Beate Herbst BSc

Allocation and integration of automotive control functions in a multi-core environment

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieurin

Master's degree programme: Telematics

submitted to

Graz University of Technology

Supervisor

Univ.-Doz., Dipl.-Ing., Dr.techn. Daniel Watzenig

Institut für Meßtechnik und Meßsignalverarbeitung

Graz, May 2015

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis dissertation.

Date

Signature

Kurzfassung

In den letzten Jahren haben sich die verwendeten Strategien in der Automobilindustrie erheblich verändert. Die Anzahl der verwendeten Software-Funktionen ist bedeutend angestiegen und aufgrund der Tatsache, das pro Steuergerät genau eine Software-Funktion integriert wurde, nahm auch die Anzahl der Steuergeräte enorm zu. Aus diesem Grund sind Themen wie Software-Verteilung und Multi-Core Anwendung im automobilen Bereich zu immer wichtigeren Forschungsthemen herangewachsen. In der Automobilindustrie sind unterschiedliche Bereiche in einen Entwicklungsprozess involviert. Um eine Vorstellung davon zu bekommen, wie man Software-Funktionen sinnvoll verteilen kann, ist das Wissen unterschiedlichster Bereich (z.B. Modell-Entwicklung, Software-Entwicklung,...) notwendig.
In dieser Masterarbeit wurde eine Toolkette entwickelt, die es einem einzelnen Ingenieur mit möglichst geringem Aufwand ermöglicht, eine Hardware-Integration von diversen Regler-Modellen, verteilt oder nicht verteilt, durchzuführen. Des Weiteren soll das resultierende Verhalten der integrierten Funktionen durch sogenannte 'Hardware in the Loop' (HiL) Tests beurteilt werden können. Um dies umzusetzen, musste im ersten Schritt ein Szenario erstellt werden, um herauszufinden welche Informationen für eine Integration von Regelfunktionen auf einer speziellen Plattform notwendig sind. Die resultierende Methodik enthält eine Schnittstelle, in Form einer XML-basierten Spezifikation, die einen einfachen Datenaustausch zwischen unterschiedlichen Entwicklungsbereichen ermöglicht. Im nächsten Schritt wurde die entwickelte Toolkette durch die Integration mehrerer, sowohl verteilter (Multi-Core), als auch nicht verteilter (Single-Core) Regelfunktionen auf verschiedenen Plattformen getestet. Dafür wurden die Regelfunktionen eines Dual Mode Energy Storage (DMES) Systems eines Lotus Evora 414E Plug-In Hybrid, sowie die Funktionen einer allgemeinen Hybridfahrzeug-Simulation verwendet. Eine Evaluierung des erziehlten Verhaltens der jeweiligen Regelfunktionen, sowie ein Vergleich der Single- und Multi-Core Ergebnisse wurden danach durch eine HiL-Simulation mit Hilfe eines Co-Simulationstools durchgeführt. Diese Ergebnisse dienen als Basis für zukünftige Verteilungs-Szenarios.

Abstract

In the last years the strategies in the automotive industry significantly changed. The number of used software functions grew and due to the fact that one software function is integrated on one electrical control unit (ECU), even the number of control units tremendously increased. Thus, software distribution and multi-core usage became an active research field. In the automotive industry, different domains are included in development processes. To get an idea of how software functions can be distributed, the knowledge of different domains (e.g. model development, software development,...) is necessary to achieve suitable results.
Within this Master's thesis a toolchain was developed, to enable one single engineer to integrate different modelled control functions, distributed as well as not distributed and furthermore, evaluate the resulting performance of this integration by performing so called Hardware in the Loop (HiL) tests with little effort. To define such a methodology, a scenario to get an overview of what information is necessary to integrate different control functions on a hardware device has been defined in the first part of this work. The resulting toolchain contains an interface, in form of an XML-based specification, to easily enable the data exchange between different development domains. In the second part, the resulting toolchain is verified by performing single- as well as multi-core integrations of various control functions on different hardware devices. Therefore, the control functions of a Dual Mode Energy Storage (DMES) system, integrated in a Lotus Evora 414E plug-in hybrid, are used as well as control functions of an energy management system of a further hybrid vehicle simulation. After this integration, the behavior of the single- and the multi-core approaches were evaluated and compared by performing Hardware in the Loop (HiL) simulations using a co-simulation platform. Finally, the achieve results were evaluated and serve as a first basis for future distribution scenarios.

Danksagung

Diese Masterarbeit wurde im Jahr 2015 am Institut für Elektrische Meßtechnik und Meßsignalverarbeitung an der Technischen Universität Graz durchgeführt.

An dieser Stelle möchte ich all jenen danken, die durch ihre fachliche und persönliche Unterstützung zum Gelingen dieser Masterarbeit beigetragen haben.

Mein besonderer Dank gilt Univ.-Doz. Dipl.-Ing. Dr.techn. Daniel Watzenig, der die Durchführung dieser Masterarbeit am Institut für Elektrische Meßtechnik und Meßsignalverarbeitung an der Technischen Universität Graz in Zusammenarbeit mit dem Virtual Vehicle Research Center ermöglicht hat. Großer Dank gilt auch Dipl.-Ing. Stephanie Messner, für das Bereitstellen dieses interessanten Themas und die gute Betreuung und Unterstützung während der gesamten Zeit der Masterarbeit. Weiters möchte ich mich bei allen Mitarbeitern des Virtual Vehicle Research Center bedanken, im Besonderen bei Dipl.-Ing. Dr.techn. Bakk.techn. Martin Benedikt und Dipl.-Ing. Dr.techn. Allan Tengg, ohne deren Hilfe und Bemühungen diese Arbeit nicht zustande gekommen wäre.

Mein besonderer Dank gilt weiters meiner Familie, insbesondere meinen Eltern, die mir mein Studium ermöglicht und mich in all meinen Entscheidungen unterstützt haben. Herzlich bedanken möchte ich mich auch bei all meinen Freunden, die stets an mich geglaubt und meine gesamte Studienzeit zu etwas ganz Besonderem gemacht haben.

In diesem Sinne: *Let a new journey begin.*

Graz, im April 2015 Beate Herbst

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

In the last years the strategies in the automotive industry significantly changed. According to lecture notes of Stanford University [1], the number of Electric Control Units (ECUs) in a vehicle steadily increased since 2002 and is still expected to grow. Today's vehicle architectures integrate up to 70 ECUs [2], so a trend towards vehicles controlled electronically rather than purely mechanically is clearly shown. Figure 1.1 cited from [3] highlights this assumption. It shows that the number of functions have increased along with the number of ECUs within a vehicle. Thus, future vehicle architectures actually become much more complex and also the number of functions obviously increase more than it would be necessary for the achieved gain in functionality.

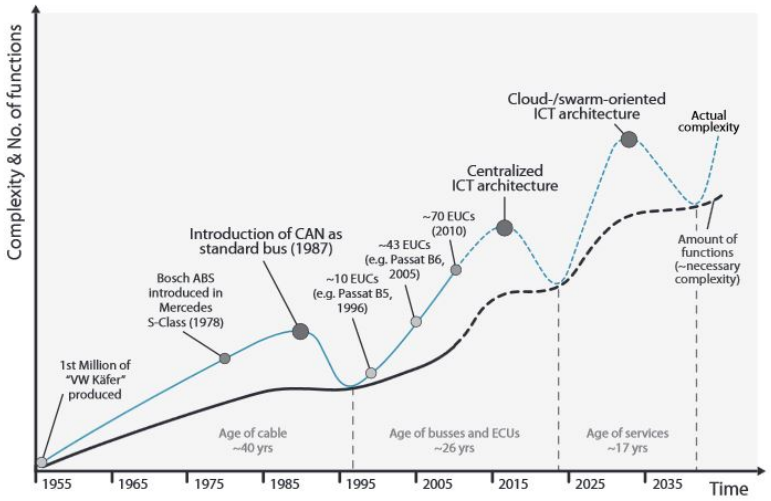

Figure 1.1: Evolution of complexity [3].

The solid part of the black line represents the approximate necessary complexity of vehicles until the year 2010, its dashed part illustrates the prediction over the next 20 years. The solid part of the blue line represents the actual complexity of vehicle's functions, the dashed part again illustrates its outlook for the next 20 years.

ECUs, consisting of a microcontroller and a set of sensors and actuators, are needed to control one or more electrical systems or subsystems within a vehicle, whereby their field of application varies from highly safety-critical operations to non-critical comfort or infotainment. A logical consequence of the highly increasing number of ECUs is the increase of costs for electronics and software in a vehicle. Due to the fact that 90% of innovations are electronic (80% in the area of software), as mentioned in [4], solutions are required to reduce the number of ECUs and as a result decrease the costs. A paper from 2005 [5] reports that today each new developed functionality is implemented on a stand-alone single-core ECU. However, the inefficiency of this approach becomes more and more clear now. Thus, distribution of functions over multiple ECUs and reuse of already existing software components becomes more and more important. Moreover, the usage of multi-core systems in the automotive domain has also gained importance over the last few years. Regarding [6], an automotive multi-core processor consists of at least two Central Processing Units (CPUs) which are connected to the same bus. This means that a multi-core system looks very like a number of single-core systems. A comparison of a single- and a multi-core architecture approach is illustrated in figure 1.2. With this approach, more centralized architecture designs can be adopted, consisting of a few powerful multi-core ECUs, each of them integrating functionalities of several single-core ECUs.

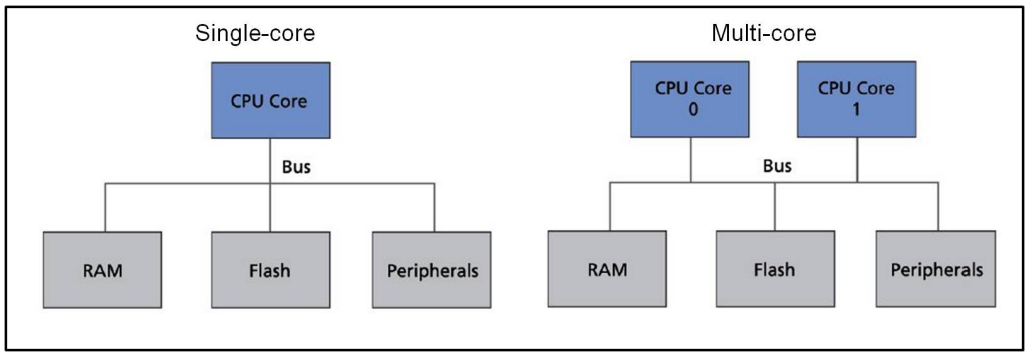

Figure 1.2: Examples of a single- and a multi-core architecture [6].

1.2 Aim of Work

This work aims for enabling development frontloading by establishing a seamless methodology for efficient hardware integration. Automotive software functions are often implemented using a model-based development approach. In most cases it is necessary to generate an executable code out of the model, which will afterwards run on an ECU. To test the correctness of the integrated software functions HiL tests are performed. Therefore, a connection between the hardware and a data source, which provides stimuli from the environment, is required to perform calculations on the ECU. Within this work most often generic stimuli are generated, representing various automotive parameters like the current vehicle velocity, its rotational speed or power as well as signals regarding a specific test track. These signals for instance can be provided by so called co-simulation. A co-simulation tool enables the coupling of different models created in different modelling and simulation tools as well as the coupling with hardware devices. The behavior of each model is not influenced by this coupling. To enable such a co-simulation including hardware devices, an appropriate communication interface is needed.
In such a process many different engineering domains are involved. An application engineer develops models. For further processing of the models a software engineer is responsible. To integrate the models on hardware in the end, an engineer for hardware integration (integrator) is needed. Hence, within this work a solution is discovered to ease the interaction of all these different engineering domains. In the end, it is possible that a hardware integration and HiL simulation can be completely performed by one person, if a simulation model is given.

The developed approaches will be tested by integrating already existing simulation models on different hardware devices. Further, within this proof-of-concept different given software distributions are tested. The achieved results are analysed to evaluate the emerged behavior and to reveal possible sources of error in terms of distribution.
This thesis is conducted at the *VIRTUAL VEHICLE Research Center*[1] in the context of the *Artemis* EU project *Embedded Multi-Core Systems for Mixed Criticality Applications in dynamic and changeable Real-time Environments (EMC^2)*[2]. Within the scope of this project a significant reduction of the number of control units in vehicles should be accomplished. Figure 1.3 shows an automotive system with many single-core ECUs, each of these units is specialized for an individual criticality level. Such criticality levels are specified by the functional safety standard ISO 26262 (Road vehicles - Functional safety) [7] and are defined as Automotive Safety Integrity Levels (ASILs). According to this safety standard, ASILs are established by performing a risk analysis of a potential hazard by considering three factors of a vehicle operation: *severity*, *exposure* and *controllability*. Thereby four safety levels can arise from ASIL A, representing the lowest one, to ASIL D dictating the highest integrity requirements.

[1] www.v2c2.at
[2] http://www.artemis-emc2.eu/

The vision of EMC^2 is the implementation of multi-core systems for mixed criticality systems. Functions with different criticality levels will be able to run on the same ECU. As a result, the number of ECUs will decrease.

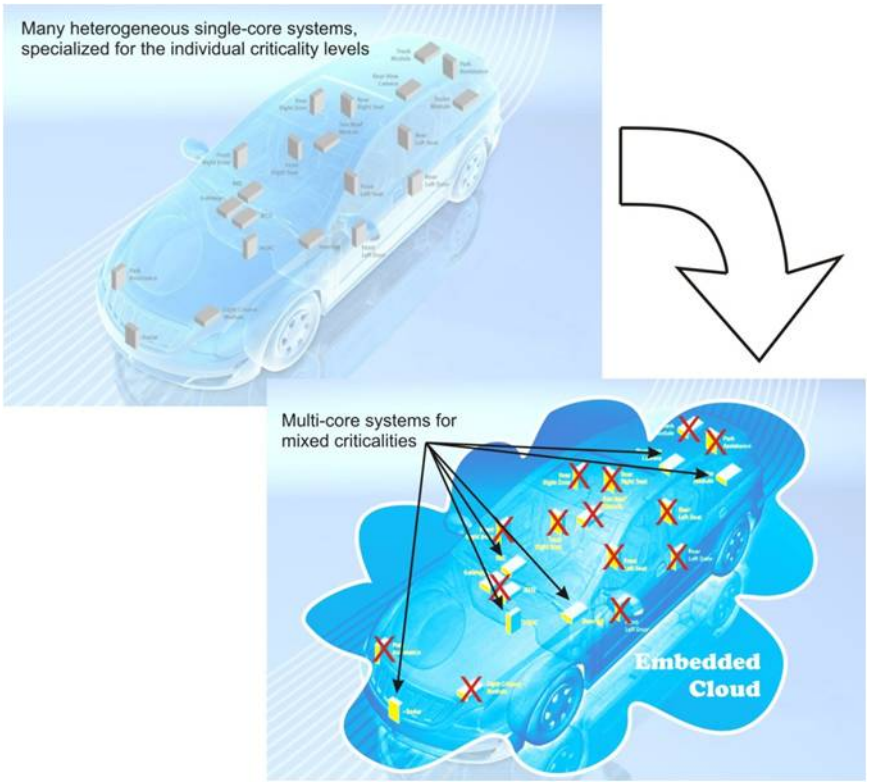

Figure 1.3: Vision of multi-core systems in the Automotive Domain [8].

Due to the fact that the EMC^2 project is in its initial phase, the results of this work will serve as a basis for a first demonstrated prototype. In figure 1.4 a rough overview of a first implementation of the prototype is shown. It represents a scenario of how to come from a Model in the Loop (MiL) to a HiL test. In more detail the figure illustrates a co-simulation model which represents the hardware integration of a model. The co-simulation tool couples a model, providing environmental data like vehicle or track data with a corresponding controller model. This model will then be converted into C-code, which needs to be extended to enable data exchange and afterwards can be integrated on an ECU. To check the correctness of the behavior of the hardware integration a different co-simulation model is needed. Instead of coupling the environmental data with a controller model running in a simulation tool (MiL), it will directly be coupled with the hardware (HiL). For the necessary data exchange between the co-simulation tool and the hardware, an appropriate communication interface needs to be defined.

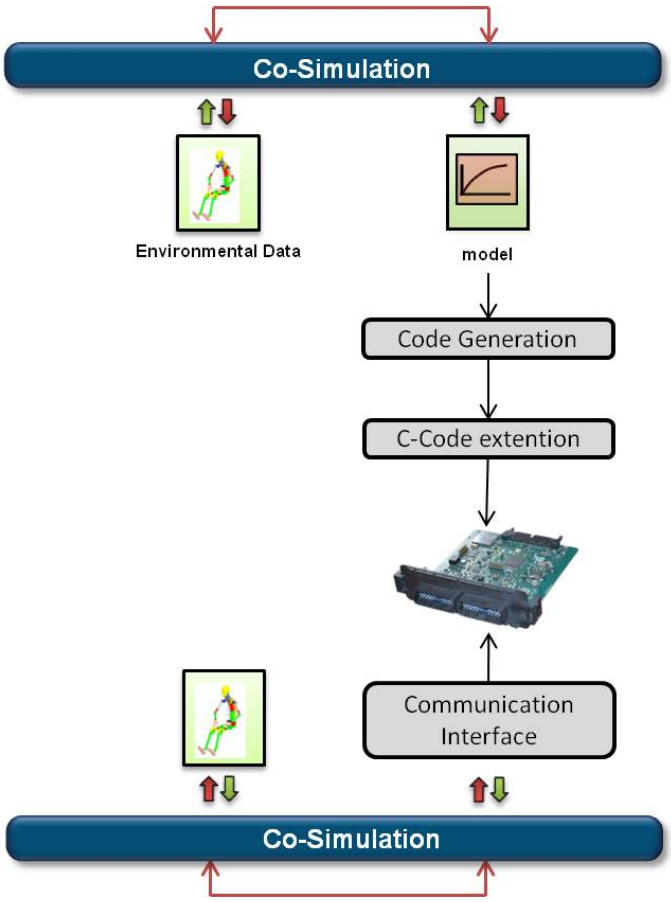

Figure 1.4: Rough overview of the hardware integration part of this Master's thesis.

This described scenario is feasible for single-core as well as multi-core applications. Within this thesis both possibilities will be implemented to be able to compare the achieved results in the end. The multi-core approach will be realized in form of a connection of multiple single-core devices.

A further project of the *VIRTUAL VEHICLE Research Center* which benefits of this Master's thesis is the *Integrated Control of Multiple-Motor and Multiple-Storage Fully Electric Vehicles (iCOMPOSE)*[3] project. Moreover, the first application example used within this thesis is provided by this project. Challenges of this project are:

- Improvement of Electric Vehicle (EV) technology to achieve
  - Adequate driving range
  - Better driveability
  - Better handling performance
- Increasing driving range
  - Increasing amount of available energy
  - Reduction of energy usage
- Vehicle components and controllers developed seperatly
  - Integration and interaction of optimising energy efficiency
  - Growing number of Information and Communication Technologies (ICT)-functions must not lead to more complexity

One goal of this project is the development of a supervisory controller which uses the informations from satellite navigation systems, the internet, and vehicle sensors, to optimally and adaptively coordinate the energy flow in an EV. A rough overview of this controller and its information cloud can be seen in Figure 1.5.

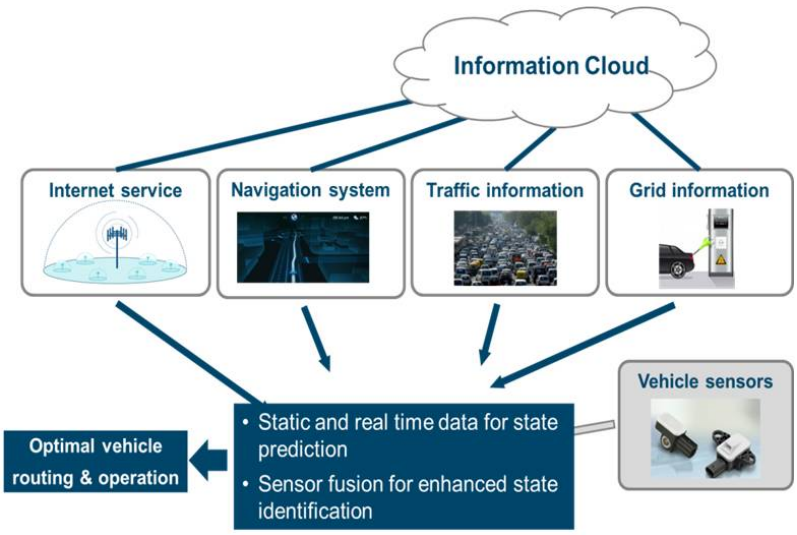

Figure 1.5: Overview of the controllers influence factors.

[3] www.i-compose.eu

One of the main objectives is the design of a *Model Predictive Controller*, to achieve an optimal vehicle operation. Due to the fact that this controller will finally be merged with other control algorithms, a very complex controller arises. As a result, it will be necessary to distribute the control function within a vehicle's ECU network to improve its performance. Figure 1.6 shows the compatibility of the integrated control software and a powerful multi-core automotive control unit. Therefore, the revealed results regarding the behavior of straightforward distributed controller functions are significant for the iCOMPOSE project and will serve as basis for an ongoing PhD thesis.

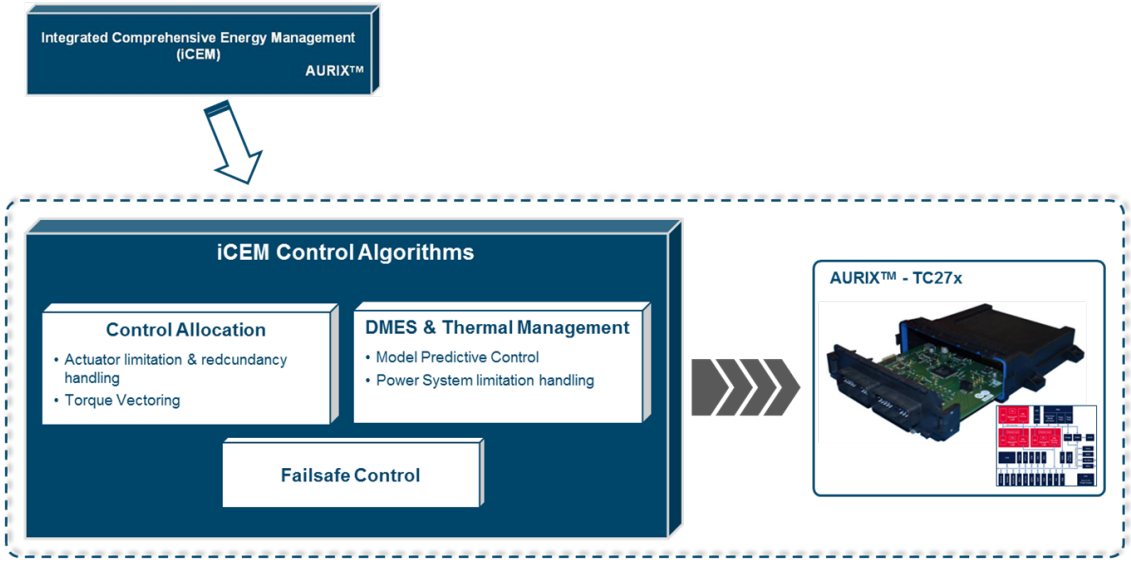

Figure 1.6: Integrated control software is compatible with a multi-core automotive ECU.

1.3 State-of-the-art

1.3.1 System architecture

System architecture and software design still are relevant topics in the automotive industry. For instance the growing number of EVs, including more and more control systems, pose new challenges to the design of in-vehicle system architectures.

In addition, *M. Lukasiewycz et al.* [9] present in their paper, that EVs consist of several components which bring along new implementation, integration and control challenges. Furthermore, they depict that current high-class vehicles consist of up to 100 ECUs, interconnected with several heterogeneous buses. This is a result of an incremental design over the last decades where new functionalities are mostly introduced by adding separate ECUs. According to the authors, this approach is reaching its limits with the growing complexity of in-vehicle networks, and thus in the future the major trend goes towards a consolidation of ECUs involving an unification of the in-vehicle networks. A further paper written by *Peti et al.* [10] also describes this *'one function per ECU'*-approach. They describe the complexity and the amount of electronics in today's luxury cars by means of the electronic infrastructure of a Fiat. Further they discuss a possible mapping of the Fiat

architecture to an integrated solution based on a specific architecture. This architecture is based on a time-triggered core architecture and a set of high-level services. In contrast to *M. Lukasiewycz et al.* [9] the presented approach also provides a foundation for mixed-criticality integration, including safety-critical as well as non safety-critical subsystems. Another different approach published by *Continental*[4] in 2009 [11], presents a so called Vehicle Control Unit (VCU), to reduce the complexity and costs of the vehicle management system. The described idea is the division of in-vehicle controllers into specific domains each of them including one head unit. Within this paper the powertrain is defined as such a domain whereby its integrated VCU is especially powerful and is responsible for the entire domain. Thus, it coordinates the control tasks of each individual powertrain control unit. To reduce the number of partner control units on the powertrain, the VCU unifies control and management functions for several powertrain parts. Further state-of-the-art investigations showed that even projects at the *Virtual Vehicle Research Center* deal with the system architecture subject. According to the iCOMPOSE project [12], today's in-vehicle control system architectures are described as follows: The automotive industry uses separate controllers to integrate different functions like energy management, vehicle dynamics, drive-ability and HVAC. Thus, consistent with the approaches discussed before, a step towards a novel integrated control structure, in this case using multi-core hardware platforms is required in the future.

1.3.2 Multi-core control systems

As described in a paper published by *D. Zhu et al.* [13], the use of multi-core processors for future automotive control systems will present new challenges for the automotive industry. According to this paper, multi-core processors have emerged to be the main computing engine for high-end servers, but also for embedded control systems. In this approach a few powerful multi-core ECUs integrate the functionalities of several single-core ECUs from the same or similar domains. Thus, in contrast to the *'one function per ECU'*-approach, a *multi-core domain control unit* is presented, reducing the number of ECUs to 10 or 20. The main integration challenge depicted in this paper concerns the scheduling of these *multi-core domain control unit*. The presented state-of-the-art solutions greatly limit the flexibility and system utilization efficiency. Thus, within this paper, new approaches to solve scheduling issues are introduced.

1.3.3 Mixed criticality

In the automotive domain it is common to evaluate the criticality level of different embedded components. Thereby, criticality is defined as the level of assurance against failure [14]. As already described before in section 1.2, such criticality levels are for instance determined as Automotive Safety and Integrity Levels (ASILs) by the ISO 26262. These levels reach from ASIL A, representing the lowest criticality level, up to ASIL D, representing the highest one. According to*K. Schmidt et al.* [15] at the moment each ECU is associated with exactly one distinct criticality level. The term 'mixed criticality' describes the new approach of integrating functions with different criticality levels on one platform. According to a review published by *A. Burns* and *R. Davis* [14] in 2015, nowadays there is

[4] www.conti-online.com

an increasing trend towards integrating such mixed-criticality components onto one ECU. Furthermore, the used platforms also mitigate from single- to multi- or even many-core architectures. According to the authors, the fundamental issue with mixed criticality systems is how to reconcile the differing needs of separation and sharing resources, to guarantee safety and to enable efficient resource usage. Along the same line, an overview of different already existing approaches to solve this issue is provided by the authors. A recurring topic in this state-of-the-art investigation is the used scheduling strategy for mixed criticality systems. In a further paper, *Ficek et al.* [16] for instance describe an approach how to use AUTomotive Open System ARchitecture (AUTOSAR) to build safe and efficient ISO 26262 mixed criticality systems. Therefore, they first evaluate three different priority-assignment strategies and afterwards show how the advantages of these three approaches can be combined to achieve safe and efficient schedules using AUTOSAR Timing Protection. Further, they present a method to determine good configurations for this AUTOSAR mechanism. Another approach regarding scheduling is described in a paper published by *S. Baruah et al.* [17]. In this paper the authors present a formal model for representing mixed criticality workloads. Further on, they demonstrate the severity of determining whether such a specified model can be scheduled to meet all its certification requirements or not.

1.3.4 State-of-the-art conclusion

These investigations showed, that in the literature different approaches, regarding ECU reduction as well as new development issues when integrating automotive functions on multi-core platforms are demonstrated. The big variety of publications regarding this topic shows the actuality and importance of this topic. Nevertheless, all these publications are rather theoretically oriented. Thus, in this Master's thesis a more practical approach shows how modelled controller solutions can be integrated and distributed on real hardware devices with little effort. A further result of this thesis is the definition of an interface, including all necessary information to realize such a distribution. In addition, this thesis examines the behavior of these integrated models and depicts possible error-sources which need to be minded.

1.4 Outline

The contents of this Master's thesis are structured as follows: chapter 2 covers a comparison of possible hardware solutions as well as a detailed description of the actually used hardware devices and software tools. Furthermore, different communication methods, which are relevant for this thesis, are presented. Chapter 3 describes different scenarios which represent the initial point of this thesis. This chapter describes the structure of the developed schema-file as well as a consistent toolchain to easily realize hardware-in-the-loop simulations. A proof-of-concept of the developed toolchain is presented in chapter 4. The conclusion and an outlook of how these results will be used in the future is presented in chapter 5.

Chapter 2

Technical Background

2.1 Hardware

For the realization of this work it is necessary to simulate an ECU network. Each ECU will at first execute a specific controller and afterwards a specific part of a distributed software. Hence, the first task is a hardware comparison to find a suitable solution.

2.1.1 Existing Hardware Solutions

The major goal of this work is figuring out how to easily integrate existing software on a given hardware. Further, the behavior of these integrated software parts (either distributed and not distributed) will be analyzed. The following investigations showed that there exist two conceivable hardware-solutions for this scope:

1. Setting up a network consisting of micro controllers. Depending on the controllers, it might be necessary to equip them with specific shields, for instance CAN- or Ethernet-Shields, to be able to communicate with each other. An example of such a hardware solution is the *Arduino UNO*[1]. However, there also exist micro controllers with already integrated communication mechanisms. An example of such a micro controller for instance is the TriBoard designed by Infineon[2].

2. Use a multi-core processor instead of creating a real ECU-network. Each core of the processor could act as a 'standalone ECU'.

Due to new projects like EMC[2] or iCOMPOSE, the deployment of multi-core ECUs is an active research field. Anyway, in this thesis different single-core boards will be used to analyze the behavior of distributed as well as not distributed simulation models running on hardware. This analysis will give an overview of potential error sources, which should later on be taken into account when integrating controllers on multi-core ECUs. As first approach a simple micro controller will be used to find a convenient procedure for the required toolchain. Afterwards, another more powerful micro controller will be used to proof this toolchain. The following sections will give an overview of the single-core

[1] http://arduino.cc/en/main/arduinoBoardUno

[2] http://www.infineon.com/cms/de/product/microcontroller/channel.html?channel=ff80808112ab681d0112ab6b64b50805

hardware components used within this Master's thesis. All these devices are state-of-the-art and are already available at the *VIRTUAL VEHICLE Research Center*.

2.1.2 VIF CAN Board V1.0

The VIF CAN Board V1.0 is a simple board designed at the *VIRTUAL VEHICLE Research Center* and is shown in Figure 2.1. Overall it contains an ATMEL processor and a NXP[3] Controller Area Network (CAN) transceiver. The processor is described in more detail in the following section.

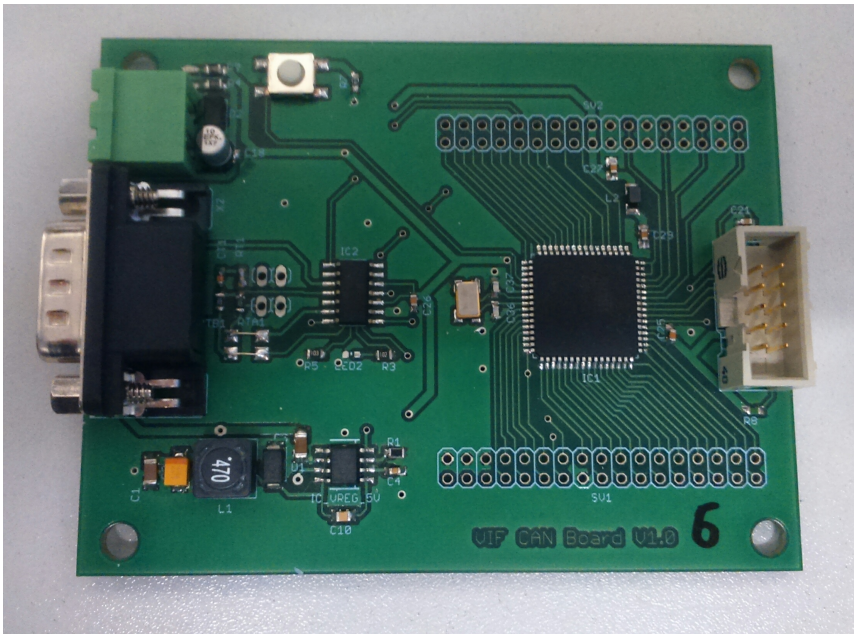

Figure 2.1: VIF CAN Board V1.0.

Atmel AT90CAN128

The AT90CAN128, described in [18], is a 8-bit micro controller based on the enhanced Reduced Instruction Set Computer (RISC) architecture. The block diagram of the AT90CAN128 is depicted in figure 2.2. The board provides the following, for this thesis relevant, features:

- 128 KB in-system programmable flash
- 4 KB EEPROM
- 4 KB SRAM
- 53 general purpose Input/Output (I/O) lines
- 32 general purpose working registers
- CAN controller

[3] www.nxp.com/

- real-time counter

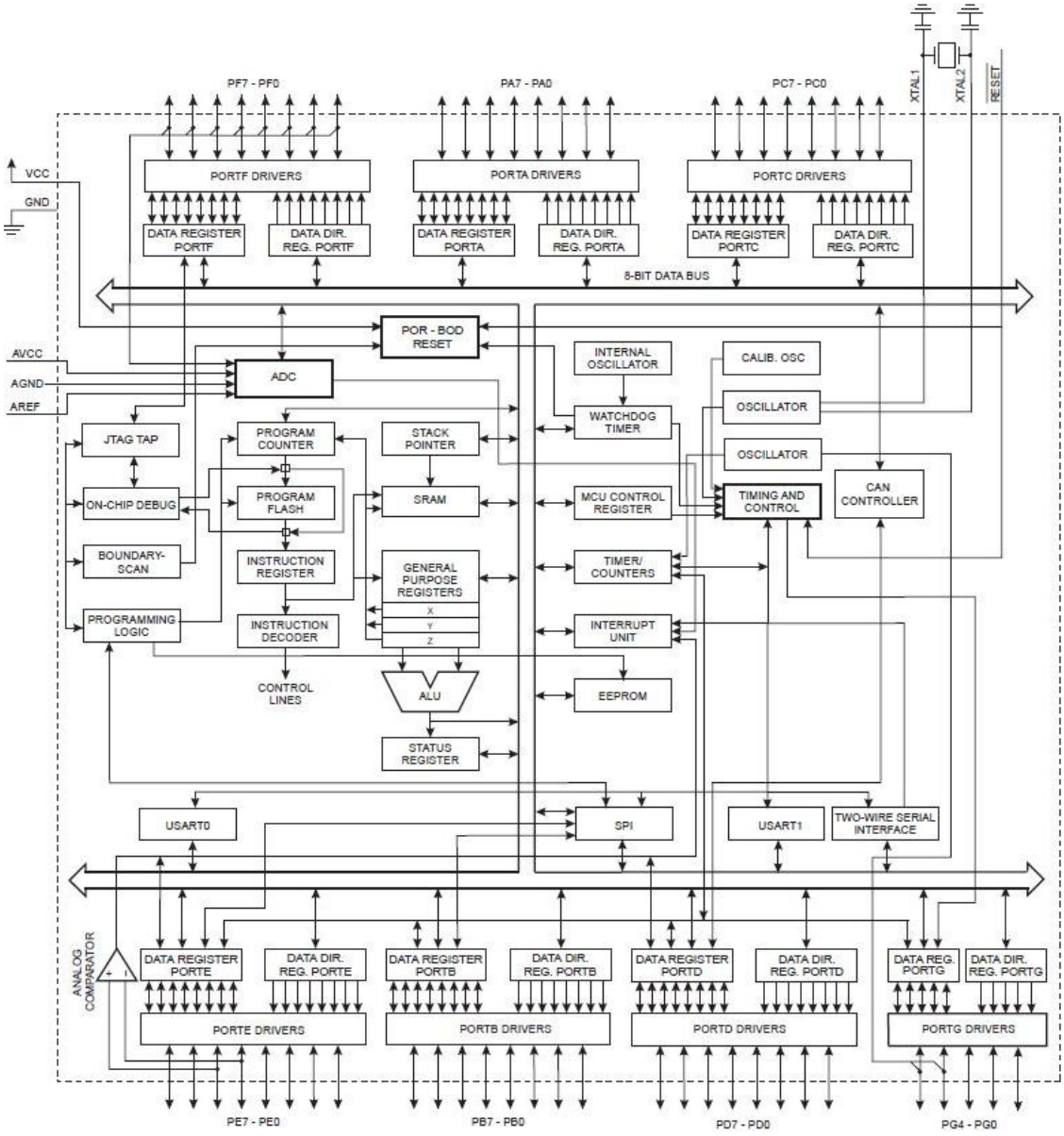

Figure 2.2: AT90CAN128 block diagram.

2.1.3 Infineon TriBoard 1797

The TriBoard 1797, described in [19], is a board designed by Infineon[4] and is illustrated in figure 2.3. This board consists of a TC1797 controller also designed by Infineon, details regarding this controller follow in the next section. Further features of this board for instance are 2 FlexRay transceivers, 2 High Speed CAN transceivers, a Universal Serial Bus (USB) to Universal Asynchronous Receiver Transmitter (UART) bridge, a Serial Peripheral Interface (SPI), and an Electrically Erasable Programmable Read-Only Memory (EEPROM).

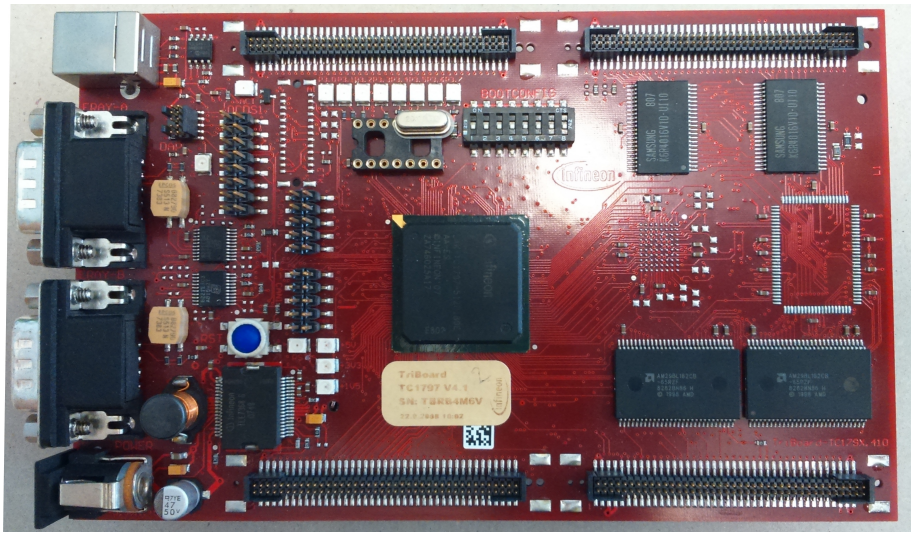

Figure 2.3: Infineon TriBoard 1797.

TC1797

According to [20], the TriCore 1797, short TC1797, is a high-performance 32-bit micro controller, based on the Infineon TriCore architecture, which mainly combines three technologies to achieve good speed, power, and economy for embedded applications. The RISC processor architecture, to provide high computational bandwidth with low system costs. Digital Signal Processor (DSP) operations and addressing modes to provide the computational power to be able to analyze complex real-world signals. And finally on chip memories and peripherals, to support even the most demanding high-bandwidth real-time embedded application tasks.

Figure 2.4 shows the block diagram of the TC1797. In general it includes a 32-bit TriCore CPU as well as a 32-bit Peripheral Control Processor (PCP) optimized for interrupt handling to unload the CPU. The following itemizations shows the main features provided by these two processors.

[4] www.infineon.com

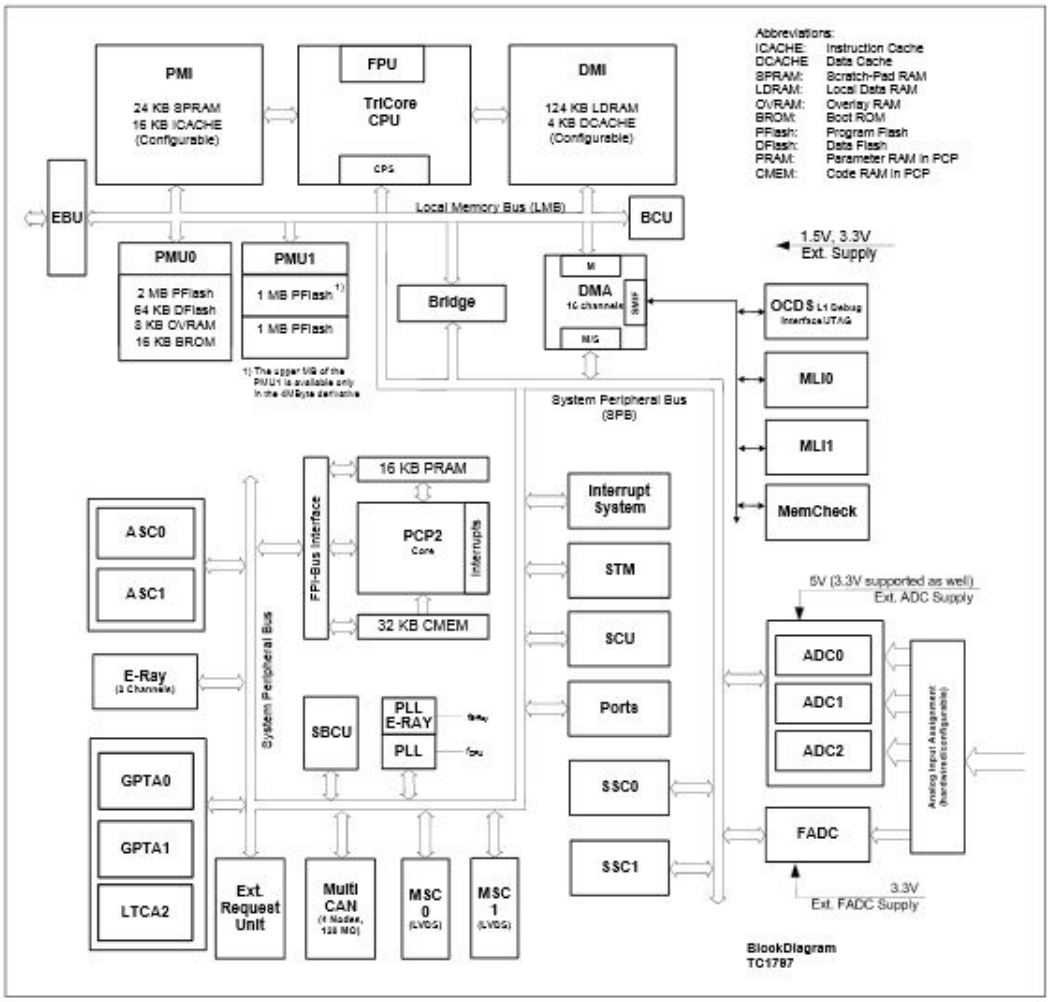

Figure 2.4: TC1797 block diagram.

TriCore CPU:

- 16/32-bit instructions for reduced code size
- Floating point unit
- Data types include: Boolean, array of bits, character, signed/unsigned integer, double-word integers, and IEEE-754 single-precision floating point
- Data formats include: Bit, 8-bit byte, 16-bit half-word, 32-bit word, and 64-bit double-word data formats
- 4 MB embedded program flash memory
- 156 KB on-chip Static Random-Access Memory (SRAM)
- 16 KB EEPROM

Peripheral Control Processor

- Read/move data and accumulate it to previously read data
- Read two data values, perform arithmetic/logical operations, store result
- Bit-handling capabilities (testing, setting, clearing)
- Dedicated interrupt system
- 32 KB code memory
- 16 KB parameter memory

2.2 Software

For the realization of this thesis various software is necessary, which will be presented in the following sections.

2.2.1 MATLAB

All used models and the tool used for code generation within this Master's thesis are based on the MathWorks tool MATLAB ®. MATLAB is a numerical computing environment, which offers many different toolboxes to extend the tools functionality. The two used toolboxes for this thesis are described in the following sections.

Simulink

All used models within this thesis are modelled in a MATLAB toolbox called Simulink®. This toolbox serves for modelling, simulating and analysing multi-domain dynamic systems. A Simulink model consists of different graphical blocks which have specific properties. A detailed understanding of all these blocks is not necessary for the performed integrations in this thesis.

Embedded Coder

With Embedded Coder® C and C++ code optimized for embedded systems can automatically be generated out of Simulink models as well as MATLAB functions. In this thesis it is used, to generate C-code out of specific model parts. This code needs to be extended by a communication mechanism in the next step. Therefore another tool is necessary.

2.2.2 Integrated Development Environment (IDE)

For further steps, a suitable Integrated Development Environment (IDE) for the used hardware is necessary. With this IDE the generated C-code can be extended by a communication mechanism and afterwards be easily flashed on the given hardware.

AVR Studio 5.1

This IDE is used for the first experiences with the AT90CAN128 micro controllers. AVR Studio[5] is a tool created by Atmel which serves to develop and debug embedded 8- and 32-bit Atmel AVR® applications. Supported languages are C, C++ and Assembler. A big advantage of this IDE is the integrated C Compiler, thus installing a separate compiler or toolchain is not necessary.

HighTec TriCore Toolchain

To develop and compile code for TriCore processors the Hightec TriCore toolchain provided by Infineon[6] is used. The Hightec TriCore toolchain consists of an Eclipse based IDE including a C/C++ cross compiler and an USB-debugger.

2.2.3 CAN tools

To comprehend in more details how the implemented CAN interface is working, the company *PEAK Systems*[7] provides different useful tools for instance to monitor the CAN traffic. Furthermore, this company provides the used *PCAN-USB* adapter, which enables simple connection to CAN networks via USB.

CAN monitor

To be able to represent the CAN data traffic, PCAN-View® is used. This software enables the user to send specific CAN messages over the channel, to test the implemented CAN interface, as well as record the transmitted data packages.

2.2.4 Co-Simulation

For data-exchange between a hardware device, executing different controller tasks, and its environment (input and output data) a co-simulation tool is needed. These kind of testing is called HiL. Such a co-simulation tool enables the user to couple different models which are generated and simulated in different simulation tools. A big advantage of co-simulation is that this coupling does not influence the used algorithm or the step size of the included models.

[5] www.atmel.com/microsite/avr_studio_5/
[6] www.infineon.com/cms/en/product/channel.html?channel=db3a304344134c7a014420d628fa76ec
[7] www.peak-system.com

ICOS

In this Master's thesis a tool developed at the *VIRTUAL VEHICLE Research Center* called Independent Co-Simulation (ICOS)[8] is used. ICOS allows the coupling of a variety of engineering tools like MATLAB/Simulink, MSC Adams, Excel or CarMaker, thus the platform allows a simulation of an entire vehicle if needed [21]. Figure 2.5 presents the coupling of different simulation tools with the co-simulation platform.

Figure 2.5: Coupling of different simulation tools via ICOS.

Coupling Methods

As described in [22] coupling of different models corresponds to a synchronisation of the models. Each model is simulated for a specific time interval whereas a data exchange takes place at certain points in time. This time interval is called *macro step size* ΔT. Within this interval each model is solved independently by its individually determined solution algorithm and its fixed or variable step size. The step size used by the solution method is called *micro step size* δT and is defined by the developer of the individual model and the used solver (variable step-size solver)[23]. Figure 2.6 shows a graphical representation of these different step sizes. A simulation which includes different models solved by different micro step-sizes is called a multi-rate simulation, in general.

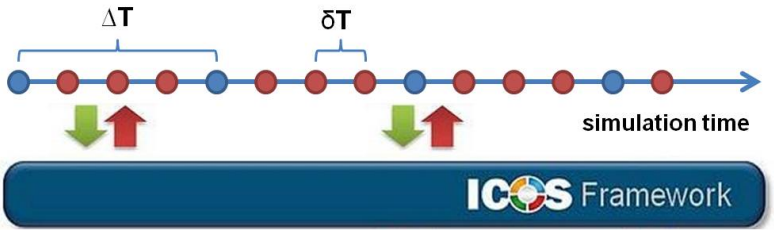

Figure 2.6: Data exchange with ICOS.

[8] www.v2c2.at/en/products/icos/

Furthermore, ICOS provides two different coupling mechanisms: *sequential* and *parallel* coupling. Examples of these two possibilities are depicted in figure 2.7. In case of a sequential execution order, illustrated on the left side of figure 2.7, each subsystem is solved after the other, thus only the input signals of the first subsystem has to be extrapolated. In contrast to that, for a parallel execution of a simulation, shown on the right side of figure 2.7, an extrapolation of the coupling data has to be done by every subsystem.

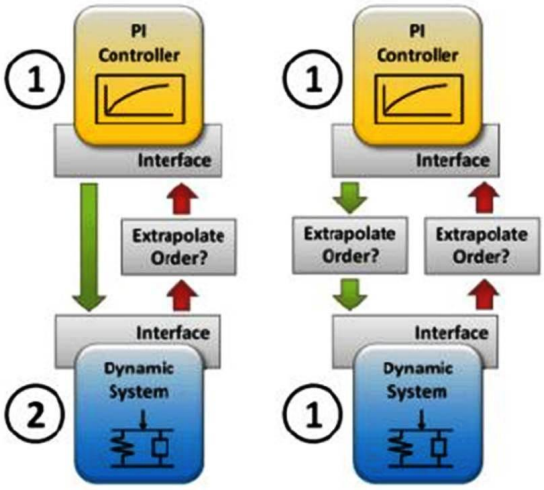

Figure 2.7: Different possibilities of coupling mechanisms.

2.3 Communication Methods

In the automotive domain, different standards exist to exchange data. Within this Master's thesis a communication method is needed to enable communication between ICOS and the used hardware. Due to the fact that ICOS only supports the transmission of User Datagram Protocol (UDP) packages and CAN messages a decision between these two technologies needs to be made. The following sections give a short introductions to the main concepts of these technologies. Based on this comparison a decision is made in the end.

2.3.1 Controller Area Network (CAN)

The Controller Area Network is a standard defined by the International Standardization Organization (ISO)[9]. It describes a serial communication bus developed for the automotive industry and at the moment is the most widely used in-vehicle network standard. In contrast to other communication networks like Ethernet or USB, CAN does not send data point-to-point from one node to another under the supervision of a central bus master, but rather broadcasts short messages to the entire network. Hence, CAN is a multi-master, message broadcast system, which covers small networks including buses with a length up to $40m$. Moreover, the CAN standard includes Carrier Sense Multiple Access/Collision

[9] www.iso.org

Reduction (CSMA/CR) as transmission mechanism to ensure reliable message delivery. Carrier Sense Multiple Access (CSMA) means, that each node needs to wait a specific time interval before attempting to send a message to avoid collisions. Nevertheless, if two nodes want to send simultaneously bit-wise arbitration is used, whereas each message keeps a preprogrammed priority in his identifier field. As a result, it is guaranteed that the message with the higher priority is allowed to send. [24]

As described in [25] and depicted in figure 2.8, the CAN architecture defines the two lowest layers regarding the Open Systems Interconnection (OSI) model [26] which consists of seven layers: the *physical* and the *data-link* layer.

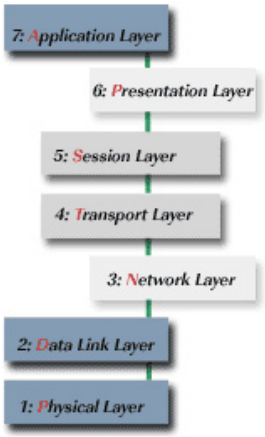

Figure 2.8: OSI model for the CAN standard [27].

These two layers are defined by ISO 11898-1:2003. The most common physical layer standards are ISO 11898-2:2003 also known as *high-speed CAN* and ISO 11898-3:2006 also known as *low-speed* and *fault-tolerant CAN*. The high-speed standard covers CAN requirements for data rates up to 1 Mbit/s and is the most used physical layer standard for CAN networks. The low-speed and fault tolerant standard covers requirements for rates up to 125 kbit/s and is mainly used for body electronics in the automotive industry. [27]

Message frame formats

The CAN standard specifies two different types of message frame formats. The only essential difference between these two formats is the identifier length. A standard CAN identifier consists of 11-bits and provides signal rates from 125 kbit/s up to 1 Mbit/s. For in-vehicle communication only this identifier is used. After the amendment of this standard identifier an extended one was created, consisting of 29-bits. This 29-bit identifier is made up of the 11-bit base frame identifier and an 18-bit identifier extension. A standard identifier offers 2^{11} or 2048 different message different identifiers whereas the extended one offers 2^{29} or 537 millions. Due to the fact that both formats have to co-exist on one bus, the CAN standard defines that a 11-bit message always has priority over a extended message. CAN controllers supporting extended frame format messages are also able to send and receive standard frame format messages, however a controller which only covers

base frame format messages is not able to interpret extended ones. The general structure of the CAN message frame format is illustrated in figure 2.9. Beside the identifier field there are several other fields set within a CAN message. Another meaningful field is the data field which includes the actual transmitted message. This data field can handle a payload of up to 8 byte. [28]

Figure 2.9: Structure of the CAN message frame format.

2.3.2 UDP (User Datagram Protocol) based on Ethernet

UDP belongs to the internet protocol suite and is defined within the Request For Comments (RFC) 768 document [29]. The main idea is the connectionless transmission (multicast and broadcast) of messages within a network with a minimum of protocol mechanism. UDP is a transaction-oriented protocol which does not guarantee any reliability and correct message delivery. In contrast to CAN, this protocol is design to enable data exchange within huge world-wide networks. Moreover, unlike the CAN standard, UDP messages are sent point-to-point from a source to a destination by using Ethernet standards whereas the Internet Protocol (IP) address of the destination needs to be known. [29]

As described in [30] for sending UDP packages four layers regarding the OSI model needs to be defined: the *physical*, the *data-link*, the *internet* and the *transport* layer.
Hence, beside defining low-level protocols and mechanisms it is also necessary to implement Ethernet and the associated address and transport mechanisms to enable data exchange with UDP.

Package structure

A UDP package is always part of an Ethernet frame. The structure of such a packet is depicted in figure 2.10. The UDP header consists of four 16-bit fields to define the *source* and *destination* port as well as the *length* of the data package and a *checksum* for error-checking. The data field of the packet can include up to 65507 byte of information. To transmit such a packet beside the UDP header two further headers for the internet protocol and Ethernet are needed. These headers include information like the acIP and Media Access Control (MAC) address of the source and destination application. [30]

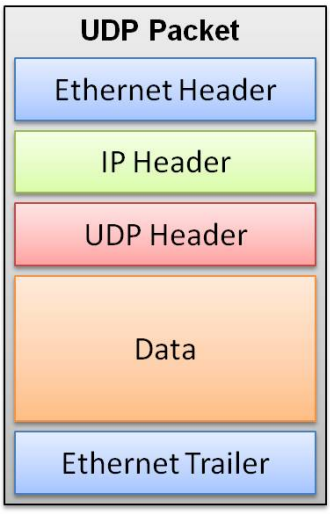

Figure 2.10: Structure of a UDP packet.

2.3.3 Comparison and Decision

A comparison of the above mentioned properties of the two message formats shows that the implementation of a CAN interface is much easier to realize than using UDP packages based on Ethernet. An argument for the usage of the CAN protocol is the fact, that this protocol enables the prediction of the resulting delay times caused by the communication mechanism. In contrast, the resulting delay times when using Ethernet with UDP messages are unpredictable. Furthermore, the CAN standard is well-established in the automotive domain and is a reliable protocol whereas UDP is unreliable and therefore is more suitable for the exchange of uncritical data between applications throughout the whole world. Due to these facts and because all used hardware devices within this Master's thesis supports CAN communication, this communication standard will be used for all implementations in this work.

Chapter 3

Design

The following sections describe the developed toolchain and the resulting interface to distribute hardware.

3.1 Hardware integration scenario

As a first approach to define a suitable toolchain for hardware integration of simulation models, a rough scenario has been defined. Figure 3.1 illustrates the first approach of this scenario.

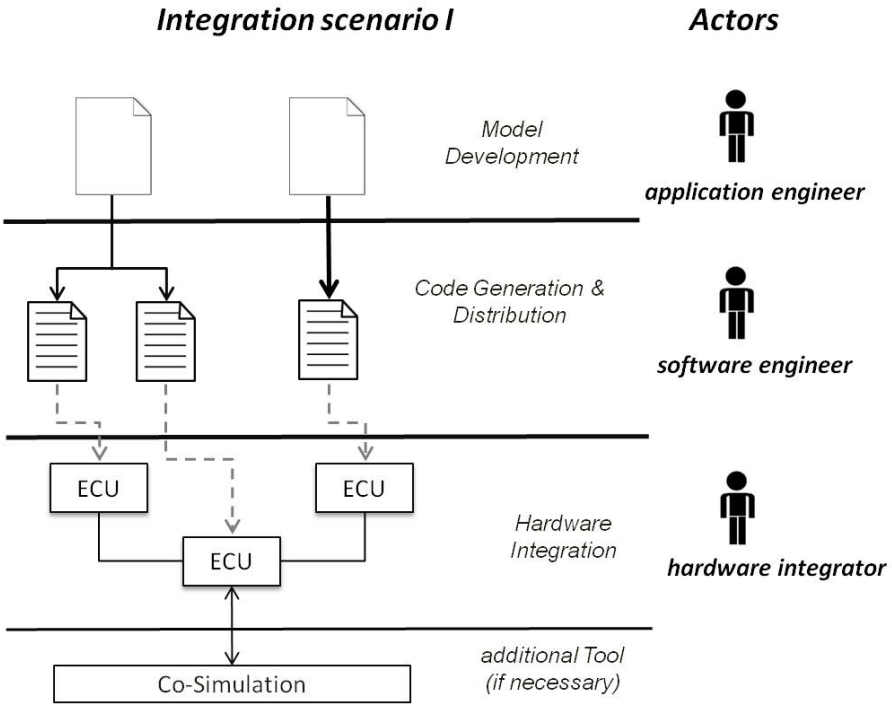

Figure 3.1: First approach of a scenario definition.

The three main steps, necessary to integrate a given model on an ECU are:

- *model development*
- *code generation and distribution*
- *hardware integration*

The depicted actors are the application engineer, the software engineer and the hardware engineer. An application engineer develops simulation models using a simulation tool. To generate code out of these models a software engineer is needed. Finally, the hardware integration is performed by an integrator.

This first approach of an integration scenario does not fulfil the requests for a proper cooperation of all appearing engineers, thus a second scenario has been defined which is shown in figure 3.2.

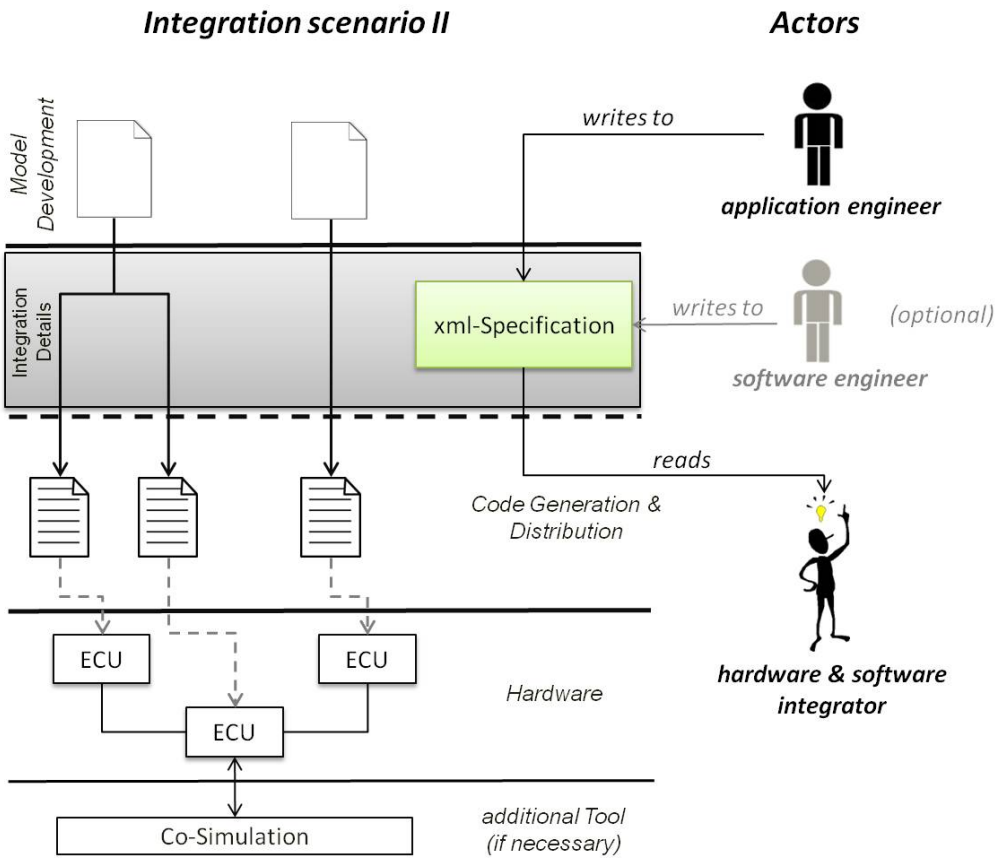

Figure 3.2: Second scenario definition.

This second approach describes the following scenario:

1. Simulation models are defined by an application engineer.
2. Application engineer adds basic information regarding the desired hardware integration to a configuration Extensible Markup Language (XML) file (e.g. desired distribution, hardware devices,...).
3. If necessary, a software engineer adds more precise information regarding the software components to the xml-file.
4. A different engineer called integrator reads the XML-file and gets all necessary information about the desired distribution and hardware components.
5. Executable code will be generated out of the controller functions/simulation model due to the desired distribution.
6. Software functions will be integrated on ECUs.
7. Communication between ECUs/co-simulation tool will be set up.
8. It is either possible to connect ECUs among themselves or with a co-simulation tool.

The main idea of this scenario is the insertion of a new phase between the model development phase and the code generation and the distribution phase. By inserting this new phase, the hardware integrator will be able to perform the whole hardware integration scenario on his own. The therefore required information is provided by an application engineer and a software engineer who add the required information regarding the requested hardware integration to an especially developed XML-file defined as XML Schema Definition (XSD). This file serves as basis for the hardware integrator. Thus, by reading this file, the hardware integrator retrieves all necessary information to be able to generate the required code parts for the following hardware integration. Furthermore, he can find out which hardware devices should be used and how these devices will communicate. A detailed description of the newly created XSD schema can be found in the following section.

3.2 Developed XSD schema

To enable a single person to perform the whole hardware integration on his own, specific input from different development fields is necessary. A state-of-the-art analysis revealed that there already exist different approaches of description formats to define and represent properties of software components. The following section gives an overview of already existing approaches.

3.2.1 State-of-the-art description formats

One approach, described in [31], is the Field Bus Exchange Format (FIBEX), which is maintained at the Association for Standardisation of Automation- and Measuring Systems (ASAM)[1]. This XML-based file format is a freely available standard used to describe complex, message-oriented communication systems, e.g. in-vehicle networks. The specified exchange format covers the functional network, the system topology and the communication level. Furthermore, this standard enables the description of gateway configurations. Thereby FIBEX tries to be widely independent from all communication controller implementations and protocols. Supported communication networks are CAN, Local Interconnect Network (LIN), Media Oriented Systems Transport (MOST), and FlexRay.

A further approach is defined in AUTOSAR[2]. In AUTOSAR, applications consist of different software components. A detailed description of these software components is given in [32]. In general, according to [33], a series of steps to create executable ECU components are defined in AUTOSAR, whereas XML is the used interchange format. These XML-files are based on a schema file, which roughly consists of four parts. With respect to this Master's thesis, the most interesting part of this schema is the system template. A detailed description of this part can be found in [34]. Within this template the overall system is defined by storing information about the bus systems, the signals, the mapping and the topology.

Further research revealed, that there also exists a well established defacto standard called Data Base CAN (DBC), which is presented in [35]. This DBC file format is used for the description of the communication of CAN networks. Hence, it serves as basis for the development of communication software for an ECU which shall be part of a CAN network, whereas the functional behavior of the ECU is not defined within the file.

This research showed that there already exist several approaches of description formats which could be used to provide the necessary information regarding hardware integration within this thesis. The implementation of one of these approaches is not in scope of this Master's thesis, but might be considered in future projects. For the proof-of-concept implementation in this thesis, a new XML-based file format has been defined in form of an XSD-schema to enable the desired data exchange.

3.2.2 Data exchange file format

As already mentioned before, a new XML-based file format to exchange all necessary information between several engineering domains has been defined within this Master's thesis. The determined specification includes all necessary information regarding the required distribution of the different model functions, the therefore used hardware type and the used communication interface. The following listing gives an overview of the determined XML-elements specified in an XSD-schema and its meaning:

[1] www.asam.net
[2] www.autosar.org/

- **distribution**
  The root node of the XML-schema. This node includes all necessary information for a specific function distribution.
- **modelPath**
  States where the required model is stored.
- **modelName**
  Name of the model.
- **modelASIL**
  Safety level of the model given as ASIL value. This attribute is defined as optional.
- **modelSolver**
  The used solver type of the simulation model. This attribute is defined as optional.
- **modelStepSize**
  The used step-size of the simulation model. This attribute is defined as optional.
- **distributionFunction**
  Specifies which specific functions of the model will be integrated on hardware. This attribute can occur multiple times in an xml-file depending on the number of functions to distribute.
- **partName**
  States how the function block is called in the model.
- **functionASIL**
  Safety level of a specific function of the model given as ASIL value. This attribute is defined as optional.
- **inputSignals**
  An overview of all input signals of a specific function block. This attribute can occur exactly once for each distribution function and consists of multiple signal definitions.
- **outputSignals**
  An overview of all output signals of a specific function block. This attribute can occur exactly once for each distribution function and consists of multiple signal definitions.
- **signal**
  A description of a specific input or output signal. This attribute can occur multiple times in the inputSignals and outputSignals attribute.
- **signalName**
  Name of a specific input or output signal.
- **signalRange**
  Range of a specific input or output signal.

- **signalPrecision**
  Required precision of a specific input or output signal. The precision is indicated as a power of ten, whereas the exponent indicates the number of necessary decimal digits.

- **hardwareDevice**
  Includes information regarding the required hardware device. This attribute occurs exactly once for each distribution function.

- **HWtype**
  Type of the desired hardware.

- **communicationInterface**
  Type of the desired communication interface.

Figure 3.3 shows an overview of the determined XSD-schema. This schema is defined within an XSD-file and serves as basic underlying schema for each generated XML-file. The content of this XSD-file can be found in Appendix C.

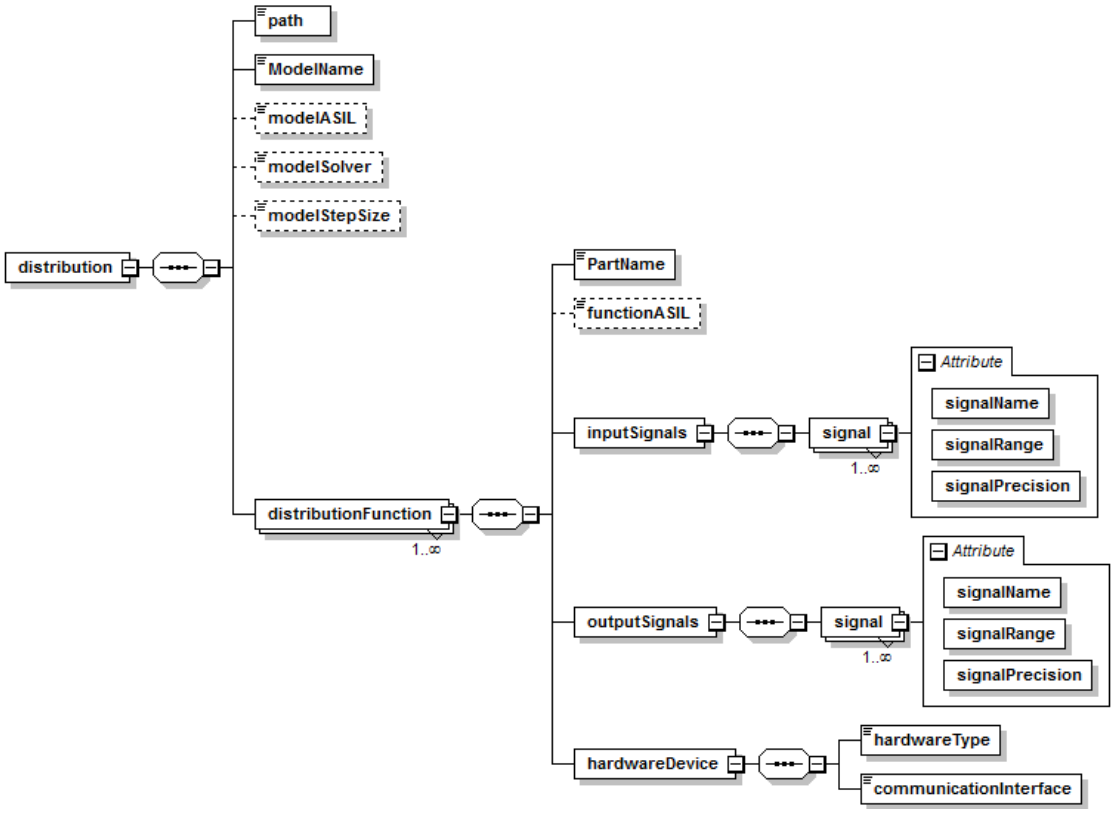

Figure 3.3: Defined xsd-schema represented as graph. Dashed-lined blocks represent an optional element, solid-lined blocks represent a required element.

This predefined XML-file will be filled in by various engineers belonging to different domains throughout a development process. In the end the file includes all information regarding a specific model and its desired distribution. At this stage of the development, this defined file format will be evaluated and therefore all necessary information is added to the file by hand. As a next step, a xml-editor could be implemented, to automatically generate XML-files out of the users input data and furthermore generate clue-code for the CAN conversion of the entered in- and output signals. However, this could be part of future work and therefore is not part of this thesis.

3.3 Hardware integration toolchain

The following sections describe which general steps are necessary to run a simulation model as single- or multi-core implementation.

3.3.1 From model to code

As a first step, the model functions, which need to be integrated as single- or multi-core solution, have to be determined. This information can be taken from the *modelPart* section of the specific XML-file. Depending on the used modelling and simulation tool, code can then be generated for each necessary model block by using an appropriate tool. Within this Master's thesis only simulation models generated in MATLAB Simulink are used, whereas Embedded Coder is used to automatically generate C-code out of each function block.

3.3.2 Code extension

In the next step, the generated C-code needs to be extended by an appropriate communication interface. Information about the desired communication interface can be taken from the *hardwareDevice* section of the XML-file. Within this Master's thesis CAN communication is used for all implementations.

To extend the previously generated code by a CAN communication interface, information regarding all in- and output signals of each function block is needed. This information can also be extracted from the XML-file. It provides information about the signal range and its desired precision. These values are used to calculate the parameters *offset* and *gain*. The formulas to calculate these values are illustrated in equation 3.1.

$$\begin{aligned} Range &= [min, max] \\ \Delta &= max - min \\ gain &= \frac{\Delta}{2^{number\ of\ bits}} \\ gain &\leq precision \\ offset &= 0 + min \end{aligned} \tag{3.1}$$

The parameters *offset* and *gain* are needed to convert input and output values to a format which can be represented in a CAN message. Thus, it is possible to transmit and receive positive as well as negative decimal values. The used formulas for this conversion at receiver- and transmitter-side are taken from the ICOS Real Time Wrapper implementation and are shown in equations 3.2 and 3.3.

CAN Transmit:

$$x = \frac{value - offset}{gain} \tag{3.2}$$

CAN Receive:

$$y = value \cdot gain + offset \tag{3.3}$$

x... converted value to transmit
y... converted received value
value... original value
offset... offset needed for the zero shift
gain... a scaling factor for the achieved precision

The number of necessary bits to represent each input and output signal is determined through the given precision. Thus, after the determination of an appropriate scaling factor and an offset for each signal the number of necessary CAN messages can be defined. As already described in section 2.3 a CAN message consists of 8 byte which comply with 64 bit. Within this Master's thesis, for reasons of simplicity it was decided that signals can only be represented by a number of bits that is divisible by 8, thus a maximum of 8 signals can be packed in a CAN message.

The previously generated C-code needs to be extended by a mechanism to pack and unpack CAN messages. Therefore, the afore mentioned equations 3.2 and 3.3 for the conversion of received and transmitted data are implemented for each signal by using the calculated parameters for *offset* and *gain*. Afterwards the adapted code is ready to run and can be flashed on an ECU.

3.3.3 Hardware-in-the-Loop testing

At last, to test the correctness and behavior of the hardware integration, a co-simulation model needs to be set up. It would also be possible to test the hardware integration by connection several ECUs or sensors and actuators, however within this thesis HiL testing is used solely. As already mentioned before, the used co-simulation tool is ICOS.

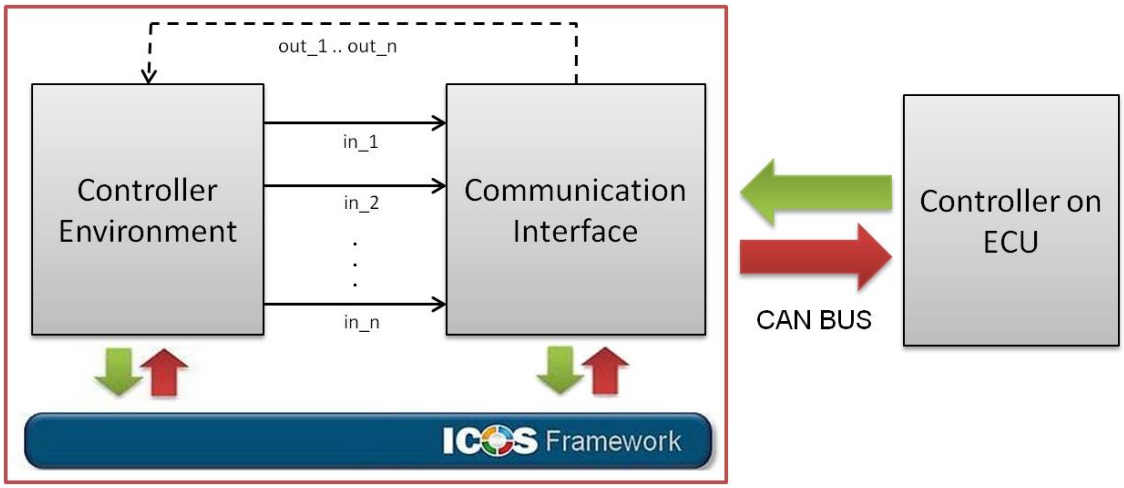

Figure 3.4: Co-Simulation realized in this work.

Figure 3.4 illustrates how ICOS is used to perform HiL testing. The first block used in the co-simulation is a controller environment, this environment provides information regarding the vehicle, the used track or the plant model coming from tools such as AVL Cruise or MATLAB/Simulink. This data is sent to a communication block, which can also be seen in figure 3.4. This block enables the creation of UDP packages or CAN messages to exchange data between ICOS and a hardware device. In this case, only the CAN features of the co-simulation tool are used. To generate such a communication block in ICOS a Real Time Wrapper model needs to be created.

Chapter 4

Proof of Concept

To get an idea of how the claimed toolchain for hardware integration could look like, two different automotive Simulink models are integrated on different single-core devices within this prove of concept. The achieved investigation results will further give information if the defined XML-properties are sufficient. All used models are provided by the *Virtual Vehicle Research Center*. To get an overview of the implemented functionalities and the main ideas behind these models, each model will shortly be introduced in the particular section. However, a detailed knowledge of the functionalities of these models is not necessary for the performed integrations.

4.1 Test setup

Figure 4.1 shows the test setup used for this proof-of-concept. It illustrates the setup, when using two TriBoards for the integration of software functions, however exactly the same setup is used when working with the VIF CAN Boards. For all following single-core integrations, only one micro controller integrates software functions, in the multi-core case both controllers integrate specific software parts and they communicate via a CAN bus.

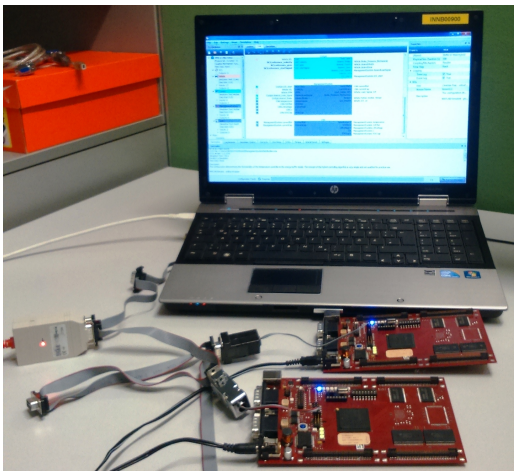

Figure 4.1: Overview of the experimental setup.

This setup includes a laptop, needed to execute the necessary co-simulation software (ICOS), which performs HiL simulations and thus provides input data for the controllers. Within ICOS the Real Time Wrapper is used to periodically transmit input values, coming from other coupled models (e.g. Simulink, AVL Cruise,...), to the hardware. Therefore, the laptop is connected to a CAN interface via USB. The controllers are coupled to this USB CAN interface by a CAN bus.

4.2 Dual Mode Energy System (DMES)

The first model, used to integrate on hardware, belongs to the iCOMPOSE project and simulates specific features of a Lotus Evora 414E. The driving-cycle used for all these simulations is based on the Lotus test track located in *Hethel/UK*.

The Lotus Evora 414E, depicted in figure 4.2, is described in [36] as a plug-in Hybrid Electrical Vehicle (HEV) including a range extender and a rear wheel drive. With this drive-train concept and an unique implementation of control systems, an eco-friendly vehicle with performance-oriented driving can be achieved. Within the iCOMPOSE project this vehicle is re-designed to a fully electric vehicle, which serves to evaluate the comprehensive energy management including the Dual Mode Energy Storage (DMES) system and semi-autonomous driving. For this Master's thesis the behavior of a specific control function of the Lotus Evora 414E running on hardware will be investigated.

Figure 4.2: Lotus Evora 414E.

Figures 4.3 and 4.4 illustrate the distance and the height profile of the *Hethel* track. The distance profile shows the vehicle velocity as a function of the time. The height profile the height as a function of the driving distance.

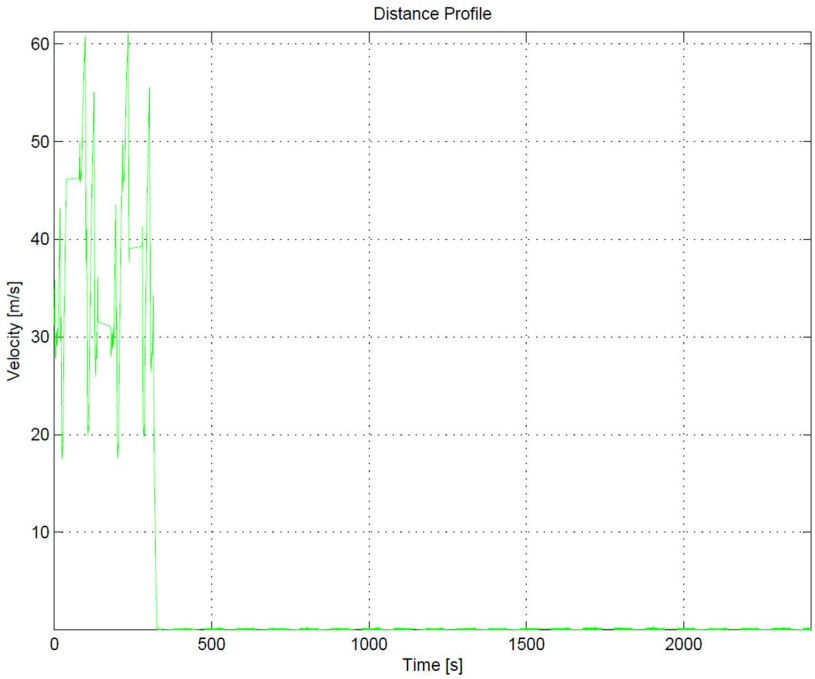

Figure 4.3: Distance profile of the *Hethel* track.

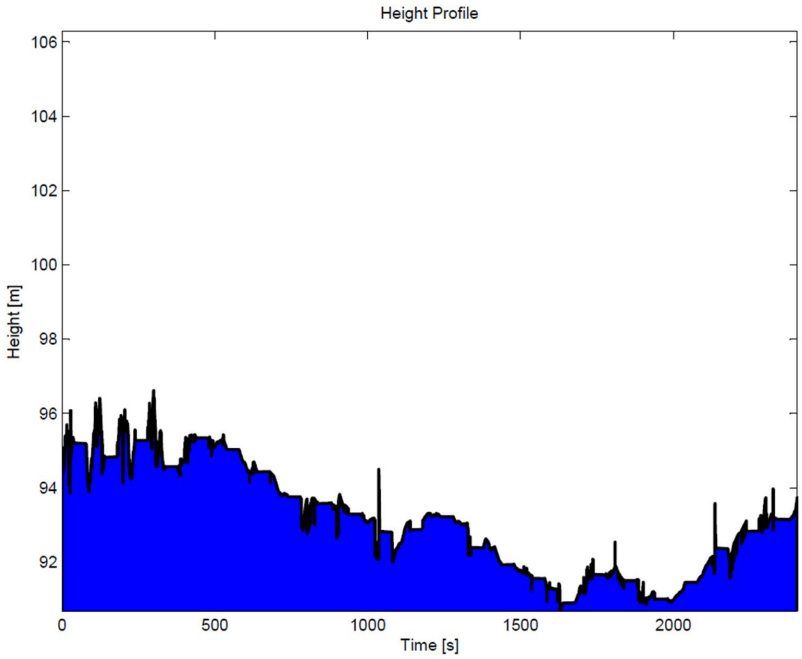

Figure 4.4: Height profile of the *Hethel* track.

For this first hardware integration the above mentioned DMES model, implemented in Simulink is used. This model contains a specific controller called Dual Mode Energy Storage Controller (DMESC) and was designed by *Fraunhofer-Institut für Verkehrs- und Infrastruktursysteme (IVI)*[1] located in Dresden. The main task of this controller is the selection of the optimal distribution of energy between battery and super capacitor, taking into account reliability of the power supply, energy efficiency and battery lifetime.

To get an idea of why this DMES system includes a battery and a super capacitor, a comparison of different energy storage technologies is shown in form of a *Ragone chart* in figure 4.5. According to [37], an energy storage device is characterized by its energy and power being available for a load. The *Ragone chart* depicts the performance of different energy-storing devices by valuing the energy density, in Wh/kg, versus the power density, in W/kg. In terms of an electrical vehicle the energy density presents the possible driving range, whereas the power density shows how quickly that energy can be delivered or stored. It is shown, that lithium-ion batteries have a high energy density but only a low power density. In contrast to that, super capacitors have a rather high power density, but a low energy density. Considering that, a combination of these two technologies, as it is done in the DMES system, leads to a solution with a rather good power density as well as a rather good energy density.

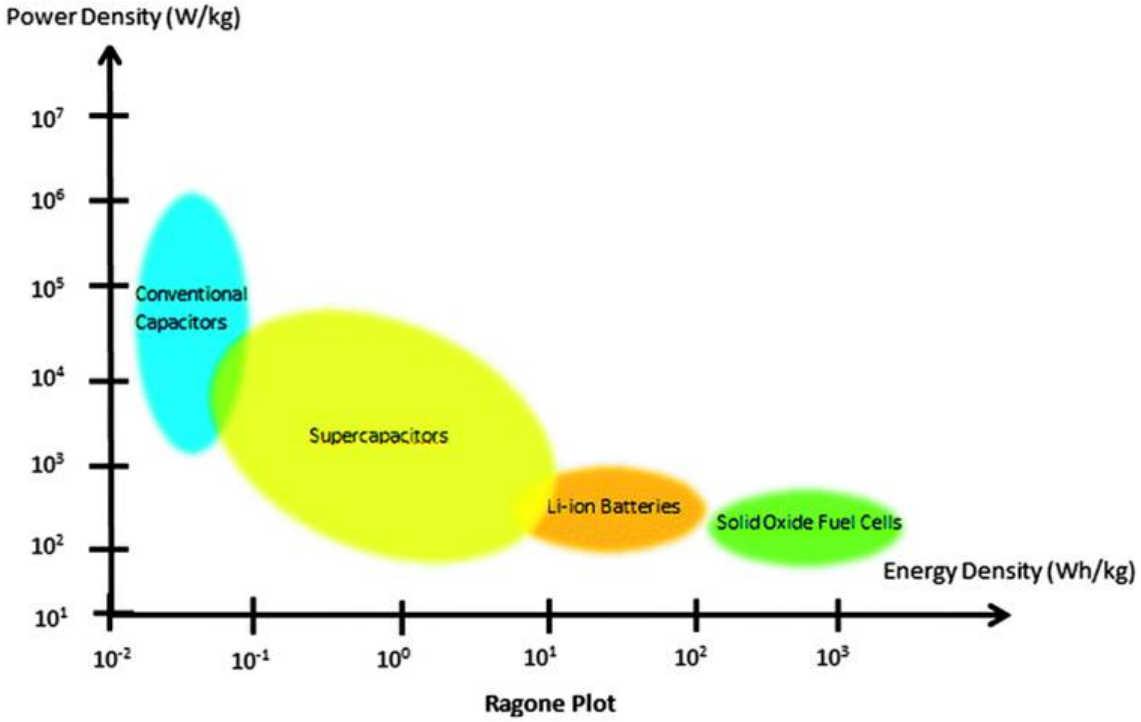

Figure 4.5: Ragone Chart [38].

[1] www.ivi.fraunhofer.de/

A comparison of the achieved battery currents with and without using the DMES system in the simulation of the Lotus Evora 414E are presented in figure 4.6. The upper sub-plot depicts the vehicle velocity as a function of time, the lower one the associated battery currents. It is shown, that the current reaches higher peaks without the DMES system (red signal) than when performing power distribution between the integrated battery and the super capacitor through the DMES system (blue signal).

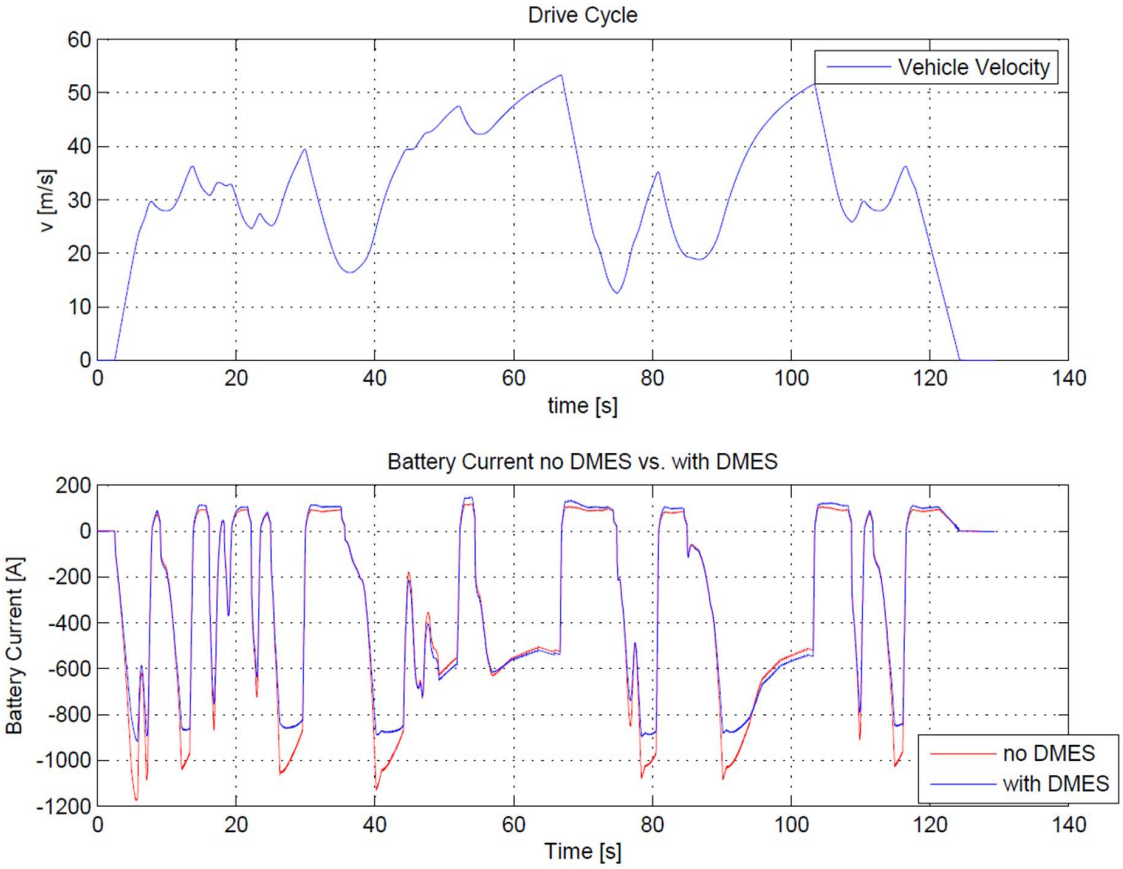

Figure 4.6: Comparison of the battery currents with and without DMES.

Figure 4.7 shows the main components of a DMES:

- A battery, as energy storage device
- A super capacitor, for quick charging/discharging (no storage)
- A Direct Current Direct Current (DCDC) converter, to convert from one voltage level to another
- An electric motor controller

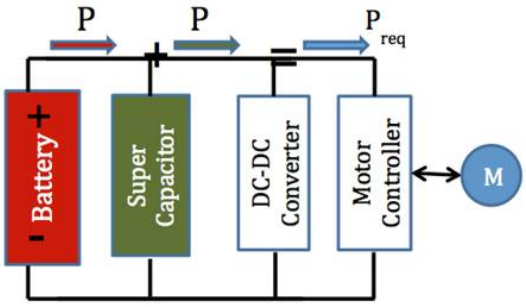

Figure 4.7: Schematic layout of the DMES System [39].

Figure 4.8 depicts the implementation of these components in MATLAB Simulink. Input signals for this model are the velocity of a given vehicle on a specific track and the power load, both values are represented over time.

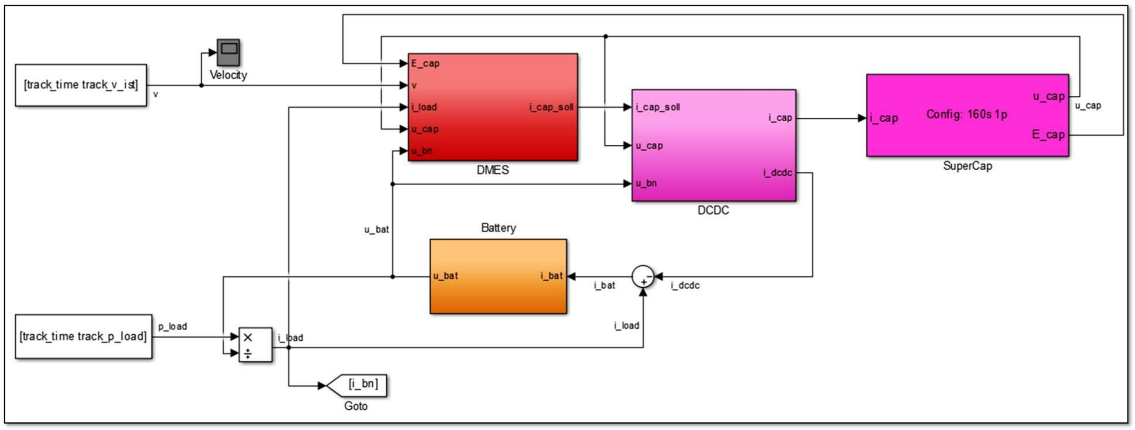

Figure 4.8: Overview of the DMES model.

The interesting part of this model regarding hardware integration and distribution is the DMESC system, which is shown in figure 4.9. This system block is responsible for the power distribution between the integrated battery and the super capacitor, to avoid voltage peaks which reduce the battery lifetime tremendously. It consists of two PI controllers, the first one, depicted as red path, to maintain a certain super capacitor State of Charge (SoC),

depending on the current vehicle velocity. The second one, depicted as green path, to take any high frequency components from the battery current.
More precisely the specified power used in this model derives from the vehicle velocity and the given vehicle model. This obtained power splits up between battery and super capacitor. As shown in figure 4.9 the high frequency components of the battery current are filtered out by a high-pass filter. By using the vehicle velocity and a lookup table the energy for the SoC calculation is determined. The resulting behavior of this controller model leads to the fact that the battery will not be used as much as the super capacitor, who is responsible for the buffering of current peaks to extent the lifetime of the battery and is able to charge and discharge very fast.

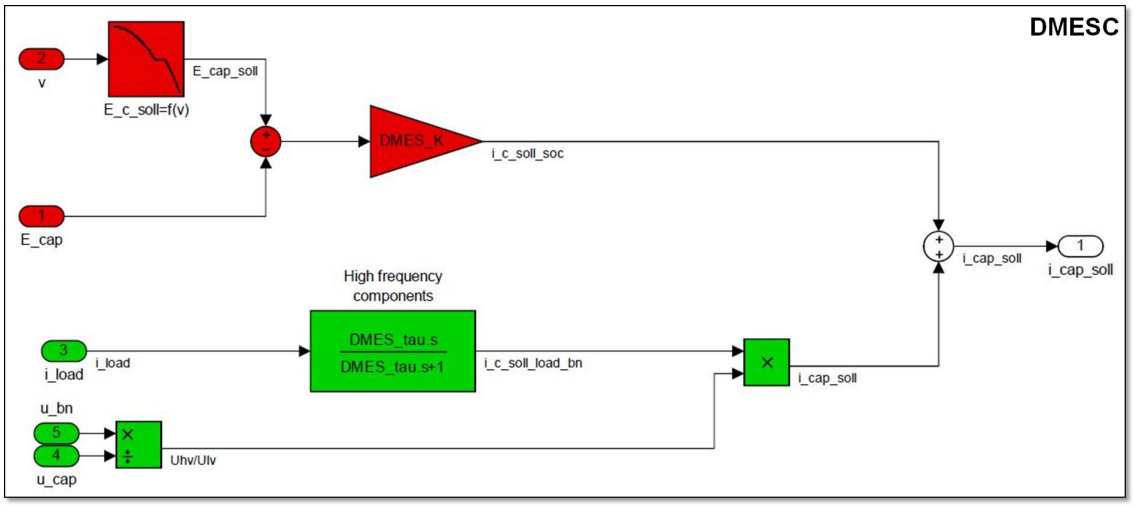

Figure 4.9: Structure of the DMESC.

For the following hardware integrations the controller part of the DMES model is used. This part, as already mentioned before, includes two simple PI controllers - see DMESC block in figure 4.9. This simulation model uses a Ordinary Differential Equations (ODE)3 (Bogacki-Shampine) solver with a step size of $0.01s$.

4.2.1 Single-core integration of DMESC

The first hardware integration is performed on a VIF CAN board as well as a TriBoard 1797, both described in section 2.1. On these two micro controllers the whole DMES controller logic implemented in the DMESC Simulink block will be integrated. Afterwards the achieved results of both boards will be compared and discussed. All necessary steps to realize this hardware integration have already been presented in section 3.3 and will be explained for this particular example in more details in the following sections. The belonging XML-file can be found in Appendix D.

From model to C-code

The first step towards integrating the DMESC on an ECU is code generation. Therefore, information regarding the required distribution are taken from the XML-file. This file specifies which model functions will afterwards run on hardware and which will not. For the first attempt C-code will be generated out of the whole DMESC block, including both PI controllers. To accomplish this, no changes in the model are required.
For the C-code generation MATLAB Embedded Coder, described in 2.2.1, is used. With this toolbox it is possible to choose a specific Simulink block and automatically generate a C-project out of it. The following section shows which extensions needs to be added to this generated code to enable CAN communication.

CAN communication

To enable communication via a CAN bus at first the required in- and output signals need to be determined. These signal information can again be taken from the XML-file in Appendix D. By means of the given range and precision appropriate values for the parameters *gain* and *offset* can be defined. The therefore used equation 3.1 can be found in chapter 3. Equation 4.1 depicts an example calculation of these two parameters for the signal *i_load*.

Parameter calculation for signal *i_ load*:

$$\begin{gathered} Range = [-1000, 200] \\ \Delta = max - min = 200 - (-1000) = 1200 \\ requiredprecision = 10^{-4} \\ gain_1 = \frac{\Delta}{2^{number\ of\ bits}} = \frac{1200}{2^{16}} = 0,0183 \\ gain_2 = \frac{\Delta}{2^{number\ of\ bits}} = \frac{1200}{2^{24}} = 0,000072 \\ offset = -1000 \end{gathered} \tag{4.1}$$

Table 4.1 shows the input and output signals of the DMES Controller and the therefore calculated parameters *offset* and *gain*, with respect to the desired range and precision, to create a correct value representation in a CAN message.

Signal	Range	Precision	Gain	Offset	Number of bits
in: E_cap	$[0, 1100000]$	10^{-1}	0.0656	0	24
in: v	$[0, 60]$	10^{-3}	0.000916	0	16
in: i_load	$[-1000, 200]$	10^{-4}	0.000072	-1000	24
in: u_cap	$[0, 300]$	10^{-2}	0.0046	0	16
in: u_bat	$[0, 400]$	10^{-2}	0.0061	0	16
out: i_cap_soll	$[-2000, 2500]$	10^{-3}	0.000268	-2000	24

Table 4.1: Details regarding the used in- and output signals.

All illustrated signal ranges are taken from the results of the model simulated in Simulink and are adjusted, to avoid possible overflow errors and added to the XML-file. The required precision for each signal needs to be achieved, by determining the number of used bits to represent a value. An example of such a calculation for the signal *i_ load* which should cover a range of $[-1000, 200]$ is shown in equation 4.1. The precision should be minimal four digits after the decimal point. The achieved precision is determined by the used scaling factor (*gain*). Equation 4.1 shows, that a representation with 16 bits ($gain_1$) would lead to a precision of only one digit after the decimal point, whereas by using 24 bits ($gain_2$) four digits precision can be achieved. The parameter *offset* defines the required value which is needed for the zero shift for the given range.

To send all required input and output signals of the DMESC through a CAN bus, 3 messages will be used in this case. Two messages including the five input signals, the third one including the output signal of the DMESC running on the hardware. The packing of the different input signals of the controller is represented in figure 4.10. A C-code snipped of this data conversion can be found in Appendix B.1.

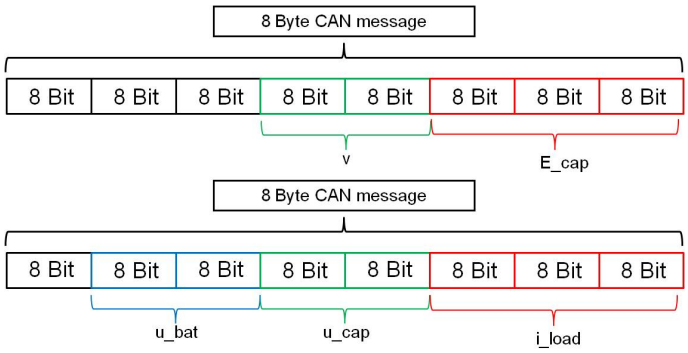

Figure 4.10: Overview of CAN messages packing for the DMESC input values.

In the end the previously generated code needs to be extended by a CAN communication interface which is able to pack and unpack messages by using the afore mentioned equations 3.2 and 3.3 and the appropriate parameters from table 4.1. The synchronization between the micro controller and the board is implemented as a if-statement in the code. The micro controller is only allowed to calculate and transmit an output value if all current input values are received. Thus, the controller needs to be able to calculate the new output value within the specified co-simulation step-size of the CAN module, otherwise some output value calculations will be missed. This code at the moment needs to be extended by hand and can afterwards be flashed to each board by using an appropriate compiler. For the VIF CAN Board AVR Studio and for the TriBoard the HighTec Toolchain is used.

Co-Simulation with ICOS

After the definition of the CAN interface and the therefore needed code adaptation an appropriate co-simulation model needs to be set up in ICOS to perform a HiL test. By performing such a test, the correctness and performance of the integrated DMES controller

function can be evaluated.

Figure 4.11 shows how this ICOS model looks like for the DMESC single-core integration. The *DMES_Environment* block includes the modified Simulink model with extracted DMESC block. Instead of this block, special ICOS input and output blocks have been added to the Simulink model. The modified model is shown in figure 4.12. The *DMES_Environment* provides the five input signals which need to be send over the CAN bus to the DMESC and further on receives the calculated output signal from the controller. These signals include data regarding battery and capacitor voltage, the capacitor energy, the loading current and the vehicles velocity as well as the desired capacitor current. These signals are coupled with the *CAN_Wrapper* block, which can also be seen in figure 4.11. This block is necessary for the CAN communication over the CAN bus between ICOS and the extracted DMESC running on the ECU. To generate such a communication block in ICOS a Real Time Wrapper is used. Within this wrapper the whole CAN communication needs to be defined within a specific *.ini-file*. This file includes information regarding the used CAN channel, the input and output signals together with their appropriate parameters, message identifiers and extrapolation mode and additionally, the desired baud rate. An example of such an *.ini-file* can be found in Appendix B.2.

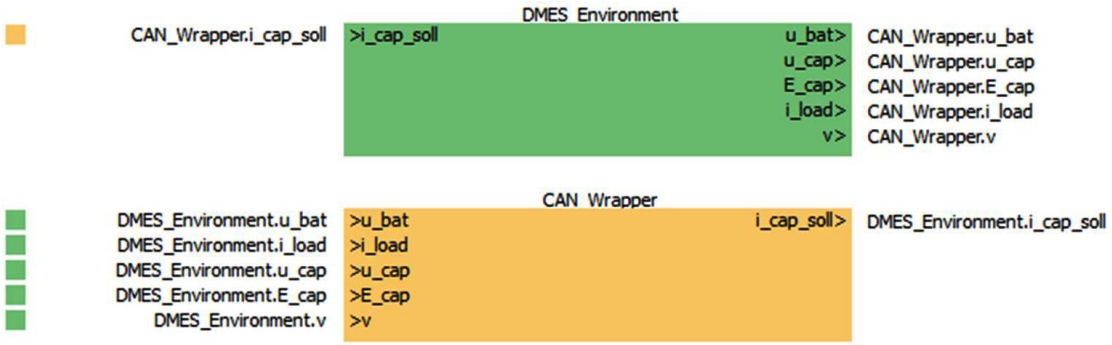

Figure 4.11: Overview of the created ICOS model.

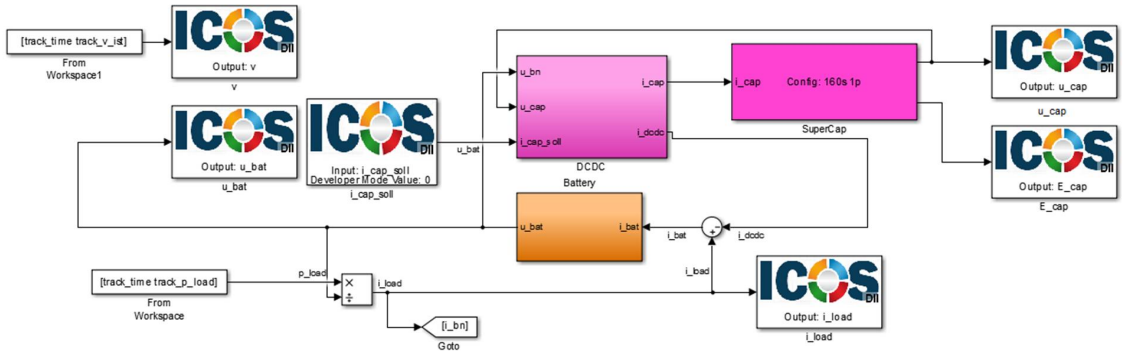

Figure 4.12: Overview of the Simulink model used for the co-simulation.

The physical simulation duration of this co-simulation model is set to $130s$, the coupling mechanism is sequential and the macro step size ΔT of both used model blocks is set to $0.01s$. This means that within the simulation duration every $0.01s$ new data values are exchanged between the *DMES_Environment* and the *CAN_Wrapper*.

Single-core results

To evaluate the HiL performance of the single-core integration of the DMESC, the achieved results are compared with the Simulink simulation results. Therefore an appropriate plot is generated, illustrating the behavior of the controller in both cases. The generated plot, depicted in figure 4.13, shows the amount of time in percent when specific battery currents occur, whereat high battery currents should occur as little as possible. In this plot, the bold grey line represents the achieved behavior without using the DMES-System, the black line shows the behavior if the DMES-System is used within a simulation model, the red line depicts the behavior of the HiL integration on a ViF CAN board, and the blue line shows the achieved results of the TriBoard 1797.

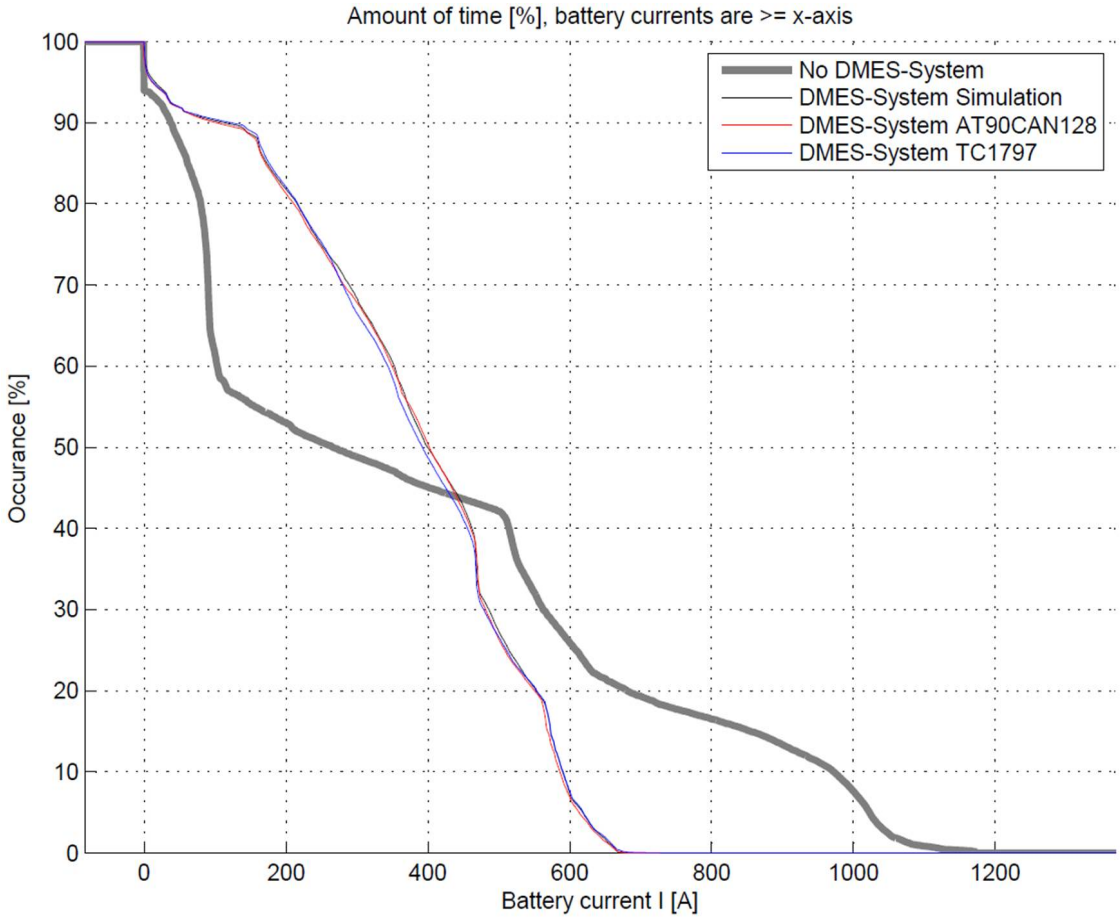

Figure 4.13: Comparison of DMESC simulation and single-core HiL results.

In this plot it can be seen that the behavior of all simulations including the DMESC is very similar but not identical. Furthermore, the plot illustrates that the AT90CAN128 controller, depicted in red, and the TC1797 controller, depicted in blue, do not achieve equal calculation results. This diverse behavior of the controllers can also be seen in figure 4.14. It represents a comparison of the DMESC output signal (*i_cap_soll*) plotted over the simulation time as well as the velocity profile of the chosen test track.

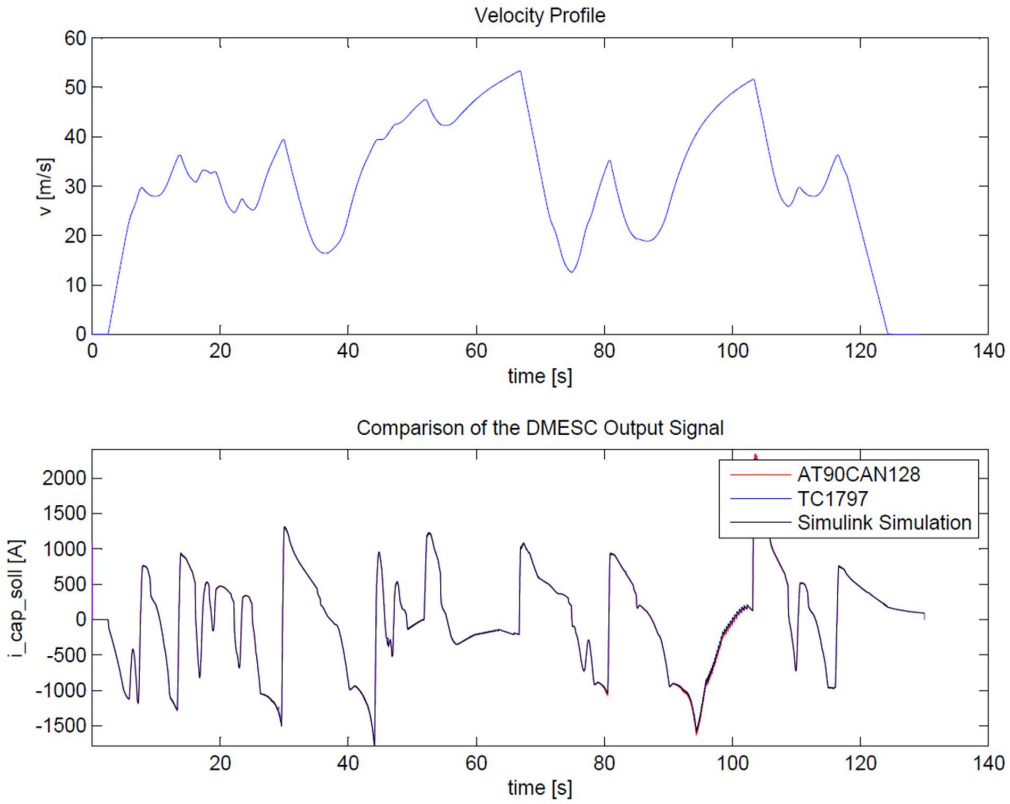

Figure 4.14: Comparison of the DMESC output signal.

This plot depicts that all three output signals are very similar throughout the whole simulation time, up to minimal deviations due to rounding and precision errors. A calculation of the percentage relative errors of the boards in comparison to the simulation results (equation 4.2) shows that the VIF CAN Board has a relative deviation of 1.8064%, the relative deviation of the TriBoard is a little bit lower with 1.0591%.

$$error_{abs} = \frac{\sum_{t=0}^{\#timesteps} |MiL(i\_cap\_soll(t)) - 0|}{\#timesteps} \hat{=} 100\%$$

$$error_{rel} = \frac{\sum_{t=0}^{\#timesteps} |MiL(i\_cap\_soll(t)) - HiL(i\_cap\_soll(t))|}{\#timesteps} \hat{=} (\frac{error_{abs}}{100} \cdot error_{rel})\% \tag{4.2}$$

$$errorAT90CAN128_{rel}[\%] = 1.8064\%$$
$$errorTC1797_{rel}[\%] = 1.0591\%$$

These different relative errors occur, because the two controllers have different restraints for doing arithmetic and logical operations. The AT90CAN128 is designed as a 8-bit micro controller without floating point unit. In contrast to that, the TC1797 is a 32-bit controller including a floating point unit. These properties affect the speed and sophistication of the controller instructions. Furthermore, these deviations are caused by the conversion of the transmitted output values into the CAN representation, thereby the achieved precision of the values deviates from the simulated one.

Another meaningful way, to evaluate the achieved quality of the hardware integration is the observation of the record of the CAN traffic. Therefore the *PCANView* tool is used. The record of the CAN traffic between the TriBoard and ICOS is shown in figure 4.15.

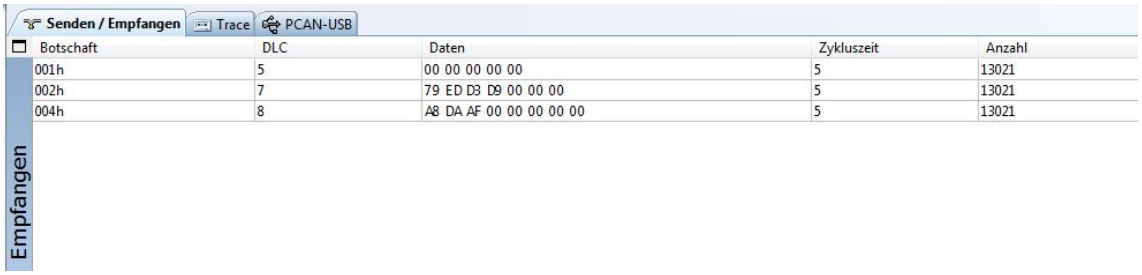

Figure 4.15: A record of the CAN traffic when performing the HiL testruns.

The CAN messages with the identifiers $0x001$ and $0x002$ are received by the micro controller, message $0x004$ is the controllers transmitted output value for each time-step. It can be seen, that 13021 input values are sent to the micro controller and 13021 are sent back. The chosen simulation time in this example is $130s$, thus with a time-step of $0.01s$, 1300 messages are exchanged through the co-simulation. The additional messages are dummy-messages sent by ICOS to establish and close the connection. This dummy-message transmission could in some cases influence the initial behavior of a control function, because it could lead to wrong initial values. All in all this CAN traffic record shows, that the used synchronization method works and no messages are lost due to timing restrictions.

After these observations it is obvious that the DMESC integration on both boards lead to a correct HiL result. The minimal differences that can be seen in both plots are reducible to the chosen precision which effects the scaling factors used for the CAN transmission and receiving. A fundamental finding of this first single-core proof of concept is, that the occurring time delays due to the used CAN communication do not have an influence on the achieved HiL results in this example, since the used DMESC does not include high dynamic parts. Nevertheless, the achieved results depend on hardware properties like the implemented register size.

4.2.2 Multi-core integration of DMESC

The second hardware integration is performed on two TriBoards, which are coupled via a CAN bus and therefore represents a multi-core integration. On these micro controllers, the a distributed form of the logic implemented in the DMESC Simulink block will be integrated. On each board one of the two PI controllers will be flashed. The general steps to realize this hardware integration have already been described in the previous sections. Hence, only a short description of the necessary steps for this hardware integration are given in the following sections. All necessary information regarding the desired distribution are again taken from a XML-file belonging to this hardware integration. This XML-file can be found in Appendix D.

From model to C-code

For this approach some little modifications need to be done in the existing Simulink model. The two PI controllers in the DMESC block need to be split up into two independent blocks, to easily generate the necessary code for each hardware devices. Thus, the blocks *PI_Controller1* and *PI_Controller2* are created and the particular controller logic is added. For the C-code generation MATLAB Embedded Coder is used again. This time, two C-projects are generated which need to be modified subsequently. Figures 4.16, 4.17 and 4.18 show, how the Simulink model looks after the modification.

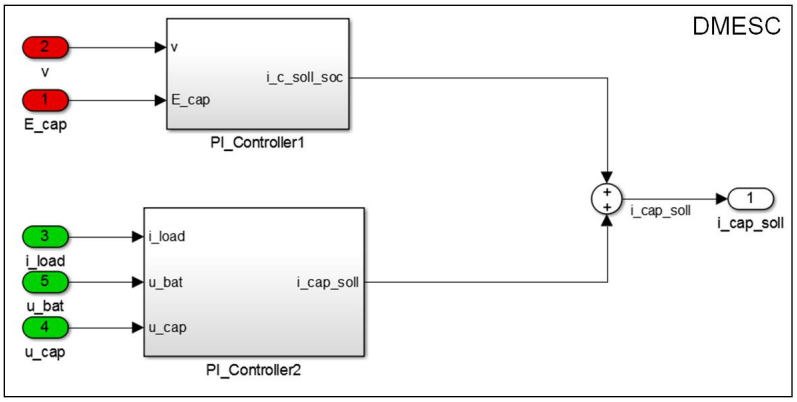

Figure 4.16: Modified DMESC block.

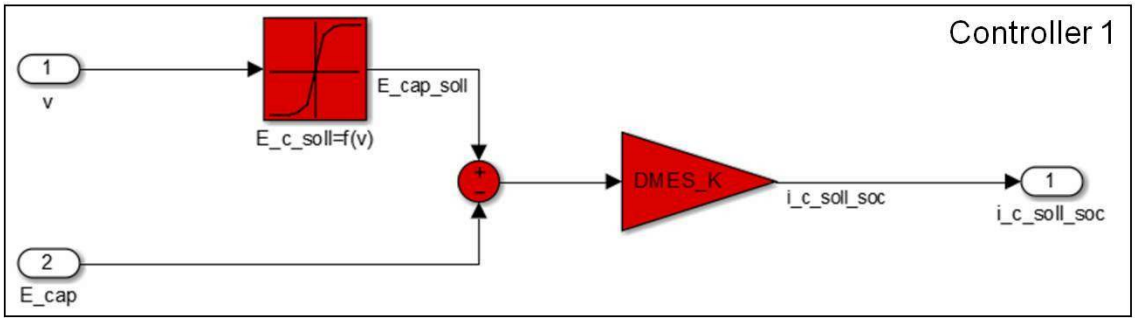

Figure 4.17: Implementation of the first PI controller block.

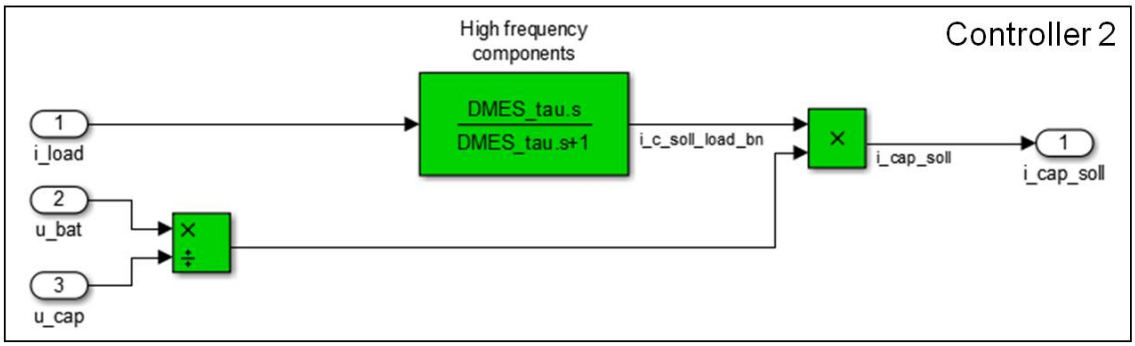

Figure 4.18: Implementation of the second PI controller block.

CAN communication

Due to the fact, that the original DMES controller block needs to be modified, the in- and output signals for both micro controllers changed as well. Tables 4.2 and 4.3 give an overview of the necessary in- and output signals for each micro controller. Further, the determined *offset* and *scaling factors* as well as the necessary signal range and number of used bits are listed.

Signal	**Range**	**Factor**	**Offset**	**Number of bits**
in: E_cap	$[0, 1100000]$	0.0656	0	24
in: v	$[0, 60]$	0.000916	0	16
out: i_cap_soll_soc	$[-1000, 1000]$	0.0001192	-1000	24

Table 4.2: Overview of the in- and output signals of the first PI controller.

Signal	Range	Factor	Offset	Number of bits
in: i_load	$[-1000, 200]$	0.000072	-1000	24
in: u_cap	$[0, 300]$	0.0046	0	16
in: u_bat	$[0, 400]$	0.0061	0	16
out: i_cap_soll	$[-5000, 5000]$	0.000596	-5000	24

Table 4.3: Overview of the in- and output signals of the second PI controller.

On the basis of these defined signal representation properties, the two C-projects need to be extended by an appropriate CAN interface as well as a synchronization method. For the synchronization again an if-Statement is used, to ensure that the controllers only calculate output values when new input values are received. For the first PI controller one CAN message needs to be received and unpacked, including the values of the two input signals *E_cap* and *v*. Further, this controller needs to transmit the calculated value of signal *i_cap_soll_soc*. The second PI controller also needs to be able to receive and unpack one CAN message including the input values for the signals *i_load*, *u_cap* and *u_bat*. Moreover, the transmission of the calculated value of the signal *i_cap_soll* needs to be added to the code by using one further CAN message. After this code modifications the controller functions can be flashed on the VIF CAN boards with AVR Studio.

Co-Simulation with ICOS

Although this multi-core approach includes two micro controller boards the ICOS co-simulation model almost looks like the one for the single-core approach. The only difference is the number of output signals of the Real Time Wrapper and the deployment of the input signals which is defined in the *.ini-file*. The CAN block input signals *E_cap* and *v* are send to the VIF CAN board that runs the C-code of *PI_Controller1*, *i_load*, *u_cap* and *u_bat* are sent to the one that runs the C-code of *PI_Controller2*. An overview of the coupling is shown in figure 4.19.

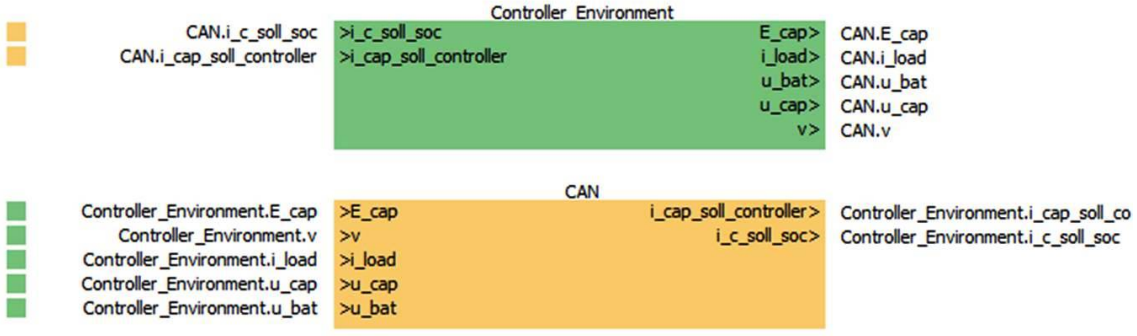

Figure 4.19: Overview of the ICOS coupling.

To enable the coupling with the DMES Simulink model again some modifications of the model are necessary. The before created new blocks *PI_Controller1* and *PI_Controller2* are replaced by two ICOS input blocks, the remaining DMES model looks the same as in the single-core approach. The modified DMESC block is shown in figure 4.20.

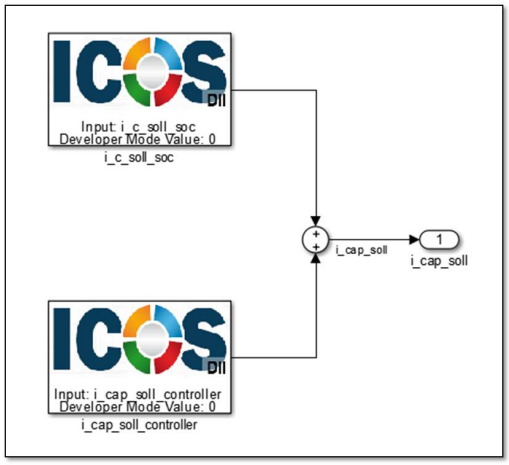

Figure 4.20: Modified DMESC block.

The physical simulation duration of this co-simulation model is set to $130s$, the coupling mechanism is sequential and the macro step size ΔT of both used model blocks is set to $0.01s$. This means that within the simulation duration every $0.01s$ new data values are exchanged between the *DMES_Environment* and the *CAN_Wrapper*.

Multi-core results

To evaluate the HiL performance of the distributed DMESC, the achieved results are again compared with the Simulink simulation results. Therefore, the same plots as in the single-core approach are generated.

Figure 4.21 illustrates the different percentage amounts of time, specific battery currents occur with and without the DMES system, when distributing the DMES controller over two TriBoards and simulating it in Simulink. In this plot, the bold grey line again represents the achieved behavior without using the DMES-System, the blue line shows the behavior if the DMES-System is used within a simulation model and the red line depicts the behavior of the HiL distributed over two TriBoards. Even in this multi-core scenario the achieved behavior complies to the simulated one. Figure 4.22 depicts a comparison of the output signals of both controllers and the calculated output signals of the DMESC block simulated in Simulink as well as the belonging vehicle velocity profile.

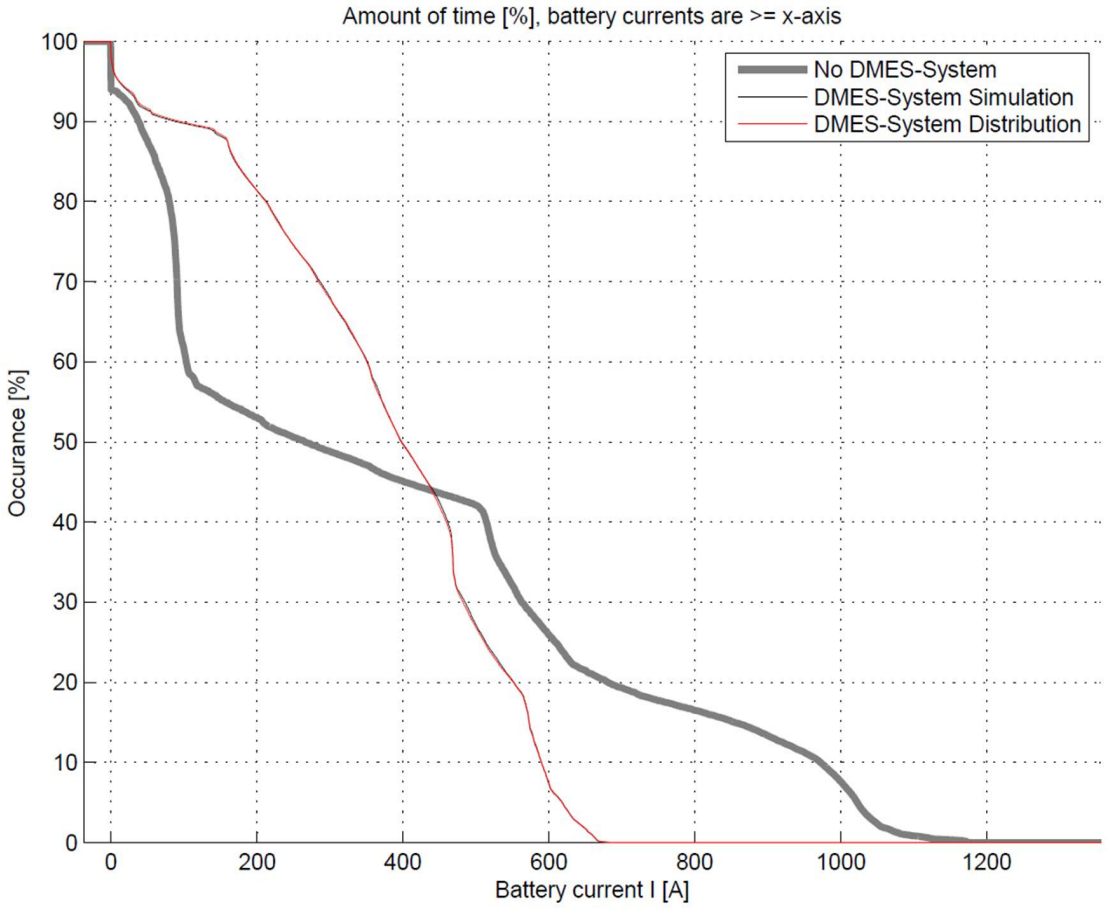

Figure 4.21: Comparison of DMESC simulation and multi-core HiL result.

In this case, the resulting percentage relative error of the output signal *i_cap_soll* computed by the simulation and the multi-core integration, calculated according to equation 4.2, lies at 1.0604%. Thus, the multi-core integration almost achieves the same result as the single-core integration, which achieved a relative percentage error of 1.0591%. The minimal deviation of this relative errors is caused by the used scaling factors for the CAN representation of the values. Another reason for these deviations is the increasing communication effort when performing a multi-core integration. Nevertheless, this result shows that the timing delay caused by this CAN communication has only a small effect on the HiL result, because the distributed model parts do not include high dynamic elements. In case of too high delay times in this scenario, the resulting DMESC output signal would show much higher deviations from the Simulink simulation, because it can not be guaranteed that the adder, included in the DMESC block, adds up values belonging to the same time-steps.

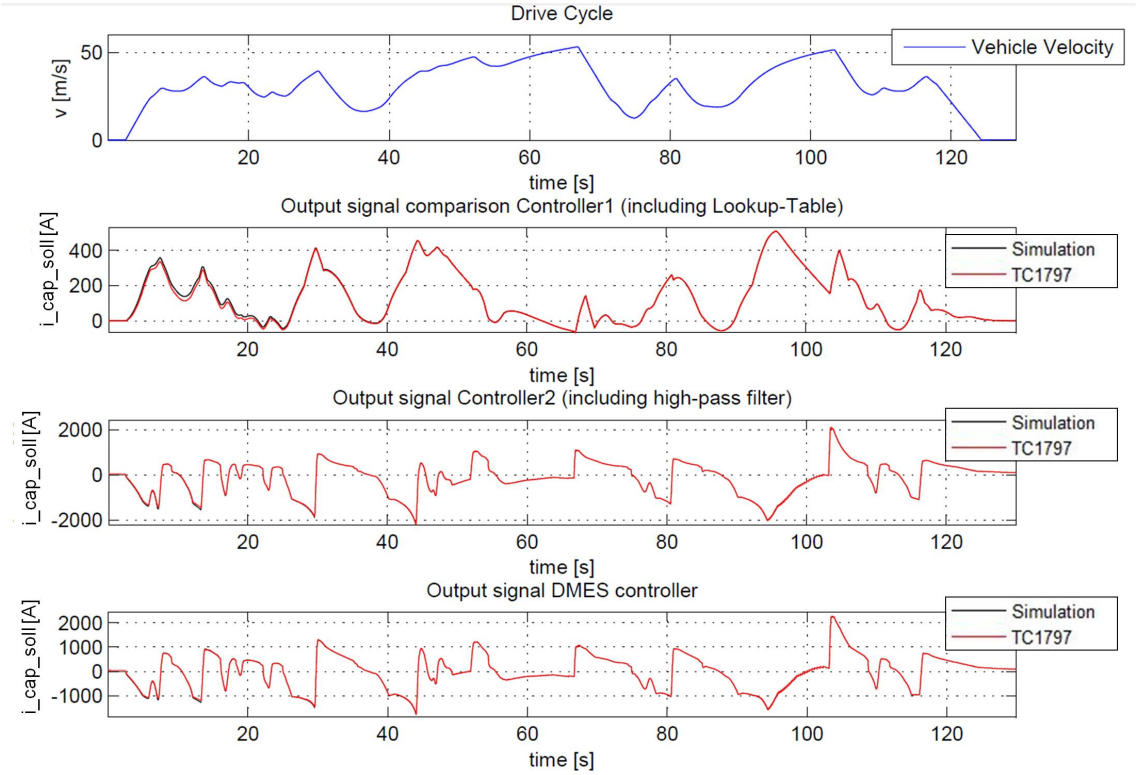

Figure 4.22: Illustration of the different output signals of both micro controllers and the output signal of the DMES system.

A further comparison of the different output signals, presented in figure 4.22 shows that the HiL simulation achieves very similar results as the simulation in Simulink. The comparison of the output signals of the micro controller, including the lookup table, depicted in the second sub-plot, shows a large deviation in its initial phase. This behavior can also be seen in the other two sub-plots when zooming on the y-axis. It looks as if the micro controllers need about 20 seconds until they reach a steady-state, from that moment both depicted signals overlap.

Figure 4.23 shows the record of the CAN communication. As in the single-core example, it can be seen that no messages are lost throughout the communication. Thus, all occurring deviations after the controllers reach a steady-state can be traced back to inaccuracies caused by the conversion of the values to the CAN representation.

Senden / Empfangen | Trace | PCAN-USB

Botschaft	DLC	Daten	Zykluszeit	Anzahl
001h	5	00 00 00 00 00	5	13003
002h	7	79 ED D3 00 00 00 00	5	13003
003h	8	E6 5D 0B 00 00 00 00 00	5	13003
004h	8	00 00 00 00 00 00 00 00	5	13003

Figure 4.23: Record of the CAN bus traffic of the DMESC multi-core approach.

4.2.3 Influence of used solver-type and step-size on HiL results

To get an impression of how the selected fixed-step solver and step-size used in a control model influences the behavior of embedded functions, further simulations using the TriBoard1797 are performed. Therefore, a short introduction to the used fixed-step solver, taken from the *MathWorks documentation*[2], is given:

- ODE1 - *Euler's Method*
  This solver uses the Euler method for its approximations. It is the least complex fixed-step solver used by MATLAB and achieves first order of accuracy.
- ODE2 - *Heun's Method*
  This solver uses the Heun method for its approximations. In contrast to ODE1 this solver achieves second order of accuracy.
- ODE3 - *Bogacki-Shampine Formula*
  This solver uses the Bogacki-Shampine formula for its approximations and achieves third order of accuracy.

All these solvers use different orders for there calculations and therefore achieve different approximation results. Moreover, non of the described fixed-step solvers has an error control mechanism and therefore, the accuracy and the duration of a simulation directly depends on the selected step-size, pre-defined by the user. *Mathworks* provides fixed-step solver up to 8^{th} order of accuracy, nevertheless the three presented solvers are sufficient for the aims of this comparison within this Master's thesis.

Solver comparison results

Figure 4.24 depicts the achieved behavior of the DMES system when using a step-size of $0.1s$ and the above described solvers. As reference signals in this case, the result of the single-core integration on the TriBoard using ODE3 and a step-size of $0.01s$, represented as black-dashed line, as well as the result of the simulation represented as black line are used.

[2] www.mathworks.com/help/simulink/ug/choosing-a-solver.html

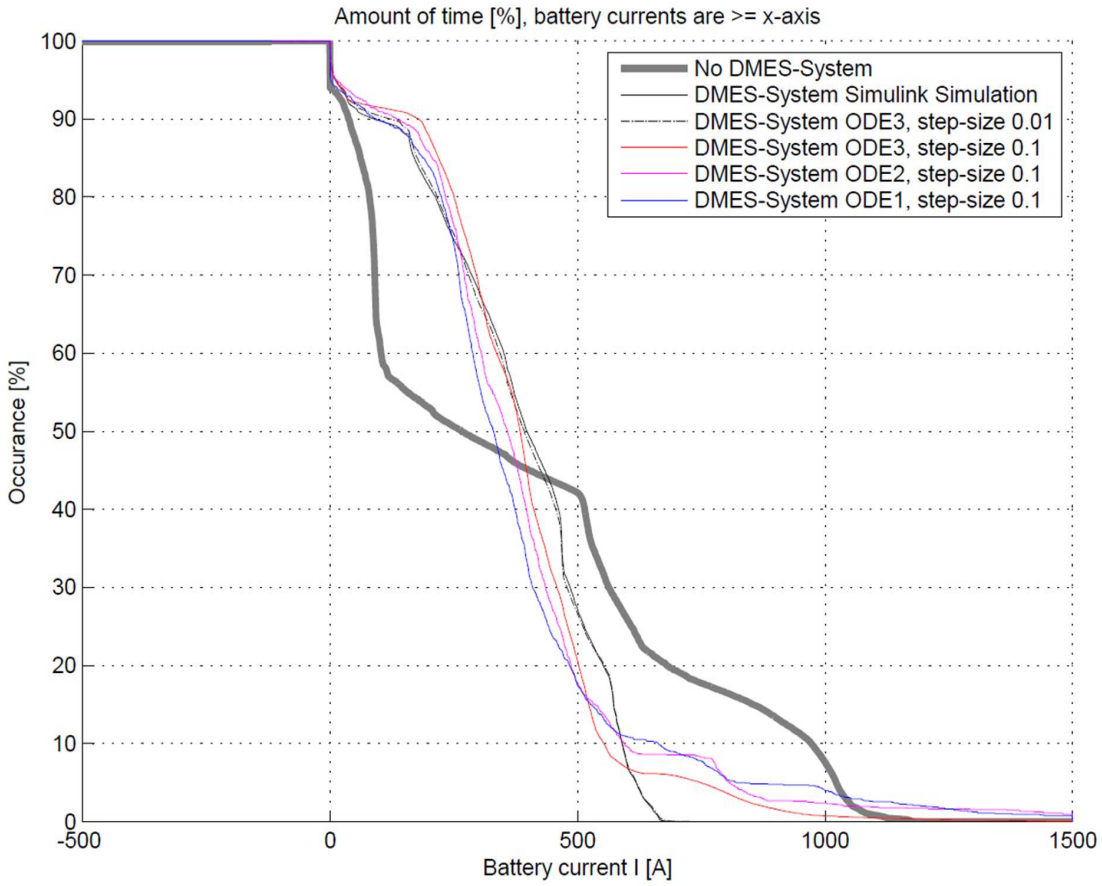

Figure 4.24: Comparison of different solver types using a step-size of 0.1s and the result of ODE3 with a step-size of 0.01s, represented as dashed-line, as well as the MiL result represented in black.

To perform a meaningful comparison between the different solver types, the DMESC output signal *i_cap_soll*, computed by the different solvers, are plotted in the second sub-plot in figure 4.25. Furthermore, the resulting error compared to the HiL simulation using ODE3 and a step-size of 0.01s, depicted in black, is plotted in the third sub-plot in figure 4.25. An overview of the relative errors in percent, calculated according to equation 4.2, when comparing the reference signal and the signal computed with different solver types, is shown in table 4.4.

Solver Type	Relative *i_cap_soll* Error [%]
ODE1	86.6279%
ODE2	84.0781%
ODE3	83.4410%

Table 4.4: Relative percentage deviation of the different solvers from the result using ODE3 and a step-size of 0.01s.

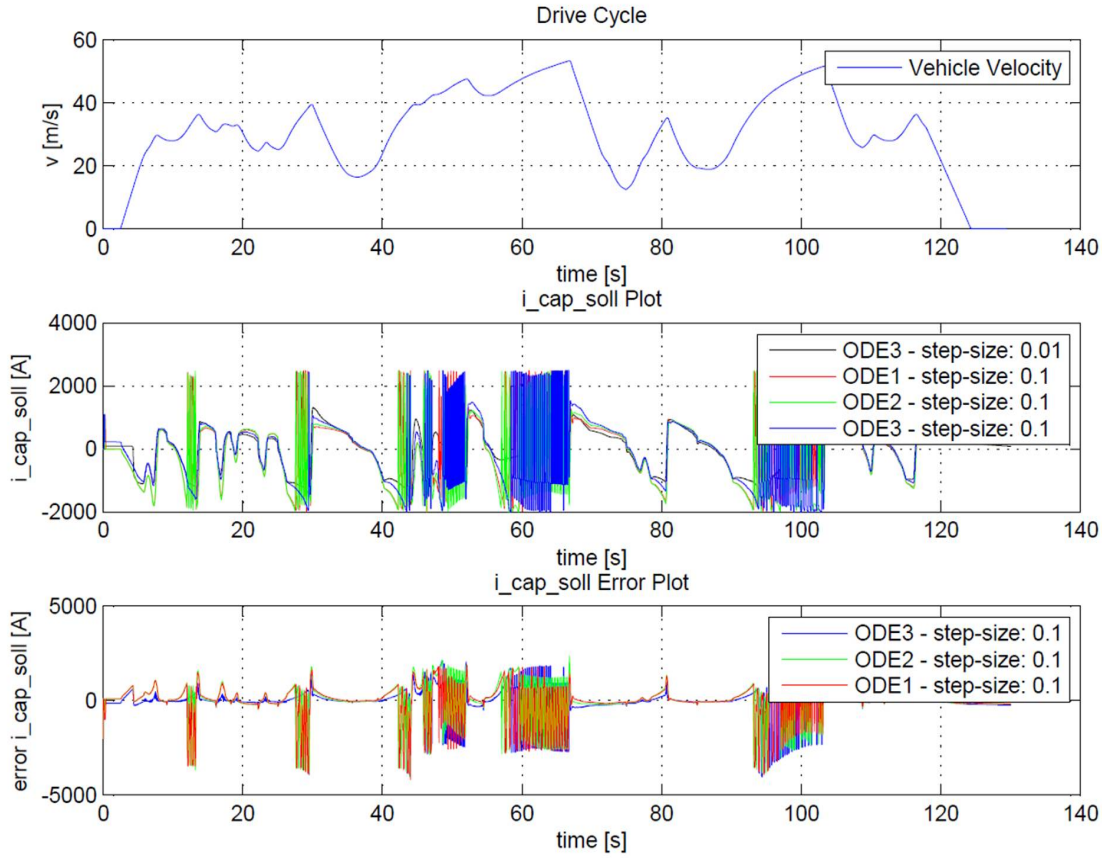

Figure 4.25: Resulting error of *i_cap_soll* when using different solver types.

The comparison of the different resulting super capacitor currents and the calculated relative errors shows that all computed signal overshoot very much and therefore have much higher relative errors than the integration using ODE3 with an appropriate step-size of $0.01s$. Moreover, it can be concluded that the most complex solver (ODE3) achieves the smallest, but even a too high error. Thus, this solver, using the Bogacki-Shampine formula, is able to compensate the chosen step-size better than the two less complex solvers. Nevertheless, even this result is not acceptable for a real application. It further can be seen, that the second complex solver (ODE2) achieve similar relative errors as ODE3 and the least complex solver (ODE1) achieves by far the worst result. All these simulations show that the used solver type as well as the related step-size has a big influence on the performance of embedded controllers and therefore needs to be considered when designing a model.

4.3 Hybrid Energy Management System model

To proof the developed toolchain a second time, another model integrating a energy management system of a hybrid vehicle is used. This model was originally designed by *Christian Paar* as part of his Bachelor thesis with the title *'Energy Management in Hybrid Electric Vehicles using Co-Simulation'* [40]. Since this Bachelor thesis has been conducted in 2009, the version used within this Master's thesis has already been modified by the *Virtual Vehicle Research Center*.

This model simulates a mild hybrid Sport Utility Vehicle (SUV) with an engine power of $150kW$. Furthermore, the vehicle is designed as series hybrid, the scheme of such a hybrid is depicted in figure 4.26. Loosely speaking, a series hybrid vehicle consists of an Internal Combustion Engine (ICE) which is connected to an electrical generator. This generator produces electric energy for the recharging of the energy storage. The Electrical Machine (EM) is powered by this energy storage and propels the drive train.

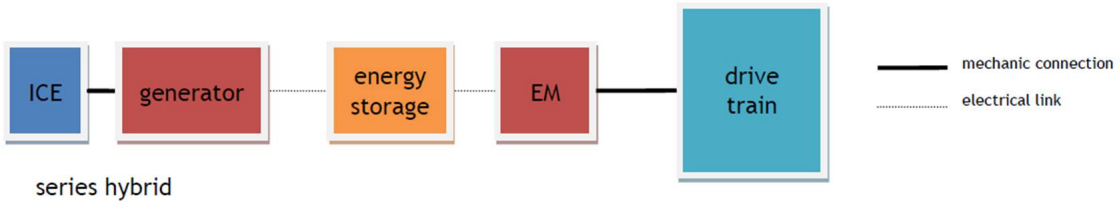

Figure 4.26: Scheme of a series hybrid electrical vehicle [40].

The used driving cycle for this simulation is the New European Driving Cycle (NEDC), illustrated in figure 4.27. This cycle is composed of two parts, an urban driving cycle which is repeated 4 times and an extra-urban driving cycle which is used once. Within the simulation the first $190s$ of this cycle are used.

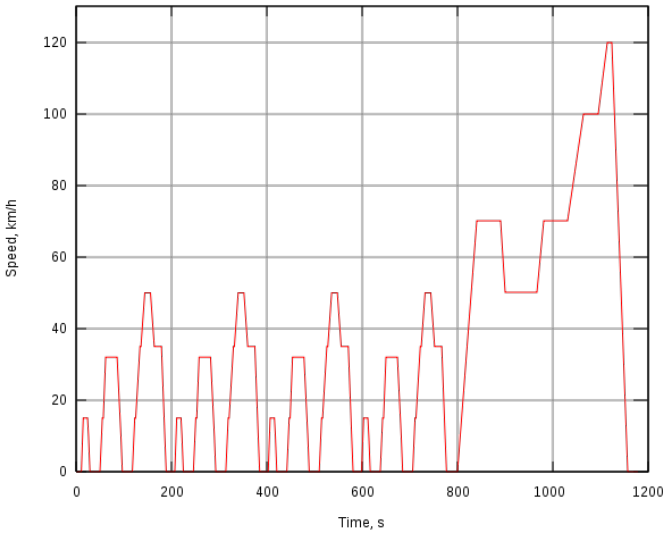

Figure 4.27: Velocity profile of the NEDC [41].

The complete vehicle simulation model is shown in figure 4.28 and consisting of three subsystems: a vehicle model designed in *AVL Cruise*[3], a cockpit modelled in Simulink, and a management system modelled in Simulink as well. Thus the management system model depicted in figure 4.29 is only one part of this simulation.

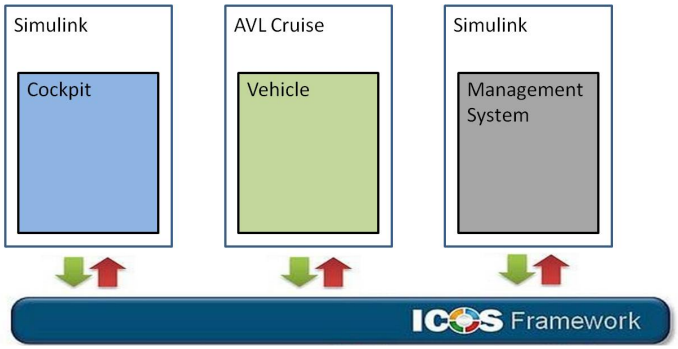

Figure 4.28: Subsystems of the whole hybrid vehicle simulation.

The management system model, illustrated in figure 4.29, includes the energy management of a hybrid vehicle. Herein especially the energy exchange between the super capacitor, the electric motor and the lithium-ion battery is represented. This system consists of a hybrid module including the parallel hybrid structure of the vehicle. Thus, the recuperation of brake energy and the partition of the desired driving torque to the combustion and electric engine are the main tasks of this module. Further, the management system consists of a lithium-ion battery and a super capacitor block, which both model the behavior of this specific component. Another module of the management system is the energy buffer which handles the used currents for the super capacitor and the battery system. All this information is taken from a paper published by *Stettinger et al.* [42] in 2013.
Characteristics of the lithium-ion battery and the super capacitor used for this simulation are taken from the Simulink model and look as follows:

Lithium-Ion Battery

- Number of cells: 15
- Initial SoC: 80%
- Initial temperature: $20°C$
- Energy capacity per cell: $16Ah$

Super capacitor

- Number of parallel capacitors: 1
- Initial SoC: 80%
- Capacity: $200F$
- Max. Voltage: $81V$

[3] www.avl.com/web/ast/cruise

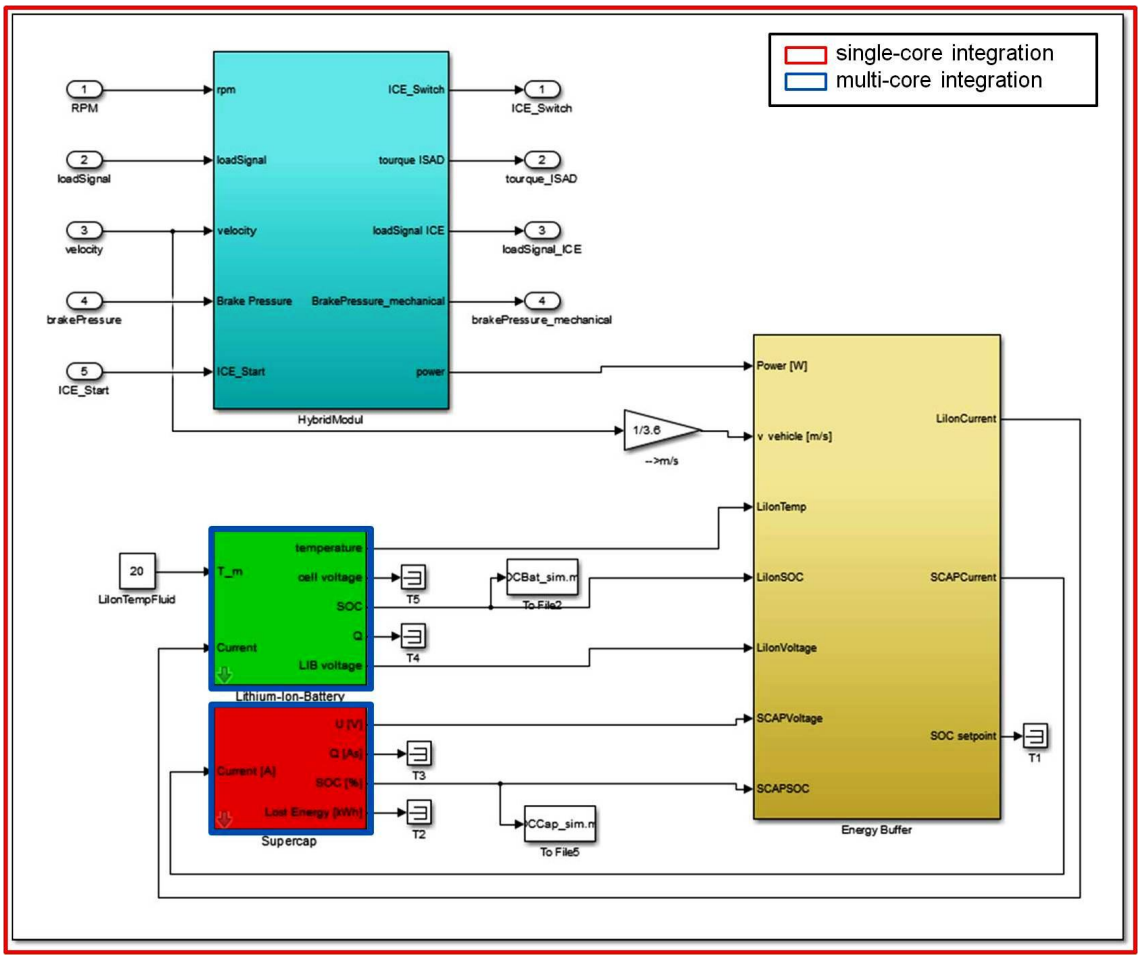

Figure 4.29: Overview of the used management system model.

As a further proof-of-concept of the developed toolchain, the whole management system model, red framed in figure 4.29, is integrated on the Infineon TriBoard1797. Afterwards, the battery as well as the super capacitor block are integrated on two TriBoards. The used blocks for this approach are framed in blue in figure 4.29. Therefore, the same steps as described in section 4.2.1 and 4.2.2 are done.

First, code is generated out of the necessary model block(s). For the single-core approach the whole management system block, including all subsystems illustrated in figure 4.29, is used. For the multi-core approach separate code is generated out of the battery and the super capacitor block. These subsystems implement different controlling strategies to simulate the behavior of these components. Figures 4.30 and 4.31 show the simulation model of the particular component, including several dynamic parts like transfer functions and integrators.

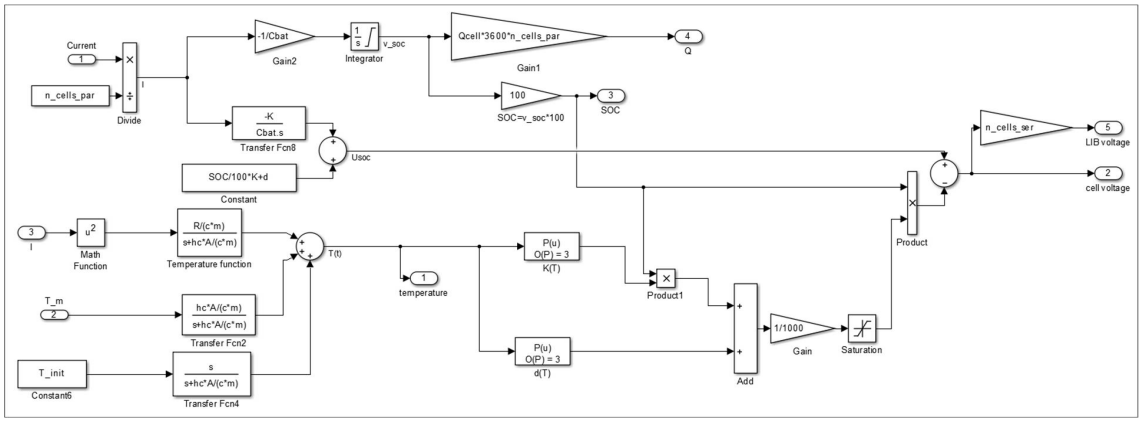

Figure 4.30: Simulink model of the battery.

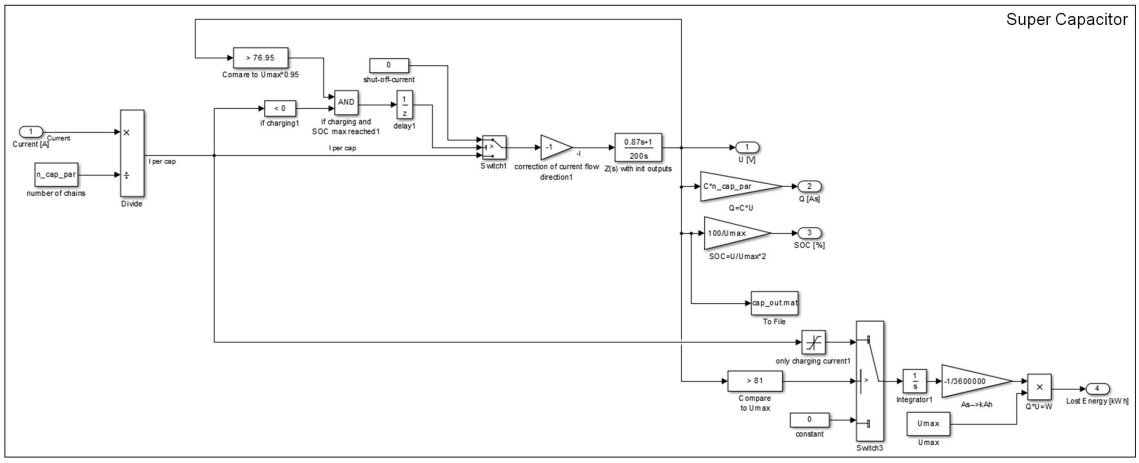

Figure 4.31: Simulink model of the super capacitor.

As a next step, the generated code is extended by a CAN interface using appropriate scaling factors and offsets. Afterwards the extended code is flashed on the hardware. As a last step, an appropriate co-simulation model needs to be created. In both case, the single-core and the multi-core one, this co-simulation model includes more parts than in the DMES examples. Figure 4.32 depicts the adapted Simulink model of the management system in the single-core case.

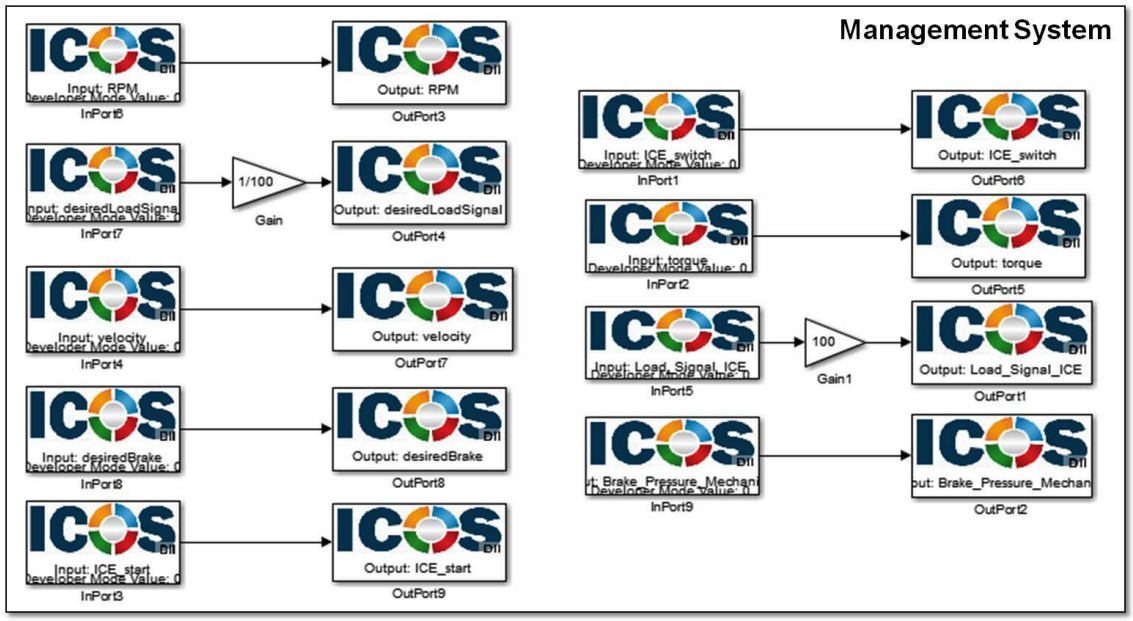

Figure 4.32: Modified Simulink model used for the co-simulation of the single-core approach.

In this case, the input signals are generated in the vehicle and the cockpit model which are connected to the management system via ICOS input blocks. These input blocks are directly connected to the Real-Time Wrapper, responsible for the CAN communication, via ICOS output blocks.
Figure 4.33 shows all needed co-simulation subsystems for the single- as well as the multi-core approach. The only difference between these two approaches is the number of input values of the CAN block.

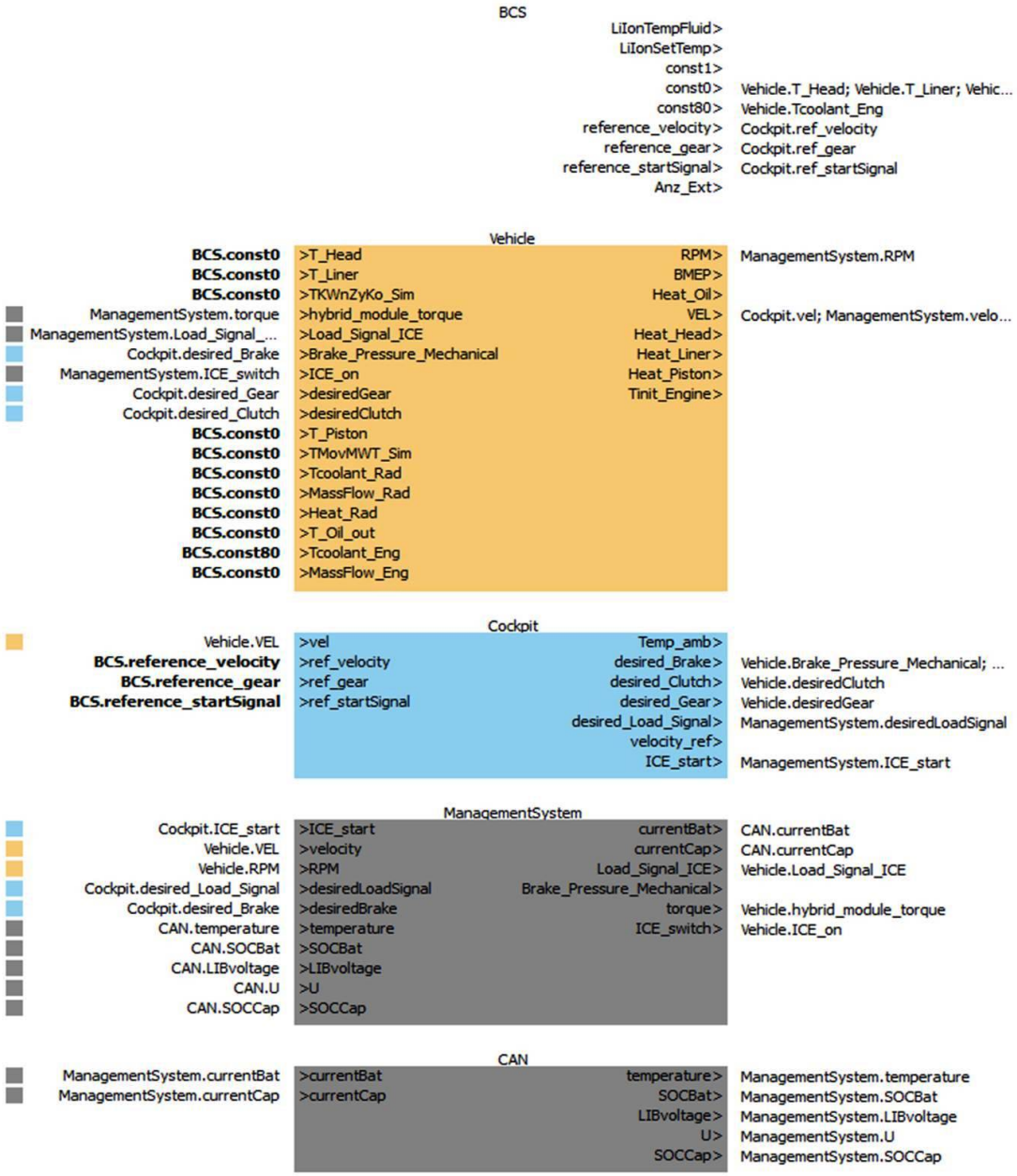

Figure 4.33: Overview of the model used for the data exchange between two TriBoards and ICOS.

4.3.1 Single-core results

To evaluate the performance of the single-core HiL simulation performed on the TriBoard1797, the resulting torque signal of the Simulink simulation is compared with the one computed by the TC1797 controller. Further on, the achieved battery and super capacitor SoCs are compared to better evaluate the performance of the dynamic parts of this model. In all these cases a step-size of $0.01s$ is used, the selected solver is ODE3 (Bogacki-Shampine). The comparison of the achieved torque results and the belonging vehicle profile are shown in figure 4.34.

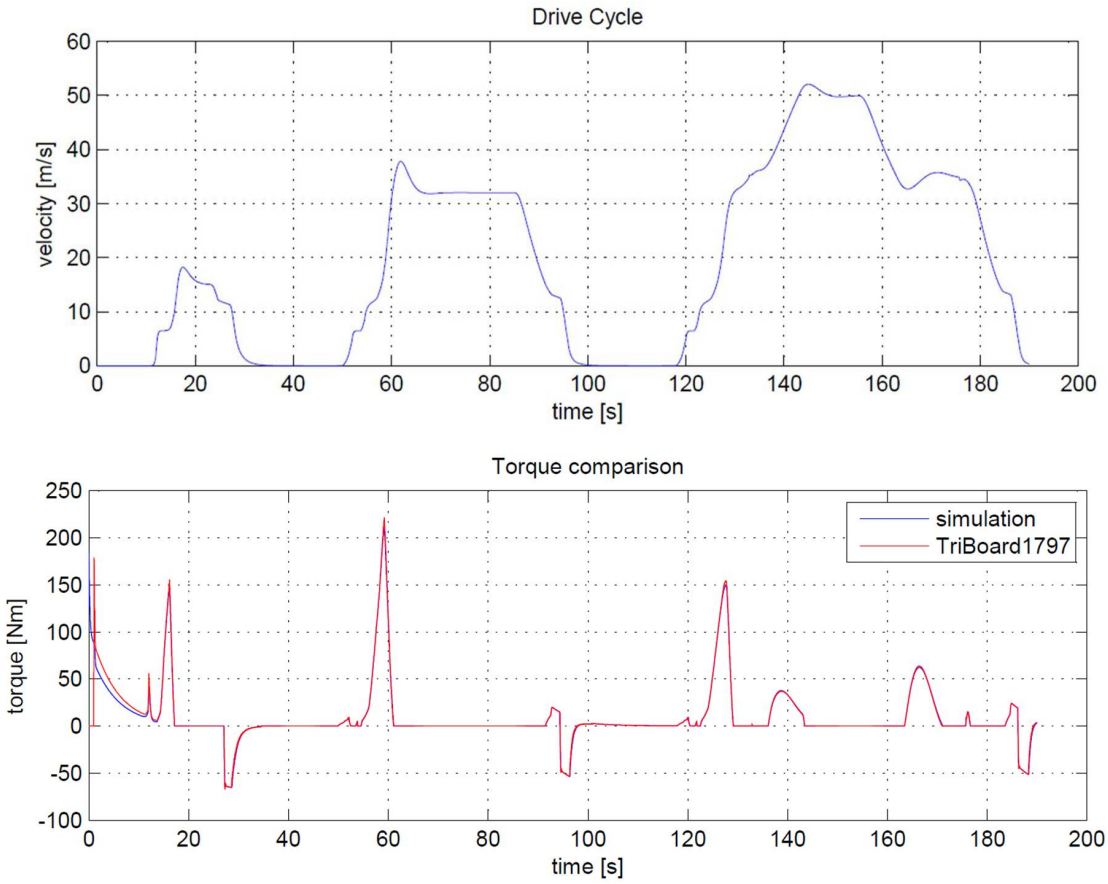

Figure 4.34: Comparison of the torque signal computed via Simulink and the TriBoard1797.

This comparison shows that the TriBoard receives a wrong initial value and therefore needs about $10s$ to compensate this error. Afterwards, the computed values match with the simulated ones. As already mentioned before, this wrong initial value is caused by the transmitted dummy messages from ICOS. This comparison further shows that the used accuracy for the transformation of controller in- and output values to a CAN representation does not lead to deviations in this example.

Figure 4.35 depicts the resulting SoC of the battery and the super capacitor in comparison to the MiL results as well as the velocity profile of the chosen test track. It is shown, that the HiL solution has as similar behavior as the MiL solution. A comparison of the achieved average SoC errors of the single-core and the multi-core integration follows in section 4.3.2.

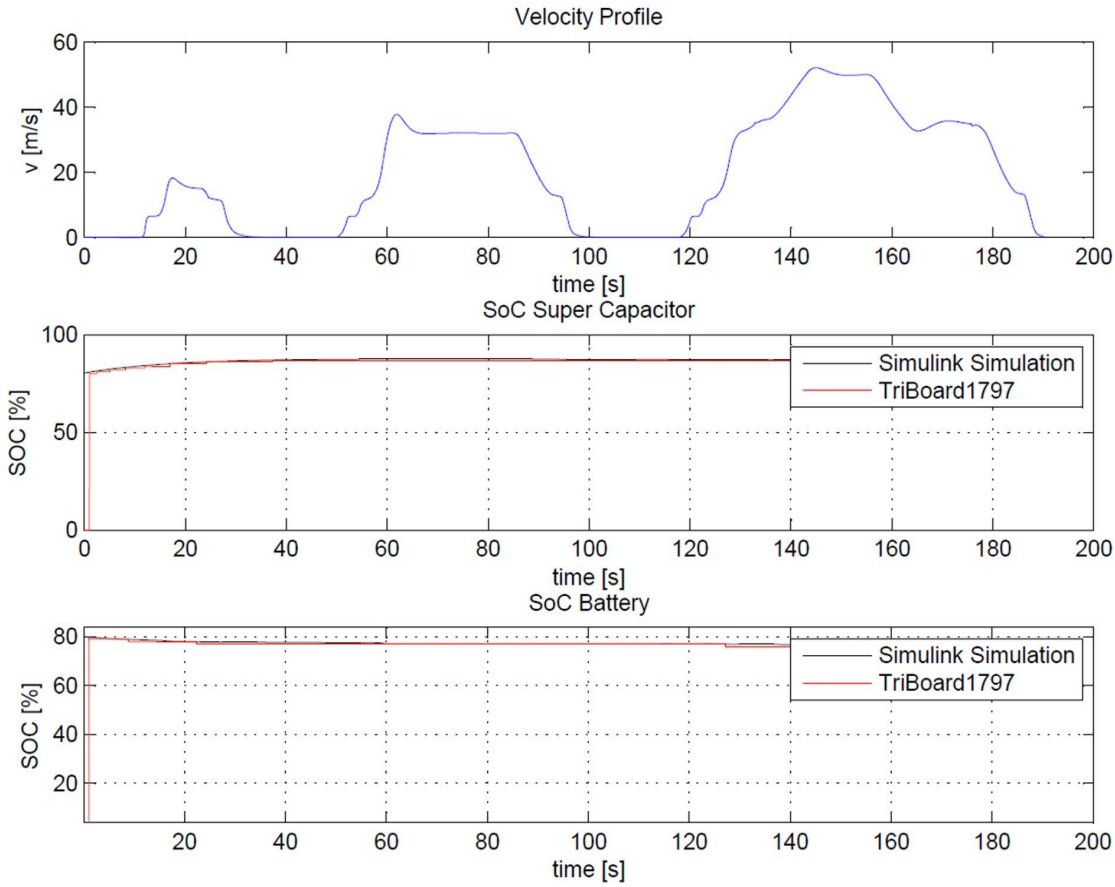

Figure 4.35: Comparison of the resulting MiL and HiL SoCs.

4.3.2 Multi-core results

To evaluate the result of the multi-core integration, first the achieved torque signals, illustrated in figure 4.36, are compared. It is shown that this time the two signals are identical throughout the whole simulation. This result shows that the distributed parts of the model do not affect the remaining Simulink simulation in any way.

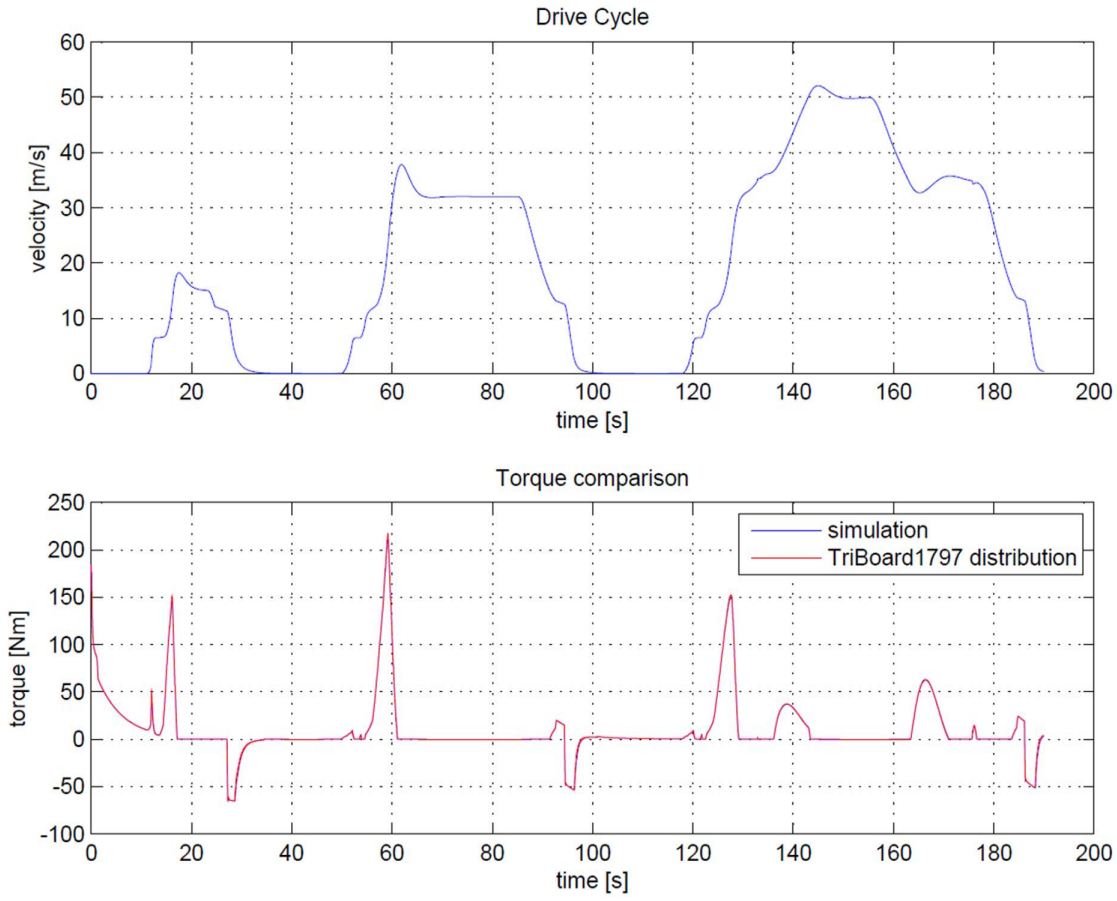

Figure 4.36: Comparison of the torque signal computed with Simulink and two TriBoard1797.

To compare the resulting average errors of the SoC achieved by the single- and multi-core integration, computed according to equation 4.2, the reference SoC simulated in Simulink, as well as the ones computed by the two TriBoards are compared. These results and the belonging velocity profile are depicted in figure 4.37. The reason why the two SoC signals are used for this comparison is that the *battery* and *super capacitor* blocks of the model include dynamic parts. Therefore, these parts are more error-prone as the block, which computes the torque. It is shown that, in comparison to the single-core result depicted in figure 4.35, the multi-core integration achieves a worse result. Table 4.5 presents the achieved relative errors of the battery and super capacitor SoCs for the single- and the multi-core approach in comparison to the Simulink simulation result.

SoC	Relative Error
single-core super capacitor	0.4902%
single-core battery	0.4500%
multi-core super capacitor	6.6627%
multi-core battery	1.8693%

Table 4.5: Relative percentage deviation of the SoCs from the Simulink simulation result in the single-core and multi-core case.

It can be seen that in both approaches the battery SoC has lower deviations than the super capacitor SoC. The reason for this is, that these two parts include different control logics and therefore different dynamics influence the behavior of the controllers. Further, it is clearly shown that the distribution of the two control functions achieve a average error of far more than one percent. This worse behavior of the distributed solution occurs because the *battery* and the *super capacitor* blocks, illustrated in figure 4.29 are part of a feedback loop. Due to the fact that the input values of these two blocks are calculated by the *energy buffer* in Simulink, the transmission of these values leads to delay times which have a big influence on the behavior of the integrated controller dynamics. Moreover, this behavior is caused by the used extrapolation (zero-order-hold) of the input values, which is done by co-simulation tool.

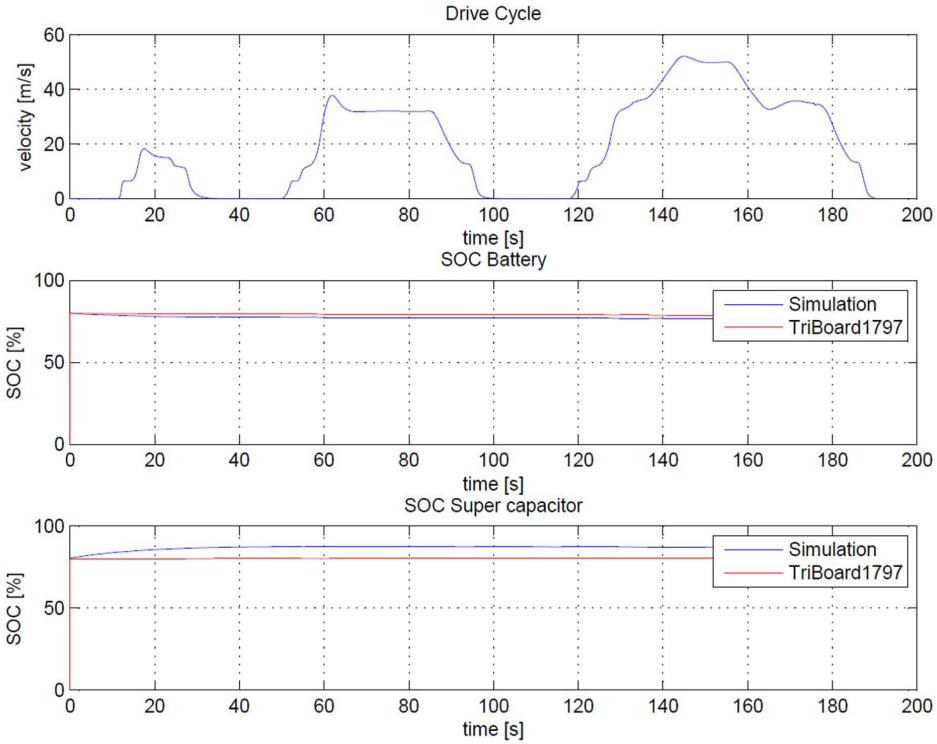

Figure 4.37: Comparison of the SoC of the battery and the super capacitor when simulated with Simulink and as multi-core integration.

Finally, the whole management system is integrated as multi-core approach on two TriBoards1797 and coupled to the cockpit and vehicle models via ICOS. Figure 4.38 shows how the simulation model is distributed in this case. The four subsystems of the management system are split up into two parts which are integrated on the micro controllers.

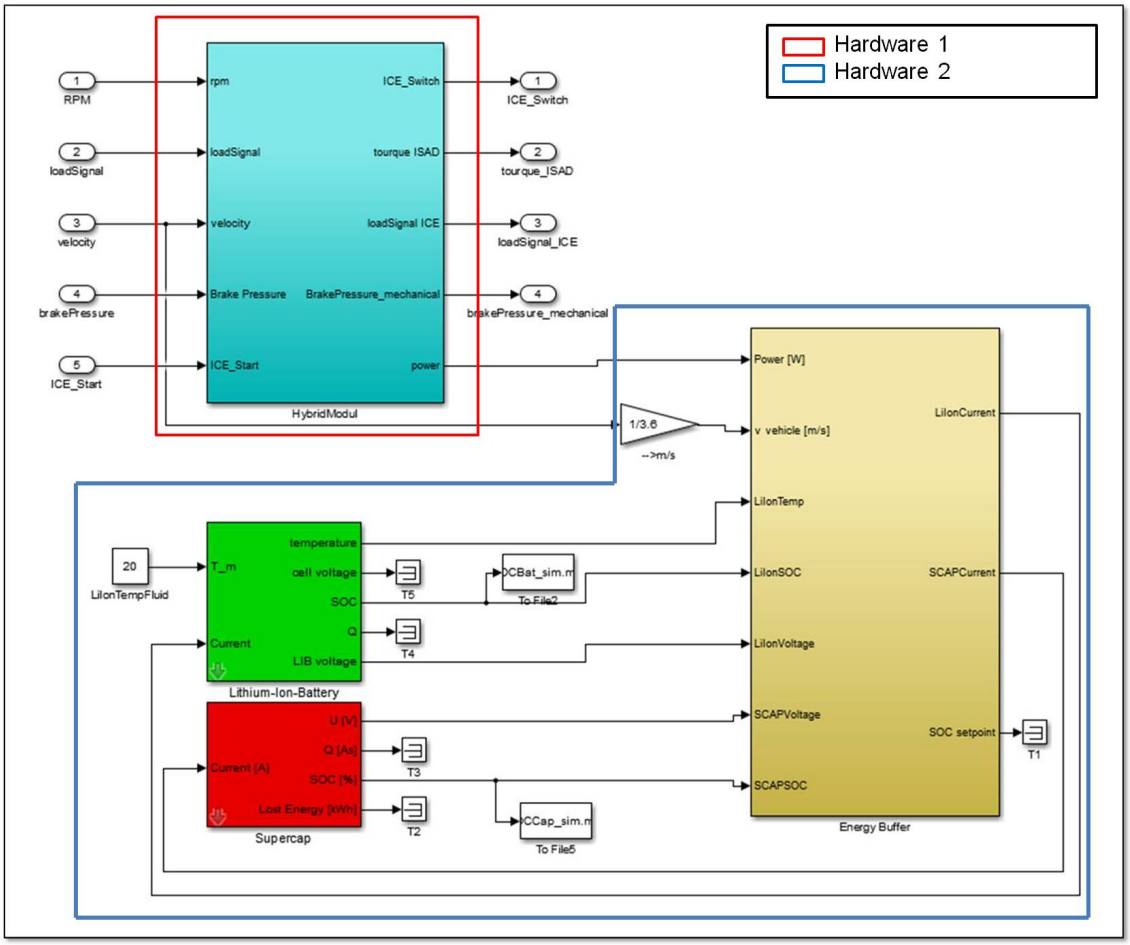

Figure 4.38: Overview of the Simulink model used for the management system distribution.

The control logic for the first TriBoard, marked in red, includes the whole *hybrid module*. The second TriBoard implements the logic of the *battery*, the *super capacitor* and the *energy buffer*, marked in blue. All these model parts are depicted in figure 4.29. Due to the fact, that the before chosen distribution did not lead to a reasonable controller behavior, in this scenario the whole feedback loop is integrated on one TriBoard. Moreover, by using this way of distribution minimal communication effort is necessary. In this scenario the power signal is the only connection between the two used boards. This signal is computed by the *hybrid module* and sent to the *energy buffer* on the second hardware through the CAN bus. The boards are able to compute and transmit the belonging output values before a new input value is sent by ICOS and thus the synchronization works properly.

The results of this distribution are represented in figures 4.39, 4.40 and table 4.6. Figure 4.39 depicts a comparison of the computed HiL and MiL torques. It is shown that even in this case the control functions on the TriBoards need some time to reach a steady-state. Afterwards, the signals are equal, thus the integrated *hybrid module* works properly on the TriBoard1797. Thus, for future HiL implementation it might be a good idea, to somehow distinguish between dummy-data sent by ICOS and real simulation data, to avoid such an initial behavior.

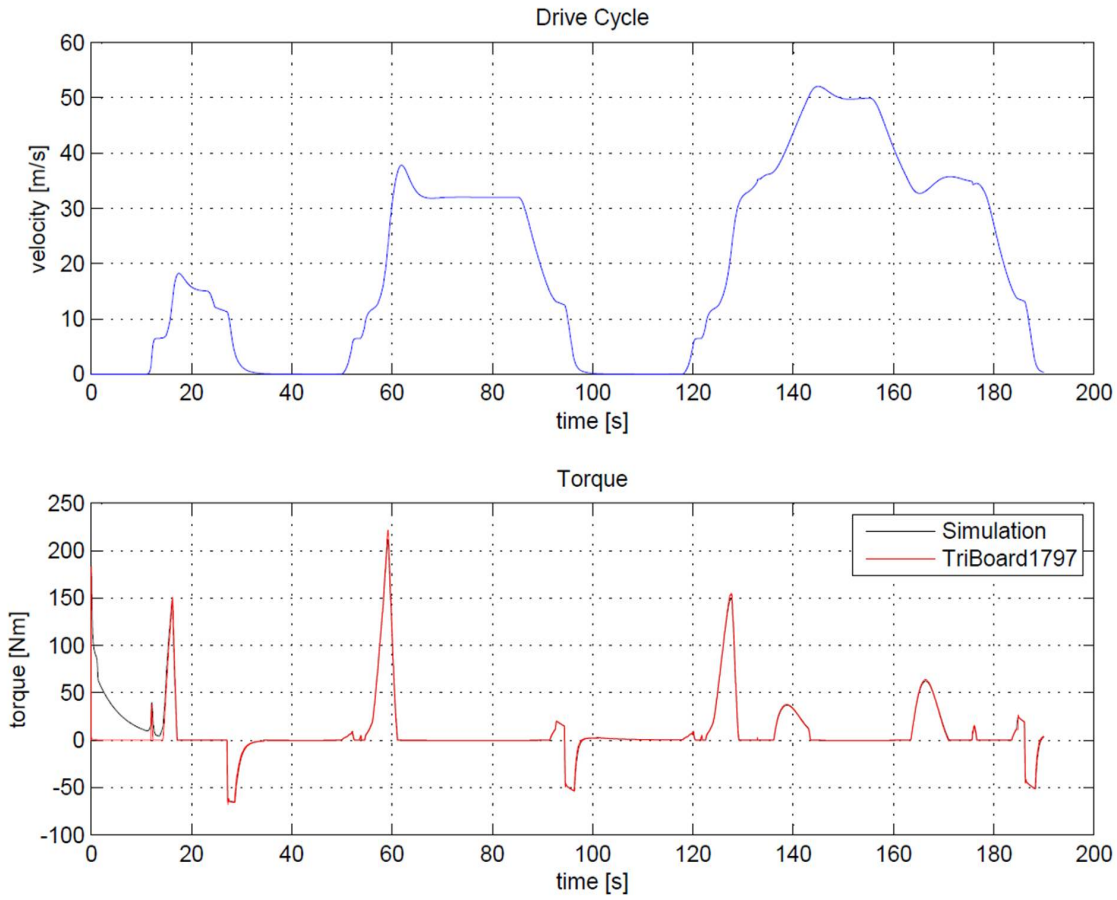

Figure 4.39: Comparison of the torque signal computed with Simulink and two TriBoard1797.

Figure 4.40 illustrates the comparison of the resulting SoCs. It can be seen that this multi-core solution leads to a much better result than the approach before. This results confirm the assumption that the bad result of the example before was caused by the distribution of the feedback loop. A comparison of the achieved average errors of the SoCs of the single- and multi-core integration are represented in table 4.6. In this case, the relative errors of the multi-core integration and the single-core integration are very similar. The reason why the distributed approach achieves a worse result is, that a multi-core integration requires a higher communication effort.

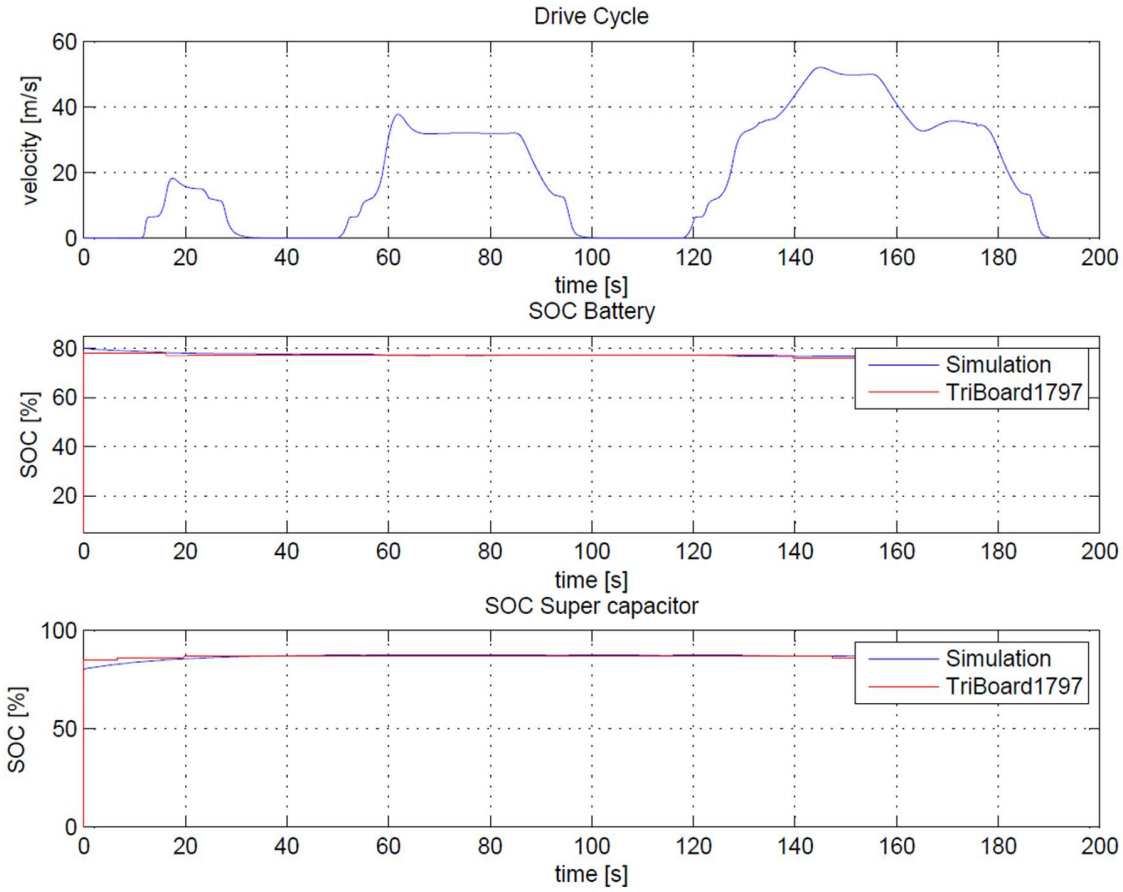

Figure 4.40: Comparison of the SoC of the battery and the super capacitor when simulated with Simulink and as multi-core integration of the whole system.

SoC	**Relative Error**
single-core super capacitor	0.4902%
single-core battery	0.4500%
multi-core super capacitor	0.6958%
multi-core battery	0.4750%

Table 4.6: Relative percentage deviation of the SoCs in the single-core and multi-core case.

All these integration examples showed that it is possible to distribute control functions over several devices, as long as appropriate parts of the model are determined to be split up. Furthermore, these HiL integrations showed, that an optimal distribution does not achieve as good results as a single-core integration, but the deviations were acceptable in these examples. The resulting deviations in all multi-core approaches are caused by the rising communication effort, which leads to timing delays, which are additional dead times within a control loop. Such dead times have a huge influence on the stability of control functions. All in all, this thesis demonstrated that it is reasonable to chose a distribution with short timing delays, to minimize dead times and therefore decrease the stability of control functions as little as possible.

Chapter 5

Conclusion and Outlook

5.1 Conclusion

In this Master's thesis a seamless methodology for efficient hardware integration and HiL simulation of modelled control functions has been developed. It has been shown that in general a process, coming from a MiL to a HiL solution consists of three main steps: the *model development*, the *code generation and distribution* and the *hardware integration*. Additionally, in this thesis a control function for a DMES for a Lotus Evora 414E plug-in hybrid vehicle with range extender has been developed. However, the main focus of this work has been the definition of an appropriate toolchain, enabling one engineer to implement a hardware integration as well as a HiL simulation, using an already existing model in an intuitive way. Therefore, an approach to ease the interaction of different engineering domains has been developed in form of a new XML-specification. The resulting XML-file format has been defined in a XSD-schema and serves as connection point between different engineering domains included in this development process. This file enables an engineer to realize a hardware integration with minimum effort, by providing all necessary information regarding a simulation model, its desired distribution and the suitable hardware. In the future all this information will be added to the XML-file by different engineers involved in a development process. Thus, each engineer shares his knowledge necessary for further integration steps with the hardware integrator in a predefined manner. This defined XSD-schema is a fundamental part of the represented toolchain developed in this thesis.
Based on the defined XML-file, the presented toolchain consists of three general steps which need to be performed to integrate a control model on a micro controller and simulate HiL tests. The first step is the code generation of the the model parts, determined in the XML-file. Afterwards, the generated code needs to be extended by a communication interface (CAN) to enable data exchange with the controller environment. Afterwards, the code can be flashed on an appropriate platform. To analyze the behavior of the integrated control functions a co-simulation model needs to be set up.
As proof-of-concept of this defined toolchain, two different simulation models have been integrated on different types of micro controllers, as single-core as well as multi-core solutions. These hardware integrations are verifying the correctness of the defined toolchain. Furthermore, these integrations have been used to get more information about the influ-

ence of specific model configurations on the behavior of embedded control functions. It has been shown that the used solver type as well as the determined step-size of a model have a huge influence on the stability of an embedded control function. Furthermore, it has been demonstrated that also the decision of how a simulation model is distributed, influences the achieved controller behavior. For instance it has been indicated that the distribution of a feedback loop should be avoided if possible. Moreover, the communication between different model parts should be reduced to a minimum by choosing a wise distribution.

5.2 Outlook

The toolchain developed in this Master's thesis serves as a first demonstrator for the EMC^2 project. It is a base for further investigations of mixed-critical systems. The focus of this work has been the development of a seamless toolchain, which has been tested through the development and integration of control functions e.g. for the DMES. An optimal allocation is beyond the scope of this thesis. Thus, an optimization in terms of timing and safety, as well as control stability (and a combination of those) could be part of future work. Furthermore, distribution needs to be considered during the modelling process, for instance in terms of possible parallelization methods.
As a next step, the generated code examples can be integrated on a real multi-core platform, therefore a real-time operating systems needs to be set-up. Furthermore, the synchronization, which now is implemented as part of the integrated code, needs to be done by this real-time operating system using a suitable scheduling mechanism. Additionally, an investigation and evaluation of different operating systems regarding different criteria (e.g. AUTOSAR, ISO26262 compliance,...) should also be considered as future work.
Finally, to improve the defined toolchain, an editor for the developed XML file format can be implemented, to ease the input of distribution information for the involved engineers. Further, a method to automatically generate clue-code for the CAN interface can be added to this editor to minimize the necessary manual work for the hardware integrator.

Appendix A

List of abbreviations

AMP Arbitration on Message Priority

ASAM Association for Standardisation of Automation- and Measuring Systems

ASIL Automotive Safety Integrity Level

AUTOSAR AUTomotive Open System ARchitecture

CAN Controller Area Network

DBC Data Base CAN

CPU Central Processing Unit

CSMA Carrier Sense Multiple Access

CSMA/CD Carrier Sense Multiple Access/Collision Detection

CSMA/CR Carrier Sense Multiple Access/Collision Reduction

DCDC Direct Current Direct Current

DAS Distributed Application Subsystem

DMES Dual Mode Energy Storage

DMESC Dual Mode Energy Storage Controller

DSP Digital Signal Processor

ECU Electric Control Unit

EEPROM Electrically Erasable Programmable Read-Only Memory

EM Electrical Machine

EMC^2 Embedded Multi-Core Systems for Mixed Criticality Applications in dynamic and changeable Real-time Environments

EV Electric Vehicle

FIBEX Field Bus Exchange Format

HiL Hardware in the Loop

HEV Hybrid Electrical Vehicle

ICE Internal Combustion Engine

iCOMPOSE Integrated Control of Multiple-Motor and Multiple-Storage Fully Electric Vehicles

ICOS Independent Co-Simulation

ICT Information and Communication Technologies

IDE Integrated Development Environment

IP Internet Protocol

ISO International Organization for Standardization

NEDC New European Driving Cycle

ODE Ordinary Differential Equations

OSI Open Systems Interconnection

I/O Input/Output

LIN Local Interconnect Network

MAC Media Access Control

MiL Model in the Loop

MOST Media Oriented Systems Transport

PCP Peripheral Control Processor

RFC Request For Comments

RISC Reduced Instruction Set Computer

SPI Serial Peripheral Interface

SRAM Static Random-Access Memory

SoC State of Charge

SUV Sport Utility Vehicle

TC TriCore

UDP User Datagram Protocol

UML Unified Modelling Language

UART Universal Asynchronous Receiver Transmitter

USB Universal Serial Bus

VCU Vehicle Control Unit

XML Extensible Markup Language

XSD XML Schema Definition

Appendix B

Code samples

B.1 CAN data conversion DMESC single-core solution

```
#define FACTOR_ECAP  0.0656
#define OFFSET_ECAP 0
#define FACTOR_V  0.000916
#define OFFSET_V  0
#define FACTOR_I_LOAD  0.000072
#define OFFSET_I_LOAD  -1000
#define FACTOR_U_CAP  0.0046
#define OFFSET_U_CAP  0
#define FACTOR_U_BAT  0.0061
#define OFFSET_U_BAT  0
#define FACTOR_I_CAP_SOLL  0.000268
#define OFFSET_I_CAP_SOLL  -2000

static int32_T E_cap_rx;
static int32_T v_rx;
static int32_T i_load_rx;
static int32_T u_cap_rx;
static int32_T u_bat_rx;
static int32_T i_cap_soll;
static int32_T E_cap_converted;
static int32_T v_converted;
static int32_T i_load_converted;
static int32_T u_cap_converted;
static int32_T u_bat_converted;

static char rxmsg1[8];
static char rxmsg2[8];
static char txmsg[8] = {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00};
static char rx1, rx2, tx1;
```

```
int handleCANmsg(void)
{
// receive ids 1 and 2
rx1 = AddCANMessageRX(1, 8, 0x001, 0, 0x3ff);
rx2 = AddCANMessageRX(1, 8, 0x002, 0, 0x3ff);

// CAN RX
GetCANMessage(rx1,(unsigned*)rxmsg1);
GetCANMessage(rx2,(unsigned*)rxmsg2);
convertRxMsg();

//.. do something


// transmit id 3
tx1 = AddCANMessageTX(1, 8, 0x003, 0);
convertTxMsg();
UpdateCANMessage(tx1,(unsigned*) txmsg);
return 0;
}

void convertRxMsg()
{

E_cap_rx  = rxmsg1[0] | (rxmsg1[1]<<8) | (rxmsg1[2]<<16);
v_rx  = rxmsg1[3] | (rxmsg1[4]<<8);
i_load_rx = rxmsg2[0] | (rxmsg2[1]<<8) | (rxmsg2[2]<<16);
u_cap_rx  = rxmsg2[3] | (rxmsg2[4]<<8);
u_bat_rx  = rxmsg2[5] | (rxmsg2[6]<<8);

E_cap_converted = E_cap_rx * FACTOR_ECAP + OFFSET_ECAP;
v_converted   = v_rx * FACTOR_V + OFFSET_V;
i_load_converted = i_load_rx * FACTOR_I_LOAD + OFFSET_I_LOAD;
u_cap_converted  = u_cap_rx * FACTOR_U_CAP + OFFSET_U_CAP;
u_bat_converted  = u_bat_rx * FACTOR_U_BAT + OFFSET_U_BAT;
}

void convertTxMessage()
{
int32_T i_cap_soll_tx = (i_cap_soll - OFFSET_I_CAP_SOLL) / FACTOR_I_CAP_SOLL;

txmsg[0] =  i_cap_soll_tx  & 0xff;
txmsg[1] = (i_cap_soll_tx>>8)  & 0xff;
txmsg[2] = (i_cap_soll_tx>>16) & 0xff;
}
```

B.2 ICOS RealTime Wrapper .ini-file - DMESC single-core solution

```
[INPUT]
E_cap    0
v        0
i_load   0
u_cap    0
u_bat    0

[OUTPUT]
i_cap_soll     0

[CAN-Settings]
CAN-Channel   = 81
CAN-Baud-Rate = 20 # 1 MBit/s

#name                          CAN-ID  start-bit  end-bit(incl.)   factor   offset
CAN-In-Parameter  = E_cap         1       0           23            0.0656       0
CAN-In-Parameter  = v             1       24          39            0.000916     0
CAN-In-Parameter  = i_load        2       0           23            0.000072  -1000
CAN-In-Parameter  = u_cap         2       24          39            0.0046       0
CAN-In-Parameter  = u_bat         2       40          55            0.0061       0

CAN-Out-Parameter = i_cap_soll    4       0           23            0.000477  -4000
```

Appendix C

XSD-Schema

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
attributeFormDefault="unqualified">
<xs:element name="distribution">
<xs:complexType>
<xs:sequence>
<xs:element name="path" type="xs:string" minOccurs="1" maxOccurs="1"/>
<xs:element name="ModelName" type="xs:string" minOccurs="1" maxOccurs="1"/>
<xs:element name="modelASIL" type="xs:string" minOccurs="0"/>
<xs:element name="distributionFunction" maxOccurs="unbounded">
<xs:complexType>
<xs:sequence>
<xs:element name="PartName" type="xs:string" minOccurs="1" maxOccurs="1"/>
<xs:element name="functionASIL" type="xs:string" minOccurs="0" maxOccurs="1"/>
<xs:element name="inputSignals" maxOccurs="1">
<xs:complexType>
<xs:sequence>
<xs:element name="signal" maxOccurs="unbounded">
<xs:complexType>
<xs:attribute name="signalName" type="xs:string" use="required"/>
<xs:attribute name="signalRange" type="xs:integer" use="required"/>
<xs:attribute name="signalPrecision" type="xs:double" use="required"/>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="outputSignals" maxOccurs="1">
<xs:complexType>
<xs:sequence>
<xs:element name="signal" maxOccurs="unbounded">
<xs:complexType>
<xs:attribute name="signalName" type="xs:string" use="required"/>
<xs:attribute name="signalRange" type="xs:integer" use="required"/>
<xs:attribute name="signalPrecision" type="xs:double" use="required"/>
</xs:complexType>
</xs:element>
```

```
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="hardwareDevice" minOccurs="1" maxOccurs="1">
<xs:complexType>
<xs:sequence>
<xs:element name="hardwareType" type="xs:string" minOccurs="1" maxOccurs="1"/>
<xs:element name="communicationInterface" type="xs:string"
minOccurs="1" maxOccurs="1"/>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:schema>
```

Appendix D

xml-specification samples

D.1 xml-file for DMESC single-core integration

D.1.1 DMESC single-core integration.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<distribution>
        <modelName> DMES </modelName>
        <modelPath> ...\DMES.mdl </modelPath>
        <modelSolver> ODE3 </modelSolver>
        <modelStepSize> 0.01 </modelStepSize>
    <distributionFunction>
            <partName> DMESC </partName>
            <inputSignals>
                <signal>
                    <sigName> E_cap </sigName>
                    <sigRange> [0, 1100000] </sigRange>
                    <sigPrecision> 10^-1 </sigPrecision>
                </signal>
                <signal>
                    <sigName> v </sigName>
                    <sigRange> [0, 60] </sigRange>
                    <sigPrecision> 10^-3 </sigPrecision>
                </signal>
                <signal>
                    <sigName> i_load </sigName>
                    <sigRange> [-1000, 200] </sigRange>
                    <sigPrecision> 10^-4 </sigPrecision>
                </signal>
                <signal>
                    <sigName> u_cap </sigName>
                    <sigRange> [0, 300] </sigRange>
                    <sigPrecision> 10^-2 </sigPrecision>
                </signal>
                <signal>
                    <sigName> u_bat </sigName>
                    <sigRange> [0, 400] </sigRange>
                    <sigPrecision> 10^-2 </sigPrecision>
```

```
                </signal>
            </inputSignals>
            <outputSignals>
                <signal>
                    <sigName> i_cap_soll </sigName>
                    <sigRange> [-2000, 2500] </sigRange>
                    <sigPrecision> 10^-3 </sigPrecision>
                </signal>
            </outputSignals>
            <hardwareDevice>
                <HWname> VIF CAN Board V1.0 </HWname>
                <commInterface> CAN </commInterface>
            </hardwareDevice>
        </distributionFunction>
</distribution>
```

.

D.1.2 DMESC multi-core integration.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<distribution>
        <modelPath> ...\DMES.mdl </modelPath>
        <modelName> Controller1 </name>
        <modelSolver> ODE3 </modelSolver>
        <modelStepSize> 0.01 </modelStepSize>
        <distributionFunction>
            <partName> DMESC </partName>
            <inputsSignals>
                <signal>
                    <signalName> E_cap </sigName>
                    <signalRange> [0, 1100000] </sigRange>
                    <sigPrecision> 10^-1 </sigPrecision>
                </signal>
                <signal>
                    <sigName> v </sigName>
                    <sigRange> [0, 60] </sigRange>
                    <sigPrecision> 10^-3 </sigPrecision>
                </signal>
            </inputSignals>
            <outputsSignals>
                <signal>
                    <sigName> i_c_soll_soc </sigName>
                    <sigRange> [-1000, 1000] </sigRange>
                    <sigPrecision> 10^-3 </sigPrecision>
                </signal>
            </outputSignals>
            <hardwareDevice>
                <HWname> VIF CAN Board V1.0 </HWname>
                <commInterface> CAN </commInterface>
            </hardwareDevice>
```

```
        </distributionFunction>
        <distributionFunction>
                <partName> Controller2 </partName>
                <inputsSignals>
                        <signal>
                                <sigName> i_load </sigName>
                                <sigRange> [-1000, 200] </sigRange>
                                <sigPrecision> 10^-4 </sigPrecision>
                        </signal>
                        <signal>
                                <sigName> u_cap </sigName>
                                <sigRange> [0, 300] </sigRange>
                                <sigPrecision> 10^-2 </sigPrecision>
                        </signal>
                        <signal>
                                <sigName> u_bat </sigName>
                                <sigRange> [0, 400] </sigRange>
                                <sigPrecision> 10^-2 </sigPrecision>
                        </signal>
                </inputSignals>
                <outputsSignals>
                        <signal>
                                <sigName> i_cap_soll </sigName>
                                <sigRange> [-5000, 5000] </sigRange>
                                <sigPrecision> 10^-3 </sigPrecision>
                        </signal>
                </outputSignals>
                <hardwareDevice>
                        <HWname> VIF CAN Board V1.0 </HWname>
                        <commInterface> CAN </commInterface>
                </hardwareDevice>
        </distributionFunction>
</distribution>
```

Bibliography

[1] L. Delgrossi, "The Future of the Automobile." Lecture notes, 2013.

[2] A.Albert, "Comparison of event-triggered and time-triggered concepts with regards to distributed control systems," *Embedded World Conf. 2004*, pp. 235–252, February 2004.

[3] M.Bernhard Ch.Buckl V.Döricht M.Fehling L.Fiege H.von Grolman N.Ivandic Ch.Janelle C.Klein K.-J.Kuhn Ch.Patzlaff B.Riedl B.Schätz Ch.Stanek, *The Software Car: Information and Communication Technology (ICT) as an Engine for the Electromobility of the Future*. ForTISS GmbH, March 2011.

[4] B. Hardung, T. Kölzow, and A. Krüger, "Reuse of Software in Distributed Embedded Automotive Systems," in *Proceedings of the 4th ACM International Conference on Embedded Software*, EMSOFT '04, (New York, NY, USA), pp. 203–210, ACM, 2004.

[5] N. Navet, Y. Song, F. Simonot-Lion, and C. Wilwert, "Trends in Automotive Communication Systems," *Proceedings of the IEEE*, vol. 93, pp. 1204–1223, June 2005.

[6] G. Morgan and A. Borg, "Multi-core Automotive ECUs: Software and Hardware Implications." 2009.

[7] "ISO 26262-1:2011 Road vehicles - Functional safety," 2011.

[8] W. Weber, "Introduction to the EMC^2 project," date of access: 02.12.2014.

[9] M. Lukasiewycz, S. Steinhorst, S. Andalam, F. Sagstetter, P. Waszecki, W. Chang, M. Kauer, P. Mundhenk, S. Shanker, S. Fahmy, and S. Chakraborty, "System architecture and software design for Electric Vehicles," in *Design Automation Conference (DAC), 2013 50th ACM / EDAC / IEEE*, pp. 1–6, May 2013.

[10] P. Peti, R. Obermaisser, F. Tagliabo, A. Marino, and S. Cerchio, "An integrated architecture for future car generations," in *Object-Oriented Real-Time Distributed Computing, 2005. ISORC 2005. Eighth IEEE International Symposium on*, pp. 2–13, May 2005.

[11] Continental, "Electronic Vehicle Management - New options for commercial vehicle controllers," tech. rep., Continental, 2009.

[12] iCompose Consortium, "iCompose - Progress beyond SOA." http://www.i-compose.eu/iCompose/index.php/project/progress. date of access: 16.01.2015.

[13] D. Zhu and C. Qian, "Challenges in Future Automobile Control Systems with Multicore Processors." 2011.

[14] A. Burns and R. Davis, "Mixed Criticality Systems - A Review." Fifth edition, 2015.

[15] K. Schmidt, M. Buhlmann, C. Ficek, and K. Richter, "Design Patterns for Highly Integrated ECUS with Various ASIL Levels," *ATZelektronik worldwide*, vol. 7, no. 1, pp. 22–27, 2012.

[16] C. Ficek, N. Feiertag, and K. Richter, "Applying the AUTOSAR timing protection to build safe and efficient ISO 26262 mixed-criticality systems." 2013.

[17] S. Baruah, V. Bonifaci, G. D'Angelo, H. Li, H. Marchetti-Spaccamela, N. Megow, and L. Stougie, "Scheduling real-time mixed-criticality jobs," *IEEE Transactions on Computers*, vol. 61, no. 8, pp. 1140–1152, 2012.

[18] ATMEL, *8-bit AVR Microcontroller with 128K Bytes of ISP Flash and CAN Controller - AT90CAN128*, 2006.

[19] Infineon Technologies AG, *TriBoard TC179X - User's Manual*, 2010.

[20] *TC1797 32-Bit Single-Chip Microcontroller - User's Manual V1.1*, 2009.

[21] M.Benedikt J.Zehetner D.Watzenig J.Bernasch, "Moderne Kopplungsmethoden - Ist Co Simulation beherrschbar?," *NAFEMS Online-Magazin, Zeitschrift für numerische Simulationsmethoden und angrenzende Gebiete*, 7 2012.

[22] S. Messner, "Einfluss von nicht-iterativer Co-Simulation auf die numerische Lösung von Systemgleichungen," Master's thesis, Technische Universität Graz, 2014.

[23] M. Benedikt, *Eine Koppelungsmethode für die nicht-iterative Co-Simulation*. PhD thesis, Kompetenzzentrum - Das Virtuelle Fahrzeug mbh Graz, 2012.

[24] S. Corrigan, "Indroduction to the Controller Area Network (CAN)," tech. rep., TEXAS INSTRUMENTS, 2008.

[25] Texas Instruments, "The ISO 11898 CAN Standard," date of access: 19.03.2015.

[26] H. Zimmermann, "OSI Reference Model–The ISO Model of Architecture for Open Systems Interconnection," *Communications, IEEE Transactions on*, vol. 28, pp. 425–432, Apr 1980.

[27] CAN in Automation (CiA), "CAN physical layer," date of access: 19.03.2015.

[28] CAN in Automation (CiA), "CAN protocol," date of access: 19.03.2015.

[29] J. Postel, "User Datagram Protocol (RFC 768)." IETF Request For Comments, Aug. 1980. Introduction to UDP.

[30] "User Datagram Protocol." http://en.wikipedia.org/wiki/User_Datagram_Protocol. date of access: 20.03.2015.

[31] T. Lorenz, J. Taube, M. Ihle, O. Manck, and H. Beikirch, "Fibex gateway configuration tool chain," *iCC 2006*, pp. 13–19, 2006.

[32] AUTOSAR Release 4.2.1, *Software Component Template*.

[33] T. Bachmann, "FIBEX XML format and AUTOSAR development," *EE Times - Connecting the Global Electronics Community*, July 2009.

[34] AUTOSAR Release 4.2.1, *System Template*.

[35] Vector, "DCB File Format Documentation," 2007. date of access: 19.03.2015.

[36] S. Doyle, "EVORA 414E HYBRID," 2011.

[37] T. Christen, M. Carlen, "Theory of Ragone plots," *Journal of Power Sources*, vol. 91, no. 2, pp. 210 – 216, 2000.

[38] Y. Bin Tan and J.-M. Lee, "Graphene for supercapacitor applications," vol. Royal Society of Chemistry, 2013.

[39] A. Mulay, Y. Vardhan pant, and R. Mangharam, "Protodrive - Rapid prototyping platform for Electric Vehicle powertrain." http://mlab.seas.upenn.edu/protodrive/. date of access: 11.02.2015.

[40] C. Paar, "Energy Management in Hybrid Electric Vehicles using Co-Simulation," Master's thesis, FH Wiener Neustadt, 2009.

[41] "NEDC."

[42] G. Stettinger, M. Benedikt, N. Thek, and J. Zehetner, "On the Difficulties of Real-Time Co-Simulation," vol. International Conference on Computational Methods for Coupled Problems in Science and Engineering, 2013.