TECHNISCHE UNIVERSITÄT WIEN

VIENNA UNIVERSITY OF TECHNOLOGY

DIPLOMARBEIT

Diagnosis of CAN-based Legacy Applications in the Time-Triggered Architecture

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines Diplomingenieurs der Technischen Informatik

unter der Leitung von

O. Univ. Prof. Dr. phil. Hermann Kopetz

und

Dipl.-Ing. Dr. techn. Roman Obermaisser

als verantwortlich mitwirkender Assistent

Institut für Technische Informatik 182

durchgeführt von

Florian Schüller
Matr.-Nr. 9925146

Breitenfurterstraße 114b/173
1120 Wien

Wien, im März 2007 _____

Abstract

The Controller Area Network (CAN) is the most widespread network protocol in the automotive domain of today's world. CAN has gained wide acceptance because of its flexibility and the low cost of hardware with CAN support. In the automotive domain it is used for control applications as well as for diagnostic purposes, comfort functions and other applications. Several protocols and software layers (e.g. TP2.0, KWP2000, DEH etc.) have been developed to be able to handle especially the complex diagnostic functions in cars. They include security mechanisms, flashing functions, and many more. Also, in the past few years more and more safety-related functions have been integrated in modern cars using CAN. Famous representatives are the airbag, the Anti-lock Brake System (ABS), the Electronic Stability Control (ESC) and so on. This trend shows us that more and more even safety-critical applications are going to be controlled by electronic devices in the future.

To handle the communication needs of future safety-critical applications (e.g. X-by-wire) a time-triggered (TT) technology will be used. The Time-Triggered Protocol (TTP) for example is a famous representative which facilitates the creation of dependable distributed embedded real-time systems. It is successfully used for example in the avionic domain for applications which have to be ultra-reliable. Nevertheless, for non safety-critical applications, it is important to reuse existing CAN hardware and software to preserve the existing investments in diagnostic equipment and the diagnostic know-how which has been gathered for several years now. The huge investment in legacy equipment also makes a solution attractive which reuses most of this hardware.

In this thesis, we show that it is possible to use the existing diagnostic tester hardware to diagnose not only CAN-based Electronic Control Units (ECUs) but also time-triggered ECUs. Another aspect of this work is to demonstrate that it is possible to use the "Hersteller Initiative Software" (HIS)-CAN interface to emulate CAN for the complete diagnosis software stack which is used today. The stack was provided by a major European carmaker for our work. Another important goal in this work is to show that it is possible to provide additional diagnostic information (e.g. membership) using the same diagnostic equipment as for CAN. By utilizing a Time-Triggered Protocol (TTP), which provides error containment in the time domain, we also tackle the trouble not identified (TNI) phenomenon which exists in CAN.

Kurzfassung

In der Automobilindustrie ist das Controller Area Network (CAN) das am weitesten verbreitete Kommunikationprotokoll. CAN hat breite Anwendung gefunden aufgrund der Flexibilität und der niedrigen Kosten von Hardware die CAN unterstützt. Im Auto wird CAN für Regelungen, Diagnosefunktionen, Komfortfunktionen und einige andere Anwendungen genutzt. Ausserdem wurden viele unterschiedliche CAN-basierende Protokolle und Softwareschichten (z.B. TP2.0, KWP2000, DEH usw.) entwickelt um die notwendigen komplexen Diagnosefunktionen zu realisieren. Die Diagnosefunktionen umfassen Sicherheitsmechanismen, Programmierfunktionen und vieles mehr. In den letzten Jahren sind mehr und mehr sicherheitsrelevante Funktionen, unter der Verwendung von CAN, in das Automobil integriert worden. Bekannte Vertreter sind der Airbag, das Antiblockiersystem (ABS), das Elektronisches Stabilitäts Programm (ESP) u.v.m.. Dieser Trend zeigt uns, dass in Zukunft immer mehr auch sicherheitskritische Anwendungen durch elektronische Bauteile gesteuert werden.

Um die Voraussetzungen an das Kommunikationssystem einer sicherheitskritischen Anwendung der Zukunft zu erfüllen, wird eine zeitgesteuerte Architektur Verwendung finden. Das zeitgesteuerte Protokoll TTP ist ein gutes Beispiel für ein zuverlässiges Bussystem das eine Implementierung von verteilten eingebetteten Systemen ermöglicht. In der Flugzeugindustrie wird TTP schon erfolgreich für sicherheits-kritische Anwendungen genutzt. Für Automobilhersteller ist es sehr wichtig bestehende Hardware und Software aus Kostengründen weiterzuverwenden. Auch das Diagnoseequipment und das Know-How zur Diagnose stellen eine große Investition dar, die es zu bewahren gilt. Durch diese Investitionen werden Lösungen bevorzugt, die diese Resourcen in größtem Maße weiterverwenden.

In dieser Arbeit wird gezeigt, dass es möglich ist existierende Diagnosehardware nicht nur für CAN-basierte Steuergeräte sondern auch für zeitgesteuerte Steuergeräte zu verwenden. Der “Hersteller Initiative Software” (HIS)-CAN Interfacestandard kann verwendet werden, um den kompletten existierenden Diagnosestack an ein zeitgesteuertes Netzwerk anzubinden. Eines der wichtigsten Ziele dieser Arbeit ist aber die Visualisierung der erweiterten Diagnosemöglichkeiten (z.B. Membership) mit dem selben Diagnoseequipment wie für CAN. Mit der Verwendung von TTP, dass eine strikte Unterteilung der Zeit beinhaltet, wollen wir auch das TNI Phänomen in Angriff nehmen, dass bei CAN existiert.

Contents

List of Figures

List of Tables

1 Introduction

In today's cars many functions are controlled electronically like control applications or comfort functions. Even safety-enhancing functions are already available in new cars like the Anti-lock Brake System (ABS) or the Electronic Stability Control (ESC). In the past years these safety-related applications have been widely introduced. It is obvious that the next steps for the future are towards safety-critical applications like steer-by-wire or brake-by-wire.

The current communication system used in the majority of the cars is the event-triggered CAN. It is a cheap communication system and much know-how has been gathered in the past years. However it is not the best choice for the implementation of dependable distributed embedded real-time applications, because does not support media redundancy, atomic broadcasts and some more[25]. Safety-critical systems require a reliable, fault-tolerant network which a time-triggered communication system fulfills [37][26][3][4][7][18, chapter 2][18, subsection 4.4][22].

Time-triggered protocols also facilitate the establishment of fault isolation by partitioning the network in the time domain which is beneficial for improved diagnosis. Partitioning also is a necessity in creating integrated safety-critical systems [37]. In this work we focus on the Time-Triggered Protocol / C (TTP/C) which has already been successfully used for ultra-dependable systems e.g. in the avionic domain.

Introducing a new network protocol is usually linked with high development costs. To reduce this costs we try to reuse as much as we can from the investment in the legacy communication system. We try to reuse hardware, software and diagnosis know how with our model. By using standardized interfaces we avoid internal changes in the applications which is very important while focusing on the problem of Intellectual Property (IP).

Numerous works focus on the migration of event-triggered systems to time-triggered systems whereas this work addresses several aspects of the diagnosis of CAN-based applications on top of a Time-Triggered Architecture (TTA). The search for diagnosis improvements of the existing CAN was inspired by the existence of the trouble not identified (TNI) problem. Approaches like media redundancy and fault isolation lead to the conclusion that TTP/C has advantages in diagnosis.

In this work we want to make use of additional diagnosis features available on TTP/C. We combine these features with legacy software in a prototype setup. We want to show that it is possible to reuse existing software and hardware while migrating to a dependable communication system. Besides reusing legacy hardware and software in the car we want to show that the existing diagnostic tester hardware available in every garage is able to interact with CAN-based ECUs as well as time-triggered ECUs. This is a very important issue to avoid equipping every garage with new diagnostic hardware because of a new communication system.

1.1 Structure Of Work

In Chapter 2 "Basic Concepts" concepts used in the remainder of the thesis including diagnosis in the automotive domain, the problem of intellectual property, event-triggered and time-triggered architectures are described. They are the fundamentals for understanding this work.

Chapter 3 "System Model" describes the architecture, protocols and standards used within our model in detail and how they are integrated. It also contains different views of our model including the concept of Linking Interfaces (LIFs).

Chapter 4 "Implementation" elaborates on the implementation of a prototype setup. It illustrates the use of our model applied to a demonstration showing its features on real hardware. Topics like "CAN Emulation", the "Two Level Design of a TTA" and the "Added Diagnostic Features" are the most important here.

Chapter 5 "Validation" tests all properties of our model and confirms our preconditions for hardware and software compatibility. It contains detailed test descriptions of our implementation as well as the investigation on the temporal properties of our setup.

2 Basic Concepts

In this chapter fundamental topics related to diagnosis, legacy applications, time-triggered and event-triggered architectures are briefly summed up. This will give an overview to understand the system model and its implementation.

2.1 Diagnosis In The Automotive Domain

In todays cars up to 70 ECUs improve traveling comfort or increase safety [8]. These units may also have malfunctions and it should be possible to determine the exact cause of the malfunction. In this section the diagnosis of malfunctions or errors concerning electronic parts of a car is addressed.

Present day cars are equipped with an On-Board Diagnostics (OBD) system. This system detects errors and either stores them for the inspection or shows them to the driver if the error is severe or influencing the behavior of the car. At the beginning of OBD systems, errors have been signaled using Malfunction Indicator Lights (MILs). In modern cars all type of information including errors are displayed on Liquid Crystal Displays (LCDs).

Each ECU in a modern car is able to detect local errors for each ECU. Correlating the errors reported by each ECU to find the real problem source is still a big problem today and will be discussed later. Each error has a specific identification number and is called Diagnostic Trouble Code (DTC) in the automotive domain. These errors are stored in the ECU and can be retrieved together with the information whether this error is permanent or transient[38]. Usually one fault (e.g. short circuit of the communication bus) causes more than one DTC so it is very important to save for example the mileage when the error occurred to be able to analyze possible correlations of DTCs.

The “freeze frame” table stores additional information about the current status of the car at the instant when an error occurs. Information like the mileage, the time-stamp and environmental values within the freeze frame allow only an expert to correlate error messages and determine the real source of the error.

The basic concept of detecting errors, storing this information and revealing it during an inspection is the same for all vehicle manufacturers.

During the car inspection the mechanic connects the diagnosis tester to the car and is able to retrieve the DTCs. This is done using special protocols which are often a mixture of standards and manufacturer specific adaptations. Important protocol standards are ISO-9141[27], J1850[16] and the Keyword Protocol 2000 (KWP2000)[38].

2.2 Integration Of Legacy Applications

Focusing on the diagnosis and integration of legacy applications leads to the need for defining what we call "integration of legacy applications". Usually legacy application or sometimes called "legacy code" is meant to be unsupported or unmaintained code. This wording originates from the development of newer platforms where the code should be executed from now on. These newer platforms are introduced, e.g., because they are faster, more reliable or more advanced. The drawback usually is the incompatibility to the old code. In our case the code is neither unsupported nor old but we tried to migrate the code to a completely different architecture and hardware platform. The reason why we call the CAN-applications "legacy-applications" is that a time-triggered system is a more suitable approach for dependable, fault-tolerant and predictable control systems. Several texts suggest to use a TTA for safety-critical systems like drive-by-wire or brake-by-wire [7] [23] [6].

2.3 Intellectual Property (IP)

The approach discussed in this work talks about software integration and migration. The title of our work contains the word "legacy" which leads us directly to this topic. "Legacy Application" implies that the software already exists and it is very probable that this software source code is not owned by the car manufacturer or the data-bus manufacturer. The source code is usually "owned" by a third party firm which means that it is protected by copyrights, patents etc. as listed in [5]. This also means that the car manufacturer who usually is responsible for merging all mechanic and electronic parts to a car has no access to the source code of these applications. Migrating or integrating software, however forces you to have at a detailed description of the interfaces used by adjacent modules or software layers. The standards used in the automotive domain are very often not self-contained but include the freedom for each manufacturer to enrich or adapt these standards. Obtaining these descriptions, including the manufacturer specific adaptations, is somtimes difficult or impossible for externals for business rivalry reasons. It is then also not possible

to develop software for several car manufacturer Because of that, the topic of "intellectual property" is one of the most important issues when talking about software migration and integration.

Sharing source code, signing business cooperation contracts and Non-Disclosure Agreements (NDAs) are not the best solution to protect the IP of a company. The solution proposed, and finally also used in this work is based on layered software and standarized interfaces between these layers. The idea of standardizing all layers used in an automobile is for example developed and pushed by automotive open system architecture (AUTOSAR)[13] since 2003. As we want to replace the network of an existing electronic system a standardized network interface has to be the link between the two systems. The HIS-Standard[1] is the standard that separates the software layers in our work. When strictly adhering to the standard it is possible to develop software layers separately and composing them afterwards. Intellectual Property (IP) is now assured and as the older software parts already match the HIS-Standard no change to the code has to be done to switch the networks.

2.4 Event-Triggered vs Time-Triggered

The decision whether to use an event-triggered or a time-triggered communication is non trivial and mainly depends on the constraints deduced from the system specification. This section is going to introduce the advantages and disadvantages of both communication techniques to get an idea of the strengths of both approaches[18, section 4.4].

2.4.1 Event-Triggered Communication

Event-triggered communication is probably the most widely used communication technique as it is intuitive, more flexible and easier to implement than time-triggered communication. Event-triggered communication can be found not only in the automotive domain but in several other domains like the world of the Personal Computers (PCs). Event-triggered architectures are used in environments where communication needs to be very flexible, should have a low average latency and high bandwidth. The idea in Event-Triggered (ET) communication is that an action or a data transmission is activated on a significant change of state or another input. All other nodes can react on this event appropriately. However these events are not synchronized and one can not predict if two or more messages caused by events will collide on the common communication medium. Depending on the network load each participating

node of the event-triggered architecture is able to use the available bandwidth of the network when needed. During low network load flashing of nodes and retrieval of big chunks of data can be done seamlessly on one network during normal operation.

The disadvantages like increased latency during increased network load are the same as it is known from the Ethernet. A very important representative of the event-triggered paradigm in the automotive world is the Controller Area Network (CAN) [31][12]. The CAN is of special importance for this work because is it the start of our migration process of improving diagnosis. It contains a simple but powerful medium access control mechanism "bit arbitration" implementing a Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA) algorithm. It is in contrast to the Carrier Sense Multiple Access with Collision Detection (CSMA/CD) algorithm which just detects collisions and retries the transmission after some random interval, e.g. Ethernet. The "bit arbitration" described in detail in Section 3.1.1 avoids collisions and prioritises messages to improve the data integrity of important communication channels between nodes.

2.4.2 Time-Triggered Architecture

The Time-Triggered Architecture (TTA) [4][20][18][3] is a network architecture suitable for creating high dependable distributed embedded real-time systems. This architecture has advantages concerning the reliability of the communication while losing some runtime flexibility and throughput of event-triggered approaches. In this section the most important features are summarized to get an idea of the possible application range where you can make use of the TTA and to be able to understand our work.

By using a time-triggered architecture a global time at a given precision is provided and kept synchronized among all nodes. An exact definition of global time and how it works, can be found in [18, chapter 3]. Another very important feature of the TTA is that all nodes judge each other if they operate as expected or not. This property of the TTA enables it to determine whether a node is faulty or not, referring to the TTA's fault hypothesis [4]. It is significant that all nodes have the same rating of all other nodes because this "global view" of the network enables a robust implementation of fail-silent behaviour and a more exact diagnosis of faulty nodes or network connections than in other networks like the standard CAN. If a node is judged to be faulty by the majority of the nodes it can be forced not to disturb the communication of the others for

example using a busguardian. This information about the "global view" is also called the "membership".

The nodes (or often called ECUs in the automotive domain) are connected by a network which is based on a transmission strategy called Time Division Multiple Access (TDMA). In this strategy time is divided into "cluster cycles" which consist of one or more "TDMA-rounds". In each "TDMA-round" every node has its dedicated "round slot" on the common network. In each slot a "frame" is embedded which contains, for example, the payload. It is very important that each node can receive the messages from all other nodes to be able to create the membership. The "TDMA-rounds" can contain different types of information sent by the nodes but the information is repeated every "cluster cycle". The allocation of the slots to the nodes is the same in each "TDMA-round" [20]. It is also possible to switch between several cluster cycle definitions e.g. other messages transmitted, longer round slot for a specific node to gain bandwidth and so on. These different cluster cycles are named "operational modes". The previously described combined information is called "cluster schedule". The common boundary between the host computer and the communication controller within a node is called Communication Network Interface (CNI). In the CNI all messages are stored and read at a priori known instants to cleanly separate the host computer from network communication.

Each ECU maintains[1] three important parameters:

- Operational mode ([20])
- Current global time
- Membership vector

The union of these is called Controller state (C-state). They indicate if (operational mode, membership vector) and when (global time) a node is allowed to send which information. To be able to create a global time and a global view of the network this C-state has to be synchronized with all other nodes. This is done with the "I-frames". I-frames are normal TTP/C frames which don't contain user defined values but the C-state of the sending node so all nodes can get a synchronous time and view of the network.

An example of a cluster schedule is shown in Figure 2.1. Here you can see a cluster cycle consisting of two TDMA rounds. In the first TDMA round each node sends an I-frame and in the second the data is sent.

[1] in the communication controller

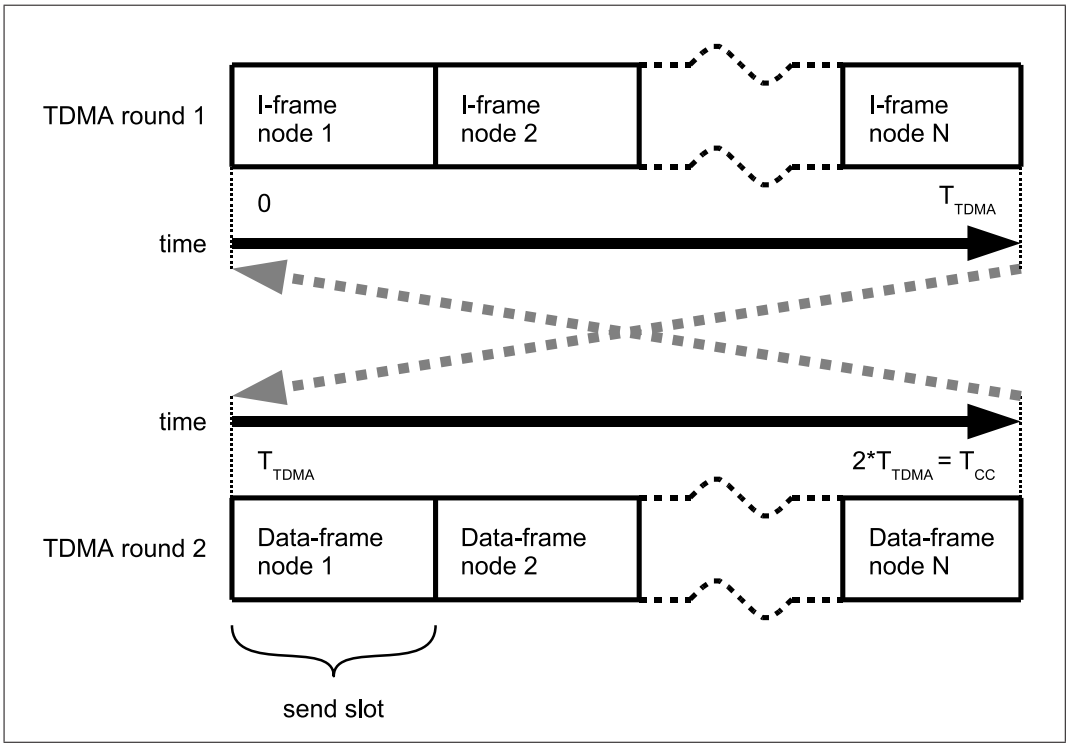

Figure 2.1: A cluster cycle with N nodes and two TDMA rounds

The send and receive instants of all nodes participating in the system are calculated in advance to satisfy all response times and all bandwidth needs of all nodes in the system. These send and receive instants are stored in the "Message Descriptor List" (MEDL) on each node.
The node's tasks are also scheduled in advance, so the whole system can guarantee timely delivery of it's output.

To guarantee that every node participating in this network sends at its assigned time slot, an independent hardware extension, called "bus guardian", is available for each node. At first it just reads the Message Descriptor List (MEDL) and listens to the network communication. The "bus guardian" is now able to connect the node's send-wires to the network only if the node is allowed to send, according to the cluster schedule described above. The bus guardian is either on the node itself or preferably on the replicated star center to maximize failure independence.

While developing a TTA it is important to stick to the "two level design approach"[4, Chapter VI] to guarantee composability. The first design phase is called the "architecture design", the second is the "component design". In

the architecture design phase the interfaces between the nodes of the TTA network are defined exactly and thus also the network interfaces of all nodes. In the second phase, the component design phase, all node specific tasks are completed like the task schedule. The separation of the development into two phases makes it possible for the TTA to offer independent development of the nodes connected to the same bus while offering composablilty [19].

2.5 Gateway

The integration of legacy hardware or legacy software implies the use of gateways. That's why the term "gateway" has to be described in detail within this work. The term gateway has multiple definitions, but we focus mainly on the communication based definition of a gateway.

> *Gateway: (n) A network participant that provides automated interfaces to another network or system using different message formats and/or communications protocols. A gateway may contain protocol translators (for message standards), and signal or medium translators, as necessary, to provide system interoperability. Gateways require mutually acceptable administrative procedures to handle gateway functions between networks, such as information management rules, command and control rules, and reporting responsibility rules.*
>
> Gateway definition from [9] inspired by [10]

The protocol translator of a gateway usually has some criteria whether to allow a package of information to pass through or not. This "selective redirection" is often simply called "filter". Medium translators can range from only a different physical layer, e.g. Ethernet over a copper wire or over fiber optic cable, up to different application layers, e.g. a software emulating a POP3 account while reading mail from a web-mail application.
A gateway is usually a network node equipped for interfacing with two networks that use different protocols. To establish this connection a gateway may consist of protocol translators, impedance matching devices, rate converters, fault isolators, or signal translators.
The last important feature a gateway can have, and which has to be mentioned in this work, is the "fault isolation". For example if in one network a physical damage leads to too high voltage on the communication medium and blocks all communication, the network on the other side of the gateway should not be bothered by this fault.

2.6 Two Level Design

The two level design approach [4, Chapter VI] is an easy way of seperating network protocol design parameters from node development. This strict separation enables independant node development while the composability of the nodes is guaranteed.
The two levels are

- cluster design and
- node design.

The cluster design incorporates specification of the number of TDMA rounds contained in a cluster cycle, TDMA round length, messages available on the bus, number of nodes connected to the bus, linkage which message is sent by which node, and some more. The output of the cluster design is called "cluster schedule". With this cluster schedule the communication between all nodes in this cluster is defined and the node design can be started. Considerations what the cluster has to control and what bandwidth is needed for this cluster to work are also done in this design level.

In the node design, according to the messages a node has to send on the bus, the tasks which calculate or process the messages on the bus are scheduled. The timestamps of the access to sensors and other inputs and outputs can also be determined because of thoughts which are the preliminaries to programming the tasks. For example the ABS ECU is going to use values like the speed of the car and the speed of the wheels to calculate the brake force, or directly control the brake force with an attached actuator. These definitions can be fixed in the very early design and the implementation of the node's tasks can then be done independently.

2.7 TTP-OS

TTP-OS is a light weight operating system developed by TTTech specifically designed for fault-tolerant real-time applications. It provides time-triggered pre-emptive task scheduling and is compatible with OSEKtime[14]. On lowest thread priority the so called 'background hook' is running. It is not recommended to run tasks inside the background hook because it's not guaranteed that the background hook is called and it's not determined how often it is called. During development it is sometimes easier to use the background hook

for testing or debugging purposes as this is the only repeated task which does not need scheduling in advance. Running event-triggered applications in the background hook is possible but not recommended. More on that in chapter 4.

All tasks are scheduled in advance so each task has its predefined time slot where it is allowed to run in. A task execution which takes longer than its reserved time slot provokes a deadline violation. A deadline violation usually causes a node to restart and continue normal operation. This mechanism provided by the TTP-OS prevents tasks from blocking other tasks or the whole node. This functionality is very useful during development and will not occur during runtime if the tasks are correctly implemented.

3 System Model

In the following sections all parts of the system model are described in detail. All features of a node described by our system model are visualised in Figure 3.1. The possible interactions for the tasks with the outside world and the connecting parts of the figure are described in the next few paragraphs. The dark grey layers of the figure are hardware parts while the white layers are software parts. The light grey layers describe interfaces (see Section 2.4.2) implemented by shared memory.

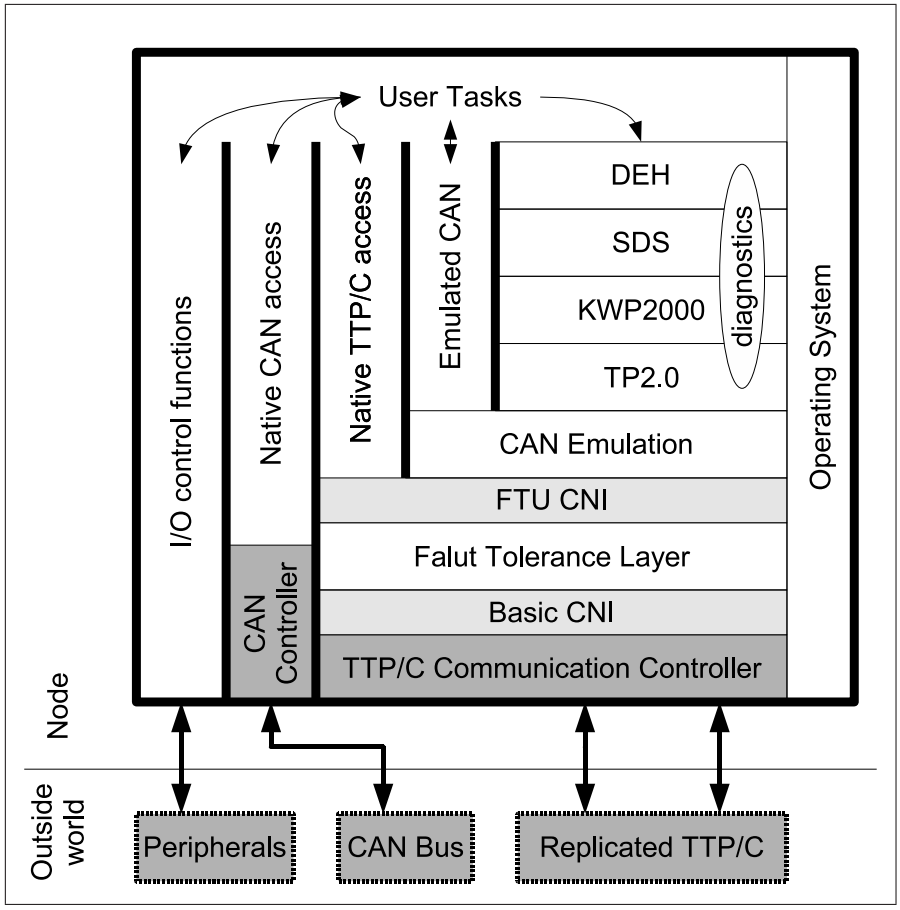

Figure 3.1: I/O interfaces of a node in our model and the associated layers

On the most left side of Figure 3.1 there are the *I/O control functions* and the interconnected peripheral. These functions are controlled directly by the application or some with a small driver simplifying the access. Some possible examples of these I/O functionalities are:

- Light Emitting Diodes (LEDs)
- Electric motor(s)
- Serial communication
- LCD display

Our system model needs to have a CAN controller for *native CAN access* to the outside world. This will be very important later for the creation of gateways between the event-triggered CAN world and the time-triggered TTP/C world. A standard CAN driver by TTTech is available for this hardware and no detailed description is needed for our system model. General description of event-triggered communication can be found in Section 2.4.1.

Native TTP/C access can be used for all sorts of communication which needs to be fail operational. The interaction with the Fault Tolerant Unit (FTU)-CNI is conceptually simple and needs no explanation here. Details can be found in Section 2.4.2.

Emulated CAN is any communication originally developed for a CAN network. It is tunnelled through TTP/C to use features like dependability and media redundancy. See Section 3.1.3 for more on this part.

The stack marked with the *diagnostics* bubble is a special, layered software originally developed for CAN diagnosis. It can be ported within our model to run on TTP/C. Details on this software can be found in the Sections 3.1.4 and following.

The most right part, the operating system, provides the basics to run our application. It is discussed in detail in Section 2.7.

3.1 Architectures And Protocols

There are several protocols which have to be mentioned and described in connection with this work. They are fundamentals when working in the automotive domain focusing on software by Audi and TTAs respectively. To understand the protocols better keep Figure 3.1 in mind while reading on. It is also important that in the automotive domain a node of a network is usually called ECU which is going to be the term used in this work, too.

		Node 1 sends	
		0	1
Node 2 sends	0	**0**	**0**
	1	**0**	**1**

On a CAN network 0 is the dominant bit while 1 is the recessive bit. The bold printed bits are the received bits if node 1 and node 2 transmit their bit at the same instant

Figure 3.2: Dominant and recessive bits on a CAN network

3.1.1 Controller Area Network (CAN)

The Controller Area Network (CAN) is a field bus system which is heavily used in the automotive industry to reduce the complexity of the cabling in a car. It is also used in automation industry in various fields.

CAN is a lightweight protocol to be able to create very small, robust and cheap ECUs communicating with each other. CAN is usually built upon a bus topology but other topologies, like a star, are possible, too. The medium is accessed using the CSMA/CA strategy. To avoid collisions, a technique is used which is called "bit arbitration" which is described later. The bits are transferred using the Non Return to Zero (NRZ) line code. CAN offers a wide range of transmission rates from 10kBd up to 1MBd. Usual transmission rates are 125kBd and 500kBd in the automotive domain.

There are two standards called "CAN 2.0A" and "CAN 2.0B". The main difference is that "CAN 2.0B" has an Message identification number of 29bits in contrast to the 11bits used in "CAN 2.0A". In this work only "CAN 2.0A" (also called "standard CAN"[1]) is needed.

Bit Arbitration is the technique to avoid collisions and respect message priorities. It relies on the fact that all CAN messages have a unique Identifier (ID) defining the priority, the content and the sender of the message distinctly. It is also very important that the ID is sent first (see Figure 3.3). When using NRZ it is possible to detect collisions and implicitly define priorities because the bit value "0" is dominant and overrules the bit value "1" which is recessive (see Figure 3.2). A message with a low identifier has higher priority and blocks the bus for all messages with a higher identifier which are tried to be sent at the same instant. With bit arbitration collisions are detected by the sender which tries to transmit the message with the lower priority.

[1] CAN 2.0A is called "standard"-CAN; CAN 2.0B is called "extended"-CAN

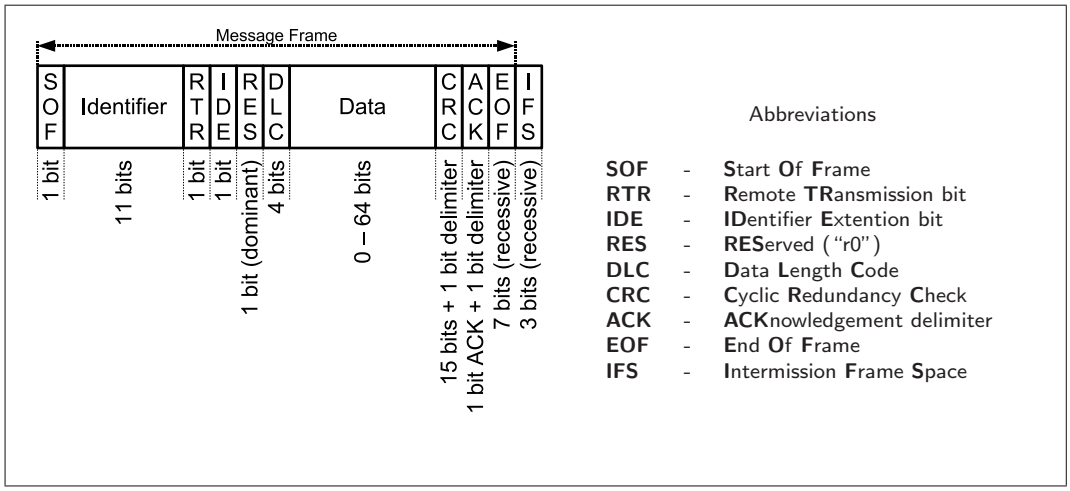

Figure 3.3: The standard CAN-Message Frame

A standard CAN frame is assembled as shown in Figure 3.3. The length of the "Data" block in a CAN message can be 0 to 64 bits in steps of 1 byte. The length is stored in the Data Length Code (DLC) which only needs to be 4bits wide to cover the value range 0-8 bytes. The Acknowledge (ACK) bit is set to confirm the reception of a CAN Frame or to signal a damaged frame. If more than one ECU acknowledges a frame at the same instant a dominant failure-indicating ACK bit (0) overrules a recessive flawless reception indication (1).

A closer description of the CAN protocol can be found in [24] while the standard for the CAN protocol is defined in ISO 11898 ([31], [12]).

3.1.2 Time-Triggered Protocol Class C (TTP/C)

The Time-Triggered Protocol Class C (TTP/C) is the advanced time-triggered protocol, in contrast to the more simple version TTP/A which is used for low data rates and simple sensors sending measure values only. TTP/C should operate on two replicated communication channels to guarantee perfect communication even if one channel fails. These channels are called "channel A" and "channel B" further on. It is possible to use only one channel but not useful for safety-critical systems. As the name TTP/C already reveals, the protocol is time-triggered and uses a cyclic transmission schedule as medium access strategy (TDMA). This avoids collisions on the medium, the timeliness of messages and the bandwidth can be guaranteed. The bus speeds for asynchronous communication available for TTP/C range from 500kBit/s up to

5MBit/s to provide enough bandwidth for all applications. Faster bus speeds are already in development. The specification of the current TTP/C can be found at the homepage of the TTA-Group[15].

The membership is a basic feature of TTP/C - see [18, chap6.4]. It is the implementation of the global state of the TTA every node is able to access. The membership indicates whether a node is valid or not. This validity contains communication failures or other node failures detectable e.g. in the CRC of the protocol. With this membership every node has the same judgement of the other nodes which enables fail-silent behaviour and can be used in diagnostics easily. Each node having the same view of communication failures also helps pinpointing a failure. Besides the membership a program can get a status information of each frame received on each channel. This status information for each single received frame is named "frame status".

The TTP/C we use here is shipped with replica deterministic algorithms ([18, chap5.6]) implemented in a layer called "Fault-Tolerant Communication Layer (FT-COM)". This ensures that if a function is replicated on more than one node, for safety reasons, all other nodes that read the output of these nodes (which might be different) use the same combined value for their further calculations. Values can be different because of measurement errors or because of a faulty node.

Two additional features increase the reliability of communication in a TTP/C network. The bus guardian is an independent hardware which enables the sending wires of the communication controller only in its predefined sending slot. This avoids an erroneous communication controller from disturbing regular communication. This is called fail silent behaviour of a Fault Containment Unit (FCU). The second feature is the life-sign which the Central Processing Unit (CPU) has to send the communication controller periodically to indicate that it is still working. A blocked or crashed CPU causes the communication controller to go to passive mode (stops sending TTP/C frames) which indicates a malfunction to all other nodes.

Topologies

The recommended topologies for TTAs are

- bus topology and
- star topology

The advantage of the bus topology is that you need less wires and money as not all ECUs need to be directly connected to a "center", although the communication is still safe. The advantage of the star topology is that here the bus guardians are collected in the star center and thus located in some distance to the ECU it self. It should be used for ultra dependable systems. As the communication channels of a TTP/C bus is replicated the star centers are also replicated not to form a single point of failure. In our work the bus topology is preferred because there is no need for a separated star center.

3.1.3 CAN Emulator

The idea of the CAN Emulator is to provide an interface for legacy CAN applications to run them on a TTP/C cluster. It is our LIF between the TTA and a legacy CAN application. One of the biggest problem here is that CAN applications usually are event-triggered. Riezler writes in his Diploma thesis [36] about this problem of writing a CAN driver for event-triggered software based on a TTA. Many approaches to tunnel or combine ET traffic with a TT network already exist because of the different features these medium access methods provide [25]. However none of them have a CAN driver interface to easily port legacy CAN applications into a TTA.

As the CAN Emulator is a time-triggered application while legacy CAN applications are not Riezler developed three possibilities to schedule the event-triggered CAN application on the TTA. The options to run a legacy CAN application on a TTA with the CAN Emulator are to

- schedule the CAN application tasks in separate new time-triggered task. This is extremely simple for the CAN Emulator as no a access conflicts can arise because all tasks run serialized.
- execute the CAN application tasks in the background hook[2]
- execute the CAN application tasks in hardware or software interrupts

[2] see Section 2.7

While the user is able to choose between these three options, scheduling the legacy CAN application in a separate time-triggered task is by far the best choice when developing a TTA. Then timeliness is guaranteed and the hard deadlines still hold. (see [2] and Section 2.4.2)

The other options the CAN Emulator offers to the programmer are only for the ease of porting and the reduction of costs while migrating any event-triggered legacy application into a non-fail-safe TTA. They are not mentioned in this work any longer.

In Figure 3.1 you can see that the CAN Emulator uses the fault tolerant layer to communicate via TTP/C so all features concerning reliability and delivery guarantees can be derived from the TTA. When the buffers and bandwidth is configured accordingly even the transmission duration can be determined independent of the network load.

3.1.4 TP2.0

As the CAN Bus is only able to transmit 8 bytes in one CAN-Packet the network layer has to segment the data for the CAN network if more data needs to be sent in a row. This segmentation, which is standardized in ISO 15765-2[33], is harmonized with the segmentation called Unacknowledged Segmented Data Transfer (USDT). USDT is defined in the OSEK-COM specification[14]. Besides these two standards, Transport Protocol 2.0 (TP2.0)[3] is an extended and proprietary version of the ISO 15765-2. This modified version (TP2.0) is a proprietary protocol made by Audi and VW. A difference of TP2.0 to the standard is the extension that the protocol has an acknowledgment and an error recovery using a retry mechanism. The abbreviation ASDT for "Acknowledged Segmented Data Transfer" is sometimes used to point out the acknowledgment feature of TP2.0.

TP2.0 is a master-slave communication protocol capable of sending connectionless messages and establishing a communication channel between the master and the slave ECU to transfer data. The reception of each packet or a block of packages has to be acknowledged to guarantee delivery. It is also possible to interrupt running data transfers.

[3] TP2.0 is also called VWTP for "Volkswagen Transport Protocol"

The protocol also introduces timeouts which have to be met while answering to a request. Several timeouts and retry times are defined in TP2.0 while for this work it is only interesting that a TP2.0 master usually gets an answer in about 50ms on a normal CAN bus and the timeout value for an answer is 500 ms. These values are very large so they will not be violated with our model (more on this in Section 5.5).

To be more flexible with TP2.0 it is possible to create dynamic channels using two CAN-identifier groups called:

- opening identifier
  used for connectionless communication (broadcast messages) and to establish a dynamic communication channel between two ECUs.
- channel identifier
  dynamic CAN-identifier used for connection setup, data transfer, acknowledgment etc.

Connectionless messages are distinguished by the opcode (see Figure 3.4). With them it is possible to initiate the following actions and receive an answer if needed.

- clear the DTC-memory of an ECU
- cause the ECU not to send any CAN message until the master sends a message to resume normal mode
- the button test. ECUs which have buttons connected to them store the information if every button was pressed after the start of the button test. The result can be read out by the "state information retrieval" used in Chapter 5
- enter energy saving mode
- initiate a dynamic channel

Each ECU in a CAN network using TP2.0 has a unique "one-byte TP2.0 identification address", in this chapter called "destination". The opening CAN-identifier for each ECU is fixed and calculated as

```
opening CAN-identifier = <offset> + <destination>
```

The offset is a manufacturer specific constant. As the CAN-identifier is also the priority of the message it is possible to set the priority of the TP2.0 messages in the system with the offset.

The TP2.0 general message structure on the opening identifier using 7 bytes payload of a CAN frame.

CAN-byte	#1	#2	#3	#4	#5	#6	#7
TP2.0	Destination	Opcode	Parameters				

Figure 3.4: TP2.0 - General Message Structure

The channel identifier is defined dynamically by the participating ECUs.

Dynamic Channel Setup

In this work the dynamic channel of TP2.0 is very important because KWP2000 uses this channel to transfer data from the diagnostic tester to the ECU and backwards.

To initiate a dynamic channel the master sends a channel setup message using its opening identifier (Figure 3.4). One of the parameters is the CAN-id the master is going to use for sending packets to the slave (channel identifier). The slave confirms this request on its opening identifier and tells the master which CAN-id the slave is going to use as channel identifier. Once this information is exchanged master and slave switch to their channel identifier and continue communication there (Figure 3.5). On the channel identifier the "connection setup" is used to acknowledge the CAN-ids used as channel identifier and the "block size". The block size is the number of data packets after which the slave needs to send an acknowledge to the master.

A usual dynamic channel setup without errors is depicted in Figure 3.6.

An important TP2.0 message on the channel identifier: The Data Message can contain up to 7 Data bytes

CAN-byte	#1	#2	#3	#4	#5	#6	#7	#8
TP2.0	TPCI[a]	D / -	D / -	D / -	D / -	D / -	D / -	D / -

[a] Transport Protocol Control Information Byte - contains

- Acknowledge request flag
- end of message indicator flag
- sequence number

Figure 3.5: TP2.0 - Data Message Structure

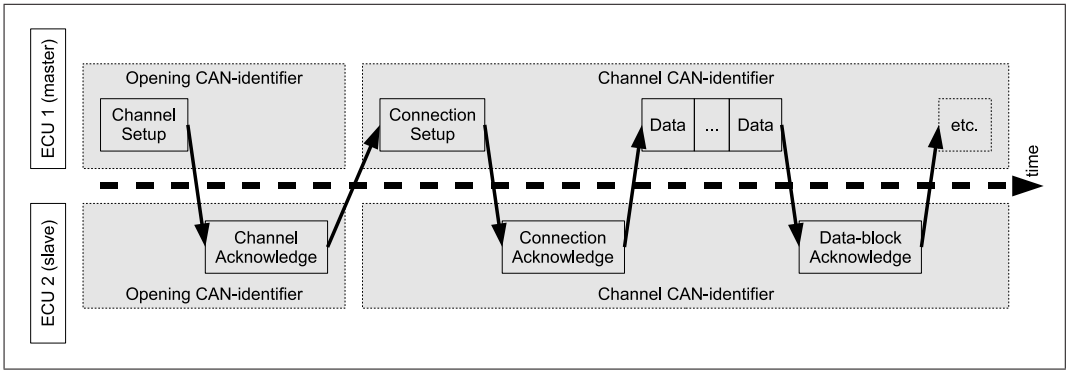

Figure 3.6: TP2.0 - Dynamic Channel Setup

3.1.5 SDS

SDS is the abbreviation for "**S**tandard **D**iagnosis **S**ervices" whereas the according standard (ISO 14229) is called Unified Diagnostic Services (UDS)[4] in the latest revision [30]. The Abbreviation Standard Diagnostic Services (SDS) is used in this document because it is used by Audi in all libraries and sources, which deal with this standard and its subordinate standard ISO 14230.

The ISO 14229 specifies generic diagnosis services and services which allow the diagnostic tester to control non-diagnostic message transmissions. In this standard the services and their parameters are specified but not the numerical values of the service identifiers and the values of the parameters. The specific values are defined in ISO 14230 and in manufacturer specific documents.

[4] in the first editions in 1998 the ISO14229 was named "Diagnostic services specification".

The services specified in the standard are subdivided into the following six units:

- **Diagnostic management functional unit**
  contains functions for controlling diagnostic sessions, security and reading ECU identification.
- **Data transmission functional unit**
  contains functions for retrieving data like sensor values or internal variables, and functions for sending data to the ECU to set transmission related parameters. For example maximum number of responses to send, in case of an answer with multiple responses.
- **Stored data transmission functional unit**
  contains functions to retrieve the DTCs and freeze frame data stored in the ECU.
- **Input/Output Control functional unit**
  contains functions to control peripherals of the ECU with the diagnostic tester.
- **Remote activation of routine functional unit**
  contains functions to initiate remote function calls and retrieve their results.
- **Upload/Download functional unit**
  contains functions to download and upload any kind of data.

The specification of these services is the basis of the Keyword Protocol 2000 (KWP2000).

3.1.6 KWP2000

Keyword Protocol 2000 (KWP2000) [28][29] is a master/slave protocol providing functionality for diagnosis information retrieval, updating ECU firmware and special development functions. It uses a request - answer scheme to poll data or acknowledge the execution of a requested function. The services for

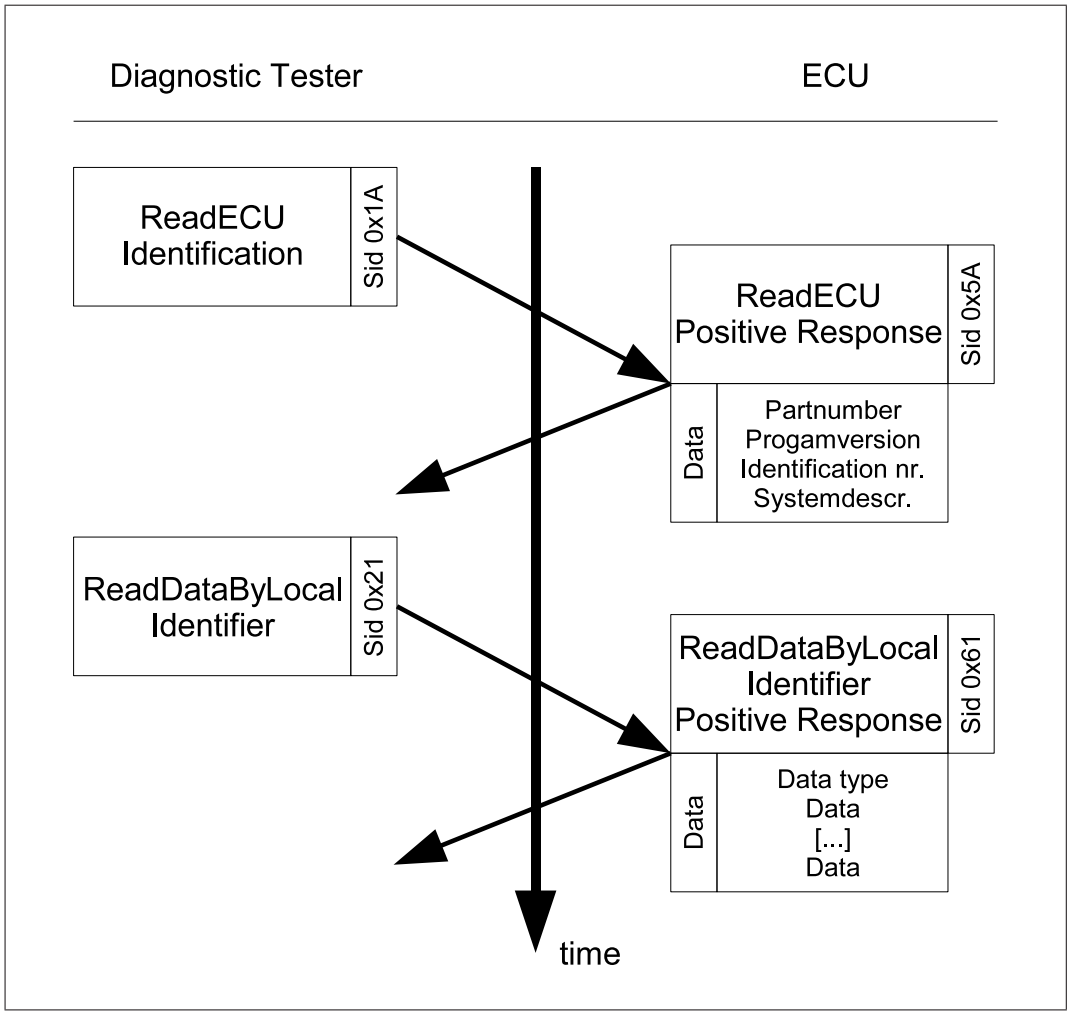

Figure 3.7: KWP2000 flow diagram

this protocol are defined in ISO 14229 described in Section 3.1.5.

The Keyword Protocol 2000 (KWP2000) is defined in the standard ISO 14230 called "Road vehicles – Diagnostic systems – Keyword Protocol 2000" which is separated into four parts:

ISO 14230-1 Physical layer

ISO 14230-2 Data link layer

ISO 14230-3 Application layer

ISO 14230-4 Requirements for emission-related systems

This standard assumes that the communication is done via the "K-Line"[5] which is a single cable serial communication bus. In newer cars the KWP2000

[5] optional an "L-Line" in addition to the "K-Line" is possible too, which is used for initialization only

protocol is on top of CAN instead of the "K-Line". Here, roughly said, the physical layer is replaced by CAN (with TP2.0). That's why in this paper only Part 2[28] and Part 3[29] of this standard are relevant.

The basic communication protocol can be depicted as in Figure 3.7. It shows the diagnostic tester (master) sending requests with a special "keyword" (=Service Identification byte (SId)) and the ECU (slave) responding to it. Some services, like updating the program flash, may also initiate a session and require more than one request and its (positive or negative) answer.

ISO 14230 Part 2: Data Link Layer

This part of the ISO 14230 standard contains the message structure (Figure 3.8) and basic services which are needed for communication (all other services are defined in ISO 14230-3).

The message structure is depicted in Figure 3.8 and uses the following abbreviations:

Fmt Format byte
Defines the format of the header. Tgt, Src and Len are optional. The length of the payload is stored here, too, in case of a payload smaller than 64 bytes. If the header contains the length byte, the length is stored there.

Tgt Target address byte
The virtual KWP address of the target (receiver).

Src Source address byte
The virtual KWP address of the source (sender).

Len Length byte
The length is stored here if the source and target addresses are used.

SId Service Identification byte
Specifies the semantic of the contents stored in the current message.

The KWP2000 message header always includes the format byte which controls the existence of the following header bytes. The Service Identification byte (SId) specifies the requested service. It may be followed by parameters and data depending on the selected service. The services and their parameters are defined in ISO 14229. The specific values of the service identifiers are

Header				Data bytes		Checksum
Fmt	Tgt[a]	Src[a]	Len[a]	Sld	Data	CS
max. 4 byte				max. 255 byte		1 byte

[a]optional, depending on format byte

Figure 3.8: KWP2000 - Basic Message Structure

defined in ISO 14230-3 for diagnostic services and in ISO 14230-2 for communication related services. Some SId ranges in the standard are marked to be defined by manufacturer specific documents.

Communication related services specified in ISO 14230-2 are

- StartCommunication
- StopCommunication
- AccessTimingParameter

The first two services contact a specific ECU to start or stop a KWP2000 communication. The third service listed here is able to retrieve or change communication timing or timeout parameters.

ISO 14230 Part 3: Application Layer

Part 3 contains the ISO OSI layer 7 including

- byte-encoding and hexadecimal values for the service identifiers
- byte-encoding for the parameters of the diagnostic service requests and responses
- hexadecimal values for the standard parameters

The services which have to be or can be implemented without specific values of the identifiers are standardized in ISO 14229.

3.1.7 DEH

The "Diagnostic Event Handler" is a software layer, in this form provided by Audi, which does the handling of the background information needed for the SDS/KWP2000 layers. It contains the failure memory (DTCs) and also handles the entries of this memory. There is a customizable form of debouncing included in this layer so the application only has to provide some state variables and the rest of the diagnosis is up to this layer.

An error can occur

- once
- several times, including arbitrary intervals of correct behavior (sometimes called sporadic occurrence)
- continuously or
- never.

Dependent on the interval between the sporadic occurrences of an error it might be useful not to report this error if it occurs rarely. Dependent on the type of error and the usecase the programmer can decide for each error separately how often or how fast an error may occur before reported. This behavior is called debouncing.

This is one of the most important functionality of this layer. The Diagnostic Event Handler (DEH) layer is also able to put an error to a passive state if it did not occur for a specific period of time. This is also part of the debouncing.

The DEH layer also handles the freeze frame table (Section 2.1). Each error can be configured whether the DEH layer should store a freeze frame containing the current state of the car (eg. mileage) at the instant of the occurrence.

3.2 Model Features

Our work focuses the enrichment of the commonly used event-based CAN applications with safety advantages of the time-triggered protocol TTP/C. To replace the CAN with an emulated CAN running on top of a TTA two common interfaces are needed to be able to reuse existing hardware and software. To reuse existing CAN software a CAN Emulation layer is needed and to reuse

existing hardware a CAN gateway is needed to connect the legacy hardware with the TTP/C network. A dependable network would be provided for any of the safety-critical applications one would want to have in a future car[11][17]. The possible features one could be able to use in the future with a TTA installed in a car, range from steer-by-wire over assisted braking towards complete drive-by-wire or automated driving.

Our model can be interpreted as simply two emulation concepts. They are explained in detail in the next sections. The model expands the dependability of this distributed control system. It can be abstracted in just one cloud like in Figure 3.9. This figure shows an abstraction of the model. This simple figure which is compatible with CAN on software and on hardware level through LIFs[21] can be extended to get a more detailed view on our model. This extended model is depicted in Figure 3.10. Here you can see that the model is compatible to the legacy CAN software and the CAN hardware. The most interesting point, however, is that CAN diagnosis algorithms, using the software layers described in the last sections, still work with CAN hardware and CAN software. Because of the fact that the whole system is now a TTA even some diagnostic advantages, like the membership (see 4.5), can be utilised with common diagnosis tools. With this model it is possible to get diagnosis information from a TTP-Node using CAN diagnosis hardware. In Figure 3.10 the TTP bus is of course meant to be a replicated communication system on two separated bus lines. These two lines are not depicted here for the sake of simplicity.

The model supports IP because of the HIS CAN driver interface for legacy software. With this clear interface development of the ECU hardware, the operating system and the communication driver can be totally separated from the actual ECU functionality and this third party software can be linked into the ECU easily.

An idea of a possible diagnosis improvement which can be realized with this model is the membership service of TTP/C. With the membership it can be determined which node is not synchronized or connected to the network, without the need for a special test procedure like querying all ECUs separately and waiting for an answer from each of them. More on that in Chapter 4.

3.2.1 Hardware Emulation

In this model hardware emulation refers to the creation of a physical CAN interface compatible to ISO 11898 [32]. With this interface it is possible to

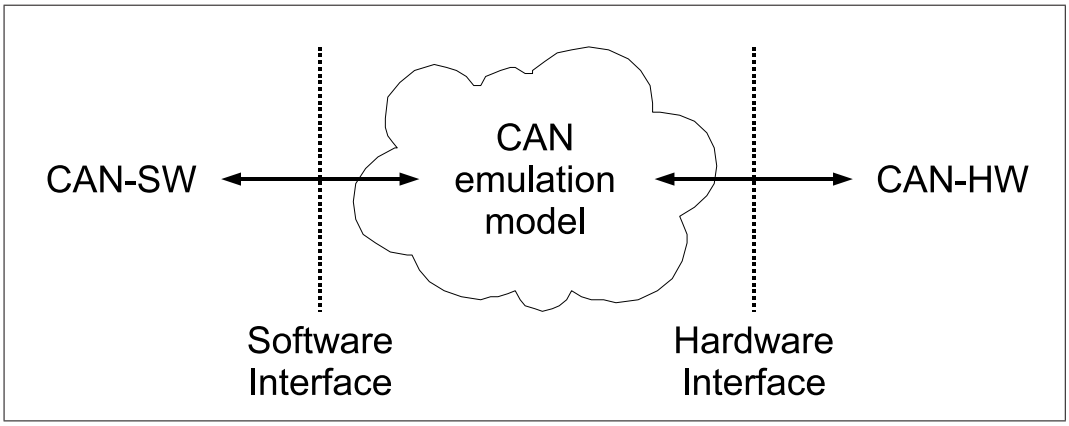

Figure 3.9: Simple abstration Model

connect legacy CAN hardware to the new network. It is very important and economically advantageous to be compatible with CAN hardware as it will not be necessary to replace all ECUs with a TTA. This hardware compatibility enables the reuse of CAN ECUs and the garage tester hardware, too. The hardware emulation is realized in this model by implementing a gateway between the TTP/C and the CAN protocol.

Gateway In our case the gateway has two important features. It is a

- protocol translator and a
- fault isolator.

The translation needed here should read CAN frames from a physical CAN-bus and transfer them onto the TTP/C Bus. The Application Programming Interface (API) of the CAN Emulator can be used here, so the gateway software only has to translate the CAN packages from the CAN Emulator to the hardware CAN driver which is just another CAN driver concerning the API.

By the fact that this gateway is a self-contained ECU faults are isolated. The second reason why this gateway is a fault isolator is that in a TTA error frames are not needed and therefore not forwarded by the gateway. In some cases error frames can disturb CAN communication[34] but not TTP/C communication with this gateway.

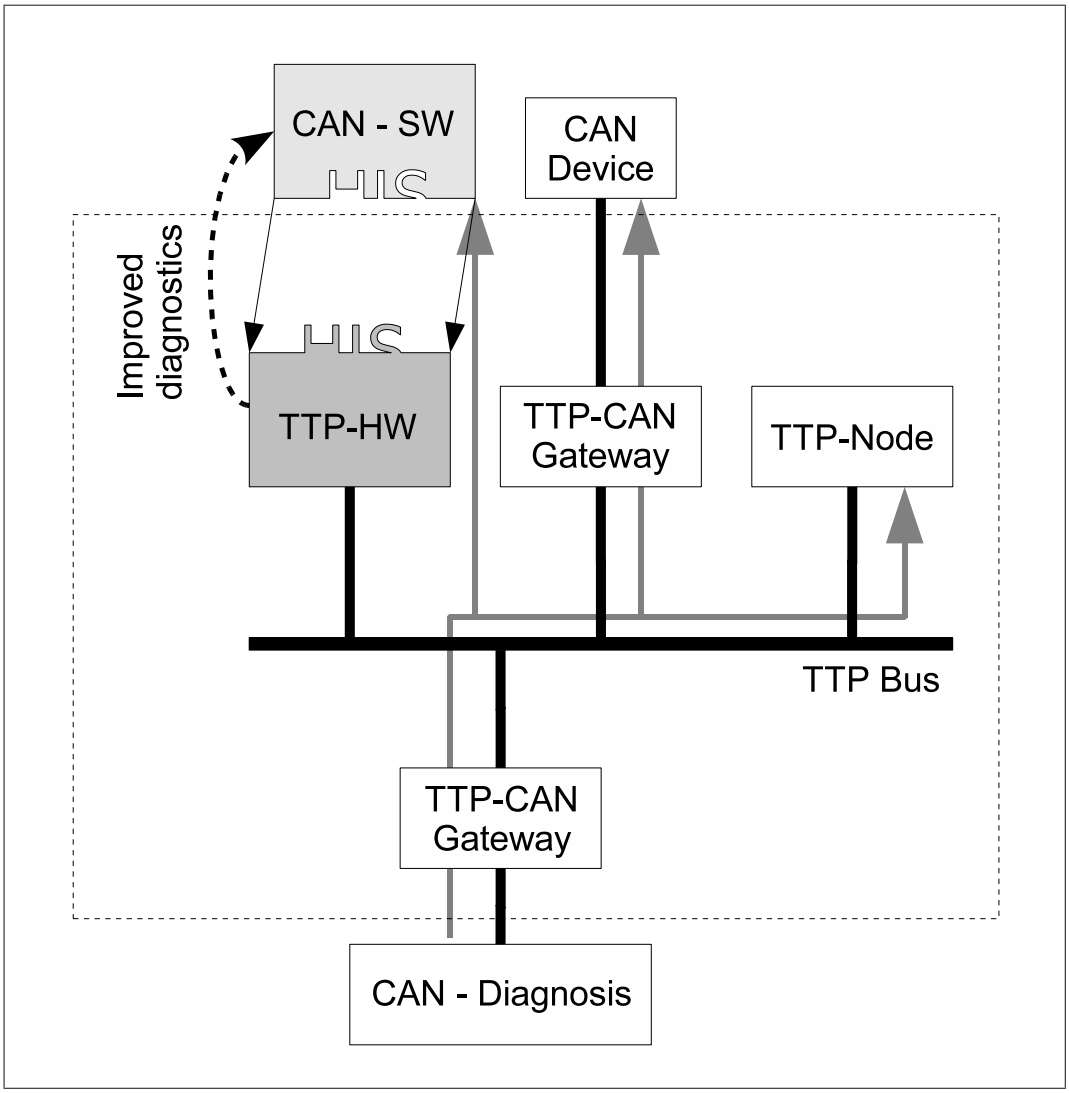

Figure 3.10: Model using dependable a Network

3.2.2 Software Emulation

In Figure 3.10 the dashed box separates the components (including software and hardware parts) of the time-triggered architecture from the parts of the event-triggered world (or here the CAN world). The interconnections between the two worlds are on the one hand the CAN connections depicted by black lines connected to the CAN-Device and the CAN-Diagnosis unit, on the other hand there is the HIS interface. With the HIS interface it is possible to reuse a CAN legacy software in the TTA.

This feature of the model is called software emulation because this interface emulates a CAN network for the migrated software. It abstracts the communication network and the fact that the software is running on a time-triggered architecture. Using this HIS interface event-triggered software including all algorithms and communication procedures need no adaptation to the new communication hardware. As shown in Figure 3.10, additionally you can easily benefit from the fact that the software is now running on a TTA. It is described in Section 3.1.2 and Section 4.5 that it is possible to extract useful information like the "membership" from the network and use it to improve diagnostics. This additional information about the state of the network can be gathered easily by legacy CAN-Diagnosis units. The grey arrows visualise the path of the diagnosis information from either a legacy software running on the TTA, a legacy hardware device connected through a gateway or a native TTP/C node to the CAN-Diagnosis unit.

4 Implementation

Investigating on the feasibility of diagnosis of CAN-based legacy applications in the TTA presumes a real legacy application to work with. In cooperation with TTTech it was possible to get an application from Audi which perfectly meets all demands for a prototype usable in this work. It is a software containing CAN communication, the diagnosis stack for the diagnostic tester and the DEH layer. In the following sections we describe a prototype containing this legacy software with extended diagnostic features.

At first the legacy application delivered by Audi was analysed to ensure all operating system and driver calls are satisfied in the new TTP environment. We created a prototype setup for the integration of the previously analysed legacy application. The CAN emulation layer was configured and compiled with the prototype source code. Afterwards the TTA schedule was to be created. The CAN emulation layer has a big impact on the network schedule because no other TTP/C messages are needed in our cluster. The TTA was created with the two level design approach as already described in Section 2.6. At that point the basis to add new diagnostic features was completed.

The prototype setup is built up out of three nodes to show the different interfaces of the model. It has four requirements listed below.

1. Demonstrating the reasonable small adaptation effort which is needed to port a CAN based legacy application to run on the TTA using the CAN Emulation layer.
2. Showing the hardware compatibility to CAN[1] based hardware. This is one of the most important requirement because of the economic need of car manufacturers to reuse existing garage hardware to save costs.
3. Testing and demonstrating the basic functionality of the CAN Emulation layer by tunnelling CAN Messages sent by the Schaltermodul Lenksäule (ger.) (SMLS)[2] through the TTP/C bus and visualizing them.
4. The gain of diagnostic precision which can be easily used even with hardware and software designed only for CAN. A nice example is the diagnostic tester in our prototype setup which is able to use features of the TTP/C network while not knowing what TTP/C is.

[1] according to ISO 11898[31]

[2] The SMLS is the ECU on the steering column including all levers and buttons like the right and left turn indicator lever

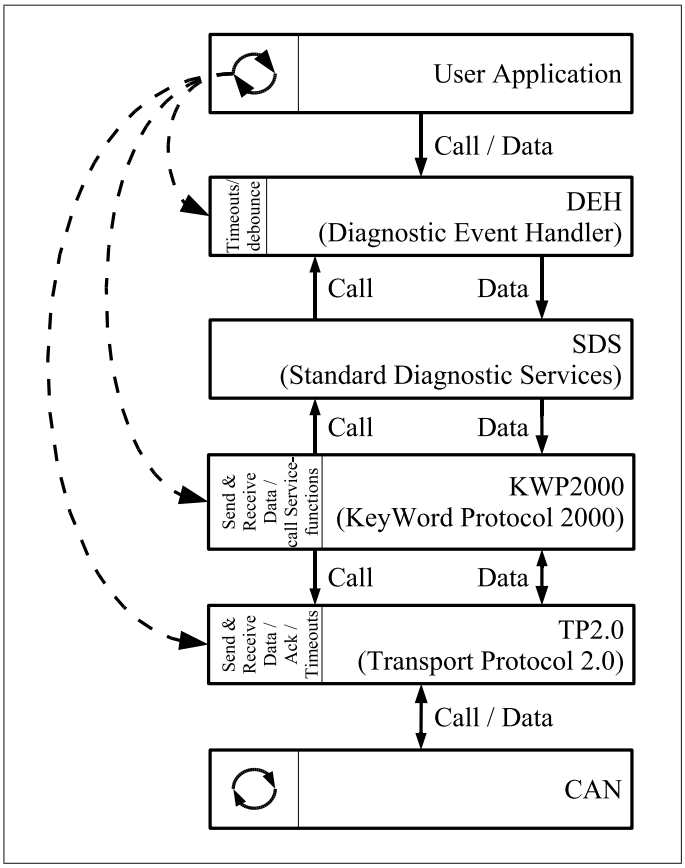

Figure 4.1: Call graph of the demo application provided by Audi.

4.1 Legacy Application

To show the integration of a legacy application a source code was needed preferably directly from a car manufacturer, to get a realistic view of the adaptation effort needed to migrate from an event-trigged system to a time-triggered one. Audi provided a demonstration code of the diagnosis stack including all important software layers and functionality to communicate with the diagnostic tester. This software will be called “original code” henceforth, while the final version of our prototype code will be called “demo application”. Audi made the original code available for this project for research purposes only. After the analysis of the original code we concluded Figure 4.1 to be the call graph of the software. In the following this figure will be described in detail from the top to the bottom. The modified legacy application was adapted to emulate the ECU ’Parking Aid’.

4.1.1 User Application Layer

In Figure 4.1 on the top you can see the user application layer with the cyclic calls of the main loop executing the lower layers. This call graph originates from the intended use of the original application which is a usual Event-Triggered (ET) CAN-ECU (in contrast to the TTA used here). The user application layer contains the main ECU functionality like a control logic or code for operating sensors and actuators. Since Audi provided the code relevant for diagnosis this application layer only contained the loop executing the lower software layers.

4.1.2 DEH Layer

The DEH layer is an Audi specific software which stores errors and does debouncing on the errors signalled by the application.

The DEH layer is part of the application layer of the OSI model but separated from the user application layer in Figure 4.1 because it is generic code used in many Audi ECUs. Each detectable error or important incidence is mapped to a Diagnostic Trouble Code (DTC) which is also called 'event' in the code. Inside the DEH layer these events can be configured to determine when and how the error should be remembered and reported. The querying of these events is then made with the diagnostic tester in the garage. An event configured in this layer can have the following parameters:

DTC Standardised diagnostic trouble code which this event is mapped to. This value is referenced directly by the diagnostic tester in the garage.

EventParameter With this parameter the so called debouncing can be configured. This parameter has two possible values: 'time debounced' or 'event debounced'. If an event is time debounced it only gets active if the event is present during the specified time period (see below). If an event is debounced it only gets active if the error is reported more than a specified number of times (see below).

EventSymPrio Configures how the event is displayed on the diagnostic tester.

QualEvent This value is either the time span in seconds or the number of repetitions an event has to be present to be reported.

DeQualEvent This value is either the time span in seconds or the number of repetitions an event has to be absent after being present for the length of 'QualEvent' to be reported as a sporadic event.

LampParameter If a lamp is directly connected to the ECU, it can be lit with this parameter in case of an error.

IndexFFrameTable The 'Freeze Frame' referred here is a snapshot of the current values of a set of predefined parameters. If a valid index is defined here the whole set of predefined parameter values is stored when the event occurs. These values can be involved in the search for the cause of the event. Typical parameters are the current speed of the vehicle, the mileage or the date and time, all mesured at the point in time when the event occurred.

A code snippet of such a DTC configuration is shown in Figure 4.2. The Application solely has to tell the DEH layer if the event occurs or not. This is done with global variables as shown in Figure 4.11.

4.1.3 SDS Layer

The SDS layer contains the functionality specified in ISO14229[30]. These are services an ECU has to provide to be able to communicate with the diagnostic tester. In ISO14229 only the services are defined not the underlying communication protocol (see 3.1.5). In the case of our prototype not all services standardised in ISO14229 are needed and therefore not all are implemented. For the prototype, specific functionality had to be added to this layer so that the diagnostic tester is able to recognize the ECU and retrieve interesting information, like the ECU description. The service 'Start Diagnostic Session' was already implemented by Audi and no changes had to be made.

While gathering diagnostic information about an ECU the first service which really transfers node specific data is 'Read ECU Identification'. Custom identification strings are sent to the diagnostic tester. Some strings and their values in our prototype ECU are

Audi Part Number '6M1234357Y0'

Part Number '12345678912'

System description 'TTT-PowerDiagNode'

All these values are transferred to the diagnostic tester which then shows them to the engineer. You can see how this information is shown on the diagnostic tester on the upper right corner of the picture 4.12.

```
typedef struct
{
 tDEH_DTC       DTC;              /* Diagnostic Trouble Code             */
 uint8_t        EventParameter;   /* time or cycle/erase/chg state detect*/
 uint8_t        EventSymPrio;     /* faulttype and priority              */
 tDEH_QualET    QualEvent;        /* time or  cycle for defect detection */
 tDEH_QualET    DeQualEvent;      /* time or  cycle for defect detection */
 tDEH_LampInfos LampParameter;    /* Parameter to control a warning lamp */
 uint8_t        IndexFFrameTable; /* index for Freeze Frame Table        */
} tDEH_EventPathParameter;

[...]

tDEH_EventPathParameter
       DEH_EventPathParameter[DEH_MAX_NUMBER_OF_EVENTPATHS]=
{
  /* DEH_EVENT_1.DTC: "Benutzer 2 defekt"                               */
  {0x07EC,
    /*       DEH_EVENT_1.EventParameter                                 */
   DEH_TIME_DEBOUNCED,
    /*       DEH_EVENT_1.EventSymPrio: symptom = E; priority = 3;       */
   0x3E,
    /*       DEH_EVENT_1.QualEvent: 0x0100 cycles                       */
   0x0100,
    /*       DEH_EVENT_1.DeQualEvent: 0x0010 cycles                     */
   0x0010,
    /*       DEH_EVENT_1.LampParameter: event is connected to lamp 1;   */
    /*                                  flashing NOT active             */
   0x08,
    /*       DEH_EVENT_1.IndexFFrameTable: first position in the table  */
                                          DEH_FreezeFrameTable         */
   0x00},

[...]
};
```

Figure 4.2: DEH - DTC configuration code

With the SDS service 'Start Routine By Local Identifier' the diagnostic tester retrieves the information which functionality is supported by the target ECU. Here the answer message was adjusted in such way as that the diagnostic tester knows that the service 'Read Data By Local Identifier' *is* supported.

The service 'Read Data By Local Identifier' was now implemented to produce TTP specific information which can lead to a possible error. In the code in Figure 4.3 you can see that two local identifier are supported.

The first identifier selects the displaying of the membership of the three other[3] nodes in the cluster. The problem now was that the diagnostic tester is not aware of TTP-membership. That's why we used the feature that it can display the terms 'WARM'[4] or 'KALT'[5]. The diagnostic tester now shows the term 'WARM' if a node has the membership and 'KALT' if it has not.

The second identifier reveals the new diagnostic feature which pinpoints the location of a cable break. It is described in Section 4.5.

4.1.4 KWP2000 Layer

The KWP2000 standard ISO14230 is split up into four parts which are:

- Part 1: Physical layer
- Part 2: Data link layer
- Part 3: Application layer
- Part 4: Requirements for emissions-related systems

In our case only the data link layer and the application layer needed to be implemented. The physical layer is replaced by TP2.0 and CAN communication or in our case CAN-tunnelling. In the original prototype code from Audi the data link layer was already implemented which includes the message structure and the communication specific services (see 3.1.6).

4.1.5 TP2.0 Layer

As described in Section 3.1.4 this layer is able to segment the data for the KWP2000 layer to fit into CAN messages. To use this layer for our prototype some adjustments had to be done. The original code was not exactly compatible to the HIS CAN driver interface which is necessary for the underlying CAN emulation layer. For example the function call `CanInterruptDisable` needs the parameter 'channel handle' in the HIS CAN driver interface whereas in the driver used by the original code no parameter was necessary. The channel handle is used to distinguish between several CAN channels. In our case only one channel is used so this parameter can be set to `0` on all appearances.

[3] other than the one which is currently under investigation
[4] 'WARM' - german word for warm
[5] 'KALT' - german word for cold

```
tSDS_Status DiagService_21(tSDS_RxTxBuffer* pBuffer,
                           tSDS_RxTxBufferLen* bufferLen) {
  char s[100];
  switch (pBuffer[0]) { //local Identifier
    case 1:
      pBuffer[1] = 0x0A; //display type: Warm/Kalt
      pBuffer[2] = (test_diag_membership[6] == '1'? 1:3);
      pBuffer[3] = 2;

      pBuffer[4] = 0x0A; //display type: Warm/Kalt
      pBuffer[5] = (test_diag_membership[7] == '1'? 1:3);
      pBuffer[6] = 2;

      pBuffer[7] = 0x0A; //display type: Warm/Kalt
      pBuffer[8] = (test_diag_membership[8] == '1'? 1:3);
      pBuffer[9] = 2;

      *bufferLen = 10;
      break;
    case 2:
      pBuffer[1] = 0x5F; //Type: long text

      if (cableDamageA || cableDamageB) {
        pBuffer[2] = 40; //number of chars
        sprintf(&pBuffer[3], "Cable break @ TTP/C-CH_%c between %d
                             and %d ", (cableDamageA?'A':'B'),
                             cableDamageLow, cableDamageHigh);
        *bufferLen = 43;
      } else {
        pBuffer[2] = 23; //number of chars
        sprintf(&pBuffer[3], "No cable break detected ");
        *bufferLen = 26;
      }
      break;
    default:
      pBuffer[1] = 0x5F; //Type: long text
      pBuffer[2] = 21; //number of chars
      sprintf(&pBuffer[3], "Service not available ");
      *bufferLen = 24;
      return SDS_SERVICE_FINISH_AN_AVAILABLE;
  }
  return SDS_E_OK;
}
```

Figure 4.3: SDS Service 'Read Data By Local Identifier'

In this layer two values have to be defined which determine how the ECU should respond and what type of ECU this is. These are the TP2.0 address and the CAN Id.

The TP2.0 address specifies the type of the ECU. In our case the TP2.0 address is set to `0x2D` which defines the ECU to be the parking aid (EPH).

To communicate with the diagnostic tester the CAN driver has to accept the CAN id `0x332` for TP2.0 layer.

These parameters have to be configured in the CAN emulation layer as this is the layer next to TP2.0 (see Figure 3.1).

4.1.6 CAN Interface

Now the most important choice we had to make was whether to run the demo application either in a separate time-triggered task, in the background hook of the TTP-OS or in some interrupts. This is the first decision one has to make when integrating a ET software into an a priori scheduled TTA. As this original code had a cyclic task which was triggered by a periodic alarm the first possibility listed here, which is offered by CAN Emulator, seemed to be the best and simplest choice. This is because the periodic alarm used in the legacy application behaves like a time-triggered task. The internal functionality of the legacy application also depended on the periodic execution, because of some internal counters.

4.2 Prototype Setup

In Figure 4.4 you can see the basic structure of our prototype setup. The Tester, located in the lower left corner of the figure is the certified diagnostic tester, which is available in every Audi garage. It is the main tool to pinpoint or even repair faults in the electric system of a car. It is directly connected to the CAN-Gateway which is usually connected to all available CAN subsystems of the car. Here the CAN-gateway is just connected to the so-called “comfort CAN” because the other CAN subsystems are not needed in this small prototype. As shown in the figure, the CAN-gateway transforms the messages from high-speed CAN, which is necessary for the tester, to low-speed CAN, which is necessary for the SMLS and therefore also for the TTP-CAN gateway which simulates the CAN bus connected to the SMLS.

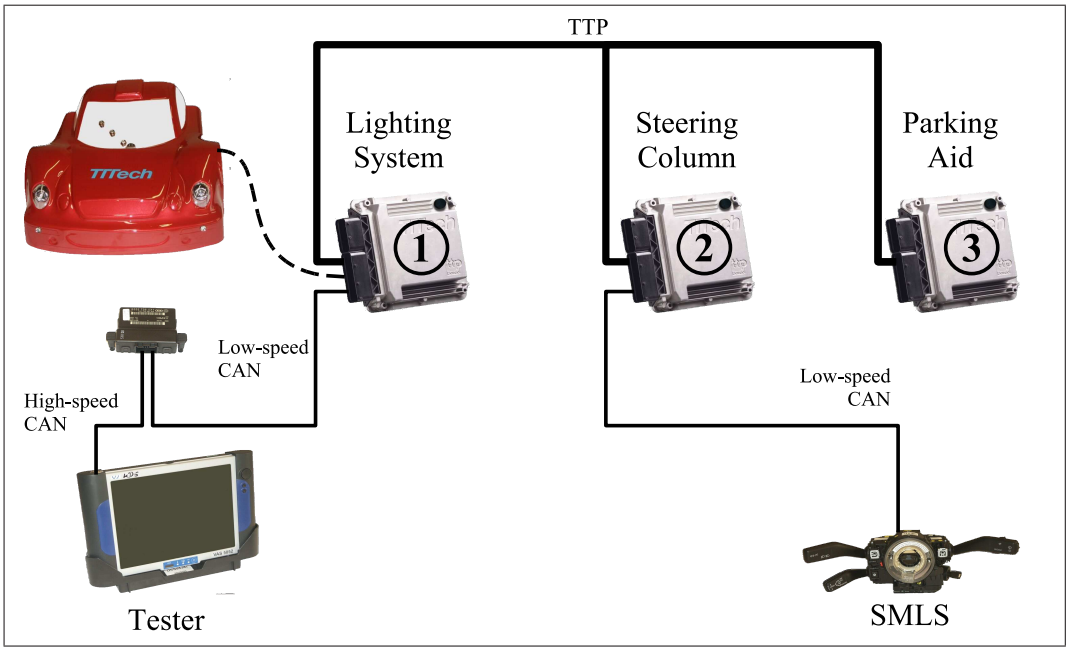

Figure 4.4: Overview of our demo setup

The three TTP nodes have the following functions:

1. The first node has two purposes: On the one hand it translates low-speed CAN messages from the CAN-gateway to the CAN emulating TTP-cluster. On the other hand it visualizes some functions controlled by the SMLS with the red car on the left side of Figure 4.4.

2. The second node shows that conventional CAN hardware can be connected to TTP-CAN gateways to send CAN messages through a TTA instead of a CAN network.

3. The last node has a particular importance to this work. It contains the nearly unmodified software stack from Audi (see Figure 4.5) which shows that migrating CAN legacy software to a TTP/C node is possible and it shows the advantages you can get from this migration. It is named "Parking Aid" because it responds to the tester as the parking aid which is installed in some modern cars. In our demonstration it has no external functionality like sound generation or blinking lights like usual parking aids.

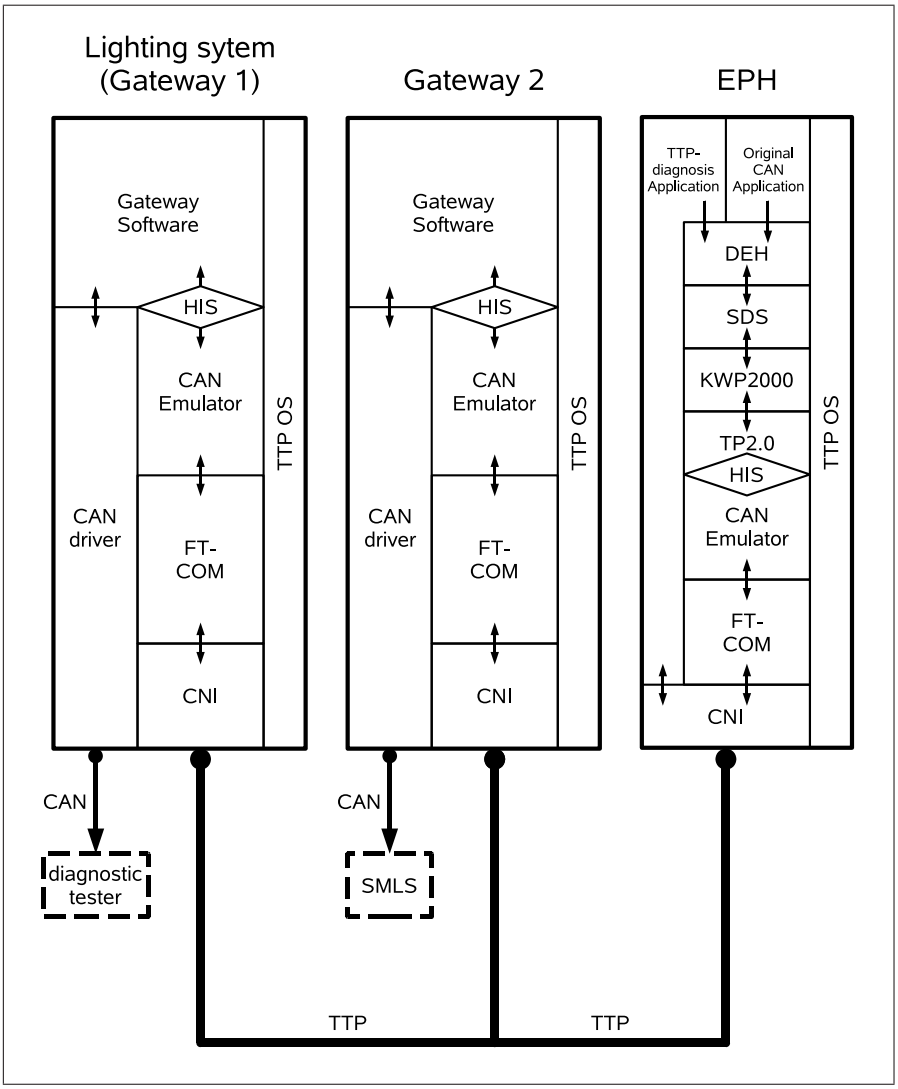

Figure 4.5: Detailed software layering of all demonstration nodes

4.3 Integration of the CAN Emulator

The CAN emulator source and configuration is split according to the two level design of a TTA like our model. The detailed description of the CAN Emulator is in [35] while in this chapter only the changes which have to be made for this prototype setup are given.

On cluster level the sources[6] of the CAN Emulator have to be included and in the header files parameters like in Figure 4.6 have to be set to configure

[6] as there was no need for a library until now we just have to use the sources

```
HIS CAN Driver Specification compliant defines

#define C_ENABLE_TRANSMIT_QUEUE
#define C_SINGLE_RECEIVE_CHANNEL
#define C_ENABLE_DYN_TX_DLC
#define C_ENABLE_DYN_TX_ID

#define C_ENABLE_RECEIVE_FCT
#define C_ENABLE_DYN_TX_OBJECTS

#define C_DISABLE_RECEIVE_FCT
#define C_DISABLE_ECU_SWITCH_PASS
[...]

Define for CAN Emulator

#define CAN_EMU_NUM_NODES            3
```

Figure 4.6: Header file parameters for CAN Emulator (cluster level)

```
HIS CAN Driver Specification compliant defines

#define CAN_EMU_CURRENT_NODE         3

#define CAN_EMU_SEND_BUFFER_SIZE     8

#define CAN_EMU_RECEIVE_BUFFER_SIZE 24
```

Figure 4.7: Header file parameters for CAN Emulator (node level)

the behavior. The HIS CAN Driver Specification describes which parameters you can set to enable or disable parts of the driver [35]. For the internal CAN Emulator buffers only the property `CAN_EMU_NUM_NODES` has to be set to the number of nodes in the TTP/C network using the CAN Emulator Driver.

On node level, three defines have to be set to configure the CAN Emulator (shown in Figure 4.7). These are which node number the current node is, how many send buffers are reserved on the TTP/C bus for this node and how many buffers the CAN Emulator has to reserve for all incoming messages of other nodes using the CAN Emulator. These parameters can be derived from the need of bandwidth and the length of the cluster cycle. Details on these parameters can be found in Section 4.4. Enhancements that these parameters are generated automatically by special tools is planned for future versions.

```
/* can message structure */
struct s_can_msg {
    /* header storing 16 bit ID, 8 bit DLC and 8 bit CAN_NR */
    ubyte4 header;
    /* data1 storing 4 databytes */
    ubyte4 data1;
    /* data1 storing 4 databytes */
    ubyte4 data2;
};
typedef struct s_can_msg ts_can_msg;
```

Figure 4.8: Can Emulator data structure storing one CAN message

4.4 Two Level Design

The next step after the assembling of the demo was to build a cluster schedule and the node schedules for three TTP/C nodes. This is done with the TTTech tools TTP-Plan[7] and TTP-Build[8]. These tools enable seamless integration of the cluster and its nodes to a TTA.

4.4.1 Cluster Level

In this design step network parameters have to be defined which are common for all nodes. Once they are defined the node development can be done independently.

In this demo the only communication is the CAN tunneling. No other applications are running and communicating on this TTP/C bus. The goal was to achieve at least the bandwidth and and at most the latency of a CAN-bus. The minimum latency of a CAN-bus can not be reached because of the time-triggered architecture using a cluster cycle length of 4ms but the latency needed for our standard software stack can be satisfied. The CAN emulator needs the size of the data type `ts_can_msg` reserved on the bus for each emulated CAN message which is going to be sent on the TTP/C bus. The documentation of the CAN emulator [35] or the include files of the CAN emulator reveal that the data type is defined as in Figure 4.8.

[7] with TTP-Plan the cluster level can be designed
[8] with TTP-Build the node level can be designed

To have at least the same bandwidth as the high-speed CAN (500kbps), 8 CAN message objects are reserved on the TTP/C bus for each node. While using a cluster cycle of 4ms to have a good response, the real data throughput without any overhead would be: $48000bit/s$. These calculations presume 3 Nodes, each sending 8 CAN messages encapsulated in TTP/C messages per cluster cycle.

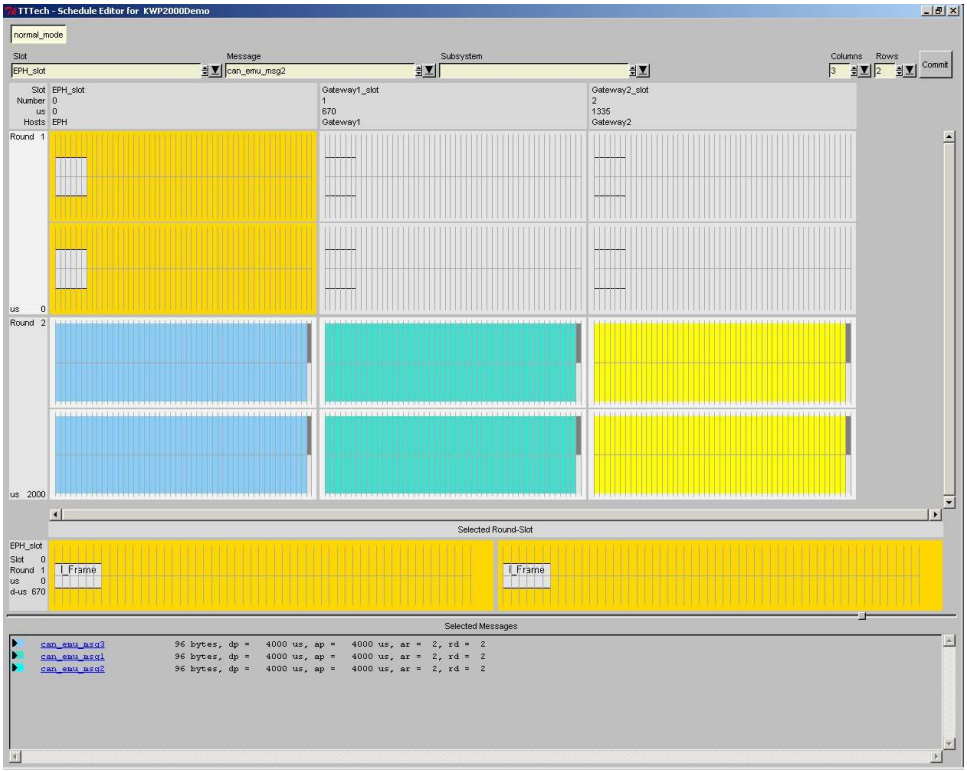

Figure 4.9: The cluster schedule of the demo application

4.4.2 Node Level

At the node level all parameters are defined which are node specific. The parameters defined in this level also have to meet all constraints of the cluster level to support composability with all other nodes. In a TTA tasks are scheduled before runtime to guarantee timeliness. This is done with the program TTP-Build. In our case we need nodes that have a similar task schedule to the nodes of the CAN world. The difference here is that in a TTA nodes are not supposed to keep running if the network is not available but they usually just reboot as this is done very quickly. In case of network unavailability nodes usually go to a safe state, reboot and try to reintegrate. In the CAN world

the nodes keep operating and give their best effort even without any communication. Time-Triggered Protocol Operating System (TTP-OS), which our nodes run with, supports mode changes to create an application running with or without network communication.

The two application modes are created for each node to keep the node running independent of network availability.

- `sync_mode` contains all tasks which are activated while the TTP/C network is available
- `drift_mode` contains all tasks which are activated while the TTP/C network is **not** available

The third node (Einparkhilfe (ger.) (EPH) node) of the demo has a task schedule defined on the node level (Figure 4.10) where you can easily see that the `sync_mode` contains the `can_emu` tasks for the communication while both modes contain the `KWP2000_task` running the KWP2000 diagnosis stack handling. This would be the behaviour on a CAN network.

4.5 Added Diagnostic Features

There are more features beside the membership and the frame status one could be able to use to enhance diagnosis or pinpoint a faulty node or cable in a TTA like a replicatively calculated value. In this demo only the membership of each node is used to gain some interesting diagnosis features. The first is a basic one where the membership is just forwarded to the DEH layer and sent to the tester on demand. In the code sample Figure 4.11 it can be seen that the check of node 2 which has to be saved in a special variable (here `uApplMsg01MsgBuffer`) which is read by the DEH layer. This check, if the node is faulty or not, is of course done by another node except node 2. With this information a faulty node can be detected and displayed on the diagnostic tester. The arguments for `TTPC_MEMBERSHIP_IS_VALID()` are the location of the current membership information in the RAM and the node index on with the membership information should be returned. This node index is the node number minus one because the information is stored in an array which starts with 0 in ANSI C.

A dependable assembly of a TTP/C network works on two replicated bus lines which are locally seperated from each other. Therefore four connectors

- Task Schedule for '`sync_mode`'
  - Task '`FT_Task_sync_mode_0`'
    receives TTP/C messages from the communication controller
  - Task '`can_emu_IO`'
    handling the sending and receiving of encapsulated CAN messages on the TTP/C bus.
  - Task '`can_emu_member`'
    simulation of the acknowledgment messages of the CAN bus based on the membership of the TTP/C bus (see [35] for details).
  - Task '`FT_Task_sync_mode_1`' sends the TTP/C messages to the communication controller
  - Task '`KWP2000_task`'
    diagnosis request handling using the software stack from Audi.
  - Task '`FT_Task_sync_mode_2`'
    updates the TTP/C controller life sign (see 3.1.2 for details)
  - Task '`mode_change_0`'
    mode change handling
  - Task '`mode_change_1`'
    time synchronization and second part of mode change handling
- Task Schedule for '`drift_mode`'
  - Task '`FT_Task_drift_mode_0`'
    updates the TTP/C controller life sign (see 3.1.2 for details)
  - Task '`drift_task`'
    Debug task and mode change request operations.
  - Task '`KWP2000_task`'
    diagnosis request handling using the software stack from Audi. In `drift_mode` the software now just does error collection of internal errors.
  - Task '`FT_Task_drift_mode_1`'
    updates the TTP/C controller life sign (see 3.1.2 for details)
  - Task '`mode_change_drift_0`'
    mode change handling
  - Task '`mode_change_drift_1`'
    time synchronization and second part of mode change handling

Figure 4.10: Task Schedule for the third demo node "EPH"

```
if (TTPC_MEMBERSHIP_IS_VALID(au4VectorMembership, 2 - 1)) {
  uApplMsg01MsgBuffer = APPL_MSG0X_EVENT_FALSE_START_VALUE;
} else {
  uApplMsg01MsgBuffer = APPL_MSG0X_EVENT_TRUE_START_VALUE;
}
```

Figure 4.11: C Code for sending membership status into DEH layer

are installed in the demo to simulate a cable break or disconnect. With these connectors the TTP/C Channel A can be disconnected between node 1 and node 2 or between node 2 and node 3. The same is possible with Channel B. When problems occur on one of the two bus lines one frame status bit is zero while the corresponding bit of the second line is one. This information can be abstracted as "bus communication error" either on channel A or channel B. As a matter of fact there are no DTCs expressing this because in the usual CAN world there is no channel A or channel B. Therefore the DTCs for "comfort CAN-High-Line defective" and "comfort CAN-Low-Line defective" are used. The mechanic has to differ if it's a TTP ECU or a CAN ECU. Then he knows if the CAN-High line is addressed or the TTP channel A. In Figure 4.12 you see a screenshot of the diagnostic tester showing such an error for a TTP ECU. In the upper right corner the model (in this demo it's `TTT-PowerDiagNode`) of the ECU is shown so a mechanic should know that this ECU uses TTP communication in stead of CAN communication. These are the only disadvantages one has while using CAN software on top of TTP hardware.

When using a bus architecture a simple algorithm (Figure 4.14) can even determine where the single cable disconnect is located. The algorithm is changed for readability. The variable `breaktype` specifies the branching whether the frame status is good on both bus lines before or after the break. This distinction is implicitly done by the membership agreement algorithm and influenced by the clique avoidance. As precondition we have the single fault hypothesis so the algorithm only has to detect one break. The information is then stored in a special variables (here `uApplMsg05MsgBuffer` and `uApplMsg06MsgBuffer`) which are read by the DEH on demand.

The code in Figure 4.14 just detects where on the bus the frame status information changes from "both buses work" to "only one bus works". This change is depicted in a sample scenario in Figure 4.13. The code would end in

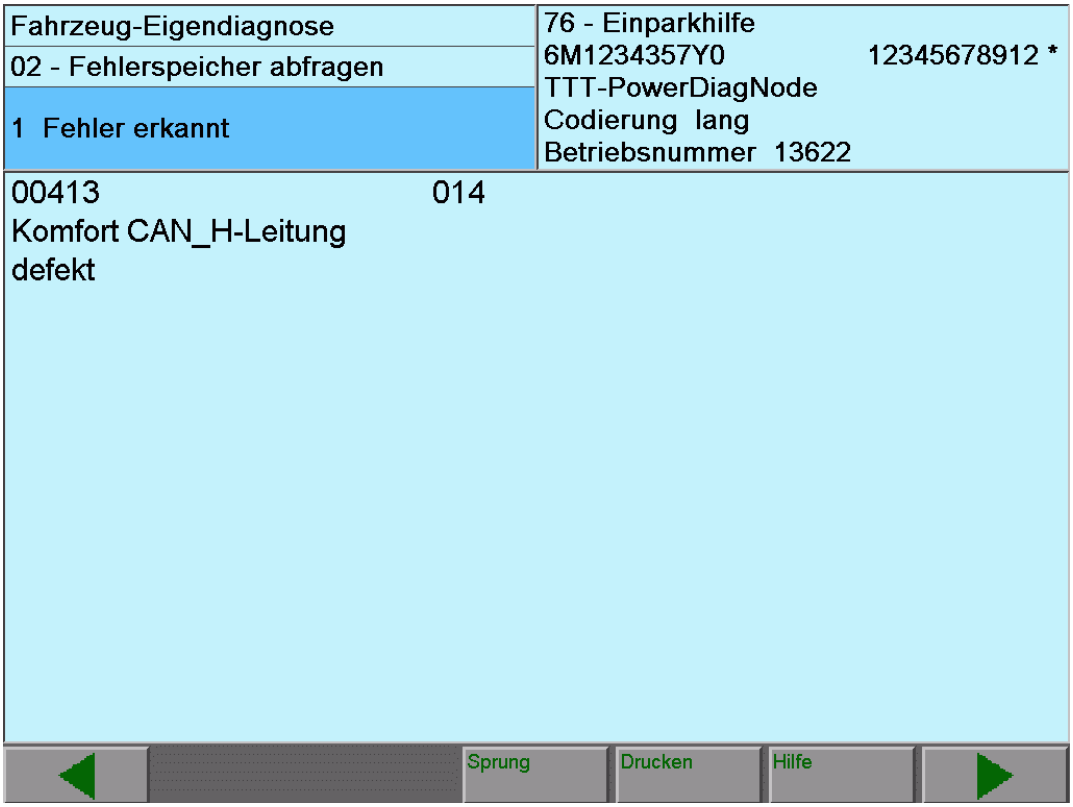

Figure 4.12: Screenshot of diagnostic tester showing a DTC on TTP hardware

this scenario with the variables

```
cableDamageA    = 0
cableDamageB    = 1
cableDamageLow  = n + 1
cableDamageHigh = n + 2
```

The Information created with

```
sprintf(&pBuffer, "Cable break @ TTP/C-CH_%c between %d and %d ",
        (cableDamageA?'A':'B'), cableDamageLow, cableDamageHigh);
```

is then shown on the diagnostic tester when searching for a malfunction in the system. This sample is shown in Figure 4.15. The most interesting thing here is that all nodes which contain this detection algorithm report the same error because of the global state of the TTA.

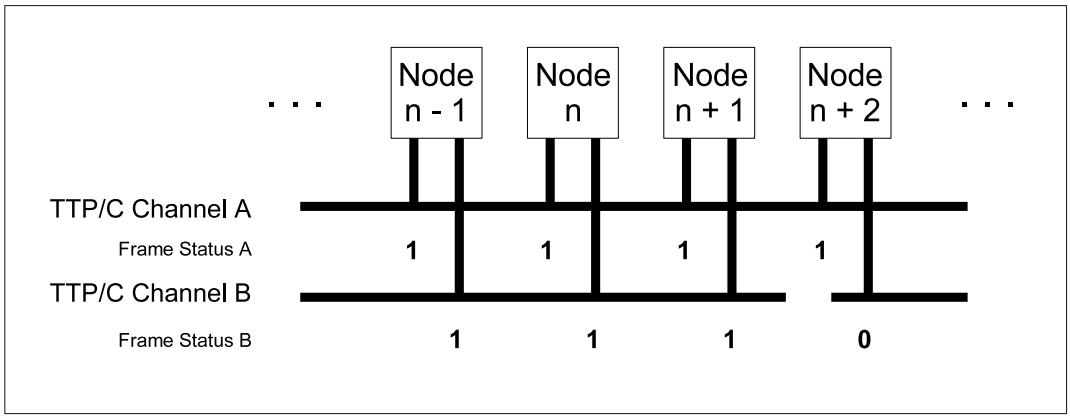

Figure 4.13: Frame statuses during simulated cable break

```
cableDamageA = 0;  cableDamageB = 0;  breaktype = 0;

if (TTPC_MEMBERSHIP_IS_VALID(au4VectorMembership, 0)) {
  cableDamageLow = -1;
  cableDamageHigh = -1;
  for (i = 1; i <= NUMBER_OF_NODES; i++) {
    if ((!frame_status('a', i)) ^ (!frame_status('b', i))) {
      if (breaktype == 0) breaktype = 1;
     if (!frame_status('a', i)) cableDamageA = 1;
      if (!frame_status('b', i)) cableDamageB = 1;
      if (breaktype == 2) {
        cableDamageHigh = i;
        break;
      } else {
        cableDamageLow = i;
      }
    }
    if (frame_status('a', i) && frame_status('b', i)) {
      if (breaktype == 0) breaktype = 2;
      if (breaktype == 1) {
        cableDamageHigh = i;
        break;
      } else {
        cableDamageLow = i;
      }
    }
  }

  if (cableDamageA) {
    uApplMsg05MsgBuffer = APPL_MSG0X_EVENT_TRUE_START_VALUE;
  } else {
    uApplMsg05MsgBuffer = APPL_MSG0X_EVENT_FALSE_START_VALUE;
  }
  if (cableDamageB) {
    uApplMsg06MsgBuffer = APPL_MSG0X_EVENT_TRUE_START_VALUE;
  } else {
    uApplMsg06MsgBuffer = APPL_MSG0X_EVENT_FALSE_START_VALUE;
  }
}
```

Figure 4.14: Cable break detection algorithm for the bus topology

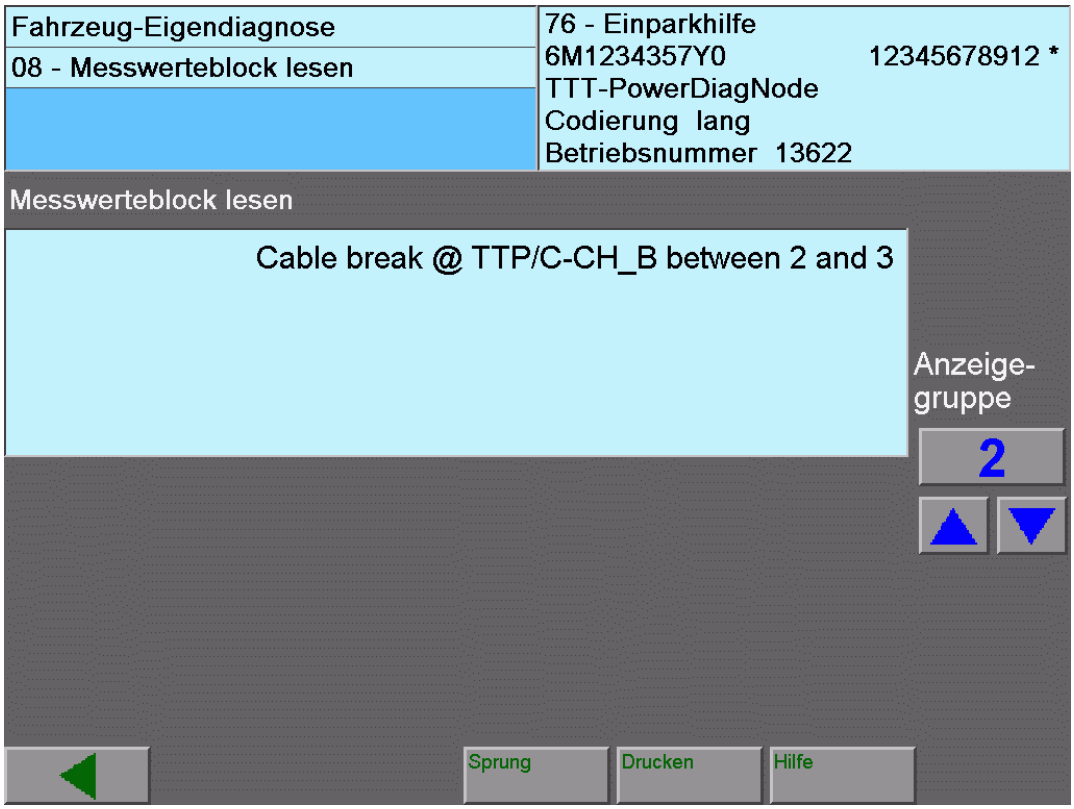

Figure 4.15: Sample diagnostic tester output during cable break

5 Validation

In this chapter our integrated system for TT and CAN-based applications is validated by performing tests covering the complete range of functionality the previously described prototype setup is offering. The tests are separated into four groups which reflect the main steps to integrate and diagnose CAN-based legacy applications in the TTA.

Each test is described in a sub-section starting with the term "Test Case". Each of them has the same prerequisites if not mentioned to be different.

- All components have to be cabled corresponding to the setup in Figure 4.4 (called "standard cabling" henceforth)
- All components have to be connected to a power source
- The VAS diagnostic tester has to be turned on

At the beginning of each test the VAS diagnostic tester has to be in the "initial state" which is defined as the screen where you can see the diagnosis button ("Fahrzeug-Eigendiagnose") on the upper right corner. The ignition switch[1] has to be in the state "ignition on".

The test groups are:

1. Validating usage of standard CAN hardware
2. Showing the legacy hardware compatibility of the implementation
3. Testing tunneling of basic CAN messages and diagnostic messages through TTP from a CAN ECU
4. Reading standard and advanced diagnostic messages from a TTP ECU which is running a ported legacy application

5.1 Validation Of Standard CAN Hardware

This scenario shows standard CAN communication and CAN diagnosis. This illustrates, mainly for public demonstrations, that the used CAN parts in the prototype are not modified, nor damaged and can operate in the usual way. This scenario is depicted in Figure 5.1. The used communication channel is highlighted.

[1] the car key

5.1.1 Test case 1: Direct CAN Connection

Test case:	1
Precondition:	The CAN cables from the SMLS have to be connected with the CAN cables from the little black gateway[2]. Now the diagnostic tester has a direct connection to the SMLS through the gateway. The used communication channel is depicted in Figure 5.1.
Test procedure:	On the diagnostic tester the diagnosis button ("Fahrzeug-Eigendiagnose") is pressed and the list entry "16 - Lenkradelektronik" has to be selected subsequently.
Result:	No error message.

As no error message occurred but a list including the items "02 - Fehlerspeicher abfragen" and "08 - Messwerteblock lesen" the test succeeded. Now it is assured than the CAN communication channel is working between the diagnostic tester and the SMLS as well as the SMLS is working sufficiently.

5.1.2 Test case 2: Direct CAN Connection - Lever State

Test case:	2
Precondition:	Test case 1
Test procedure:	To read state information the special list entry "08 - Messwerteblock lesen" has to be selected. Now the section "1" of the state information has to be selected and confirmed with the "Q"[3]-button.
Result:	No error message. Lever state visible.

If no error message occurs within this test you are able to see the lever positions on the diagnostic tester. Any change of the indicator lever or the dipped beam lever of the steering column will instantly change the diagnostic tester's screen. For example the indicator lever position causes the tester to show either the word "right" or "left".

[3] "Q" here means acknowledge (quittieren - ger.) and is in total contrast to the english "quit"

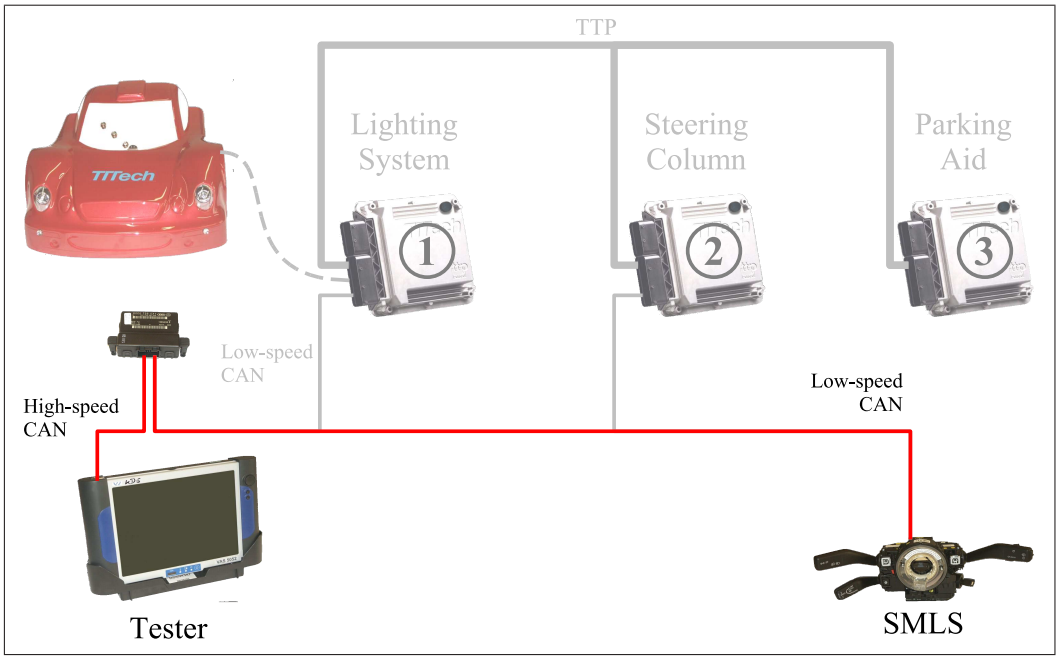

Figure 5.1: Standard CAN communication

5.2 CAN Hardware Compatibility

The CAN hardware compatibility is tested in the following experiment explicitly while in the other sections of this chapter it is implicitly tested, too. CAN hardware compatibility means that a legacy CAN ECU can be connected to the prototype and is able communicate with it concerning the physical layer and the CAN protocol. This communication is highlighted in Figure 5.2.

5.2.1 Test case 3: Heterogeneous Devices

Test case:	3
Precondition:	Standard cabling
Test procedure:	Changing lever positions on the steering column.
Result:	The red prototype car's LEDs light up according to the lever positions on the steering column.

The SMLS is sending its control signals

- turn left
- turn right
- headlamp flasher and
- windshield wiper (four different operation modes)

through CAN and TTP/C to the red prototype car. This red car is directly controlled by a custom software running on the TTP/C node one. This demonstrates also a possible interaction between legacy hardware and a TTP/C node. The red car has seven LEDs which are able to visualize the control signals originating from the ECU "SMLS".

5.3 CAN Tunneling

Basic tunneling shows that normal CAN messages can be tunneled through TTP/C without problems. In the prototype this is achieved by the same procedure as described in Section 5.1. The lever positions and the diagnostic messages from the SMLS can be queried by the tester. The big difference here is that the CAN communication is broken up into two parts (compare Figures 5.1 and 5.3). Instead of the CAN communication the CAN frames are transferred into the TTA and back again by two gateways. These gateways are called "Lighting System" and "Steering Column" in all figures. This communication tunneling is highlighted in Figure 5.3. This assembly shows the backward compatibility of our system.

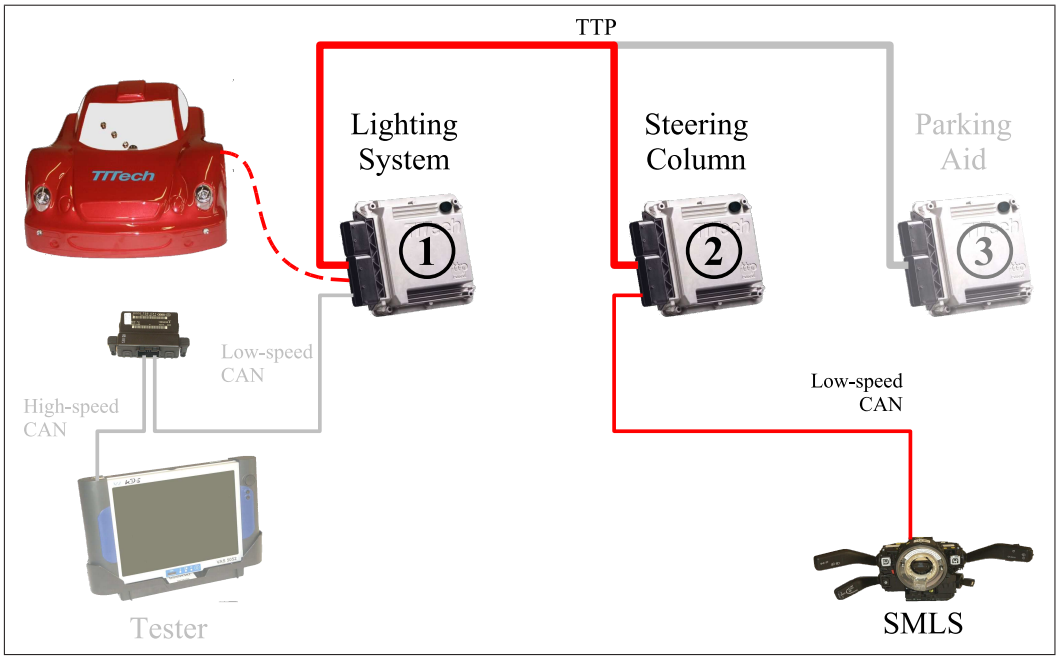

Figure 5.2: Controlling a TTP ECU with a legacy CAN ECU

5.3.1 Test case 4: Tunneled Connection

Test case:	4
Precondition:	Standard cabling
Test procedure:	On the diagnostic tester the diagnosis button ("Fahrzeug-Eigendiagnose") is pressed and the list entry "16 - Lenkradelektronik" has to be selected subsequently.
Result:	No error message.

As no error message occurred but a list including the items "02 - Fehlerspeicher abfragen" and "08 - Messwerteblock lesen" the test succeeded. Now it is assured than the CAN communication channel is working between the diagnostic tester and the SMLS as well as the SMLS and the two gateways "Lighting System" and "Steering Column" are working sufficiently.

5.3.2 Test case 5: Tunneled Connection - Lever State

Test case:	5
Precondition:	Test case 4
Test procedure:	To read state information the special list entry "08 - Messwerteblock lesen" has to be selected. Now the section "1" of the state information has to be selected and confirmed with the "Q"[4]-button.
Result:	No error message. Lever state visible.

If no error message occurs within this test you are able to see the lever positions on the diagnostic tester. Any change of the indicator lever or the dipped beam lever of the steering column will instantly change the diagnostic tester's screen coherently. It is also shown that continuous transmissions of data are tunneled fluently even with a rather big delay in contrast to ordinary CAN transmissions. (see Section 5.5)

5.3.3 Test case 6: Tunneled connection - Diagnosis 1

Test case:	6
Precondition:	Test case 4
Test procedure:	The list entry "'02 - Fehlerspeicher abfragen"' has to be selected.
Result:	No error message and a notification about the number of detected errors.

Several DTCs like sporadic communication failure can be displayed on the tester when querying diagnostic information by selecting "02 - Fehlerspeicher abfragen". As the connection was interrupted between the last test group and the current the display now shows an error message about an sporadic communication failure.

[4] "Q" here means acknowledge (quittieren - ger.) and is in total contrast to the english "quit"

5.3.4 Test case 7: Tunneled connection - Diagnosis 2

Test case:	7
Precondition:	Standard cabling.
Test procedure:	On the diagnostic tester the diagnosis button (“Fahrzeug-Eigendiagnose”) is pressed and the list entry “05 - Fehlerspeicher lï¿½chen” has to be selected subsequently. This causes the SMLS to delete all entries of its error memory.
Result:	After the confirmation to delete all entries of the error memory in the left upper corner the Text “Fehlerspeicher gelï¿½cht” and “0 Fehler erkannt” appear.

In this test it is shown that the communication is working through the tunneling and that the SMLS is deleting it’s error entries out of it’s flash memory flawlessly.

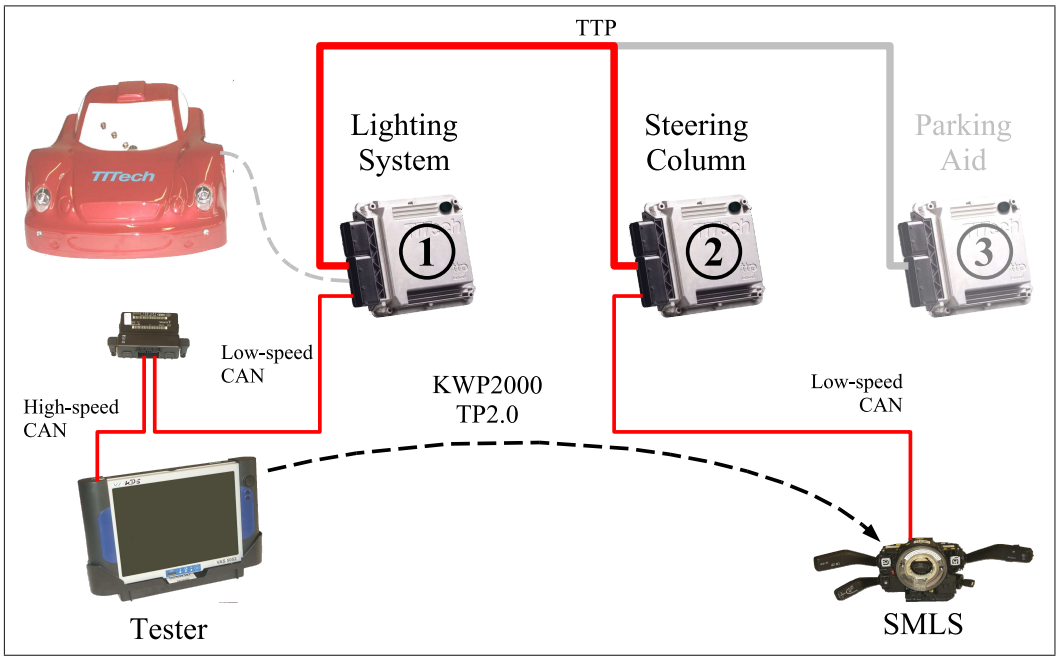

Figure 5.3: Tunneling CAN messages and diagnostic messages

5.4 Legacy Application Diagnosis

Here the main feature of the prototype is illustrated. To be able to show legacy application diagnosis the tester has to be back in the initial state either

by several back button pushes if some of the tests above have been done or by powering on the device.

5.4.1 Test case 8: Parking Aid

Test case:	8
Precondition:	Standard cabling.
Test procedure:	After pressing the diagnosis button ("Fahrzeug-Eigendiagnose") the parking aid ECU ("76 - Einparkhilfe") has to be picked from the list.
Result:	The term 'TTT-PowerDiagNode' appears in the upper right corner of the tester.

This indicates that the communication between the tester and the ported legacy diagnosis application is working (see Figure 5.4). From the diagnostic tester's point of view the TTP/C ECU now acts as a usual CAN ECU where diagnosis information can be gathered.

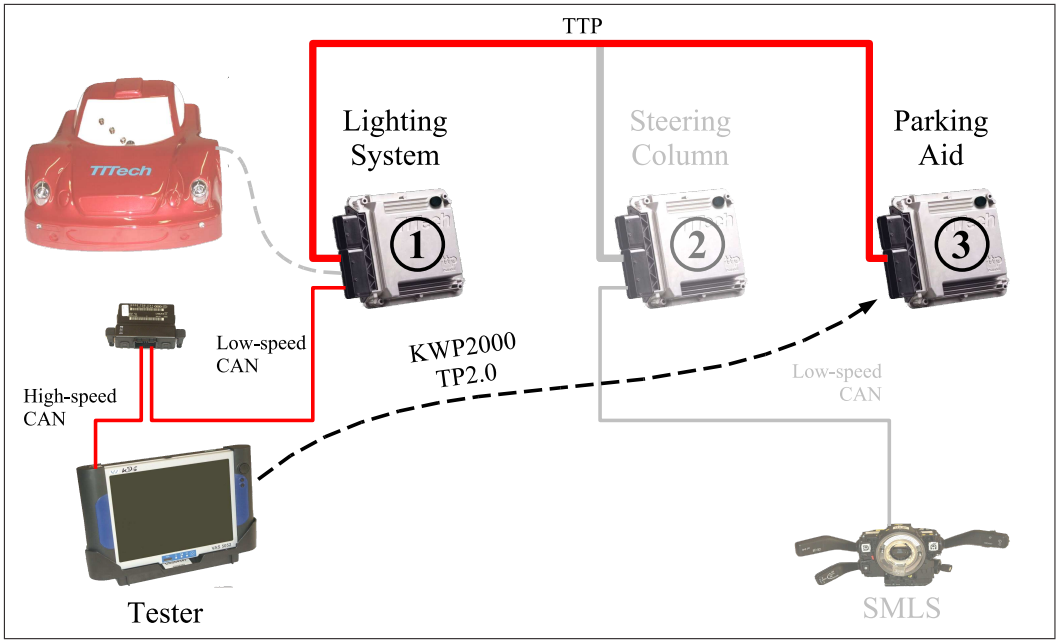

Figure 5.4: Legacy application diagnosis

5.4.2 Test case 9: Parking Aid - Diagnosis 1

Test case:	9
Precondition:	Test case 8
Test procedure:	The list entry "'02 - Fehlerspeicher abfragen"' has to be selected.
Result:	No error message and a notification about the number of detected errors which is 0 in this test. ("0 Fehler erkannt" (ger.))

Several DTCs like sporadic communication failure can be displayed on the tester when querying diagnostic information by selecting "02 - Fehlerspeicher abfragen". A special DTC which can appear on the tester is "Komfort CAN_H-Leitung defekt" in the next test case.

5.4.3 Test case 10: Parking Aid - Diagnosis 2

Test case:	10
Precondition:	Test case 8
Test procedure:	The power supply of the 2nd ECU is disabled for about 6 seconds. Afterwards the TTP/C channel B is disconnected between the 2nd and the 3rd ECU. Now the TTP/C channel A between the 2nd and the 3rd ECU is disconnected for about 6 seconds. Finally the list entry "02 - Fehlerspeicher abfragen" is selected on the garare tester
Result:	A list of error messages (containing "Komfort CAN_H-Leitung defekt") matching to the actions of the test procedure.

In this test the need for disconnecting the power supply and the TTP/C cable for about 6 seconds originates from the debouncing algorithm explained in Section 3.1.7. If disconnection is much shorter than this amount of time the DEH layer is configured to ignore it as if it was only a "sporadic" error. This test shows that the DEH layer works as expected.

When reading the error messages on the diagnostic tester the skilled mechanic now knows that while querying a TTP node - indicated by the term "TTT-PowerDiagNode" in the right upper corner - there is no "CAN high" line connected to it. The reason is derived from the limited number of possible

DTCs. When this TTP node informs you about an error on the “CAN high” line the TTP/C channel B is addressed, so there is a communication problem on the TTP/C channel B (Figure 4.12). This case is depicted in Figure 5.5.

5.4.4 Test case 11: Parking Aid - Diagnosis 3

Test case:	11
Precondition:	Test case 10
Test procedure:	On the diagnostic tester click the left arrow to go back to the selection of diagnostic features. Then choose the entry of state information retrieval “08 - Messwerteblock lesen” and select section “2”. Confirm the section “2” with the “Q” button.
Result:	Special error message about the cable break discovered in the test case 10.

We still have the disconnected channel as depicted in Figure 5.5. On the screen of the diagnostic tester the information “Cable break TTP/C-CH_B between 2 and 3” appears (Figure 4.15). This exact localisation of the cable break (or disconnect) is possible because the information is retrieved via the replicated TTP/C channel A and the exact position can be derived from the frame status bits and the fact that we have a bus architecture.

5.4.5 Test case 12: Parking Aid - Diagnosis 4

Test case:	12
Precondition:	Test case 8
Test procedure:	Disconnect the power supply of ECU 2. Choose the entry of state information retrieval “08 - Messwerteblock lesen” and select section “1”. Confirm the section “1” with the “Q” button.
Result:	Keywords cold (“KALT”) and warm (“WARM”)

The keywords cold (“KALT”) and warm (“WARM”) represent the membership of the participating TTP/C notes. As there is no membership in the CAN protocol the terms cold and warm have to represent this information.

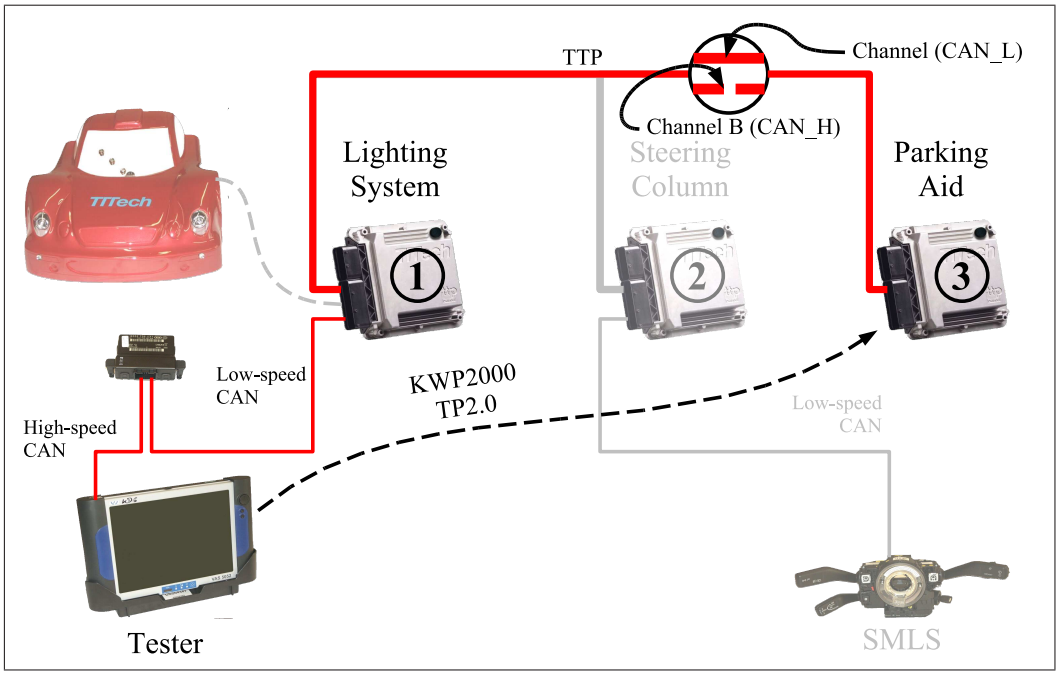

Figure 5.5: Cable break diagnosis using legacy tester hardware

5.4.6 Test case 13: Parking Aid - Diagnosis 5

Test case:	13
Precondition:	Test case 8
Test procedure:	Choose the entry of state information retrieval “08 - Messwerteblock lesen” and select section “3”. Confirm the section “3” with the “Q” button.
Result:	The text “Service not available”.

This last test simply shows that the EPH ECU only supports the state information blocks “1” and “2”. All other should produce the test “Service not available” on the diagnostic tester’s screen. It demonstrates the correctness of this specific function inside the EPH.

5.5 Timing Analysis

A very important value in communication systems for real-time applications is the latency. It is the time one message needs from one node to the other including all communication overhead. On a legacy CAN network this time is mainly affected by the arbitration algorithm and only a little by physical

from \ to		Diagnostic Tester	Steering Column	Parking Aid
Diagnostic Tester	BC	0 ms	3.67 ms	4.83 ms
	WC	0 ms	7.59 ms	8.76 ms
Steering Column	BC	4.58 ms	0 ms	no comm.
	WC	8.50 ms	0 ms	no comm.
Parking Aid	BC	7.29 ms	no comm.	0 ms
	WC	7.29 ms	no comm.	0 ms

Table 5.1: Network Latencies

properties like cable length, terminating resistor, edge sharpness etc.. On low or medium network load it is usually very small. When talking about timeouts in a higher software layer (like in TP2.0) the collisions on a CAN network are relevant because they may add an unpredictable value to the time one messages needs to arrive at the receiver. In general the latency is small enough to fulfill the required timeouts in all cases.

In a time-triggered network the latency is mainly influenced by the send instant in relation to the cluster cycle. It can be determined a priori when the cluster schedule and the task schedule of each participating node is fixed. As our tasks are only executed once in a cluster cycle it is obvious that the latency is a multiple of our cluster cycle (4ms) plus some time derived from the scheduled task activation instants. The Figures 5.6 to 5.13 depict this latency.

The latency values are very important to legacy CAN applications because they have timeouts while waiting for an answer. The communication system has to guarantee that the timeout restrictions are not violated. When comparing the latency times given in table 5.1 with the TP2.0 timeout of 500ms then there is enough time left for the computations requested by the TP2.0-master, which also have to take place during this timeout (see Section 3.1.4).

In this section all latencies are described which are related to a TP2.0 communication channel in our prototype. These are

- diagnostic tester communicating with SMLS (both directions)
- diagnostic tester communicating with ECU (both directions)

In table 5.1 the latencies are shown. As no ECU is communicating with oneself the values in the diagonal are zero. The cells containing the term “no comm.” indicate that the corresponding ECUs do not communicate with each other directly.

The values in the table differ because the latency depends on the used send slot in the cluster cycle and the exact execution instants of the tasks which are calculated by TTP-Build. The only values which are the same are the Worst Case (WC) and the Best Case (BC) latencies from the "Parking Aid" to the "Diagnostic Tester". That's because in this case there is no temporal variety of the sending instant as the sending task is time-triggered. All other values are different because the initial CAN message can arrive at an arbitrary instant.

The Figures 5.6 to 5.13 show how the values are calculated. Each row in these pictures shows one cluster cycle (lasting 4ms). If more than one cluster cycle is needed for a message to arrive they are numbered on the left hand side. (At least parts of two cluster cycles are needed in all cases.) The boxes with the bold borders are tasks or actions executed on one of the three participating time-triggered ECU of the prototype. On which ECU the message is on its way to the target is marked with the curly brackets on the right. The temporal positions of the tasks in the cluster cycle are predefined by TTP-Build.

5.5.1 Latency Figures

Below the following paragraphs all figures containing the latency analysis can be found. A detailed description is only given for the first figure as the others are analogous.

In Figure 5.6 the communication starts on the left upper corner with the Tester trying to send a CAN message to the Steering Column. At first the message is sent on the CAN-Bus between the Tester and the Lighting System. It arrives at the CAN-receive-buffer of the Lighting System, just before the execution of the CanEmuIO task (Best Case (BC)). The CanEmuIO task reads out this buffer and the message is transferred to another task which transmits the message on the TTP/C network in the slot where the Lighting System is allowed to send. Now the message "leaves" the Lighting System and arrives at the Steering Column Gateway. This is visualized with the curly brackets on the right. The curly brackets also show that e.g. the two boxes "CanEmuIO Task" are not the same task as they are on different nodes.

On the Steering Column Gateway the TTP/C Buffer, containing the message from the Tester, is read by the CanEmuIO task and transferred to a task which actually sends the message on the CAN bus between the Steering Column Gateway and the SMLS. Finally, after about 3.67 ms the message reaches its destination.

The Worst Case (WC) of this communication path, shown in Figure 5.7, is that the Tester sends the CAN-message just after the CanEmuIO task examines the CAN-receive-buffers and the message resides in the buffer for nearly 4ms additionally.

As the boxes in the figures have to contain a description it may look as if the transmission lasts longer than in the Table 5.1.

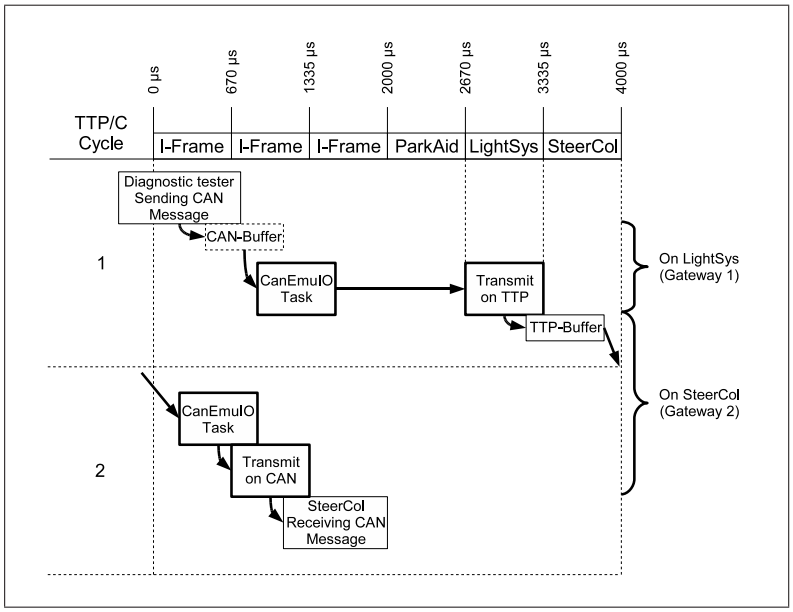

Figure 5.6: Latency: Tester to Steering Column (Best Case)

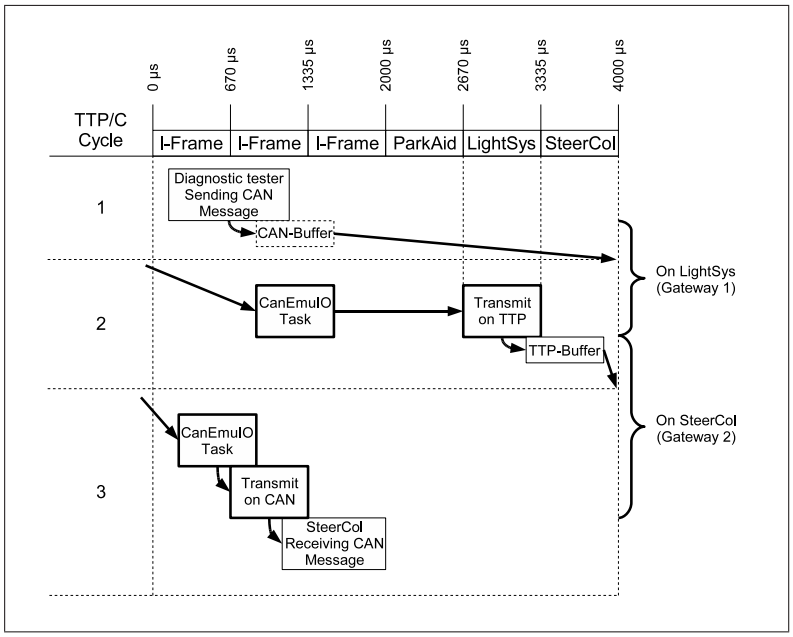

Figure 5.7: Latency: Tester to Steering Column (Worst Case)

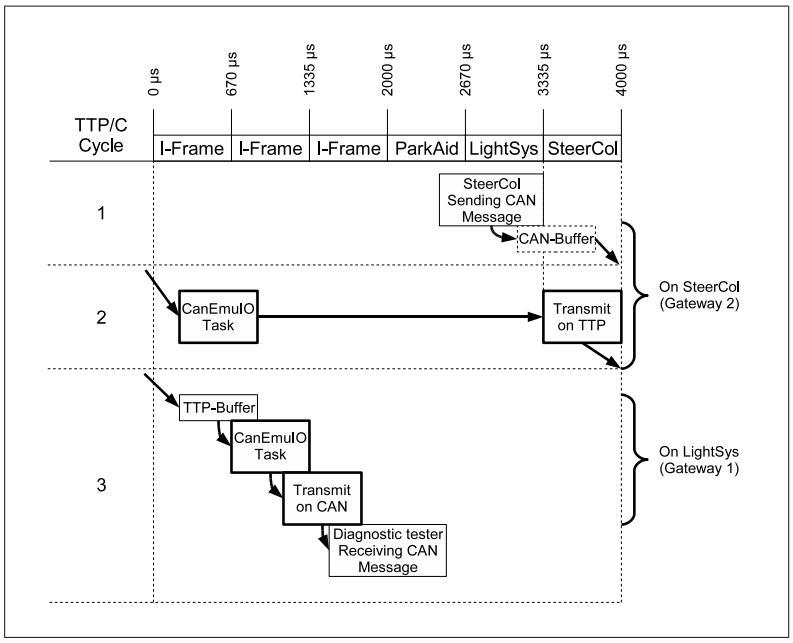

Figure 5.8: Latency: Steering Column to Tester (Best Case)

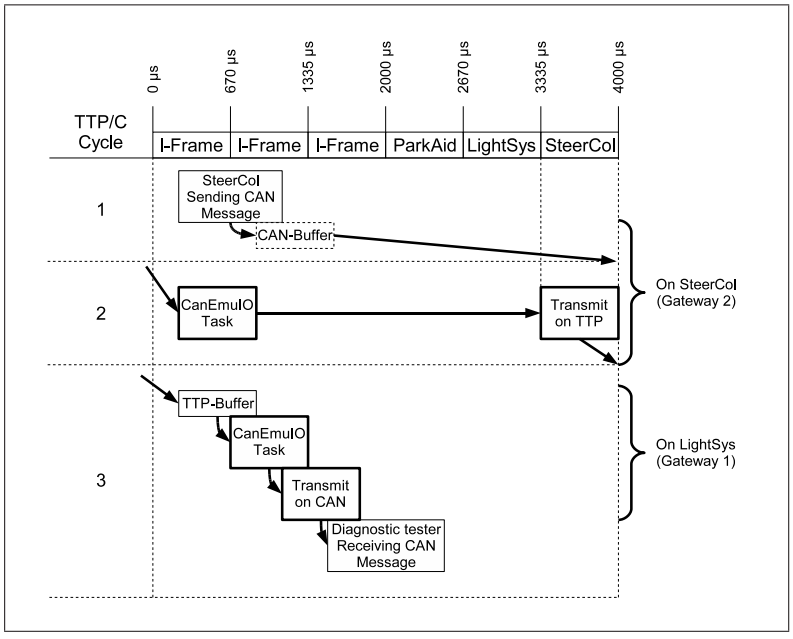

Figure 5.9: Latency: Steering Column to Tester (Worst Case)

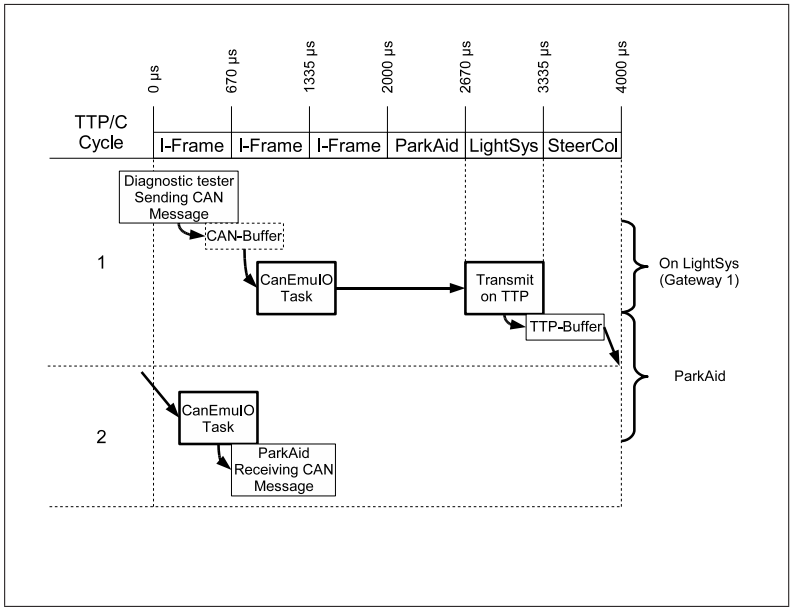

Figure 5.10: Latency: Tester to Parking Aid (Best Case)

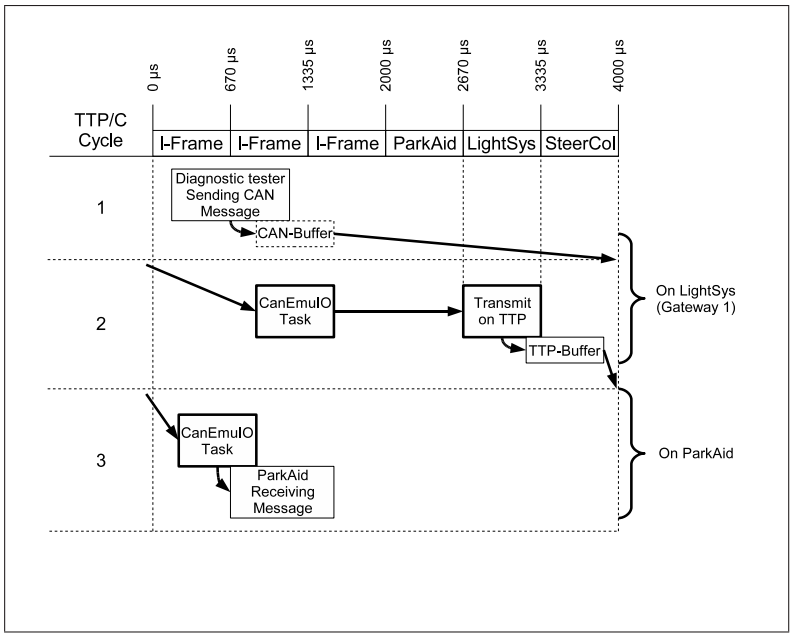

Figure 5.11: Latency: Tester to Parking Aid (Worst Case)

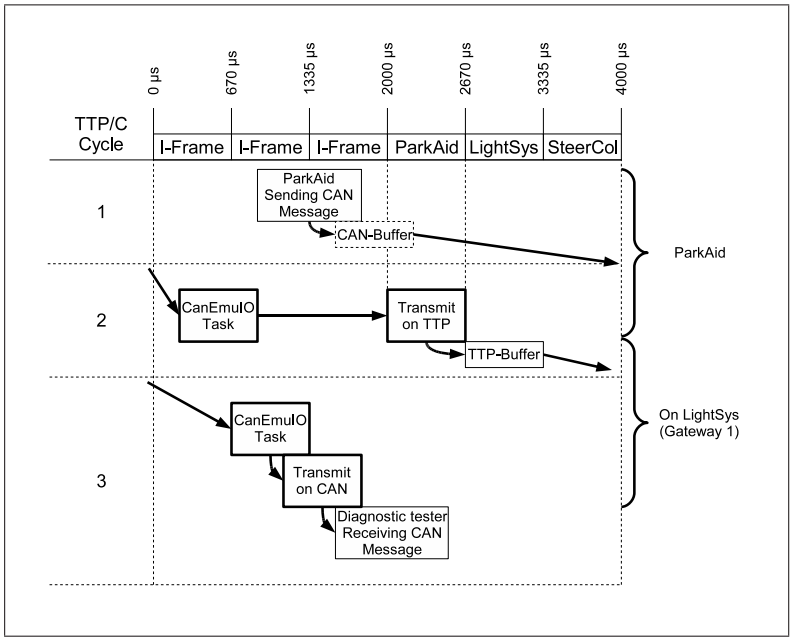

Figure 5.12: Latency: Parking Aid to Tester (Best Case)

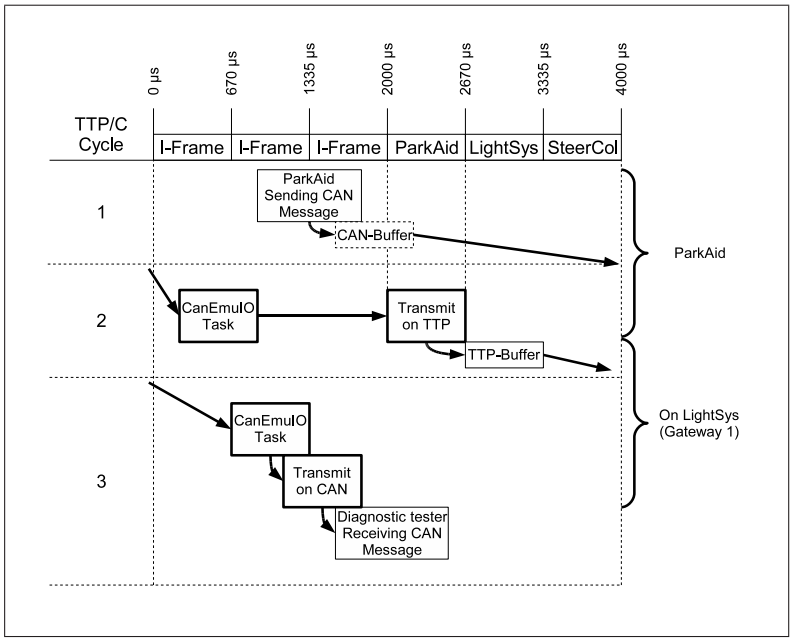

Figure 5.13: Latency: Parking Aid to Tester (Worst Case)

6 Conclusion

In the automotive domain CAN is the most widely used event-triggered communication protocol. It is suitable and efficiently used for comfort functions as well as motor control and safety enhancing functions in a car. A better approach for safety-critical applications would be a time-triggered one, because of properties like dependability and better fault isolation.

The Time-Triggered Protocol / C (TTP/C) is a well known representative of a network protocol suitable for Time-Triggered Architectures (TTAs). It is used for the creation of ultra-dependable systems for highly dependable distributed embedded real-time applications for example in avionic and railway applications.

The idea of this work was to reuse diagnosis equipment and software developed for CAN while gaining advantages from a time-triggered protocol. We use standardized layers and interfaces to allow the transfer of a legacy CAN application into a Time-Triggered Architecture (TTA). Our prototype setup also enables the usage of legacy hardware to save costs while migrating to a time-triggered system in order to increase safety and reliability of the communication subsystem. Thereby, the garage tester is able to query legacy CAN ECUs as well as time-triggered TTP/C ECUs.

Our integrated system for TT and CAN-based applications improves diagnosis by layering the Event-Triggered (ET) CAN communication on top of the dependable TTP/C. This has been demonstrated using the diagnostic protocol stack of a major European carmaker. Mechanisms facilitating better diagnosis are available in this TT protocol like media redundancy, fault isolation and global time. Besides that the model separates legacy hardware and software from the communication subsystem using Linking Interfaces (LIFs). This avoids the need for internal changes of legacy hardware or software when migrating to the time-triggered system. This property is also very important when focusing on the problem of Intellectual Property (IP).

In the implementation chapter it is shown that it is possible to migrate an existing legacy CAN application with a reasonable amount of changes into a TTA. Improvements on diagnosis are shown together with this migration. One of the diagnosis improvements utilizes the membership mechanism of the TTP/C which helps to pinpoint nodes which are e.g. not synchronized with the other nodes.

It is possible to merge a TTA with a legacy CAN network to increase dependability, improve diagnosis and save costs by reusing hardware and software. Even the whole diagnosis stack used in today's CAN ECUs can be migrated easily using our model. Safety-critical functions can be implemented and existing software and hardware can still be used to save money and continue to benefit from the experiences available on CAN.

7 Acronyms

ABS Anti-lock Brake System

ABS Antiblockiersystem

ACK Acknowledge

API Application Programming Interface

AUTOSAR automotive open system architecture

BC Best Case

CAN Controller Area Network

CNI Communication Network Interface

CPU Central Processing Unit

CSMA/CD Carrier Sense Multiple Access with Collision Detection

CSMA/CA Carrier Sense Multiple Access with Collision Avoidance

C-state Controller state

DEH Diagnostic Event Handler

DLC Data Length Code

DTC Diagnostic Trouble Code

ECU Electronic Control Unit

EPH Einparkhilfe (ger.)

ESC Electronic Stability Control

ESP Elektronisches Stabilitäts Programm

ET Event-Triggered

FCU Fault Containment Unit

FT-COM Fault-Tolerant Communication Layer

FTU Fault Tolerant Unit

HIS "Hersteller Initiative Software"

ID Identifier

IP Intellectual Property

ISO International Organization for Standardization

KWP2000 Keyword Protocol 2000

LCD Liquid Crystal Display

LED Light Emitting Diode

LIF Linking Interface

MEDL Message Descriptor List

MIL Malfunction Indicator Light

NDA Non-Disclosure Agreement

NRZ Non Return to Zero

OBD On-Board Diagnostics

OSEKtime OSEK/VDX Time-Triggered Operating System

PC Personal Computer

SDS Standard Diagnostic Services

SId Service Identification byte

SMLS Schaltermodul Lenksäule (ger.)

TDMA Time Division Multiple Access

TNI trouble not identified

TP2.0 Transport Protocol 2.0

TT time-triggered

TTA Time-Triggered Architecture

TTP Time-Triggered Protocol

TTP/A Time-Triggered Protocol / A

TTP/C Time-Triggered Protocol / C

TTP-OS Time-Triggered Protocol Operating System

UDS Unified Diagnostic Services

USDT Unacknowledged Segmented Data Transfer

WC Worst Case

Bibliography

[1] Volkswagen AG. HIS/Vector CAN Driver Specification, August 2003. Version 1.0.

[2] N. Audsley, K. Tindell, and A. Burns. The end of the line for static cyclic scheduling? In *Fifth Euromicro Workshop on Real-Time Systems*, pages 36 – 41, 1993. ISSN: 1068-3070.

[3] G. Bauer and H. Kopetz. Transparent redundancy in the time-triggered architecture. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN 2000), NY, USA*, pages 5–13, June 2000.

[4] G. Bauer and H. Kopetz. The Time-Triggered Architecture. In *Proceedings of the IEEE Special Issue on Modeling and Design of Embedded Software*, Oct. 2002.

[5] Berreth, Christensen, O'Connor, Johnson, and Kindness. Methods for protecting intellectual property. In *Northcon/96*, pages 419–424, 1996. ISBN: 0-7803-3277-6.

[6] R. Birgisson, J. Mellin, and S.F. Andler. Bounds on test effort for event-triggered real-time systems. In *Sixth International Conference on Real-Time Computing Systems and Applications. RTCSA '99*, pages 212 – 215, 13-15 Dec. 1999.

[7] V. Claesson, C. Ekelin, and N. Suri. The Event-Triggered and Time-Triggered Medium-Access Methods. In *ISORC'03*, 2003. ISBN: 0-7695-1928-8.

[8] A. Deicke. The electrical/electronic diagnostic concept of the new 7 series. In *Convergence International Congress & Exposition On Transportation Electronics, SAE*, Oct. 2002.

[9] D.S. Dodge. Gateways - 101. In *IEEE Military Communications Conference, 2001. MILCOM 2001.*, pages 532–538, oct 2001.

[10] D.S. Dodge and S. David. Gateways, A Necessary Evil? In *Simulation Interoperability Standards Organization (SISO) Fall Simulation Interoperability Workshop*, September 2000.

[11] B. Hedenetz and R. Belschner. Brake-by-wire without Mechanical Backup by Using a TTP-Communication Network. In *SAE World Congress, Detroit Michigan.*, 1998.

[12] The homepage for all CAN related topics by Robert Bosch GmbH. http://www.can.bosch.com. Internetaddress.
Related to this work is the free CAN-specification available on this page at: http://www.can.bosch.com/docu/can2spec.pdf.

[13] Homepage of the Automotive Open System Architecture. http://www.autosar.org.

[14] The homepage of the joint project OSEK/VDX. http://www.osek-vdx.org. Internetaddress.

[15] Homepage of the TTA-Group, a group for exchanging cross-industry experience with TTP. TTP Specification 1.1. http://www.ttagroup.org.

[16] SAE Standard J-1850 Class B Data Communications Network Interface. http://www.sae.org.

[17] R. Isermann, R. Schwarz, and S. Stolzl. Fault-tolerant drive-by-wire systems. In *IEEE Control Systems Magazine*, pages 64–81, October 2002.

[18] H. Kopetz. *Real-Time Systems*. Kluwer Academic Publishers, 1997.

[19] H. Kopetz and G. Bauer. The time-triggered architecture. *IEEE Special Issue on Modeling and Design of Embedded Software*, January 2003.

[20] H. Kopetz and G. Grünsteidl. *TTP* - A Time-Triggered Protocol for Fault-Tolerant Real-Time Systems. In *Proceedings of the 23rd International Symposium on Fault-Tolerant Computings*, 1993.

[21] H. Kopetz and N. Suri. Compositional Design of RT Systems: A Conceptual Basis for Specification of Linking Interfaces. In *Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, ISORC 03*, 2003.

[22] J.C. Laprie, A. Avižienis, and B. Randell. Fundamental concepts of dependability, 2001.

[23] R. Maier. Event-Triggered Communication On Top Of Time-Triggered Architecture. In *Digital Avionics Systems Conference*, 2002.

[24] N. Navet. Controller Area Network [automotive applications]. *Potentials, IEEE*, pages 12 – 14, Oct-Nov 1998. ISSN: 0278-6648.

[25] R. Obermaisser. CAN Emulation in a Time-Triggered Environment. In *Proceedings of the 2002 IEEE International Symposium on Industrial Electronics (ISIE)*, volume 1, pages 270–275, 2002.

[26] R. Obermaisser. *Event-Triggered and Time-Triggered Control Paradigms*. Springer, 2005.

[27] http://www.iso.org. ISO 9141 - Road vehicles - Diagnostic systems - Requirements for interchange of digital information, 1989.

[28] http://www.iso.org. ISO 14230-2 - Road vehicles - Diagnostic systems - Keyword Protocol 2000 - Part 2: Data link layer, 1999.

[29] http://www.iso.org. ISO 14230-3 - Road vehicles - Diagnostic systems - Keyword Protocol 2000 - Part 3: Application layer, 1999.

[30] http://www.iso.org. ISO 14229 - Road vehicles - Diagnostic systems - Part 1: Diagnostic services, 2001.

[31] http://www.iso.org. ISO 11898 - Road vehicles – Interchange of digital information – Controller Area Network (CAN) for high-speed communication, 2003.

[32] http://www.iso.org. ISO 11898-1 - Road vehicles - Controller Area Network (CAN) - Part 1: Data link layer and physical signalling, 2003.

[33] http://www.iso.org. ISO 15765-2 - Road vehicles - Diagnostics on Controller Area Networks (CAN) - Part 2: Network layer services, 2003. Transport Protocol.

[34] S. Punnekkat, H. Hansson, and C. Norström. Response time analysis under errors for can, 2000. Märlardalen University, Sweden. ISBN 0-7695-0713-1/00.

[35] D. Riezler. Funktionsbeschreibung CAN Emulator. Technical report, TTTech Computer AG, 2002.

[36] D. Riezler. Integrating CAN-based Legacy Applications in the TTA. Master's thesis, Technical University Vienna, 2004.

[37] J. Rushby. A Comparison of Bus Architectures for Safety-Critical Embedded System, 2003. NASA Langley Research Center.

[38] VW-Konzernlastenheft. KeyWord Protokoll 2000 - Rahmenbeschreibung der Dienste auf der K-Leitung, 2000.