector)
       void vIsrFunction ( void )
302 #define portINTERRUPT_HANDLER_BANKSEL( vIsrFunction, vector, bank ) __registerbank(bank) __interrupt(vector) void
      vIsrFunction ( void )
303
304 /*-----------------------------------------------------------*/
305
306 /* Miscellaneous functions */
307 #define portNOP()                           __nop()
308 #define portPUSH(val)                       { __asm( "push  %0 \n" : : "=w" (val) ); }
309 #define portPOP(val)                        { __asm( "pop   %0 \n" : "=w" (val) : ); }
310 #define portPushToStack(stack, val)         {stack--; *stack = val;}
311 #define portPopFromStack(stack, val)        {val = *stack; stack++;}
312 #define portPushToStackCnt(stack, val, cnt) { \
313     for(unsigned portSHORT __uxPushToStackCnt=0; \
314         __uxPushToStackCnt < (unsigned portSHORT)cnt; \
315         __uxPushToStackCnt++) \
316             portPushToStack(stack, val); \
317 }
318
319 #define __atomic(number)                    __asm( "$nowarning(735)" ); __asm( "atomic  #" # number " " )
320 #define __endatomic()                       __asm( "$warning(735)" );
321
322 #define __switchregbank(number)             PSW_BANK = number + ((number > 0) ? 1 : 0)
323 #define __dpof(dpp,address)                 ((((unsigned int)(dpp) & 0x3) << 14) | __pof(address))
324 #define __mknp(dpp,address)                 ((void __near *)__dpof(dpp,address))
325 /*-----------------------------------------------------------*/
326
327 /* Run time stats functions */
328 #if ( configGENERATE_RUN_TIME_STATS == 1 )
329 extern void vPortConfigStatsTimer( void );
330 extern portStatsTickType uSystemTotalRunTime;
331 #define portLU_PRINTF_SPECIFIER_REQUIRED    1   /* sizeof( int ) != sizeof( long ) */
332 #define portCONFIGURE_TIMER_FOR_RUN_TIME_STATS() \
333     vPortConfigStatsTimer()
334 #define portALT_GET_RUN_TIME_COUNTER_VALUE(ulTotalRunTime) \
335     { \
336         portDISABLE_INTERRUPTS(); \
337         ulTotalRunTime = uSystemTotalRunTime; \
338         portENABLE_INTERRUPTS(); \
339     }
340 #endif /* ( configGENERATE_RUN_TIME_STATS == 1 ) */
```

```
341
342
343 #ifdef __cplusplus
344 }
345 #endif
346
347 #endif /* PORTMACRO_H */
```

Listing G.2: Portable Layer File *portmacro.h*

G.2.2 port.c

Listing G.3: Portable Layer File *port.c*

```
/*
    FreeRTOS V7.3.0 - Copyright (C) 2012 Real Time Engineers Ltd.

    FEATURES AND PORTS ARE ADDED TO FREERTOS ALL THE TIME.  PLEASE VISIT
    http://www.FreeRTOS.org TO ENSURE YOU ARE USING THE LATEST VERSION.

    ***************************************************************************
     *                                                                         *
     *    FreeRTOS tutorial books are available in pdf and paperback.          *
     *    Complete, revised, and edited pdf reference manuals are also         *
     *    available.                                                           *
     *                                                                         *
     *    Purchasing FreeRTOS documentation will not only help you, by         *
     *    ensuring you get running as quickly as possible and with an          *
     *    in-depth knowledge of how to use FreeRTOS, it will also help         *
     *    the FreeRTOS project to continue with its mission of providing       *
     *    professional grade, cross platform, de facto standard solutions      *
     *    for microcontrollers - completely free of charge!                    *
     *                                                                         *
     *    >>> See http://www.FreeRTOS.org/Documentation for details. <<<      *
     *                                                                         *
     *    Thank you for using FreeRTOS, and thank you for your support!        *
     *                                                                         *
    ***************************************************************************


    This file is part of the FreeRTOS distribution.

    FreeRTOS is free software; you can redistribute it and/or modify it under
    the terms of the GNU General Public License (version 2) as published by the
    Free Software Foundation AND MODIFIED BY the FreeRTOS exception.
    >>>NOTE<<< The modification to the GPL is included to allow you to
    distribute a combined work that includes FreeRTOS without being obliged to
    provide the source code for proprietary components outside of the FreeRTOS
    kernel.  FreeRTOS is distributed in the hope that it will be useful, but
    WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY
    or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License for
    more details. You should have received a copy of the GNU General Public
    License and the FreeRTOS license exception along with FreeRTOS; if not it
    can be viewed here: http://www.freertos.org/a00114.html and also obtained
    by writing to Richard Barry, contact details for whom are available on the
    FreeRTOS WEB site.

    1 tab == 4 spaces!

    ***************************************************************************
     *                                                                         *
     *    Having a problem?  Start by reading the FAQ "My application does     *
     *    not run, what could be wrong?"                                       *
     *                                                                         *
     *    http://www.FreeRTOS.org/FAQHelp.html                                 *
     *                                                                         *
    ***************************************************************************


    http://www.FreeRTOS.org - Documentation, training, latest versions, license
    and contact details.

    http://www.FreeRTOS.org/plus - A selection of FreeRTOS ecosystem products,
    including FreeRTOS+Trace - an indispensable productivity tool.

    Real Time Engineers ltd license FreeRTOS to High Integrity Systems, who sell
    the code with commercial support, indemnification, and middleware, under
    the OpenRTOS brand: http://www.OpenRTOS.com.  High Integrity Systems also
    provide a safety engineered and independently SIL3 certified version under
    the SafeRTOS brand: http://www.SafeRTOS.com.
*/

/*-----------------------------------------------------------
 * Implementation of functions defined in portable.h for the INFINEON C166S V2 port.
 *----------------------------------------------------------*/

/* Standard includes. */
#include <string.h>

/* Define TASK_INCLUDED_FROM_KERNEL_FILE to import the definition of the task
control block (TCB). That should only be done when task.h is included from a
kernel or port layer file. */
#define TASK_INCLUDED_FROM_KERNEL_FILE

/* Scheduler includes. */
#include "FreeRTOS.h"
#include "task.h"
```

```
84  #include "StackMacros.h"
85
86  #undef TASK_INCLUDED_FROM_KERNEL_FILE
87
88  /* Platform includes */
89  #include "STM/STM.h"
90  #include "IO/IO.h"
91
92  #if ( configGENERATE_RUN_TIME_STATS == 1 )
93  // uSystemTotalRunTime can count to
94  portStatsTickType uSystemTotalRunTime = 0;
95  #endif /* ( configGENERATE_RUN_TIME_STATS == 1 ) */
96
97  /* We require the address of the pxCurrentTCB variable. */
98  extern tskTCB * volatile pxCurrentTCB;
99
100 /*
101  * Configure a timer to generate the RTOS tick at the frequency specified
102  * within FreeRTOSConfig.h.
103  */
104 static void prvSetupTimerInterrupt(unsigned short usILVL, unsigned short usXGLVL);
105
106 /* Calls to portENTER_CRITICAL() can be nested.  When they are nested the
107 critical section should not be left (i.e. interrupts should not be re-enabled)
108 until the nesting depth reaches 0.  This variable simply tracks the nesting
109 depth. Initialize uxCriticalNesting to zero. */
110 volatile unsigned portBASE_TYPE uxCriticalNesting = 0;
111
112 /*-----------------------------------------------------------------------------------------------*/
113
114 /*
115  * Pointers to top of system and user stack, saved by vPortStartFirstTask()
116  * and restored by vPortEndScheduler()
117  */
118 portSTACK_TYPE *pxTopOfSystemStack  = NULL;
119 portSTACK_TYPE *pxTopOfUserStack    = NULL;
120
121 /*-----------------------------------------------------------------------------------------------*/
122
123 /* support for runtime system stack overflow checking */
124 #if configCHECK_FOR_BOOT_STACK_OVERFLOW == 1
125     signed char pcBootTaskName[configMAX_TASK_NAME_LEN] = "BOOT";
126     #if configCHECK_FOR_STACK_OVERFLOW >= 1
127         extern __huge   char    _lc_ue_system_stack[];  // bottom of system stack
128     #endif
129     #if configCHECK_FOR_STACK2_OVERFLOW >= 1
130         extern __near   char    _lc_ue_user_stack[];    // bottom of user stack
131     #endif
132 #endif
133
134 /*-----------------------------------------------------------------------------------------------*/
135
136 /*
137  * Inline function to save a task context to the task stack.  This simply
138  * pushes all the general purpose registers onto the stack. Finally the
139  * resultant stack pointer value is saved into the task control block so
140  * it can be retrieved the next time the task executes.
141  */
142 void __always_inline__ portSAVE_CONTEXT( void )
143 {
144 register portSTACK_TYPE * pxTopOfStack;
145
146     /*
147      * PSW, CSP and IP are already on the stack by a call to
148      * an ISR or a Software Trap
149      */
150     __asm ( "\n"
151             " push DPP0 \n"
152             " push DPP1 \n"
153             " push DPP2 \n"
154             " push DPP3 \n"
155             " push r15 \n"
156             " push r14 \n"
157             " push r13 \n"
158             " push r12 \n"
159             " push r11 \n"
160             " push r10 \n"
161             " push r9 \n"
162             " push r8 \n"
163             " push r7 \n"
164             " push r6 \n"
165             " push r5 \n"
166             " push r4 \n"
167             " push r3 \n"
168             " push r2 \n"
169             " push r1 \n"
170             " push r0 \n"
171             " push MDC \n"      // MDx registers are only safe to push
```

```
172             " push MDH \n"      // if at least 17 cycles have passed
173             " push MDL \n" );
174
175     /* update RTOS stack pointers */
176
177     // system stack pointer
178     pxTopOfStack = __mkhp(SP, SPSEG);
179     pxCurrentTCB->pxTopOfStack  = pxTopOfStack;
180
181     // user stack pointer
182     pxTopOfStack = __mkfp((unsigned int)__getsp(), DPP1);
183     pxCurrentTCB->pxTopOfStack2 = pxTopOfStack;
184 }
185
186 /*
187  * Inline function to restore a task context from the task stack.  This is
188  * effectively the reverse of portSAVE_CONTEXT().  First the stack pointer
189  * value is loaded from the task control block.  Next the value of all the
190  * general purpose registers are retrieved.
191  */
192
193 void __always_inline__ portRESTORE_CONTEXT( void )
194 {
195 register portSTACK_TYPE * pxTopOfStack;
196
197     pxTopOfStack = (portSTACK_TYPE *)pxCurrentTCB->pxTopOfStack;
198
199     // let CPU stack pointer point to stack of task to be restored (write atomically)
200     __atomic(2);
201     SP    = __sof(pxTopOfStack);
202     SPSEG = __seg(pxTopOfStack);
203     __endatomic();              // used as fence for the compiler
204
205     // restore registers saved by portSAVE_CONTEXT
206     __asm ( "\n"
207             " pop MDL \n"
208             " pop MDH \n"
209             " pop MDC \n"
210             " pop r0 \n"
211             " pop r1 \n"
212             " pop r2 \n"
213             " pop r3 \n"
214             " pop r4 \n"
215             " pop r5 \n"
216             " pop r6 \n"
217             " pop r7 \n"
218             " pop r8 \n"
219             " pop r9 \n"
220             " pop r10 \n"
221             " pop r11 \n"
222             " pop r12 \n"
223             " pop r13 \n"
224             " pop r14 \n"
225             " atomic #4 \n" // atomically restore user stack pointer
226             " pop r15 \n"
227             " pop DPP3 \n"
228             " pop DPP2 \n"
229             " pop DPP1 \n"
230             " pop DPP0 \n"
231             "\n" );
232
233     /*
234      * IP, CSP and PSW are restored by the RETI instruction at the
235      * end of the ISR/Software Trap
236      */
237 }
238
239 /*-----------------------------------------------------------------------------*/
240
241 /*
242  * Setup the stack of a new task so it is ready to be placed under the
243  * scheduler control.  The registers have to be placed on the stack in
244  * the order that the port expects to find them.
245  *
246  */
247 portSTACK_TYPE *pxPortInitialiseStack( portSTACK_TYPE *pxTopOfStack,  portSTACK_TYPE *pxTopOfStack2, pdTASK_CODE
        pxCode, void *pvParameters )
248 {
249 __PSW_type initialPSW;
250
251     /*
252      * Place a few bytes of known values on the bottom of the stack.
253      * This is just useful for debugging and can be included if required.
254      */
255     /*
256     portPushToStack(pxTopOfStack, ( portSTACK_TYPE ) 0x1111U);
257     portPushToStack(pxTopOfStack, ( portSTACK_TYPE ) 0x2222U);
258     portPushToStack(pxTopOfStack, ( portSTACK_TYPE ) 0x3333U);
```

```
259     */
260
261     /*
262      * Simulate the stack frame as it would be created by a context
263      * switch interrupt.
264      *
265      *           System Stack (FreeRTOS Stack1)
266      *            +-------+
267      *            | xxxxx | <-- Stack Pointer (SP) before
268      *            |  PSW  |
269      *            |  CSP  |
270      *            |  IP   | <-- SP after ISR/TRAP called
271      *            |  DPP0 | --> from here registers saved
272      *            |  DPP1 |     by portSAVE_CONTEXT()
273      *            |  DPP2 |
274      *            |  DPP3 |
275      *            |  R15  | dedicated to User Stack Pointer[1]
276      *            |  R14  |
277      *            |  R13  |
278      *            |  R12  |
279      *            |  R11  |
280      *            |  R10  |
281      *            |  R9   |
282      *            |  R8   |
283      *            |  R7   |
284      *            |  R6   |
285      *            |  R5   |
286      *            |  R4   |
287      *            |  R3   | \
288      *            |  R2   | / first 32-bit parameter (pvParameters*)[1]
289      *            |  R1   |
290      *            |  R0   |
291      *            |  MDC  |
292      *            |  MDH  |
293      *            |  MDL  | <-- SP after portSAVE_CONTEXT()
294      *            +-------+
295      *
296      *  [1] ... per TASKING C-compiler calling convention
297      *
298      * The C166S V2 automatically pushes the PSW followed by the CSP
299      * and then IP onto the stack before executing an ISR.
300      *
301      */
302
303     // PSW - processor status word
304     initialPSW.U = 0;       // reset initialization value
305     initialPSW.B.ien = 1;   // enable interrupts
306     portPushToStack(pxTopOfStack, initialPSW.U);
307     // CSP - code segment pointer
308     portPushToStack(pxTopOfStack, __seg(pxCode));
309     // IP - instruction pointer
310     portPushToStack(pxTopOfStack, __sof(pxCode));
311
312     // initialize registers DPP0 to DPP3
313     // DPP1 is used to access the user stack (stack2)
314     // see the project LSL file how this is accomplished
315     portPushToStack(pxTopOfStack, DPP0);
316     portPushToStack(pxTopOfStack, __pag(pxTopOfStack2));
317     portPushToStack(pxTopOfStack, DPP2);
318     portPushToStack(pxTopOfStack, DPP3);
319
320     // initialize general purpose register R15
321     // as user stack pointer (stack2), per TASKING
322     // C-compiler calling convention, use DPP1
323     portPushToStack(pxTopOfStack, __dpof(1, pxTopOfStack2));
324
325     // initialize general purpose registers R14 to R4
326     portPushToStackCnt(pxTopOfStack, 0x00, 11);
327
328     // initialize general purpose registers R3 and R2 as
329     // input parameter pvParameters
330     // ATTENTION: we assume the huge memory model here!
331     portPushToStack(pxTopOfStack, __seg(pvParameters));
332     portPushToStack(pxTopOfStack, __sof(pvParameters));
333
334     // initialize registers R1 and R0
335     portPushToStackCnt(pxTopOfStack, 0x00, 2);
336
337     // initialize multiply/divide unit registers MDC, MDH and MDL
338     portPushToStackCnt(pxTopOfStack, 0x00, 3);
339
340     return pxTopOfStack;
341 }
342 /*-----------------------------------------------------------------------------*/
343
344 /*
345  * See header file for description.
346  */
```

```
347 portBASE_TYPE xPortStartScheduler( void )
348 {
349     /*
350      * configMAX_SYSCALL_INTERRUPT_PRIORITY must not be set to 0,
351      * and it must be greater or equal to configKERNEL_INTERRUPT_PRIORITY.
352      */
353     configASSERT( ( configMAX_SYSCALL_INTERRUPT_PRIORITY ) );
354     configASSERT( ( configMAX_SYSCALL_INTERRUPT_PRIORITY >= configKERNEL_INTERRUPT_PRIORITY ) );
355 
356     /*
357      * Start the timer that generates the tick ISR at the kernel
358      * interrupts priority. Interrupts are disabled here already.
359      */
360     prvSetupTimerInterrupt(configKERNEL_INTERRUPT_PRIORITY, 0);
361 
362     /* Start first task. */
363     portSTART_FIRST_TASK();
364 
365     /* Should never reach here. */
366     return pdFALSE;
367 }
368 /*-----------------------------------------------------------------------------------------*/
369 
370 void vPortEndScheduler( void )
371 {
372 register unsigned portSHORT reg;
373 
374     /* restore system stack pointer */
375     SP    = __sof(pxTopOfSystemStack);
376     SPSEG = __seg(pxTopOfSystemStack);
377 
378     /* restore user stack pointer */
379     DPP1  = __pag(pxTopOfUserStack);
380     __setsp(__mknp(1, pxTopOfUserStack));
381 
382     /* stop RTOS timer tick */
383     STM_vStopSTM();
384 
385     /* portSTART_FIRST_TASK() placed IP and CSP on the stack, remove them */
386     portPOP(reg);
387     portPOP(reg);
388 
389     /* simulate return value of xPortStartScheduler() */
390     reg = pdPASS;
391     __asm ( " movw r2,%0 \n"        // set function return value
392             :: "w" (reg) );
393 
394     /* portSTART_FIRST_TASK() placed PSW on the stack, restore it */
395     portPOP(PSW);
396 }
397 /*-----------------------------------------------------------------------------------------*/
398 
399 // __frame() does what __naked__ usually does, but gives a warning.
400 #pragma warning 796
401 __interrupt(TRAP126_VECT) __frame() void vPortStartFirstTask( void )
402 {
403 register portSTACK_TYPE * pxStack;
404 register __near char * us;
405 
406     /* interrupts are still disabled */
407 
408     /* save TopOfStack for system and user stack */
409     pxTopOfSystemStack  = __mkhp(SP, SPSEG);
410     pxTopOfUserStack    = __mkfp((unsigned int)__getsp(), DPP1);
411 
412     /* switch to local register bank 1 */
413     __switchregbank(1);
414 
415     /* set user stack pointer of local register bank (using DPP1) */
416     __setsp(__mknp(1, pxTopOfUserStack));
417 
418     #if ( configCHECK_FOR_BOOT_STACK_OVERFLOW == 1 )
419         /* NOTE: The following code depends on the initialisation of system and
420         user stack in cstart.c */
421 
422         /* check system stack */
423         pxStack = (portSTACK_TYPE *)_lc_ue_system_stack;
424         taskFIRST_CHECK_FOR_STACK_OVERFLOW( pxStack, pxTopOfSystemStack, NULL, vApplicationStackOverflowHook, NULL,
                pcBootTaskName );
425         taskSECOND_CHECK_FOR_STACK_OVERFLOW( pxStack, pxTopOfSystemStack, NULL, vApplicationStackOverflowHook, NULL,
                pcBootTaskName );
426 
427         /* check user stack */
428         pxStack = __mkfp((unsigned int)_lc_ue_user_stack, DPP1);
429         taskFIRST_CHECK_FOR_STACK_OVERFLOW( pxStack, pxTopOfUserStack, NULL, vApplicationStack2OverflowHook, NULL,
                pcBootTaskName );
430         taskSECOND_CHECK_FOR_STACK_OVERFLOW( pxStack, pxTopOfUserStack, NULL, vApplicationStack2OverflowHook, NULL,
                pcBootTaskName );
```

```
431     #endif /* ( configCHECK_FOR_SYS_STACK_OVERFLOW == 1 ) */
432
433     /* Initialise unused part of user stack */
434     us = (__near char *)__getsp();
435     do
436     {
437             *(--us) = 0xa5;            // initialise with fill value
438     }
439     while (us != _lc_ue_user_stack);   // bottom of user stack reached?
440
441 #if 1
442     /* initialise all registers of local register bank */
443     __asm ( "\n"
444             "       movw   r0,#0 \n"
445             "       movw   r1,#1 \n"
446             "       movw   r2,#2 \n"
447             "       movw   r3,#3 \n"
448             "       movw   r4,#4 \n"
449             "       movw   r5,#5 \n"
450             "       movw   r6,#6 \n"
451             "       movw   r7,#7 \n"
452             "       movw   r8,#8 \n"
453             "       movw   r9,#9 \n"
454             "       movw   r10,#10 \n"
455             "       movw   r11,#11 \n"
456             "       movw   r12,#12 \n"
457             "       movw   r13,#13 \n"
458             "       movw   r14,#14 \n" );
459 #endif
460
461     /* switch to global register bank */
462     __switchregbank(0);
463
464     /* restore context of first task */
465     portRESTORE_CONTEXT();
466
467     /* interrupts are enabled by returning to the first task */
468 }
469 #pragma warning restore
470 /*-----------------------------------------------------------*/
471
472 #if configUSE_PREEMPTION == 0
473     // __frame() does what __naked__ usually does, but gives a warning.
474     #pragma warning 796
475     __interrupt(TRAP127_VECT) __frame() void vPortYield( void )
476     {
477         /* save current task's context */
478         portSAVE_CONTEXT();
479
480         /* do the context switch */
481         vTaskSwitchContext();
482
483         /* restore new task's context */
484         portRESTORE_CONTEXT();
485     }
486     #pragma warning restore
487 #endif /* configUSE_PREEMPTION == 0 */
488 /*-----------------------------------------------------------*/
489
490 /*
491  * Setup the system tick timer to generate the tick interrupts at the required
492  * frequency.
493  */
494 void prvSetupTimerInterrupt(unsigned short usILVL, unsigned short usXGLVL)
495 {
496     /* STM_vInit() must have been called during hardware setup */
497
498     /* enable STM1 interrupt node */
499     STM_vNodeEnable(STM1_NODE, usILVL, usXGLVL);
500
501     /*  enable STM1 interrupt */
502     STM_vEnableSTM1();
503
504     /* start the System Timer Module (STM) */
505     STM_vStartSTM();
506 }
507 /*-----------------------------------------------------------*/
508
509 #if ( configGENERATE_RUN_TIME_STATS == 1 )
510 portINTERRUPT_HANDLER_PROTO(STM_viSTM0I,STM0_VECT);
511 void vPortConfigStatsTimer( void )
512 {
513     /* STM_vInit() must have been called during hardware setup */
514
515     /* enable STM0 interrupt node */
516     STM_vNodeEnable(STM0_NODE, configMAX_INTERRUPT_PRIORITY, 0);
517
518     /* enable as fast interrupt */
```

```
519     STM_vNodeEnableFast(STM0_NODE, (INT_FUNC)STM_viSTM0I);
520
521     /*  enable STM0 interrupt */
522     STM_vEnableSTM0();
523
524     /* STM module will be started in prvSetupTimerInterrupt() */
525 }
526 #endif /* ( configGENERATE_RUN_TIME_STATS == 1 ) */
527 /*-----------------------------------------------------------------------------------------------*/
528
529 /*
530  * STM_viSTM1I() timer ISR.
531  *
532  * Note: This ISR has no frame so it has to
533  *  - call portSAVE_CONTEXT() first and
534  *  - call portRESTORE_CONTEXT() at the end.
535  */
536 // __frame() does what __naked__ usually does, but gives a warning.
537 #pragma warning 796
538 __interrupt(STM1_VECT) __frame() void STM_viSTM1I( void )
539 {
540 register unsigned portBASE_TYPE hw_tick;
541
542     /* if any other SCU interrupt is activated and
543     points to this interrupt, we are not interested */
544     if ( !SCU_INTSTAT_STM1I )
545     {
546         return;
547     }
548
549     /* save current task's context */
550     portSAVE_CONTEXT();
551
552     /* if set, this is a hardware tick, otherwise a software yield */
553     hw_tick = SCU_DMPMIT_STM1;
554
555     /* Clear interrupt status */
556     STM_vSTM1Clr(hw_tick);
557
558     /* hardware tick, otherwise a software yield */
559     if ( hw_tick )
560     {
561         // increment Tick
562         vTaskIncrementTick();
563     }
564
565     #if configUSE_PREEMPTION == 1
566     vTaskSwitchContext();
567     #endif /* configUSE_PREEMPTION == 1 */
568
569     /* restore new task's context */
570     portRESTORE_CONTEXT();
571 } //  End of function STM_viSTM1I
572 #pragma warning restore
573 /*-----------------------------------------------------------------------------------------------*/
574
575 /*
576  * STM_viSTM0I() timer ISR.
577  */
578 #if ( configGENERATE_RUN_TIME_STATS == 1 )
579     portINTERRUPT_HANDLER(STM_viSTM0I,STM0_VECT)
580     {
581         /* if any other SCU interrupt is activated and
582         points to this interrupt, we are not interested */
583         if ( !SCU_INTSTAT_STM0I )
584         {
585             return;
586         }
587
588         // Clear interrupt status
589         STM_vSTM0Clr();
590
591         uSystemTotalRunTime++;
592     }
593 #endif /* ( configGENERATE_RUN_TIME_STATS == 1 ) */
```

Listing G.3: Portable Layer File *port.c*

G.3 Port Configuration File FreeRTOSConfig.h

Listing G.4: Port Config File *FreeRTOSConfig.h*

```
/*
    FreeRTOS V7.3.0 - Copyright (C) 2012 Real Time Engineers Ltd.

    FEATURES AND PORTS ARE ADDED TO FREERTOS ALL THE TIME.  PLEASE VISIT
    http://www.FreeRTOS.org TO ENSURE YOU ARE USING THE LATEST VERSION.

    ***************************************************************************
     *                                                                         *
     *    FreeRTOS tutorial books are available in pdf and paperback.          *
     *    Complete, revised, and edited pdf reference manuals are also         *
     *    available.                                                           *
     *                                                                         *
     *    Purchasing FreeRTOS documentation will not only help you, by         *
     *    ensuring you get running as quickly as possible and with an          *
     *    in-depth knowledge of how to use FreeRTOS, it will also help         *
     *    the FreeRTOS project to continue with its mission of providing       *
     *    professional grade, cross platform, de facto standard solutions      *
     *    for microcontrollers - completely free of charge!                    *
     *                                                                         *
     *    >>> See http://www.FreeRTOS.org/Documentation for details. <<<       *
     *                                                                         *
     *    Thank you for using FreeRTOS, and thank you for your support!        *
     *                                                                         *
    ***************************************************************************


    This file is part of the FreeRTOS distribution.

    FreeRTOS is free software; you can redistribute it and/or modify it under
    the terms of the GNU General Public License (version 2) as published by the
    Free Software Foundation AND MODIFIED BY the FreeRTOS exception.
    >>>NOTE<<< The modification to the GPL is included to allow you to
    distribute a combined work that includes FreeRTOS without being obliged to
    provide the source code for proprietary components outside of the FreeRTOS
    kernel.  FreeRTOS is distributed in the hope that it will be useful, but
    WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY
    or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License for
    more details. You should have received a copy of the GNU General Public
    License and the FreeRTOS license exception along with FreeRTOS; if not it
    can be viewed here: http://www.freertos.org/a00114.html and also obtained
    by writing to Richard Barry, contact details for whom are available on the
    FreeRTOS WEB site.

    1 tab == 4 spaces!

    ***************************************************************************
     *                                                                         *
     *    Having a problem?  Start by reading the FAQ "My application does     *
     *    not run, what could be wrong?"                                       *
     *                                                                         *
     *    http://www.FreeRTOS.org/FAQHelp.html                                 *
     *                                                                         *
    ***************************************************************************


    http://www.FreeRTOS.org - Documentation, training, latest versions, license
    and contact details.

    http://www.FreeRTOS.org/plus - A selection of FreeRTOS ecosystem products,
    including FreeRTOS+Trace - an indispensable productivity tool.

    Real Time Engineers ltd license FreeRTOS to High Integrity Systems, who sell
    the code with commercial support, indemnification, and middleware, under
    the OpenRTOS brand: http://www.OpenRTOS.com.  High Integrity Systems also
    provide a safety engineered and independently SIL3 certified version under
    the SafeRTOS brand: http://www.SafeRTOS.com.
*/


#ifndef FREERTOS_CONFIG_H
#define FREERTOS_CONFIG_H


/*-----------------------------------------------------------
 * Application specific definitions.
 *
 * These definitions should be adjusted for your particular hardware and
 * application requirements.
 *
 * THESE PARAMETERS ARE DESCRIBED WITHIN THE 'CONFIGURATION' SECTION OF THE
 * FreeRTOS API DOCUMENTATION AVAILABLE ON THE FreeRTOS.org WEB SITE.
 *
```

```
 * See http://www.freertos.org/a00110.html.
 *----------------------------------------------------------*/

 /*
 * The following #error directive is to remind users that a batch file must be
 * executed prior to this project being built.  The batch file *cannot* be
 * executed from within the IDE!  Once it has been executed, re-open or refresh
 * the Eclipse project and remove the #error line below.
 */
//#error Ensure CreateProjectDirectoryStructure.bat has been executed before building.  See comment immediately above.

#define configUSE_PREEMPTION                    1
#define configUSE_IDLE_HOOK                     1
#define configUSE_TICK_HOOK                     1
#define configCPU_CLOCK_HZ                      ( 80000000UL )
#define configTICK_RATE_HZ                      ( ( portTickType ) 1000 )
#define configMAX_PRIORITIES                    ( ( unsigned portBASE_TYPE ) 5 )

/*
 * The stack is divided into a (system) stack and a user stack space
 * per task. The tasks (system) stack stores the return addresses for
 * function calls and its context when suspended. Local variables in
 * functions are placed on the task user stack.
 */
/* sufficient for a call-depth of 7 plus 2 nested IRQs plus full context */
#define configMINIMAL_STACK_SIZE                ( ( unsigned short ) 104 )
/* sufficient for the task, 7 calls and 2 nested IRQs each pushing 7
registers and 6 variables of type word onto the user stack */
#define configMINIMAL_STACK2_SIZE               ( ( unsigned short ) 264 )
/*
 * The heap has to be max 16kB minus 256 bytes for the initial
 * system user stack or it would not be accessible by near
 * addressing via Data Page Pointer (DPP).
 */
#define configTOTAL_HEAP_SIZE                   ( ( size_t ) ( 16*1024 - 256 ) )
/*
 * The heap has to be placed right after the system user stack at
 * absolute address 0xE00100 (see project LSL file for user stack
 * definition).
 */
#define configHEAP_ABSOLUTE_ADDR                __at ( 0xE00100 )

#define configMAX_TASK_NAME_LEN                 ( 8 )
#define configUSE_TRACE_FACILITY                1
#define configUSE_16_BIT_TICKS                  0
#define configIDLE_SHOULD_YIELD                 1
#define configUSE_MUTEXES                       1
#define configQUEUE_REGISTRY_SIZE               0
#define configCHECK_FOR_STACK_OVERFLOW          2
#define configCHECK_FOR_STACK2_OVERFLOW         2
/* support boot stack checking to know when to increase
system and/or user stack size in the LSL file */
#define configCHECK_FOR_BOOT_STACK_OVERFLOW     1
#define configUSE_RECURSIVE_MUTEXES             1
#define configUSE_MALLOC_FAILED_HOOK            1
#define configUSE_APPLICATION_TASK_TAG          1
#define configUSE_COUNTING_SEMAPHORES           1
#define configGENERATE_RUN_TIME_STATS           1

/* Co-routine definitions. */
#define configUSE_CO_ROUTINES                   0
#define configMAX_CO_ROUTINE_PRIORITIES         ( 2 )

/* Software timer definitions. */
#define configUSE_TIMERS                        1
#define configTIMER_TASK_PRIORITY               ( configMAX_PRIORITIES - 1 )
#define configTIMER_QUEUE_LENGTH                5
#define configTIMER_TASK_STACK_DEPTH            ( configMINIMAL_STACK_SIZE )
#define configTIMER_TASK_STACK2_DEPTH           ( configMINIMAL_STACK2_SIZE *2 )

/* Set the following definitions to 1 to include the API function, or zero
to exclude the API function. */
#define INCLUDE_vTaskPrioritySet                1
#define INCLUDE_uxTaskPriorityGet               1
#define INCLUDE_vTaskDelete                     1
#define INCLUDE_vTaskCleanUpResources           1
#define INCLUDE_vTaskSuspend                    1
#define INCLUDE_vTaskDelayUntil                 1
#define INCLUDE_vTaskDelay                      1
// test compile extended features
#define INCLUDE_xTaskGetIdleTaskHandle          1
#define INCLUDE_xTimerGetTimerDaemonTaskHandle  1
#define INCLUDE_xTimerGetTimerDaemonTaskHandle  1
#define INCLUDE_eTaskStateGet                   1
#define INCLUDE_pcTaskGetTaskName               1
#define INCLUDE_uxTaskGetStackHighWaterMark     1
#define INCLUDE_xQueueGetMutexHolder            1

```

```
171 #define configPRIO_BITS                         4        /* 15 priority levels */
172
173 /* The absolute highest interrupt level */
174 #define configMAX_INTERRUPT_PRIORITY            ( (1 << configPRIO_BITS) - 1 )
175
176 /* The highest interrupt priority that can be used by any interrupt service
177 routine that makes calls to interrupt safe FreeRTOS API functions.  DO NOT CALL
178 INTERRUPT SAFE FREERTOS API FUNCTIONS FROM ANY INTERRUPT THAT HAS A HIGHER
179 PRIORITY THAN THIS! */
180 #define configMAX_SYSCALL_INTERRUPT_PRIORITY    11
181
182 /* Lowest Interrupt priority, used by the kernel port layer itself. */
183 #define configKERNEL_INTERRUPT_PRIORITY         1
184
185 /* Task level priority */
186 #define configTASK_LEVEL_INTERRUPT_PRIORITY     0
187
188 /* Normal assert() semantics without relying on the provision of an assert.h
189 header file. */
190 #define configASSERT( x ) \
191     if( ( x ) == 0 ) { taskDISABLE_INTERRUPTS(); for( ;; ) portNOP(); }
192
193 #endif /* FREERTOS_CONFIG_H */
```

Listing G.4: Port Config File *FreeRTOSConfig.h*

Bibliography

[1] Nicolas Navet and Francoise Simonot-Lion, editors. *Automotive embedded systems handbook*. Industrial Information Technology Series. CRC Press, Boca Raton, 2009. ISBN 9780849380266.

[2] Dip Goswami, Reinhard Schneider, Alejandro Masrur, Martin Lukasiewycz, Samarjit Chakraborty, Harald Voit, and Anuradha Annaswamy. Challenges in Automotive Cyber-physical Systems Design. In *International Conference on Embedded Computer Systems (SAMOS), 2012*, pages 346–354, 2012.

[3] Bruce Powel Douglass. *Design Patterns for Embedded Systems in C*. Newnes, Oxford, UK, 1. edition, 2011. ISBN 9781856177078.

[4] Jörg Schäuffele and Thomas Zurawka. *Automotive Software Engineering*. ATZ/MTZ-Fachbuch. Vieweg + Teubner, Wiesbaden, 4. edition, 2010. ISBN 3834803642.

[5] David John Smith and Kenneth G. L. Simpson. *Functional safety*. Elsevier/Butterworth-Heinemann, Oxford, 2. edition, 2004. ISBN 9780750662697.

[6] Ron Bell. Introduction to IEC 61508. In *Proceedings of the 10th Australian workshop on Safety critical systems and software – Volume 55*, SCS '05, pages 3–12, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc. ISBN 1920682376.

[7] Raija Koivisto, Nina Wessberg, Annele Eerola, Toni Ahlqvist, Sirkku Kivisaari, Jouko Myllyoja, and Minna Halonen. Integrating FTA and Risk Assessment Methodologies. Third International Seville Seminar on Future-Oriented Technology Analysis: Impacts and implications for policy and decision-making – SEVILLE 16-17 OCTOBER 2008, October 2008. URL http://forera.jrc.ec.europa.eu/fta_2008/papers_parallel/theme_1/1-9%20Koivisto-Paper.pdf. Last accessed 2012-09-26.

[8] Instrument-Net.co.uk. Standards/Safety/Integrity Levels ANSI/ISA S84.01 & DRAFT IEC 61508 – How this Standard will affect your business, 2010. URL http://www.instrument-net.co.uk/safety_integrity.html. Last accessed 2012-09-26.

[9] Pirmin Netter. Wie die Sicherheit laufen lernte – Entwicklung der funktionalen Sicherheit in Deutschland. *atp edition*, pages 46–55, 1-2/

2011. URL http://www.namur.de/fileadmin/media/Pressespiegel/atp/atp_01_02_2011_Sicherheit_Netter.pdf. Last accessed 2012-09-22.

[10] Wilfried Grote. Basics of Functional Safety in Process Industry. Slides, September 2011. URL http://www.emo.org.tr/ekler/7b58fca3ca5baf4_ek.pdf. Last accessed 2012-09-25.

[11] ABB Asea Brown Boveri Ltd. ABB drives – Technical guide no. 10 – Functional safety. ABB Asea Brown Boveri Ltd, 2011. URL http://www05.abb.com/global/scot/scot201.nsf/veritydisplay/7532c44d2abe431bc12578620045c2ba/$file/en_technicalguideno10_revd.pdf. Last accessed 2012-10-02.

[12] Christoph Braeuchle. Safety critical development of software-intensive automotive systems. *John Day's Automotive Electronics News – Insight for Engineers*, May 16th 2012. URL http://johndayautomotiveelectronics.com/?p=10362. Last accessed 2012-08-21.

[13] ISO 26262-1:2011 - Road vehicles - Functional safety - Part 1: Vocabulary. International Standard, November 15th 2011.

[14] ISO 26262-2:2011 - Road vehicles - Functional safety - Part 2: Management of functional safety. International Standard, November 15th 2011.

[15] ISO 26262-3:2011 - Road vehicles - Functional safety - Part 3: Concept phase. International Standard, November 15th 2011.

[16] ISO 26262-4:2011 - Road vehicles - Functional safety - Part 4: Product development at the system level. International Standard, November 15th 2011.

[17] ISO 26262-5:2011 - Road vehicles - Functional safety - Part 5: Product development at the hardware level. International Standard, November 15th 2011.

[18] ISO 26262-6:2011 - Road vehicles - Functional safety - Part 6: Product development at the software level. International Standard, November 15th 2011.

[19] ISO 26262-7:2011 - Road vehicles - Functional safety - Part 7: Production and operation. International Standard, November 15th 2011.

[20] ISO 26262-8:2011 - Road vehicles - Functional safety - Part 8: Supporting processes. International Standard, November 15th 2011.

[21] ISO 26262-9:2011 - Road vehicles - Functional safety - Part 9: Automotive Safety Integrity Level (ASIL)-oriented and safety-oriented analyses. International Standard, November 15th 2011.

[22] ISO 26262-10:2012 - Road vehicles - Functional safety - Part 10: Guideline on ISO 26262. International Standard, August 1st 2012.

[23] Jürgen Sauler and Stefan Kriso. ISO 26262 – Die zukünftige Norm zur funktionalen Sicherheit von Straßenfahrzeugen. *Elektronikpraxis*, August 31st 2011. URL http://files.vogel.de/vogelonline/pdfarticles/ep/themen/elektronikmanagement/projektqualitaetsmanagement/articles/242243/242243.pdf. Last accessed 2013-07-05.

[24] Andrea Piovesan and John Favaro. Experience with ISO 26262 ASIL Decomposition. Automotive SPIN, Milano, February 17th 2011. URL http://www.automotive-spin.it/uploads/8/8W_favaro.pdf. Last accessed 2013-05-15.

[25] Olaf Kath, Rudolf Schreiner, and John Favaro. Safety, Security, and Software Reuse: A Model-Based Approach. In *Fourth International Workshop in Software Reuse and Safety (RESAFE 2009), Washington, DC, USA*, 2009. URL http://www.favaro.net/john/RESAFE2009/results/RESAFE2009%20Intecs%20ikv%20object%20security.pdf. Last accessed 2013-07-05.

[26] Terry L. Fruehling. Delphi Secured Microcontroller Architecture. In *2000 SAE World Congress*, pages pp. 1–12. SAE International, March 6-9 2000. URL http://am.delphi.com/pdf/techpapers/2000-01-1052.pdf. SAE Technical Paper 2000-01-1052. Last accessed 2012-08-10.

[27] Massimo Baleani, Alberto Ferrari, Leonardo Mangeruca, Alberto Sangiovanni-Vincentelli, Maurizio Peri, and Saverio Pezzini. Fault-tolerant platforms for automotive safety-critical applications. In *Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, CASES '03, pages 170–177. ACM, 2003. ISBN 1581136765.

[28] Jean-Louis Dufour. Automotive safety concepts : 10^{-9}/h for less than 100EUR a piece. *6th Braunschweig Conference AAET 2005 – Automation, Assistance and Embedded Real Time Platforms for Transport*, page 10, February 2005. URL http://j-l-dufour.voila.net/05_AAET_article.pdf. Last accessed 2012-10-01.

[29] Padma Sundaram and Joseph G. D'Ambrosio. Controller Integrity in Automotive Failsafe System Architectures. In *2006 SAE World Congress*, page pp. 10. SAE International, April 3-6 2006. URL http://delphi.com/pdf/techpapers/2006-01-0840.pdf. SAE Technical Paper 2006-01-0840. Last accessed 2012-08-10.

[30] Riccardo Mariani and Peter Fuhrmann. Comparing fail-safe microcontroller architectures in light of IEC 61508. In *Proc. 22nd IEEE Int. Symp. Defect and Fault-Tolerance in VLSI Systems DFT '07*, pages 123–131, 2007.

[31] Xi Chen. *Requirements and concepts for future automotive electronic architectures from the view of integrated safety*. PhD thesis, Universität Karlsruhe (TH), February 2008. URL http://uvka.ubka.uni-karlsruhe.de/shop/download/1000007763. Last accessed 2013-07-29.

[32] Marco Bellotti and Riccardo Mariani. How future automotive functional safety requirements will impact microprocessors design. *Microelectronics Reliability*, 50 (9-11):1320–1326, 2010. ISSN 00262714.

[33] Eldon G. Leaphart, Barbara J. Czerny, Joseph G. D'Ambrosio, Christopher L. Denlinger, and Deron Littlejohn. Survey of Software Failsafe Techniques for Safety-Critical Automotive Applications. In *2005 SAE World Congress*, page pp. 16. SAE International, April 11-14 2005. URL http://delphi.com/pdf/techpapers/2005-01-0779.pdf. SAE Technical Paper 2005-01-0779. Last accessed 2012-08-10.

[34] Simon Brewerton, Rolf Schneider, and Denis Eberhard. Implementation of a Basic Single-Microcontroller Monitoring Concept for Safety Critical Systems on a Dual-Core Microcontroller. SAE Technical Paper 2007-01-1486, April 2007.

[35] Josef Börcsök. Models to calculate Safety and Reliability Parameters for Embedded Systems. In *ICAT 2009. XXII International Symposium on Information, Communication and Automation Technologies, 2009.*, pages 1–8, 2009.

[36] Ashraf Armoush. *Design Patterns for Safety-Critical Embedded Systems*. Phd thesis, Embedded Software Laboratory – RWTH Aachen University, June 2010. URL http://aib.informatik.rwth-aachen.de/2010/2010-13.pdf. AIB-2010-13. Last accessed 2012-08-13.

[37] Arbeitskreis EGAS. Standardisiertes E-Gas Überwachungskonzept für Benzin und Diesel Motorsteuerungen. Version 5.5, May 16th 2013. URL http://www.iav.com/publikationen/technische-veroeffentlichungen/e-gas-monitoring-concepts. Last accessed 2013-06-11.

[38] Arbeitskreis EGAS. Standardized ETC Monitoring Concept for Gasoline and Diesel Engine Control Systems. Version 5.5, May 16th 2013. URL http://www.iav.com/en/publications/technical-publications/etc-monitoring-concepts. Last accessed 2013-06-11.

[39] Thomas Zurawka and Joerg Schäuffele. Verfahren zur Überprüfung der Sicherheit und Zuverlässigkeit softwarebasierter elektronischer Systeme. Robert Bosch GmbH, International Patent No. WO2005003972, January 2005. URL http://worldwide.espacenet.com/publicationDetails/biblio?CC=WO&NR=2005003972. Last accessed 2012-10-24.

[40] Frank Bederna and Thomas Zeller. Verfahren und Vorrichtung zur Steuerung der Antriebseinheit eines Fahrzeugs. Robert Bosch GmbH, German Patent No. DE4438714, May 1996. URL http://worldwide.espacenet.com/publicationDetails/biblio?CC=DE&NR=4438714. Last accessed 2013-05-25.

[41] Christian Miedl. Verfahren zur Überprüfung einer Funktion einer Recheneinheit und Anordnung mit zwei Recheneinheiten. Continental Automotive GmbH, German Patent No. DE102009023112, December 2012. URL http://worldwide.espacenet.com/publicationDetails/biblio?CC=DE&NR=102009023112. Last accessed 2012-10-01.

[42] Torsten Bauer. Verfahren und Vorrichtung zur gegenseitigen Überwachung von Steuereinheiten. Robert Bosch GmbH, German Patent No. DE19933086, January 2001. URL http://worldwide.espacenet.com/publicationDetails/biblio?CC=DE&NR=19933086. Last accessed 2013-06-04.

[43] Wolfgang Haas, Andreas Frank, and Thomas Meier. System und Verfahren zur Überwachung einer Einrichtung zum Messen, Steuern und Regeln. Robert Bosch GmbH, German Patent No. DE10018859, October 2001. URL http://worldwide.espacenet.com/publicationDetails/biblio?CC=DE&NR=10018859. Last accessed 2012-10-24.

[44] Thorsten Juenemann, Bernd Doerr, Holger Niemann, and Per Hagman. Verfahren zur Steuerung einer Fahrzeug-Antriebseinheit. Robert Bosch Gmbh, German Patent No. DE102005040783, March 2007. URL http://worldwide.espacenet.com/publicationDetails/biblio?CC=DE&NR=102005040783. Last accessed 2013-06-11.

[45] Holger Niemann, Per Hagman, and Daniel Dammm. Verfahren zum Betreiben einer Steuer- und/oder Regeleinrichtung, vorzugsweise für eine Antriebsmaschine eines Kraftfahrzeugs. Robert Bosch GmbH, German Patent No. DE102005062873, July 2007. URL http://worldwide.espacenet.com/publicationDetails/biblio?CC=DE&NR=102005062873. Last accessed 2012-10-24.

[46] Malte Jacobi and Edwin Böhm. Monitoring concept in a control device. Conti Temic Microelectronic GmbH, US Patent No. US20120316728, December 2012. URL http://worldwide.espacenet.com/publicationDetails/biblio?CC=US&NR=2012316728. Last accessed 2013-05-25.

[47] Werner Zimmermann and Ralf Schmidgall. *Bussysteme in der Fahrzeugtechnik*. ATZ/MTZ-Fachbuch. Vieweg, Wiesbaden, 2. edition, 2007. ISBN 9783834802354.

[48] Robert Bosch GmbH. Introduction to Automotive Bus Systems, October 2011. URL http://users.nik.uni-obuda.hu/mobil/tantargyak/bir/bsc/2011/intro_farkas_external_2011_10_21.pdf. Last accessed 2013-09-26.

[49] Yolanda XI. Freescale Solutions for Advanced Driver Assistance, 2012. URL http://www.freescale.com.cn/media/download/Freescale_ADAS_solutions%202012_customer.pdf. Last accessed 2013-04-16.

[50] Josef Noebauer. Is Ethernet the Rising Star for In-vehicle Networks? *16th IEEE Inter. Conference on Emerging Technologies and Factory Automation (ETFA2011)*, September 2011. URL http://www.iestcfa.org/presentations/etfa2011/Noebauer_Ethernet.pdf. Last accessed 2012-09-18.

[51] Florian Hartwich. CAN with Flexible Data-Rate. In *Proceedings of the 13th iCC 2012 in Hambach Castle (Germany)*, pages 14–1–14–9. CAN in Automation e.V., March 2012. URL http://www.can-cia.org/fileadmin/cia/files/icc/13/hartwich.pdf. Last accessed 2012-08-20.

[52] FreeRTOS – Market leading RTOS (Real Time Operating System) for embedded systems supporting 33 microcontroller architectures. Website, 2013. URL http://www.freertos.org/. Last accessed 2013-02-13.

[53] About FreeRTOS. Website, 2013. URL http://www.freertos.org/RTOS.html. Last accessed 2013-02-13.

[54] Phillip A. Laplante. *Real-time systems design and analysis*. Wiley, Hoboken, N.J, 3. edition, 2004. ISBN 0471228559.

[55] Matthias Homann. *OSEK. Betriebssystem-Standard für Automotive und Embedded Systems*. mitp-Verlag, Bonn, 2. edition, 2005. ISBN 9783826615528.

[56] Eugen Brenner. Echtzeit-Betriebssysteme. Entwurf von Echtzeitsystemen, 1. Teil, November 2012. URL https://www.iti.tugraz.at/cms/index.php?option=com_phocadownload&view=category&download=130:entwurfechtzeitsystemeteil1&id=15:entwurf-von-echtzeitsystemen&Itemid=69. Last accessed 2013-02-15.

[57] Eugen Brenner. Echtzeit-Betriebssysteme. Entwurf von Echtzeitsystemen, 4. Teil, November 2012. URL https://www.iti.tugraz.at/cms/index.php?option=com_phocadownload&view=category&download=133:entwurfechtzeitsystemeteil4&id=15:entwurf-von-echtzeitsystemen&Itemid=69. Last accessed 2013-02-15.

[58] Peter Mandl. *Grundkurs Betriebssysteme*. Studium. Springer Vieweg, Wiesbaden, 3. edition, 2013. ISBN 9783834818973.

[59] Richard Barry. *Using the FreeRTOS Real Time Kernel: A Practical Guide – Standard Edition*. Real Time Engineers Limited, Bristol, UK, 1. edition, 2010. ISBN 9781446169148.

[60] Richard Barry. *FreeRTOS Reference Manual: API Functions and Configuration Options*. Real Time Engineers Limited, Bristol, UK, 2011.

[61] Richard Barry. FreeRTOS API Reference, 2013. URL http://www.freertos.org/a00106.html. Last accessed 2013-02-27.

[62] Tom Barrett. Real-Time Operating Systems: On schedule. *Micro Technology Europe (MTE) Magazine*, pages 26–29, August 2012. URL http://www.embeddednews.co.uk/files/pdfs/104fd4bb-eec0-42b8-ad87-6ccb58542520/mteaugust2012forweb.pdf. Last accessed 2013-03-01.

[63] Christopher Svec. FreeRTOS. *The Architecture of Open Source Applications, Volume II*, 2012. URL http://www.aosabook.org/en/freertos.html. Last accessed 2013-02-26.

[64] Rich Goyette. An Analysis and Description of the Inner Workings of the FreeRTOS Kernel. Unpublished course work in SYSC5701: Operating System Methods for Real-Time Applications, 2007. URL http://richardgoyette.com/Papers/FreeRTOSPaper.pdf. Last accessed accessed 2013-02-28.

[65] Richard Barry. FreeRTOS Memory Management, 2012. URL http://www.freertos.org/a00111.html. Last accessed 2013-03-04.

[66] Richard Barry. FreeRTOS-MPU – Memory Protection Unit (MPU) Support, 2012. URL http://www.freertos.org/FreeRTOS-MPU-memory-protection-unit.html. Last accessed 2013-03-04.

[67] Richard Barry. FreeRTOS-MPU Specific Functions, 2012. URL http://www.freertos.org/FreeRTOS-MPU-specific.html. Last accessed 2013-03-04.

[68] Richard Barry. FreeRTOS Blocking on Multiple RTOS Objects, 2013. URL http://www.freertos.org/Pend-on-multiple-rtos-objects.html. Last accessed 2013-03-04.

[69] Richard Barry. FreeRTOS Queue Set API Functions, 2013. URL http://www.freertos.org/RTOS-queue-sets.html. Last accessed 2013-03-04.

[70] Richard Barry. FreeRTOS Run Time Statistics, 2012. URL http://www.freertos.org/rtos-run-time-stats.html. Last accessed 2013-03-04.

[71] Richard Barry. Tickless Low power features in FreeRTOS, 2012. URL http://www.freertos.org/low-power-tickless-rtos.html. Last accessed 2013-03-06.

[72] Richard Barry. FreeRTOS Trace Hook Macros, 2012. URL http://www.freertos.org/rtos-trace-macros.html. Last accessed 2013-03-04.

[73] Martin Schmidt, Marcus Rau, Ekkehard Helmig, and Bernhard Bauer. Functional Safety – Dealing with Independency, Legal Framework Conditions and Liability Issues. Professional article, 2011. URL http://www.sgs-tuev-saar.com/pdf/Article-ISO-26262-Law-08-2011.pdf. Last accessed 2013-07-16.

[74] Stefan Teuchert. ISO 26262 – Fluch oder Segen? *ATZelektronik*, 7(6):pp 410–415, December 1st 2012. doi: 10.1365/s35658-012-0223-x. URL http://www.springerlink.com/index/10.1365/s35658-012-0223-x.

[75] Stefan Teuchert. ISO 26262 – Blessing or Curse? *ATZelektronik worldwide*, 7(6):pp 4–9, October 1st 2012. doi: 10.1365/s38314-012-0128-8. URL http://link.springer.com/article/10.1365/s38314-012-0128-8.

[76] Simon Fürst. ISO 26262 and AUTOSAR. Requirements and Solutions for Safety Related Software. 5th Vector Congress, December 2010. URL https://www.vector.com/portal/medien/cmc/events/commercial_events/VectorCongress_2010/AUTOSAR_2_Fuerst_Lecture.pdf. Last accessed 2013-07-29.

[77] Rolf Schneider, Wolfgang Brandstaetter, Marc Born, Olaf Kath, Tobias Wenzel, Rafael Zalman, and Johann Mayer. Safety Element out of Context – A Practical Approach. SAE Technical Paper 2012-01-0033, April 16th 2012.

[78] Simon P. Brewerton, Natalia Willey, Swapnil Gandhi, Thorsten Rosenthal, Claus Stellwag, and Matthieu Lemerre. Demonstration of Automotive Steering Column Lock using Multicore AutoSAR® Operating System. SAE Technical Paper 2012-01-0031, April 16th 2012. URL http://delphi.com/pdf/techpapers/2012-01-0031.pdf. Last accessed 2013-09-08.

[79] Mike Ellims, Helen Monkhouse, and Angus Lyon. ISO 26262: Experience applying part 3 to an in-wheel electric motor. In *6th IET International System*

Safety Conference 2011, pages 1–8, Birmingham UK, September 2011. URL http://skicambridge.com/papers/IET_2012.pdf. Last accessed 2012-11-21.

[80] SAE International. 2012 Formula SAE® Rules, September 2011. URL http://www.fsaeonline.com/content/2012_FSAE_Rules_Version_90111K.pdf. Last accessed accessed 2013-09-26.

[81] Formula Student Germany. Formula Student Electric Rules 2012. Version 1.0.1, March 2012. URL http://www.formulastudent.de/uploads/media/FSE_Rules_2012_v1.0.1.pdf. Last accessed 2012-11-06.

[82] Huw C. Davies and B. Gugliotta. Investigating the injury risk in frontal impacts of Formula Student cars: a computer-aided engineering analysis. In *Proceedings of the Institution of Mechanical Engineers, Part D: Journal of Automobile Engineering 2012*, pages 181–193. SAGE Publications, September 15th 2011.

[83] David Rising, Jason Kane, Nick Vernon, Joseph Adkins, Craig Hoff, and Janet Brelin-Fornari. Analysis of a Frontal Impact of a Formula SAE Vehicle. Automotive Testing Expo North America, 2006. URL http://users.telenet.be/AudiR8/Analysis%20of%20a%20Frontal%20Impact%20of%20a%20Formula%20SAE%20Vehicle.pdf. Last accessed 2012-11-27.

[84] Witchawut Pumchaloen, Rachan Chumueang, Apiwat Kialon, and Chawin Chantharasenawong. Assessment of Student Formula driver's safety through optimization of impact attenuator sizing. The 7th International Conference on Automotive Engineering (ICAE-7), Bangkok, Thailand, March 28th–April 1st 2011. URL http://www.drchawin.com/FTPdrive/Publications/ICAE-7.pdf. Last accessed 2012-11-27.

[85] Steve Hartley, Ireri Ibarra, and Gunwant Dhadyalla. Functional Safety & Diagnostics of Hybrid Vehicles. Regenerative Braking & Vehicle Supervisory Control Dissemination Event, October 27th 2011. URL http://www2.warwick.ac.uk/fac/sci/wmg/research/lcvtp/documents/presentations/lcvtp072-ws6dissemination eventfunctionalsafetyanddiagnosticsfinalversion_3.pdf. Last accessed 2013-09-05.

[86] Bernhard Kaiser. Approaches Towards Reusable Safety Concepts. VDA Automotive SYS Conference 2012, May 15th-16th 2012. URL http://www.berner-mattner.com/cms/upload/pdf/Praesentationen/BernerMattner_Vortrag_VDA_2012.pdf. Last accessed 2013-09-17.

[87] Robert Bosch GmbH. CAN Specification Version 2.0B, 1991. URL http://www.bosch-semiconductors.de/media/pdf_1/canliteratur/can2spec.pdf. Last accessed 2013-09-26.

[88] Robert Bosch GmbH. CAN (Controller Area Network), September 2011. URL http://users.nik.uni-obuda.hu/mobil/tantargyak/bir/bsc/2011/_CAN_OE_Babosa_2011_11_14.pdf. Last accessed 2013-09-26.

[89] AUTOSAR development partnership. AUTOSAR Specification of SW-C End-to-End Communication Protection Library. V2.0.0 R4.0 Rev 3, December 21th 2011. URL http://www.autosar.org/download/R4.0/AUTOSAR_SWS_E2ELibrary.pdf. Last accessed 2013-09-26.

[90] William Fordham Cooper. Electrical control of dangerous machinery and processes. *Electrical Engineers – Part II: Power Engineering, Journal of the Institution of*, 94 (39):216–232, June 1947.

[91] ISO 13850:2006 - Safety of machinery - Emergency stop - Principles for design. International Standard, May 25th 2008.

[92] Infineon Technologies AG. XC2300 and CIC61508 – Cost-Optimized Safety Computing Platform. Product Brief, 2012. URL http://www.infineon.com/dgdl/Safety-Computing-Platform-XC2300-CIC61508-Product-Brief.pdf?folderId=db3a304317a748360117f45a9c863e84&fileId=db3a3043353fdc16013543303497315d. Last accessed 2013-10-03.

[93] Infineon Technologies AG. Infineon safety computing platform for ISO26262 compliant motor control. Presentation, 2011. URL http://www.infineon-ecosystem.org/download/solution.php?act=down&topic=3&item=19. Last accessed 2012-10-19.

[94] Infineon Technologies AG. Infineon safety computing platform scalable platform up to ASIL-D. Presentation, 2011. URL http://www.infineon-ecosystem.org/download/schedule.php?act=detail&item=93. Last accessed 2013-10-03.

[95] Hitex (UK) Ltd. XC2000 For ASIL-B Configuration Concept, 2012. URL http://www.hitex.com/fileadmin/uk-files/downloads/CIC%20Support/ASIL-B(D)%20Configurations.pdf. Last accessed 2013-10-03.

[96] Infineon Technologies AG. XC2385A, XC2387A – 16/32-Bit Single-Chip Microcontroller with 32-Bit Performance. Datasheet V2.1, 2011. URL http://www.infineon.com/dgdl/xc238xa_ds_v2+1_2011_07.pdf?folderId=db3a30441 2b407950112b409d4b00386&fileId=db3a3043353fdc16013557e024517b1e&ack=t. Last accessed 2013-10-04.

[97] Infineon Technologies AG. TLE6368-G2 Multi-Voltage Processor Power Supply. Datasheet V2.32, 2010. URL http://www.infineon.com/dgdl/TLE6368_G2_DS_rev2+32.pdf?folderId=db3a30431400ef68011421b54e2e0564&fileId=db3a30432239cccd01225a67e02b6c60&ack=t. Last accessed 2013-10-04.

[98] Infineon Technologies AG. CIC61508 – Functional Safety Companion Chip. Datasheet V1.2, 2011. URL http://www.infineon.com/dgdl/CIC61508_ds_v1.2_2011_06.pdf?folderId=db3a304317a748360117f45a9c863e84&fileId=db3a30432f29829e012f449b02301f1f&ask=t. Last accessed 2013-10-04.

[99] Infineon Technologies AG. PROFET® BTS 6163 D Smart Highside Power Switch. Datasheet V1.0, 2007. URL http://www.infineon.com/dgdl/BTS6163D_DS_v10.pdf?folderId=db3a304314dca389011537739e37155f&fileId=db3a3043183a95550118606cdf2336a2&ack=t. Last accessed 2013-10-06.

[100] Infineon Technologies AG. TLE6250 High Speed CAN-Transceiver. Datasheet V4.0, 2008. URL http://www.infineon.com/dgdl/TLE6250G_DS_rev40_Green%5B1%5D.pdf?folderId=db3a30431ed1d7b2011edadb13813703&fileId=db3a30431ed1d7b2011edadbb20d3704&ack=t. Last accessed 2013-10-06.

[101] Bender GmbH & Co. KG. ISOMETER® IR155-3203/IR155-3204 – Insulation monitoring device (IMD) for unearthed DC drive systems (IT systems) in electric vehicles. Datasheet V004, 2013. URL http://www.bender-emobility.com/fileadmin/products/doc/iso-F1-IR155-32xx-V004-electric-vehicles_DB_en.pdf. Last accessed 2013-10-07.

[102] Infineon Technologies AG. C166S V2 – 16-Bit Microcontroller. User's Manual V1.7, 2001. URL http://www.infineon.com/dgdl/c166sv2um.pdf?folderId=db3a304412b407950112b41d4e492fe2&fileId=db3a304412b407950112b41d4ea32fe3&ack=t. Last accessed 2013-10-04.

[103] Altium Limited. *TASKING VX-toolset for C166 v3.1 User Guide*, October 2012. URL http://tasking.com/support/c166/c166_user_guide_v3.1.pdf. Last accessed 2013-03-07.

[104] Infineon Technologies AG. XC2300A Derivatives – 16/32-Bit Single-Chip Microcontroller with 32-Bit Performance. User's Manual V2.0, 2009. URL http://www.infineon.com/dgdl/xc2300a_um_v2.0_2009_03.pdf?folderId=db3a30431375fb1a0113864e0cc701b6&fileId=db3a304320d39d590120f681ca126a3b&sId=db3a30433d346a2d013d58f943a3183a. Last accessed 2013-10-04.

[105] Richard Barry. FreeRTOS Porting Guide, 2012. URL http://www.freertos.org/FreeRTOS-porting-guide.html. Last accessed 2013-03-07.

[106] Mahindra Satyam. Commercial vehicles – Functional safety implementation process and challenges. Presentation, 2013. URL https://s3.amazonaws.com/automotiveworld/events/cvm-india-2013/1-3-Dr-Chitra-Rajan-Mahindra-Satyem.pdf. Last accessed 2013-10-08.

[107] Verband der Automobilindustrie (VDA). Auto & Normung NAAutomobil Jahresbericht 2012, 2012. URL http://www.vda.de/de/downloads/1113/. Last accessed 2013-10-08.

[108] Infineon Technologies AG. MC-ISAR XC2000. Product Sheet, 2013. URL http://www.infineon.com/cms/de/product/microcontrollers/development-tools-software-and-kits/embedded-software-solutions/infineon-autosar-software/product-sheet-mc-isar-xc2000/channel.html?channel=db3a304329a0f6ee0129f561e79530ce. Last accessed 2013-10-08.

[109] Georg Macher. Drive-by-Wire System eines Formula Student Electric Boliden. Master thesis, Graz University of Technology, Graz, 2010.

Index