Advanced Gateways in Automotive Applications

vorgelegt von
Diplom-Ingenieur
Tobias Lorenz
aus Berlin

von der Fakultät Fakultät IV – Elektrotechnik und Informatik
der Technische Universität Berlin
zur Erlangung des akademischen Grades

Doktor-Ingenieur
- Dr.-Ing. -

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender:	Prof. Dr.-Ing. Georg Böck, TU-Berlin
Berichter:	Prof. Dr. rer. nat. Otto Manck, TU-Berlin
Berichter:	Prof. Dr.-Ing. Reinhold Orgelmeister, TU-Berlin

Tag der wissenschaftlichen Aussprache: 15.07.2008

Berlin 2008

D 83

Acknowledgment

While I was working on my thesis over the last few years, I was fortunate enough to have had many fruitful and inspiring discussions in a congenial atmosphere with my colleagues both at the Technical University of Berlin as well as at Bosch, the business company I have been working for since 2004.

I am also deeply indebted to Prof. Otto Manck, my supervisor and mentor. During my time at his microelectronics division, he supported me with fruitful discussions and sound advice on research and technical issues, as well as on a personal level.

I would also like to thank Prof. Golze and Prof. Weinerth, which unfortunately were not able to attend the exam, Prof. Orgelmeister and Prof. Manck as the examiners and Prof. Böck as the chairman of the exam.

A special word of thanks goes also to the complete AE/EIP5 group at Bosch, which not only supported my work on the technical side, but more over we shared much of our recreational time. I also with to express my sincere thanks to Jan Taube for many discussions we had on the structure and implementation, Markus Ihle for the project management and Kay Hammer for helpful detailed information. The friendships that grew in the last years will last beyond my time in Reutlingen.

Furthermore, I would also like to thank my father, Jürgen Lorenz, and my uncle, Gert Lorenz, as neutral reviewers of my work and again Markus Ihle and Jan Taube regarding the scientific contents itself.

Without the steady support of my family during the complete time of education and studies, I never would have been able to do my dissertation in a relatively short time.

My special thanks also goes to all the other people, whom I have not directly named, but who also helped me both directly and indirectly during this period of my life.

Declaration

Abstract

Today's devices in automotive environments are linked by several independent networks, such as CAN, MOST and in the future FlexRay. The interconnection technologies between these networks is one of the fastest developing areas in automotive electronics. In recent years the improved capabilities enabled many new features, e.g. driver assistance systems.

For the interconnection, complex gateway systems are necessary. Most of them are implemented as a System-On-Chip with high-speed CPUs and large buffer RAMs to provide just this functionality in software. With increasing throughput and number of interfaces on the gateway, this leads to high latency and jitter. Hardware based solutions can provide gateway functionality with hard real-time requirements and, additionally, circumvent the given problems.

The developed gateway concept is versatile, thus usable by different network types. It offers gateway functionality in hardware without the need to have a powerful CPU or large buffer RAM. CPU access to the underlying network interfaces is still possible to enable electronic control units (ECUs) to have further functionality.

For the gateway configuration, many non-standardized description formats are available. Unfortunately data exchange between different formats is inherently error-prone and time-consuming. A common description format is necessary and is found in the Field Bus Exchange Format (FIBEX).

On the other side gateway implementations have proprietary internal formats in which they store their configuration data. An optimal gateway toolchain should be based on the FIBEX configuration data of all connected networks and translate the routing information directly into the internal format of the target implementation.

The verification of complex gateway systems is problematic. Programming a remaining bus simulation is usually done manually, takes a long time and almost never covers the complete range of possible gateway mappings. Having an automated verification configuration generator was necessary. Such a regression test environment results in error logs containing patterns, which helps to find design or configuration bugs. Detailed statistics about latency and jitter of every mapping may be generated in parallel. Using FIBEX as common configuration format for the gateway and for the verification environment can eliminate inconsistencies.

Zusammenfassung

Heutige Steuergeräte in Kraftfahrzeugen sind über eine Reihe von unabhängigen Netzwerken, wie CAN, MOST und zukünftig FlexRay, miteinander verbunden. Die zunehmende Verknüpfung stellt derzeit eine der am stärksten wachsenden Bereiche in der Automobilelektronik dar. In den vergangenen Jahren sind durch die dadurch gestiegenen Möglichkeiten viele neue Funktionen entstanden, insbesondere im Bereich der Fahrerassistenzsysteme.

Für die Verbindung von Netzwerken sind komplexe Gateway-Systeme notwendig. Derzeit sind diese System-On-Chips mit schnellen CPUs und großen RAMs ausgestattet, speziell um die Gateway-Funktionen in Software zu realisieren. Mit der Zunahme von Datendurchsatz und Schnittstellenanzahl im Gateway, führt dies mittlerweile zu nicht mehr tragbarem Latenz- und Jitterverhalten. Hardwarebasierte Lösungen können die Gateway-Funktionen mit harten Echtzeitanforderungen erfüllen und die bekannten Probleme verhindern.

Das entwickelte Gateway-Konzept ist vielseitig und dadurch mit verschiedenen Netzwerktypen nutzbar. Es ermöglicht die Verlagerung der Gateway-Funktion in Hardware, ohne die Notwendigkeit einer schnellen CPU und eines großen Puffer-Speichers. Der CPU-Zugang zu den darunter liegenden Netzwerkschnittstellen ist weiterhin möglich, um weitere Funktionen im Steuergerät zu ermöglichen.

Für die Gateway-Konfiguration gibt es viele nicht-standardisierte Konfigurationsformate. Das macht den Austausch von Daten zwischen verschiedenen Formaten fehleranfällig und zeitaufwendig. Ein einheitliches Beschreibungsformat war notwendig und fand sich im Field Bus Exchange Format (FIBEX).

Auch intern weisen Gateways verschiedene proprietäre Konfigurationsformate auf. Die entwickelte Gateway-Toolchain ist daher in der Lage eine FIBEX-Konfiguration mit allen darin enthaltenen Konfigurationen zu laden und diese in das interne Format des Gateways zu übersetzen.

Die Verifikation eines solchen komplexen Gateway-Systems ist problematisch. Die Programmierung einer Restbussimulation wird üblicherweise manuell vorgenommen, was viel Zeit erfordert und fast nie den kompletten Bereich möglicher Gateway-Funktionen testet. Ein Tool zur automatisierten Erzeugung von Konfigurationen für die Verifikationsumgebung wurde entwickelt. Diese Regressionstestumgebung liefert Fehlermeldungen, deren Muster helfen können Fehler in Hard- und Software zu finden. Detaillierte Statistiken über Latenz und Jitter jeder Gateway-Funktion werden parallel erhoben. Die Benutzung von FIBEX als einheitliches Format für Gateway und Testumgebung hilft dabei Inkonsistenzen zu vermeiden.

Contents

1 Introduction **10**

2 Communication Networks **12**

2.1 Basic Concepts 12

2.1.1 Physical Transmission 12

2.1.2 Media Access Control 13

2.1.3 Logical Link Control 14

2.1.4 Upper Layers 16

2.1.5 Communication Controller 16

2.2 Controller Area Network 18

2.2.1 Protocol Description 18

2.2.2 Higher Layer Protocols 19

2.2.3 Communication Controller 20

2.2.4 Configuration Format 21

2.3 Time-Triggered Controller Area Network 21

2.3.1 Protocol Description 22

2.3.2 Communication Controller 23

2.4 FlexRay 24

2.4.1 Protocol Description 24

2.4.2 Communication Controller 25

2.4.3 Configuration Formats 27

2.5 Media Oriented System Transport 27

2.5.1 Protocol Description 28

2.5.2 Communication Controllers 29
2.6 Network Architectures 30

3 Current Gateways 32
3.1 Requirements 32
3.2 Tasks 34
3.3 Gateway-Optimized Microcontroller 36
3.4 Software Gateways 38
3.4.1 K2L Gateway 39
3.4.2 Automotive Open System Architecture 39
3.4.3 X2E Gateway 41
3.5 Configuration Data Formats 41
3.5.1 Field Bus Exchange Format 42
3.5.2 AUTOSAR System Constraint Templates 43
3.6 Problems and Issues 43

4 Advanced Gateway Architectures 45
4.1 Advanced Gateways 45
4.1.1 Requirements 45
4.1.2 CAN-CAN Gateway 46
4.1.3 Multi-Protocol Gateway 52
4.1.4 Software Gateway 59
4.2 Configuration Toolchain 61
4.2.1 FIBEX Engine 62
4.2.2 Import/Export of CANdb++ DBC databases 63
4.2.3 Configuration of Communication Controllers 64
4.2.4 Gateway Configuration 64
4.2.5 Assembler 65
4.2.6 Verification and Tests 68
4.2.7 Assembler Instruction Simulator 69
4.3 Verification Environment 70

4.3.1 Gateway Executive Model 70
4.3.2 Automatic Test Generator 73

5 Implementation and Results 75

5.1 Gateway Development Hardware 75
5.1.1 Development Board 75
5.1.2 Physical Layer Board 76
5.1.3 Clock Frequencies 77
5.2 CAN-CAN Gateway Comparisons 78
5.2.1 Instruction Set 78
5.2.2 Comparison of Implementations 78
5.2.3 Description of Test System 79
5.2.4 Implemented Tests 82
5.2.5 Results 85
5.3 Multi-Protocol Gateway Implementation 85
5.3.1 RAM Configuration 85
5.3.2 Instruction Set 86
5.3.3 Finite State Machines 87
5.3.4 Gate Count 92
5.4 Toolchain Optimization Results 93
5.4.1 Optimization Levels 93
5.4.2 Code Size and Execution Time 93
5.5 Verification Environment Implementation 94
5.5.1 Test Description 96
5.5.2 Test Scenarios 98
5.5.3 Implementation Requirements 99
5.5.4 Results 100
5.5.5 Future Improvements 102
5.6 Gateway Evaluation in OEM Environments 102
5.6.1 Vector RAM Usage Estimation 103
5.6.2 Instruction RAM Usage Estimation 104

5.6.3 GCU Utilization Estimation 105
5.6.4 Routing Latency 108
5.6.5 Worst Case Execution Time 108
5.6.6 Evaluation of a High-Class Automotive Network 108
5.6.7 Evaluation of a Mid-Class Automotive Network 109
5.6.8 Conclusion 111

6 Summary and Outlook 112

Glossary 115

List of Figures 119

List of Tables 122

Bibliography 123

Curriculum Vitae 134

Chapter 1

Introduction

The need for data transparency and information exchange within the overall in-car network has increased with the continuous improvement of electronic systems [1]. Just one of the numerous examples is the electronic stability program (ESP), monitoring the drive dynamics of the vehicle and taking control over engine management and brake systems when the vehicle is in danger of tipping over or skidding. A low latency gateway system is needed to connect the networks of both systems.

Nowadays most gateways are software gateways, based on standard communication controllers and a high performance CPU running appropriate software. This is a flexible concept, but on the other hand the performance is affected, since the hardware structure of a usual microcontroller is not optimized for gateway operations and the electronic control unit (ECU) often has to do other tasks as well. As the number of interfaces and the total bandwidth of a gateway increases continuously, software gateways will soon become bottlenecks, too slow to handle all incoming traffic in the required time. Currently these gateways are using CPUs with a clock frequency of about 150 MHz or even more [67], causing other problems, like high power consumption [75] and high electro-magnetic emission/radiation.

For the described network environment such a low-latency gateway system has been implemented in hardware. It currently allows processing of frame and signal mappings between multiple CAN and FlexRay networks.

The major part of the configuration of such a gateway system is defining the routing information for frames and signals between the channels. Before the emergence of standardized gateway description formats, the configuration data was necessarily specific for a gateway product and prohibited easy data exchange with other applications. Checking configurations was time-consuming and error-prone.

A common description format was necessary and is found in the Field Bus Exchange Format (FIBEX). FIBEX is the upcoming XML-based data exchange standard with the capability to describe complete car networks, composed of different communication protocols. Currently full support for CAN, LIN and FlexRay networks is provided. The support of further protocols is recently under development. FIBEX also covers the configuration of gateway ECUs and defines operation tolerances in terms of relative or absolute timings.

Based on these FIBEX files a gateway configuration toolchain was developed that interprets the network description and produces configuration images for both, the gateway and the communication controllers. An implemented abstraction layer allows the adaptation to different input formats in the future. The code can be generated optimized for execution time or memory size.

Safe operation of a gateway system is crucial. This has to be proven by an intensive test plan. The usual approach is to manually implement a test environment using a remaining bus simulation. Manual programming has several disadvantages. Checking the complete range of possible gateway mappings and test parameters is very expensive. Also manual programming can lead to errors not only in the implementation of the gateway function in hardware and software, but additionally in the test environment. This is especially true when changes to the gateway configuration also affect the test environment. Manual changes are always error-prone and often result in inconsistencies between gateway and test environment. These disadvantages can be eliminated by using tools to automate most of these tasks and by referring to a common source of information for the gateway and test configuration.

A verification environment was developed for black box testing of arbitrary gateways. It includes checks of the gateway mappings and therefore tests the gateways data consistency and timing behavior. A test generator has been implemented as an enhancement to the existing gateway configuration toolchain. This generator allows the usage of FIBEX files as a common source of information for both, the configuration of the gateway and the test environment. The test environment is implemented to test and compare different gateway implementations including the developed hardware gateway in realistic environments.

Chapter 2

Communication Networks

Gateways are based on communication networks. Some of the important protocols in automotive and industrial applications are explained in this chapter, along with their controllers, transport protocols and configuration formats [54] as available.

2.1 Basic Concepts

Each communication protocol can be explained by describing its specific function in the ISO/OSI reference model. Additionally the characteristics of their communication controllers (CC) can be described.

ISO/OSI Reference Model OSI stands for Open System Interconnection, an initiative that tries to standardize network communication. The OSI Reference Model is nowadays the most often model used to describe communication networks and protocols. Meanwhile this model is standardized in DIN and ISO/IEC 7498-1 [10]. The model is shown in table 2.1.

2.1.1 Physical Transmission

The physical layer describes the media, signal and binary transmission characteristics. The connection to the upper layer is most often implemented in a separate physical layer transceiver chip (PHY).

Media The most common media types are physical coupling (copper or fiber optic), radio bands or infrared.

Layer	Function
7	application
6	presentation
5	session
4	transport
3	network
2.5	logical link control
2	media access control
1	physical

Table 2.1: ISO/OSI Reference Model

Signal Codings Different signal codings are used to encode the signal on the wire, e.g. bit stuffing, NRZ, BiPhase or Manchester. They can reduce the frequency of the bit stream or enhance clock recovery from the signal.

Mapping Mechanism Mapping mechanisms can be used additionally to enhance clock recovery, transmission safety or providing DC freeness. This is done by adding additional bits to the data stream, but not by using calculations to generate these bits. Instead coding tables are used, such as 4B5B and 8B10B. In case of 4B5B a group of 4 source bits can be used to have a group of 5 target bits. As not every target combination is valid, this enhances the error detection or provide space for additional control characters for the Logical Link Control layer.

2.1.2 Media Access Control

The data link layer can be divided into two sublayers named the Media Access Control (MAC) and the Logical Link Control (LLC) layers. This separation is defined in IEEE802 for Ethernet.

The MAC sublayer determines who is allowed to access the physical media at any given time.

CSMA The most often used media access protocol is Carrier Sense Multiple Access (CSMA). Carrier Sense describes the fact that a transmitter listens for a carrier wave to prevent sender interference. Multiple Access describes the fact that multiple nodes send and receive on the medium. Transmissions by one node are generally received by all other nodes using the medium.

In pure CSMA systems collisions can still occur, when two nodes start sending nearly at the same time. Receivers then only see frame errors and don't acknowledge the transmissions, causing the transmitters to timeout and try again. One example using pure CSMA is the ALOHA network.

To solve this problem multiple additions have been developed.

CSMA/CD Carrier Sense Multiple Access With Collision Detection (CSMA/CD) is a method to reduce timeouts on collisions. This is not possible with all media and requires special PHYs. When a collision is detected the sender stops immediately and repeats the attempt after a random time. This is the technology used in every Local Area Network (LAN) using Ethernet without switches.

CSMA/CA Carrier Sense Multiple Access With Collision Avoidance (CSMA/CA) goes a step further as it tries to avoid collisions at all. Prior to sending, the network node informs the network of the following transmission. This is the technology used in every Wireless Local Area Network (WLAN). Still collisions can occur in the preceding information or transmission stage.

CSMA/CD+CR Carrier Sense Multiple Access With Collision Detection and Collision Resolution (CSMA/CD+CR) is collision free. This mechanism is often implemented by using priority based arbitration. When multiple nodes want to transmit, only the node with the highest priority is allowed to send, all other have to wait. This mechanism is used in the Controller Area Network (CAN).

2.1.3 Logical Link Control

The Logical Link Control (LLC) as defined in IEEE 802.2, can modify data on transmit or receive, f.e. to add and remove additional source and target address information. Optionally flow control, detection and retransmission of dropped packets (data integrity) and acceptance filtering can be done. Most Ethernet bridges and switches work on the LLC.

Data Integrity Data integrity always means to put additional bits of redundant information to the data stream to detect or even correct erroneous data.

Parity The simplest form of error detection is using parity bits. Based on the number of high or low bits and depending on even or odd parity, an additional bit is added to the data stream. Of course the error detection only works with an odd number of wrong bits. More wrong bits will make all results meaningless. This also means, that multiple wrong bits can have a correct parity and the errors are not detected at all.

CRC The Cyclic Redundancy Check (CRC) provides a larger number of bits to detect errors. It is a type of a hash function to generate a checksum using polynomial division. The relevant data bits and the number of CRC bits are flexible.

Message- and Address-oriented Communication A transmission is a broadcast on the network media. Depending on the filter mechanism on the receiver side, a communication can be message-oriented or address-oriented.

Message-oriented Communication means in this context, that a Message ID is transmitted with every frame and is used to filter the messages on the receiver side. The receiver selects, which broadcasts are relevant for this network node. Most of the time the Message ID is sufficient to describe the contents. Then transport protocols are not necessary. An example for message-oriented communication is CAN.

Address-oriented Communication means that every message is provided with a Sender or Receiver ID. The sender selects, which nodes should receive the message. As the message contents are not described by the IDs, further transport protocols are necessary. An example for message-oriented communication is Ethernet and IP as transport protocol.

Message Filtering In either case filtering can be done in hardware or software, mainly depending on the required functionality and the available communication controllers and CPUs.

Message Trigger Mechanism Depending on the message trigger mechanism, the protocol can be event-triggered or time-triggered.

Event-Triggered Protocols are protocols, where new messages are sent as soon as the wire is free. This allows low latency communication as long as the network utilization is kept to a minimum. Also event-triggered communication provides the maximum usage of bandwidth on collision free networks. An arbitration is done with each transmission to select which message will be the next. Low priority messages can lose the arbitration, when the bus load is high. The sending delays for these messages are then not deterministic any more.

Time-Triggered Protocols are protocols, where messages are send in fixed time slots defined in a schedule table. This is called Time Division Multiple Access (TDMA). When the time slots can be flexible in duration, this is sometimes referred to as Flexible Time Division Multiple Access (FTDMA), e.g. FlexRay. Arbitration and prioritization are still possible in every slot, e.g. TTCAN. Latency times are usually longer, as the time between a message send request and the actual transmission depends on the message position in the schedule table. Time-Triggered Protocols are predestined for real-time applications, as the message transmission can be made deterministic.

2.1.4 Upper Layers

Most often upper layers are implemented in software and are used in address-oriented networks.

Network Layer Path determination and logical addressing is provided by the network layer. In Ethernet the Internet Protocol (IP) is an example of such a network layer protocol.

Session Layer Mechanisms for host-to-host communication are provided by the session layer. One example is the Transmission Control Protocol (TCP/IP), which is based on the Internet Protocol.

Presentation Layer The presentation layer defined the representation of data. Encryption is a possible presentation provided in this layer. XML, the Extensible Markup Language, is another one.

Application Layer Network processes, which communicate using the network, are contained in the application layer.

2.1.5 Communication Controller

Communication controllers can be grouped into two main categories based on their mechanism of storing messages for receiving and transmitting as shown in figure 2.1.

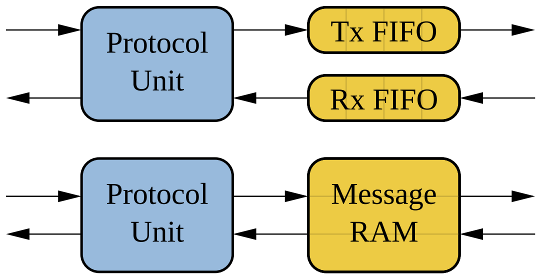


Figure 2.1: FIFO- / MRAM-based Communication Controller

Message RAM Message RAM (MRAM)-based communication controllers have their own RAM in which they store all relevant messages and control information. Every message can be accessed whenever it is necessary and in no specific order. As the Message RAM needs to be accessible from the network side and from the host side for loading and storing messages, access protection must be implemented.

Memory Mapped Access In memory mapped implementations, where the Message RAM is mapped in the address space of the host CPU, mutual exclusion (Mutex) bits are necessary. As long as this locking bit for a specific message is set, the other side cannot access the message. This can lead to conflicts, especially when having time-triggered communication controllers, which need to have full access at any time.

Message Handler Access In this implementation direct access to the Message RAM is not possible. Instead a message handler is used to arbitrate between concurrent accesses to the Message RAM from the network and host side. At least one message buffer register is used to signal requests for load and store operations to the message handler from the host side.

FIFO Opposite to MRAM-based communication controllers, FIFO-based controllers store their messages in a FIFO buffer. A prioritization for message priority or message age is possible, but very uncommon. Therefore only the oldest message can be read or send at one time.

Basic A basic communication controller is a sub-type of a FIFO-based communication controller with only limited FIFO sizes, normally one or two messages per direction. These controllers are small in size, but require a high performance CPU in order not to lose any messages on receive and possible wait before transmitting new messages.

DMA All types can be enhanced by a DMA controller to transfer messages between the system memory and the internal storage. This reduces data transfers done by the CPU significantly.

2.2 Controller Area Network

The Controller Area Network (CAN) is a serial communications protocol developed by Robert Bosch GmbH, efficiently supporting distributed real time control with a very high level of security [30].

Its domain of applications ranges from high speed networks to low cost multiplex wiring. For example in automotive electronics engine control units, sensors, anti-skid-systems, e.g. are connected using CAN. At the same time it is cost effective to install CAN into vehicle body electronics, such as lamp clusters, to replace the wiring harness otherwise required.

DeviceNet DeviceNet [26] is a CAN based Layer 7 protocol, which was originally developed by Rockwell Automation, former Allen Bradley. Operation of the DeviceNet is based on an object-oriented communication model. DeviceNet is maintained by Open DeviceNet Vendor Association (ODVA). It is mainly used in industrial applications, especially in factory automation.

Further Specifications Further interesting specifications are ISO 11783 (ISOBUS) for agricultural engineering, ISO 11992 for trucks and trailers, NMEA 2000 [22] for maritime electronics, CANaerospace [19] for avionics and space technology and CanKingdom [6] for machine control.

2.2.1 Protocol Description

CAN is best described using the ISO/OSI reference model.

Physical Layer The most common media type for CAN is twisted pair [31, 38]. Further derivatives also work over single wires [9], optical wires and radio bands [39]. The bit rates are normally between 5 kBit/s and 1 MBit/s.

MAC Layer On the MAC layer CAN uses NRZ encoding with bit stuffing. Multi master access to the media is provided by the CSMA/CD+CR protocol. Collision avoidance is done by bitwise arbitration, where the arbitration field is used to define a static priority. The mechanism is therefore sometimes referred to as CSMA/BA for Bitwise Arbitration.

LLC Layer The controller automatically tries to retransmit corrupted messages or messages which lost arbitration, as soon as the bus is idle again. A distinction between temporary errors and permanent failures of nodes is made and defect nodes switched off autonomously.

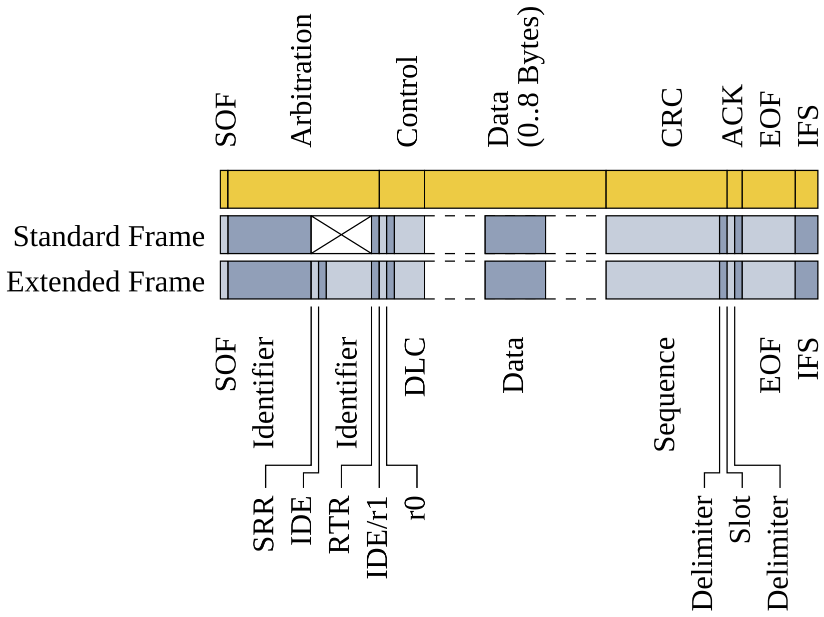


Figure 2.2: CAN Standard and Extended Frame Format

The CAN protocol version 2.0B defines message identifiers to have 11-Bit or 29-Bit depending on standard or extended frame format (see figures 2.2). Messages can have a payload of up to 8 bytes.

2.2.2 Higher Layer Protocols

The CAN protocol is specified in ISO 11898. Several more standards from transport protocols to higher layer specifications are based on this and have been developed towards specific application areas.

CANopen The CANopen [2] profile family specifies standardized communication mechanisms and device functionalities. CANopen is maintained by CAN in Automation (CiA). Application areas are industry automation and embedded applications.

SAE J1939 SAE J1939 [40] is a communication protocol based on CAN for real-time data exchange between electronic control units (ECUs) in the area of commercial vehicles.

2.2.3 Communication Controller

Most CAN controllers are either implemented as basicCAN controllers or as fullCAN controllers [3, 4]. BasicCAN controllers usually have one send buffer and one receive buffer, sometimes also with shadow buffers. The message filter mechanism and remote frame support is very limited. FullCAN controllers instead have multiple send and receive buffers and are often implemented using a Message RAM with full support for message filtering and remote frame support. Some variants already allow the connection to multiple CAN channels [18].

Overview C_CAN The most common Message RAM based CAN controller is the Bosch C_CAN shown in figure 2.3. The C_CAN [32] is a single channel CAN module that can be integrated as stand-alone device or as part of an ASIC and is written in VHDL. It consists of the components CAN Core, Message RAM, Message Handler, Control Registers and a Module Interface. The Control Registers also include the CPU Interface Control Registers (CPU IFC Registers) for message transfers from and to the Message RAM.

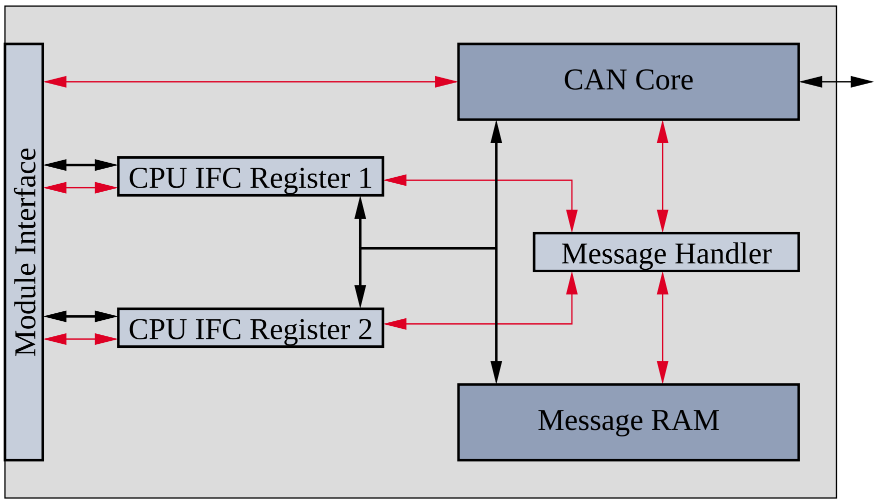


Figure 2.3: C_CAN Block Diagram

CAN Core The CAN Core implements the functionality required to perform the serial communication on the CAN bus according to the protocol specification.

Message RAM For the communication on a CAN network, individual message objects can be (pre-)configured. The message objects and related configuration data are stored in the Message RAM with 32 message objects. Successors of the C_CAN module can be given different numbers of message objects at synthesis time, such as 16, 32, 64, 128.

Message Handler All functions concerning the handling of messages are implemented in the Message Handler. This includes the transfer of messages between the CPU Interface Control Registers, CAN Core and Message RAM, acceptance filtering and the handling of transmission requests and interrupts.

Interface Registers Two sets of CPU Interface Registers are used for the data transfer between the CPU's peripheral bus and the Message RAM. They consist of the complete data, header and control information used in the C_CAN module. All host CPU accesses to messages in the Message RAM are made through the Interface Registers.

2.2.4 Configuration Format

From all the CAN-supporting configuration formats, only one is widely used.

CANdb++ The most common configuration format for CAN is CANdb++ from Vector-Informatik [50, 51], also known as DBC files. CANdb is able to define messages, signals and signal groups. Each information and signal state can be given a name and a comment. Additional information can be defined for physical units and linear conversion formula. Unfortunately there was never a public specification of this format. Only a CANdbLib Class Library (DLL) for Microsoft Visual C++ [52, 53] is available.

2.3 Time-Triggered Controller Area Network

Time-Triggered CAN (TTCAN) is a higher level protocol having a Time Division Multiple Access (TDMA) mechanism on top of the event-triggered CAN protocol.

As TTCAN is based on CAN, only the additional protocol definitions and communication controllers are described in the following sections.

2.3.1 Protocol Description

For Time-Triggered networks a global time information must be available to all connected nodes.

Reference Message In TTCAN this global time is provided by a reference message send by one time master (and up to seven potential time masters). The slaves synchronize to this reference message.

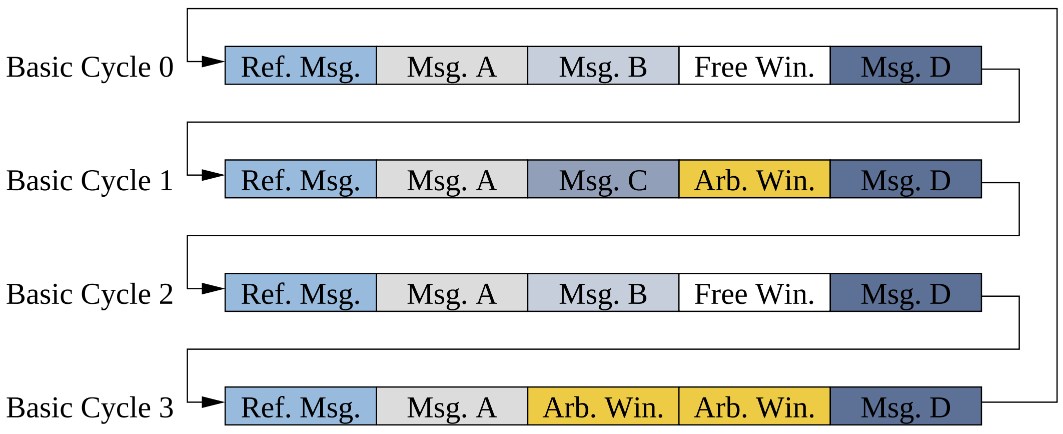


Figure 2.4: TTCAN Matrix Cycle

Transmission Columns Each reference message starts a basic cycle with a constant number of transmission columns. These transmission columns can be used for reference messages, time triggered windows, arbitration windows or can be free windows. The arbitration windows are available for event-triggered communication.

Matrix Cycle A frame can have a repeat factor higher then one, thus doesn't need to be sent in every basic cycle. Therefore up to 64 different basic cycles can be defined. All basic cycles together are called a matrix cycle.

Specification TTCAN is specified in ISO 11898-4 in addition to the CAN protocol itself.

2.3.2 Communication Controller

The TTCAN module (also provided by Bosch as VHDL code to be synthesized in FPGAs and ASICs) is based on the described C_CAN module. Only two functional blocks, (see Figure 2.5) the Trigger Memory and the Frame Synchronization Entity, have been added.

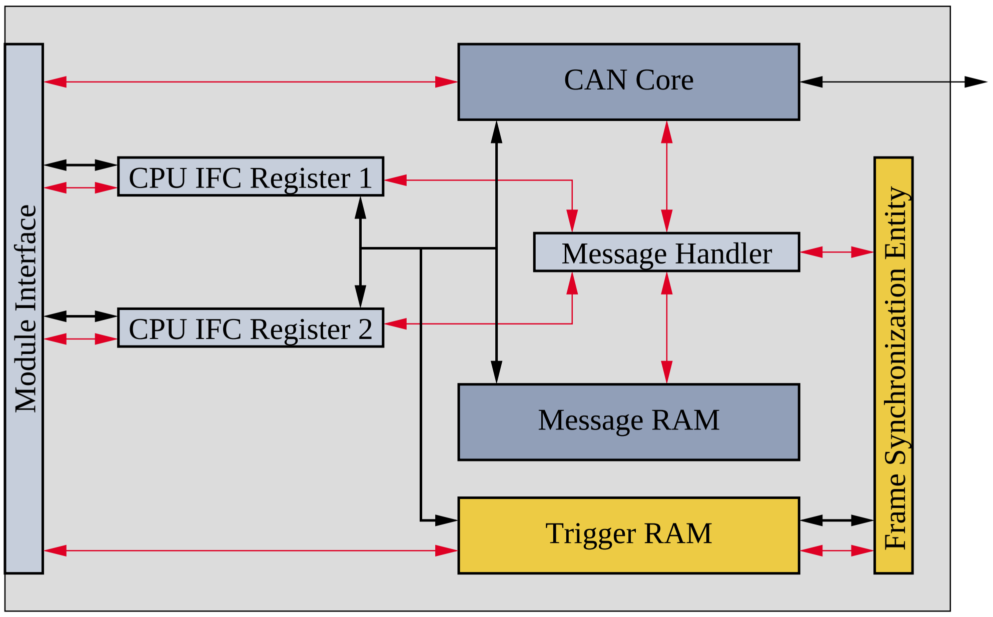


Figure 2.5: TTCAN Block Diagram

Trigger Memory The Trigger Memory stores the time marks of the system matrix that are linked to the messages in the Message RAM. The data is provided to the Frame Synchronization Entity.

Frame Synchronization Entity The Frame Synchronization Entity is the state machine that controls the time triggered communication. It synchronizes itself to the reference message on the CAN bus, adjusts the cycle time and generates Time Triggers.

Interface Register The TTCAN module also uses two interface register sets to transfer the data between the Message RAM and the CPU. Both can be accessed independently from each other. The function of the registers is identical with the exception of the access from the second Interface Register to the Trigger Memory in the initialization process.

2.4 FlexRay

The FlexRay protocol aims to be fast and redundant. It was developed towards safety and fault-tolerant systems, e.g. for the use in X-by-Wire applications (Brake-, Drive- and Steer-by-Wire) [29, 7].

2.4.1 Protocol Description

The FlexRay protocol is derived from several other protocol standards, especially ByteFlight [27] and TTP/C [49] and has therefore many similarities.

Physical Layer Two physical layer transceivers [28] connect the communication controller to two channels of twisted pair wires for redundancy. Three different bit rates are normally used: 2.5 MBit/s, 5 MBit/s or 10 MBit/s.

Data Link Layer FlexRay uses the NRZ in combination with 8B10B encoding for the data parts of the frame.

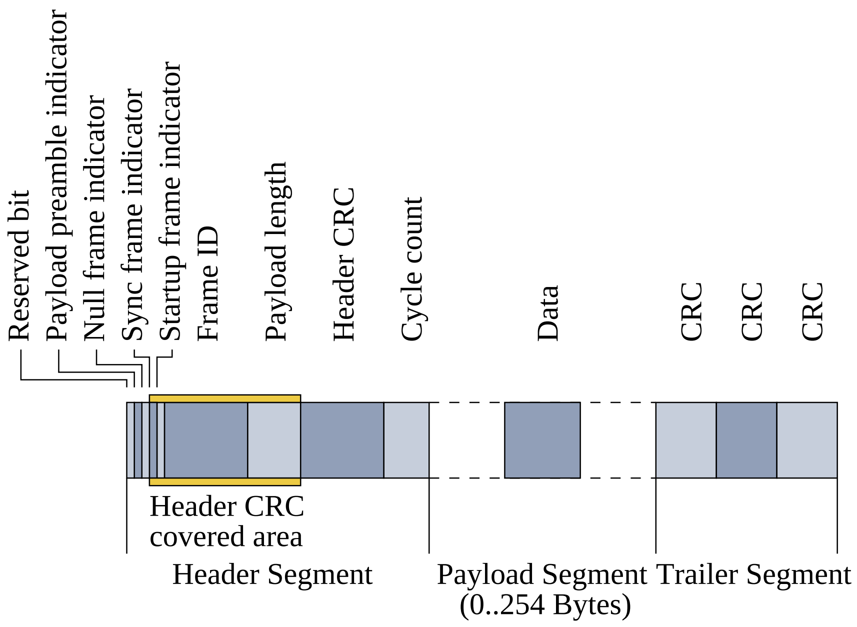


Figure 2.6: FlexRay Frame Format

Communication Cycle A FlexRay communication cycle consists of four parts: The static segment, the dynamic segment, the symbol window and the network idle time. A frame in the communication cycle is shown in figure 2.6.

Macrotick/Microtick All messages are sent at defined times measured in macroticks and microticks. This concept was derived from TTP/C [49].

Static Segment The static segment provides synchronous time-triggered communication using a TDMA protocol.

Dynamic Segment In the following asynchronous dynamic segment minislots allow event-triggered communication using prioritization based CSMA/CA similar to ByteFlight [27].

Symbol Window The symbol window provides basic communication with a bus guardian. A bus guardian terminates nodes which do not act conform to the protocol specification.

Network Idle Time The network idle time allows to calculate clock corrections and other cycle related tasks before the next communication cycle starts.

Cycle Repetition A frame doesn't need to be sent in every basic cycle, but can also be sent with another repeat factor. Therefore up to 64 different basic cycles can be defined. This is similar to TTCAN.

Clock Synchronization The multi-master distributed clock synchronization mechanisms, necessary for the timing of the communication cycle, differentiates the FlexRay cluster from other networks with independent clocks. In a cycle, up to 15 time masters can send their synchronization message. All nodes adjust their view of the global time using a voting mechanism during the network idle time. The synchronization algorithm in use is the Fault-Tolerant Midpoint Algorithm.

Cold Startup / Integration Startup To connect to the network every node has to run a startup, whereas the first two nodes are the cold start nodes and the following are integration nodes.

2.4.2 Communication Controller

The Bosch E-Ray module is a FlexRay IP-module that can be integrated as stand-alone device or as part of an ASIC. The components of the module are shown in figure 2.7.

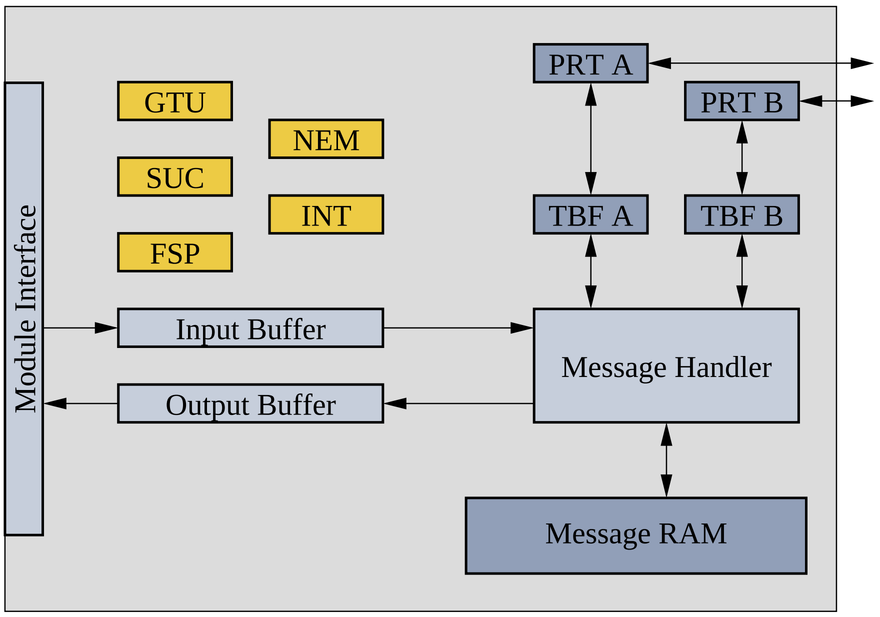


Figure 2.7: E-Ray Block Diagram

Customer and Generic Interface The Customer and Generic Interface (CIF, GIF) connects the E-Ray to the customer's specific CPU peripheral bus.

Input and Output Buffers The Input and Output Buffers (IBF, OBF) are used for the data transfer between the CPU and the Message RAM.

Message RAM The Message RAM is a single-ported RAM that stores up to 128 FlexRay message buffers together with related configuration data (header and data partition).

Protocol Controller, Transient Buffer RAM The FlexRay Channel Protocol Controller (PRT) consists of a shift register and FlexRay protocol FSM. They are connected to the Transient Buffer RAMs (TBF) for intermediate message storage and transfer to the Message RAM.

Message Handler The Message Handler (MHD) controls data transfers between the Input Buffers, Output Buffers and Transient Buffers and the Message RAM.

Global Time Unit The Global Time Unit (GTU) is used for the clock generation and synchronization and for the timing control of the static and dynamic segments.

System Universal Control The System Universal Control (SUC) controls the wakeup, startup and operation modes.

Frame and Symbol Processing The correctness of received frames is checked by the Frame and Symbol Processing (FSP) unit.

Network Management The network management vector contains a value which is OR'ed with each received network management frame. This is arithmetic is handled by the Network Management (NEM) unit.

Interrupt Control Unit The Interrupt Control unit(INT) handles the interrupt flags lines by assigning the flags to the status and error interrupt lines and setting and resetting the interrupt registers.

2.4.3 Configuration Formats

The configuration of FlexRay is very complex as it does not only cover the frame configuration, but also the complex cluster and controller configurations.

xCDEF The XML Cluster Definition File (xCDEF) was defined by DeComSys for their FlexRay tools. This file format is also used by other tool manufacturers as well. Detailed information of this format are not officially available from DeComSys.

FIBEX Nowadays FlexRay nodes are configured by the FIBEX format, described in section 3.5.1.

2.5 Media Oriented System Transport

The Media Oriented System Transport (MOST) [21, 20] is a networking standard intended for interconnecting multimedia components in automobiles and other vehicles. It differs from existing vehicle bus technologies in that it connects via Plastic Optical Fiber (POC), thus providing a ring bus-based networking system at bit-rates higher than available on most previous vehicle-bus technologies.

2.5.1 Protocol Description

The MOST specification defines all seven layers of the ISO/OSI Reference Model for data communication.

Cable MOST networks normally work over optical fibers using an optical PHY (oPHY). Very new and still unused is the possibility to also use unshielded twisted pair wires (UTP) using an electrical PHY (ePHY).

Physical The MOST network often employs a ring topology, but star configurations and double rings for critical applications are possible and may include up to 64 devices or nodes. BiPhase is used for the signal coding.

TDMA, CSMA/CA In the ICM segment TDMA is used, CSMA/CA in the MCM and MDP segments.

Timing Master A Timing Master is one of the nodes, which continuously feeds data frames into the ring or acts as the gate for data. The preamble or packet header repeatedly synchronizes the rest of the nodes called Timing Slaves.

Bandwidth The total bandwidth (including streaming data and packed data) is approximately 24.8 MBit/s. Up to 16 stereo channels with a sample frequency of 44.1 kHz can be configured.

Data Link Layer Data is transfered in blocks of 16 frames. Beside administrative information and control data, a MOST frame contains time slots for synchronous and asynchronous transfer channels. Figure 2.8 shows the frame format of a MOST frame. The boundary between synchronous and asynchronous time slots can be selected using the boundary descriptor specified in quadlets (4 Bytes).

The control data and the asynchronous channel contain frames itself, which are transferred in fragments over the MOST frames in a block. A control frame is 32-Bit long and therefore contained in two MOST frames. An asynchronous frame is 58 byte long including source address, target address and CRC.

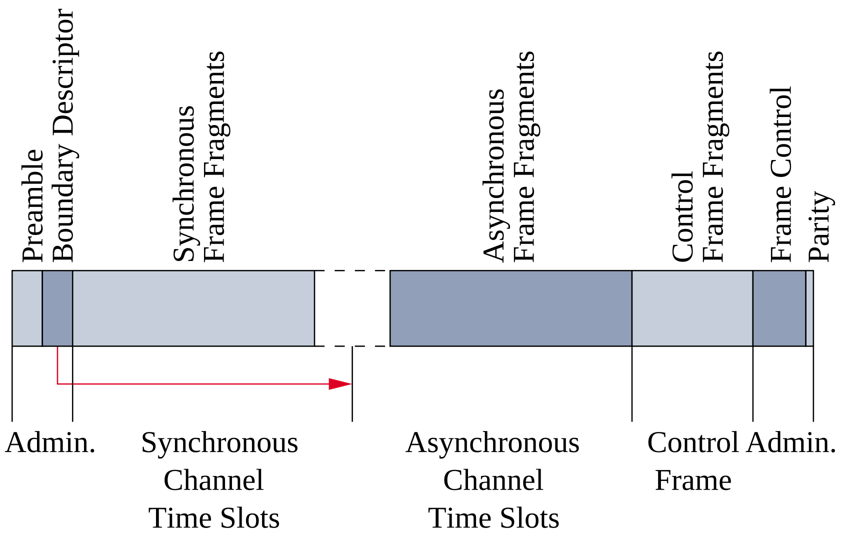

Preamble
Boundary Descriptor
Synchronous Frame Fragments
Asynchronous Frame Fragments
Control Frame Fragments
Frame Control
Parity
Admin.
Synchronous Channel Time Slots
Asynchronous Channel Time Slots
Control Frame
Admin.

Figure 2.8: MOST Frame Format

2.5.2 Communication Controllers

Currently MOST communication controllers are available only from SMSC, former Oasis and Xilinx.

OS8104 The first Network Interface Control (NIC available for MOST was the OS8104 [45, 24, 23]. This controller connects on one side to the optical PHY and on the other to an interface that could either be used for connecting four I^2S and one I^2C connection or for connecting a packet combined mode (PCM) interface.

API/SDK When using the OS8104, most of the transport protocol functions have to be implemented in software. Therefore an Application Programming Interface (API) and a Software Development Kit (SDK) are available. The advantage of having the API running on the host CPU and the reason, why many OEMs still use the OS8104, is the upgrade possibility together with the rest of the ECU software.

The MOST Network contains the following simultaneous operating networks and frames:

- MOST Control Messages (MCM) are used to manage the network and communicate control data across the network.
- MOST Data Packet (MDP) are used to support nodes that communicate asynchronous data.

- INIC Control Messages (ICM) are used to support high-speed synchronous data with extremely low overhead, such as audio and video.

OS81050 The OS81050 [46, 42] (also called INIC@25) comes with an improved host interface called the Media Local Bus (MLB [25]). It is called Intelligent Network Interface Controller (INIC), because of the fact, that most of the protocol stack previously running in software now runs as firmware on a specialized core in the NIC itself. This makes upgrading the ECU more complicated or even impossible, as not only the software needs to be handled, but also the INIC firmware.

OS81082 The newest development is the OS81082 (also called INIC@50) [47]. The main advantage compared to the OS81050 is the possibility to connect an electrical PHY [43].

2.6 Network Architectures

Todays automobiles consist of multiple communication networks of different type. Each network is implemented to handle a specific function domain, such as powertrain or infotainment. The interconnection between these networks is provided by gateways. When analyzing complete network architectures, two can be described as being most important: Central Gateway Architectures and Backbone Architectures.

These network architectures will be described in the following paragraphs. For simplicity the following figures present the physical topology only as a simple bus line, but in fact it depends on the type of communication network.

Central Gateway Architecture A central gateway architecture has only one gateway, that connects the communication networks. This creates high demands on the performance, but also provides a latency minimized communication over the networks. The demands for safety are very high, as the gateway is a single point of failure. An advantage lies in the possibility to provide one central diagnostic access to the automotive network.

Backbone Architecture In contrary, in a backbone architecture every communication network has its own gateway connected to a central high speed backbone network, such as FlexRay or IDB1394. This reduces the demands on performance and safety as every gateway has only two network connections. Backbone architectures will therefore become more established in future [68].

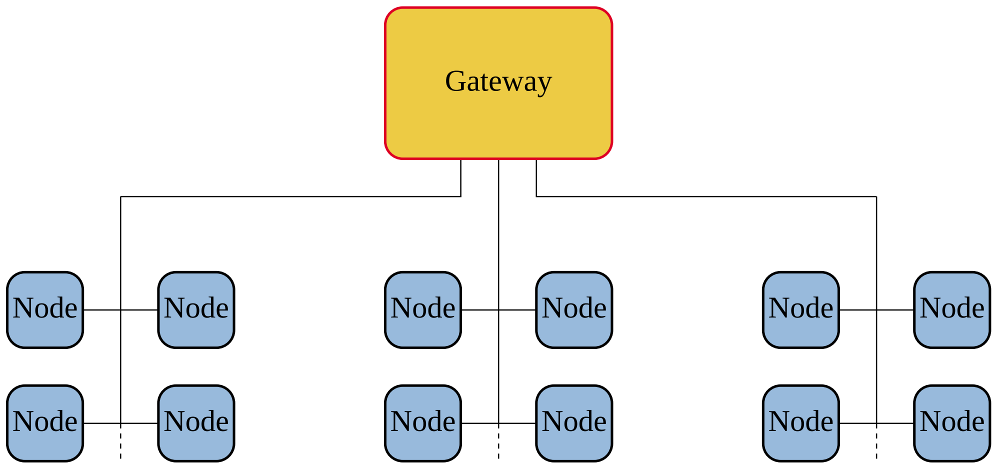


Figure 2.9: Central Gateway Architecture

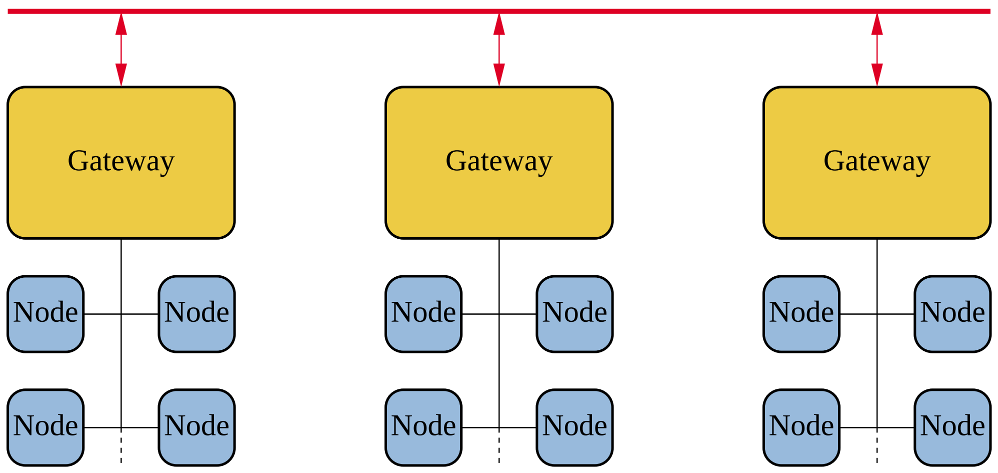


Figure 2.10: Backbone Architecture

Mixed Architecture In fact many automotive networks have mixed architectures, somewhere between a central gateway and backbone architectures. The low-speed LIN bus for example is usually connected to a body controller ECU that itself is connected to a central gateway or backbone gateway.

Impact on the Gateway A gateway designed as central gateway with high performance and safety requirements can easily be used as a backbone gateway that has lower requirements. Therefore the difference has an impact mainly on the scalability of the gateway.

Chapter 3

Current Gateways

Automotive gateways are developed within their limitation of rough physical environment, required efficiency and high production quantities.

In recent years many new automotive features became available because of improved interconnection techniques between different car networks. Therefore new gateways were necessary with almost every car generation.

As hardware usually has longer product cycles, most gateways are implemented in software. Unfortunately most microcontrollers are developed towards different applications. Only few microcontrollers have specialized instruction sets dedicated to provide gateway functionality.

3.1 Requirements

A gateway has to fulfill multiple requirements not only regarding technical, but also economical issues. These can also be used in comparing different implementations.

Proven Communication Controllers In automotive applications, the safety of passengers is directly influenced by failures of critical systems. Gateway systems are the most critical ones as they connect multiple ECUs and networks. Therefore it is very difficult to introduce completely new hardware components, such as communication controllers.

Low Latency Generally spoken latency is the period of time between one component generating and sending a message until another component is receiving and handling this message.

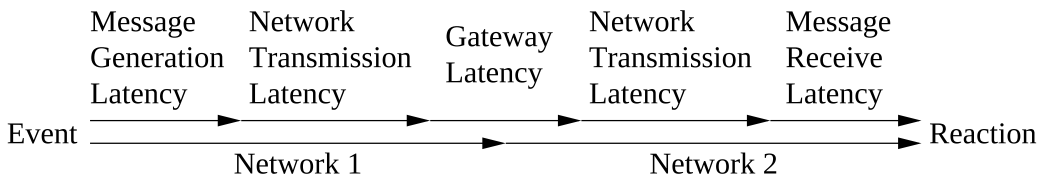

Message Generation Latency
Network Transmission Latency
Gateway Latency
Network Transmission Latency
Message Receive Latency
Event
Reaction
Network 1
Network 2

Figure 3.1: Latency between two networks

In a gateway environment the latency between two networks can be defined as a sum of

- generation latency - to compute a new message after an event,
- transmission latency - to transmit the message over the first network (dependent on schedule in time-triggered networks or message prioritization in event-triggered networks),
- gateway latency - to process and transfer the content of the received message to a second message,
- transmission latency - to transmit the message over the second network and
- receive latency - to process and handle the received data.

The generation latency, transmission latencies and receive latency cannot be influenced by the application. These latencies depend on the used bus systems, the transmitting and receiving nodes and the synchronization of the time masters. The only latency that can be actively influenced is the gateway latency, which is in software terms identical to the task's Worst-Case-Execution-Time (WCET) [61]. It depends on the gateway concept, the number of interconnected networks, bus loads and the complexity of the gateway function as described below.

Low Jitter Jitter is the time in which the latency varies during normal message transmissions. Having a high jitter is more a problem of event-triggered communication systems, where high utilization of the communication link can lead to high delays and unpredictable latencies. For real-time applications a precise prediction of message delays and receive times is much more important then fast message transmits. Therefore gateways intended to be used in real-time applications are required to have low jitter.

Scalability, High Bandwidth and Throughput Gateway systems are often used in environments where high-speed networks need to be connected. FlexRay, MOST and Ethernet each have a bandwidth of more than 10 MBit/s. The throughput needs to be high enough to be able to process these data even under worst case conditions. Scalability can therefore be described as the impact of high network utilization and throughput on latency, jitter and frame loss.

Low Power Consumption, Temperature, Frequency and Electromagnetic Radiation On the other hand especially in automotive environments, low power consumption is a crucial factor, not only to be economical and efficient. High power consumption also means high temperatures that makes it difficult to fulfill the requirement for the large automotive temperature range from -40 C up to 125 C. It is often a result of complex hardware, where signal changes and state transitions happen with high clock frequencies. High frequencies are also causing high electromagnetic radiation which can affect other ECUs, sensors and actuators as well.

Low Cost and Gate Count In applications with high production quantities, the benefits can be increased dramatically by reducing the production cost of each part. The only way to do this in hardware is to reduce the gate count and RAM requirements by an innovative concept and an optimized implementation. Optimization is reflected in a reduced software code and hardware die size, which allows more dies per waver and therefore higher quantities at the same price. Also smaller die sizes often have a positive influence on the power consumption. On the other hand, a lower gate count allows to integrate more functionality without changing the die size.

3.2 Tasks

The movement of large amounts of complete messages between networks is only one task a gateway has to perform. Transfers can include operations on the signal level and usually need to be done with a specific time behavior.

Frames and Signals In many cases one frame consists of some few independent e.g. sensor and/or actuator signals of one or several bits. To minimize the used bandwidth of the connected buses, messages transmitted contain only signals necessary on the target bus. Therefore usually these transmitted messages are created from signals out of several received frames. However, the bit position can vary in both messages so that many bit manipulating operations are necessary.

Periodic Transmissions To increase the safety of in-car systems, periodic transmissions can be used. Therefore the gateway has to buffer the message data and a timer is used to periodically transmit the message.

Immediate Reaction The gateway can react differently to the reception of a message. It is not only possible to immediately forward the frames and signals therein, but it is also possible to only send new messages, if specific parts of the received message have changed. The gateway has to compare the old and the new message and has to react accordingly. It is also possible to store just the received data for periodic transmission without immediate forwarding.

Timeout Handling The gateway can also react to messages, that are not received within a specified time. It is then possible to set timeout bits in certain messages, mark signals as obsolete, send additional timeout frames or change the interval duration of periodic transmissions.

Debouncing Some ECUs try to increase transmission safety by sending the same message multiple times in a short period. As this mainly wastes precious bandwidth, a gateway can be configured to react to these frames only once. Once a debounce time has passed, frame transmissions are allowed again.

Transport Protocols Some transport protocols require the gateway to not only forward frames, but to process them. This is especially true, when the gateway should transparently convert one transport protocol to another, which is always the case when different network types and therefore protocols are involved.

Network Management A gateway also needs to handle network management requests and responses on all connected networks. If these networks are of different type, a conversion is necessary.

Further Control Applications The previous tasks have shown the complexity of gateway implementations. Additionally microcontrollers performing the gateway function have to process several control applications, such as the evaluation of sensor data or the adjustment of actuators.

3.3 Gateway-Optimized Microcontroller

To support software gateways in fulfilling the requirements and manage their tasks, few microcontrollers have been developed especially for gateway applications. A characteristic of these controllers is that they already have many communication controllers integrated. The connection to the CPU is provided by high speed on-chip buses to reduce latency and increase throughput. In addition DMA controllers or network coprocessors can be integrated.

Freescale MPC5567 The MPC5567 [66] is a microcontroller developed by Freescale, shown in figure 3.2.

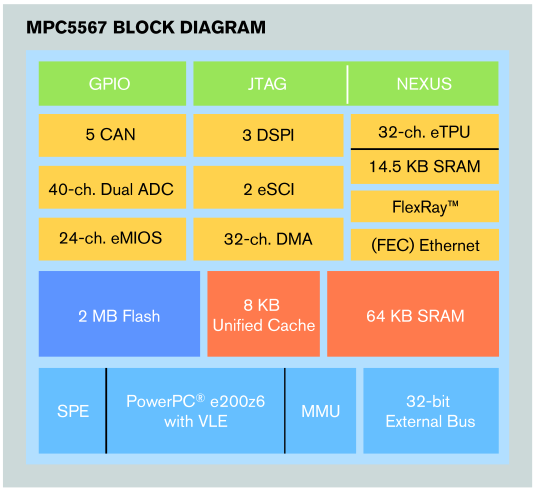


Figure 3.2: Freescale MPC5567 Block Diagram

It has a high-performance 32-Bit PowerPC core with a clock frequency of up to 132 MHz and a MMU with 32-entry fully associative translation lookaside buffer. This processor core is enhanced by a signal processing extension (SPE), which brings additional capabilities for DSP, SIMD and floating point operations.

The enhanced Time Processor Unit (eTPU) is a programmable unit on which most timer handling functions of the gateway application can be offloaded. Additionally a 32-channel enhanced DMA controller can be programmed to automatically copy data from and to the communication controllers. This is one of the first device in which Freescale integrated their own FlexRay controller. It acts beside five CAN cores and an Ethernet core.

Infineon TriCore TC1130 The Infineon TriCore TC1130 [69, 70, 71, 72], as shown in figure 3.3, provides four CAN controllers [11] and one Ethernet controller.

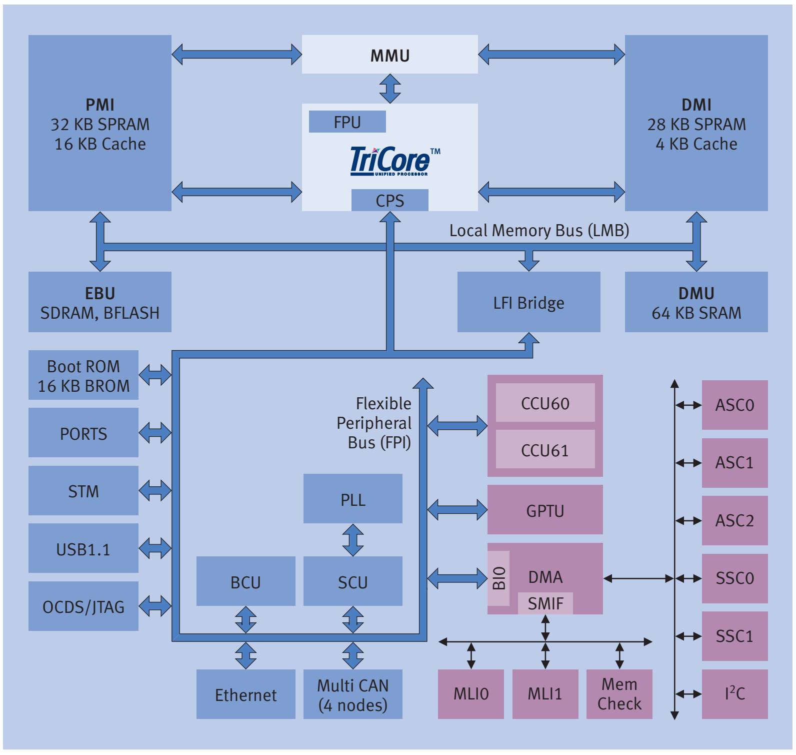


Figure 3.3: Infineon TriCore TC1130 Block Diagram

The 32-Bit TriCore architecture combines RISC, CISC and DSP functionality with a MMU and a FPU in a single chip. It can run with a core clock frequency of up to 150 MHz.

The DMA controller can be used to transfer data to and from the four CAN controllers and the Ethernet core. As the device supports hardware controlled context switches for tasks and interrupts, the gateway latency and jitter is reduced.

NEC V850 The NEC V850 [76] family is especially well suited for high-end chassis applications in the automotive area. One family member, the V850E/PH3 "Phoenix-FS", is shown in figure 3.4.

The V850E CPU is a 32-Bit RISC core with integrated FPU, a clock frequency of up to 128 MHz and a maximum performance of 166 MIPS.

The two CAN controllers and one FlexRay controller can transfer their data from and to the system RAM using an 8-channel DMA controller.

Freescale HCS12X The HCS12X [65] contains a 16-Bit HCS12 compatible CPU on which the main application software runs. The suffix X stands for an

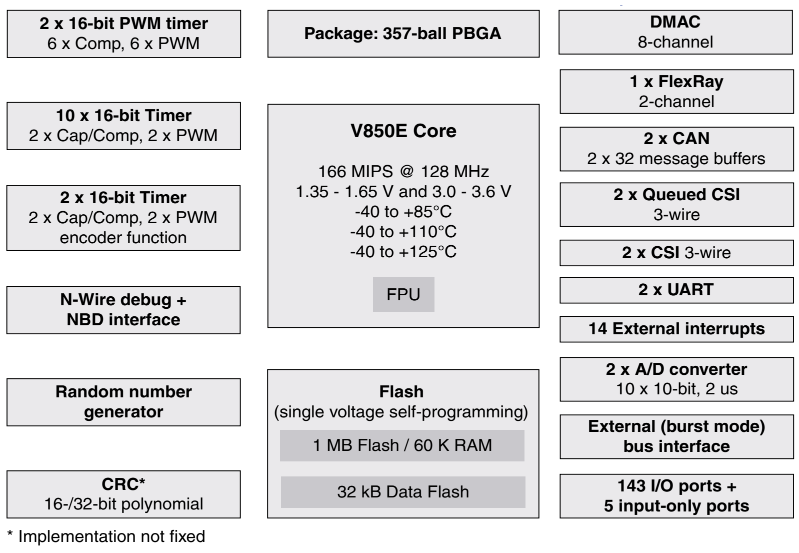


Figure 3.4: NEC V850E/PH3 Phoenix-FS Block Diagram

additional XGate [64] peripheral coprocessor module, that has been added to the system. This parallel processing module offloads the CPU by providing high speed data processing functions for transfers between peripheral modules, RAM and I/O ports.

The XGate offers a slight performance increase, but requires specialized software [63]. Standard gateway software, as described in the following section, is currently not available on the XGate and would not run fast enough on the HCS12. Therefore the HCS12X is currently only used without the XGate in gateway implementations. Although the XGate can be used to offload certain tasks (interrupt, timers) of the HCS12 software application.

3.4 Software Gateways

For a long time the software structure of a gateway was not standardized. Therefore many different vendors had their own incompatible implementations of the same functionality, e.g. Vector Informatik, 3Soft or K2L.

Operating System The real-time operating system (RTOS) used by almost all automotive ECUs is OSEK/VDX as standardized in ISO 17356 [73]. It provides task management, different application modes, interrupt handling, event control mechanisms, resource management, alarms, messages, different hook routines for error handling, tracing and debugging.

Driver Every hardware module, such as the communication controllers, do need a software driver to adapt to this hardware module for low-level functions and network management functions. The CAN Driver for example provides services for initiating transmissions and callback functions for notifying receive events independently from the hardware.

Network Management provides functions for network wide shut down of the communication system, for determining the network configuration at start-up, for monitoring the network configuration during operation and for providing information of the current network status.

Communication Module provides signal and frame routing functions between equal (e.g. CAN/CAN) or different (e.g. CAN/FlexRay) vehicle network systems. It is therefore also responsible for fragmentation and defragmentation of data.

Transport Protocol Modules provide segmentation of data in transmit direction, collection of data in receive direction, control of data flow and detection of errors (e.g. messages loss/doubling/sequence).

3.4.1 K2L Gateway

K2L developed a software gateway with the described components. Their demonstration system is called MoCCa Vario II [74]. It is able to connect up to six CAN, two FlexRay, one MOST and multiple low speed bus systems. The software is portable to different Real-Time Operating Systems. The signal conversion and routing characteristics are defined in a table format, which contains operations for bit, byte, word and various complex conversion methods like scaling signal values or adding constants to messages. A configuration tool is provided for this proprietary tabular gateway format. The overall gateway application is embedded in a modular framework, suitable also to integrate other, non gateway applications.

3.4.2 Automotive Open System Architecture

The Automotive Open System Architecture (AUTOSAR) Initiative tries to standardize complete software architectures [58, 59] to improve and fasten application developments [62]. Additionally, it does not only specify the current basic software, such as the OSEK/VDK operating system [79, 77, 78], but also parts of the applications, the toolchain and the configuration formats. Several components are already available as shown in figure 3.5.

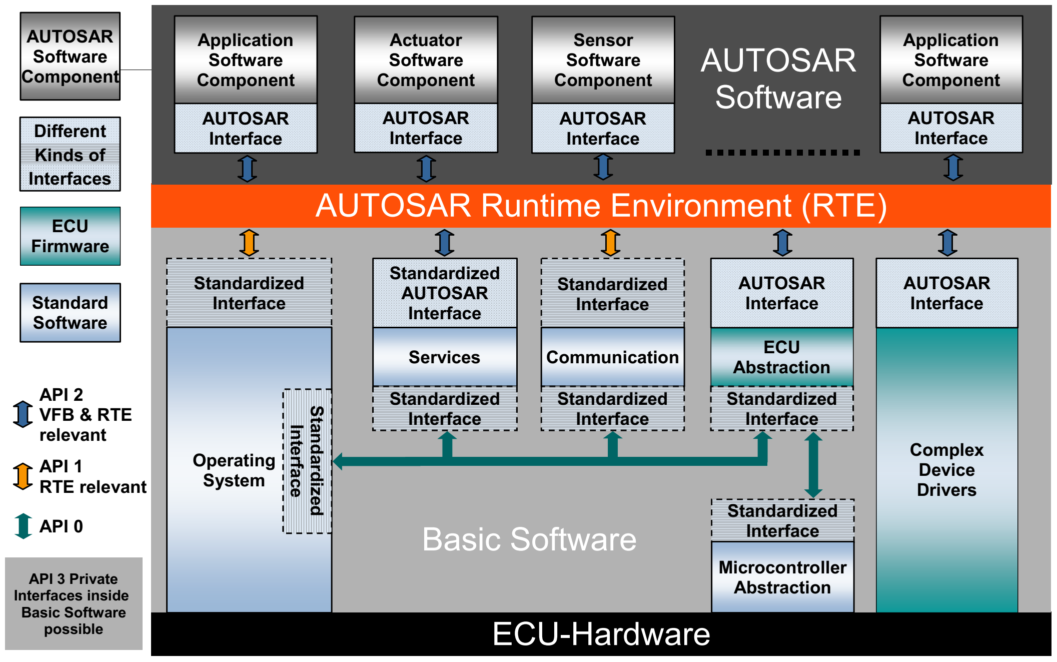


Figure 3.5: AUTOSAR Components

Microcontroller Abstraction Layer is the lowest software layer of the Basic Software. It contains drivers, software modules with direct access to the microcontrollers internal peripherals and memory mapped microcontroller external devices.

ECU Abstraction Layer interfaces the drivers of the Microcontroller Abstraction Layer. It also contains drivers for external devices. It offers an application programming interface (API) for access to peripherals and devices regardless of their location (microcontroller internal/external) and their connection to the microcontroller (port pins, type of interface).

Services Layer is the highest layer of the Basic Software and has the highest relevance for the application software. While access to I/O signals is covered by the ECU Abstraction Layer, the Services Layer offers operating system functionality, vehicle network communication and management services, memory services (NVRAM management), diagnostic services (including UDS communication and error memory) and ECU state management.

Complex Device Drivers implement complex sensor evaluation and actuator control with direct access to the microcontroller using specific interrupts and/or complex microcontroller peripherals, e.g injection control, electric valve control and incremental position detection.

Runtime Environment The RTE is a middleware layer providing communication services for the application software (AUTOSAR Software Components or AUTOSAR Sensor/Actuator components). The AUTOSAR Software Components communicate with other components (internal or external) or services via the RTE.

3.4.3 X2E Gateway

X2E integrated and ported the K2L software gateway to an Altera FPGA using a NIOS II softcore processor. This combination makes it possible to easily implement various parts of the software in hardware.

The NIOS processor allows to integrate user-defined instructions for gateway specific tasks. With recent versions (since 6.0) of the Quartus synthesis software [90], it is also possible to use a technique called C2H [88]. C2H allows to select certain C functions and to synthesize them as function blocks in hardware, which improves the gateway performance even more.

3.5 Configuration Data Formats

Gateway configurations are based on non-standardized description formats. Data exchange between these formats is inherently error prone and time consuming.

Simple frame and signal relations between different CAN channels can already be described in CANdb (see section 2.2.4). When more complex functions are necessary, non-standardized user-defined attributes can be used. CANdb is limited to only one CAN network, however multiple files can be used.

A common description format usable for different networks types and gateway configurations was necessary. The Field Bus Exchange Format (FIBEX) provides these information.

The Automotive Open System Architecture adapted this format for their own System Constraint Templates.

3.5.1 Field Bus Exchange Format

The Field Bus Exchange Format (FIBEX) is an XML-based file format, the upcoming standard for network configurations [60], which combines information about each aspect of a complete in-car network including controllers, channels, frames and signals [55, 56]. It is also the first standard to describe gateway configurations. The exchange format covers the functional network, the system topology and the communication level.

Standardization FIBEX is based on an initiative of BMW and was developed in cooperation with automobile manufacturers, suppliers and tool producers, proposed for data exchange between tools that deal with message-oriented bus communication systems. It is now being maintained at the Association for Standardisation of Automation- and Measuring Systems (ASAM e.V.).

Supported Networks FIBEX tries to be widely independent from all communication controller implementations and protocols. In the version 1.2 it is usable for CAN, LIN and FlexRay networks. A MOST extension is available since version 2.0. A standardization for TTCAN is still pending.

FIBEX Object Model In FIBEX the description of all elements is split into different objects as shown in figure 3.6.

Data Format and Instantiation The cluster, channel, frame and signal objects describe the format and configuration of data, whereas the connector, frame-triggering and signal-instance objects instantiate these data descriptions by providing timing and position information. In FIBEX, the definition of gateway configurations is done by defining one-way mappings between these data instance objects. Timing attributes can be defined for each mapping, e.g. message timeout, debounce time or cyclic sending. Trigger-conditions define the immediate reaction on frame receives, such as send immediate or on-change of the received data.

Manufacturer Extensions Although FIBEX already allows comprehensive descriptions, manufacturer extensions may expand the capability even further.

Tools The description in FIBEX is very complex, resulting in files, which evade easy comprehension. Therefore many tool manufacturers developed FIBEX editors with assistance functions and support for specific configuration aspects, e.g. FlexRay parameters and frame schedules.

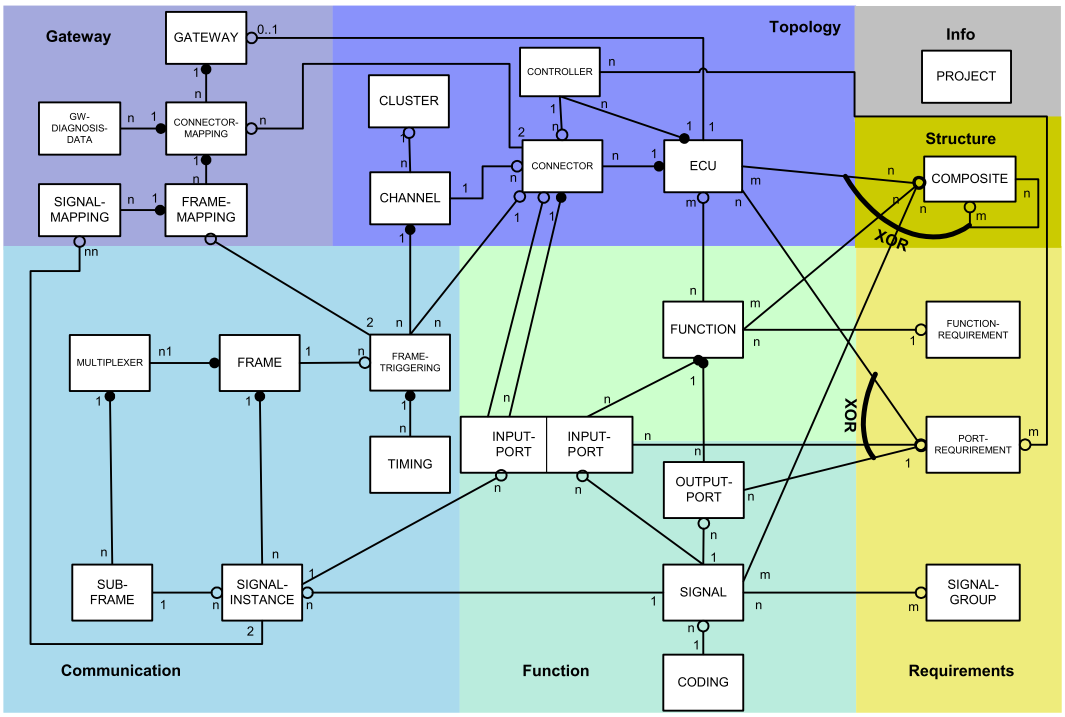


Figure 3.6: FIBEX Object Model

3.5.2 AUTOSAR System Constraint Templates

The Automotive Open System Architecture (AUTOSAR) Initiative, as described in section 3.4.2, provides another XML-based standard, that can also describe complete networks including node configurations and software layers. They are called AUTOSAR System Constraint Templates [57] and are in fact based on FIBEX with specific adaptations to reflect the AUTOSAR template naming regulations.

3.6 Problems and Issues

Implementing the gateway functionality in software provides high flexibility in the connection of new hardware components and the expandability of the gateway functionality. The software modules of a software gateway soon become standardized and will be provided by multiple manufacturers, that also give support and can implement specific functionality. Most transport protocols or modules for diagnostic applications are easier to integrate in the software application.

However, software implementations have some disadvantages. Large memories are required to store the functionality in form of software applications. It is less scalable regarding the bandwidth of the connected networks, as the CPU frequencies (up to 150 MHz) have already reached the limit for automotive environments. Although the requirements on latency and jitter for a usual central gateway are difficult to achieve. As a consequence the microcontrollers have high power consumptions and problematic electromagnetic radiations and compatibility. The real-time ability is affected and even packet loss can be expected.

Chapter 4

Advanced Gateway Architectures

In this chapter two concepts for advanced gateways with configuration toolchain and verification environment will be developed based on the requirements and limitations described in the previous chapters.

4.1 Advanced Gateways

In previous chapters the requirements and limitations of current gateways have been described. Based on that information the concept of an advanced gateway architecture can be derived.

The developed gateway concept is versatile, thus usable by different network types. It offers gateway functionality in hardware without the need to have a powerful CPU or large system RAM.

A first implementation of the Gateway Control Unit utilizes the Bosch IP-modules for CAN and TTCAN. The second implementation adds multi-protocol support using the Bosch IP-module for FlexRay.

Accessing and interacting with the underlying network interfaces is still possible from the host CPU to provide further independent functionality in software.

4.1.1 Requirements

An advanced gateway should be able to process most gateway tasks in hardware without the need to inform the host CPU about it. The reduction of interrupt load and gateway functionality in software should reduce the demand for a high performance host CPU.

Access to the communication controller should still be possible in parallel without the need to setup blocking Mutexes.

4.1.2 CAN-CAN Gateway

The CAN-CAN gateway is the first implementation of an advanced gateway. This hardware solution can be a set of functions implemented in hardware to support the software or an autonomous system that provides the functionality completely in hardware. Example implementations of both solutions will be described in this section.

It is based on a modular gateway structure which interconnects several single channel Bosch C_CAN (see section 2.2.3) or TTCAN (see section 2.3.2) IP modules. A block diagram of the used CAN controller was shown in figure 2.3.

Gateway Interface The (TT)CAN modules are expanded by several functional blocks to add a specific gateway interface as shown in figure 4.1. These functional blocks are an input multiplexer (In-Mux) and an output multiplexer (Out-Mux) together with the necessary control signals to direct the data flow.

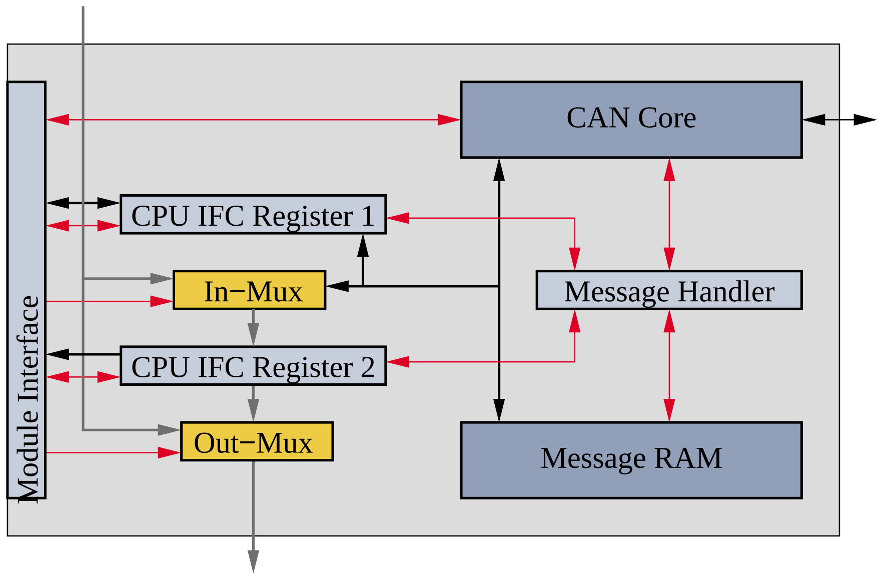


Figure 4.1: C_CAN Controller with Gateway Interface

Two multiplexers give access to the internal data bus, making it possible to load the complete CPU Interface Control Register (IFC) in parallel from a wide input port (Cascade-Input) and to export the contents of the CPU IFC Register over an equally wide output port (Cascade-Output). The Cascade-Input may also be routed directly to the Cascade-Output.

Cascade Ring Bus When several instances of this adapted single channel CAN module are cascaded into a gateway module, the wide input and output ports are connected to the cascade ring bus (Figure 4.2). This allows the transfer of a complete CAN message and corresponding control information directly from one CAN cell to all connected cells in one clock cycle, avoiding the bottleneck of the CPU's peripheral bus.

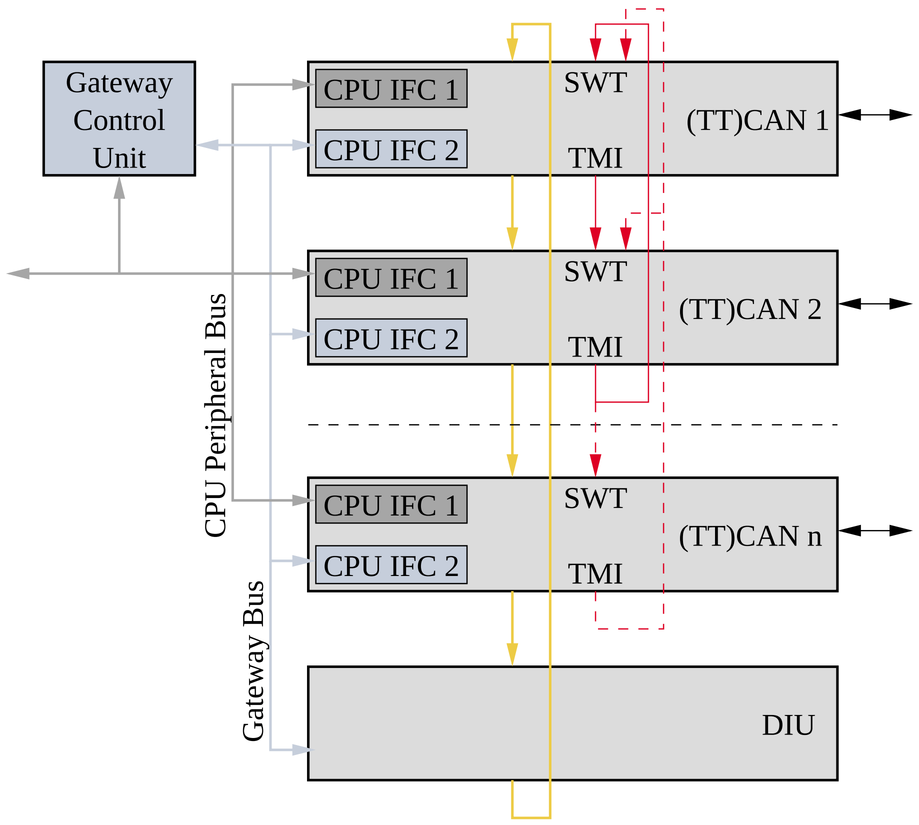


Figure 4.2: Gateway Module with cascaded CAN/TTCAN cells

TTCAN Cycle Time Synchronization When using TTCAN communication controllers, it is possible to configure a constant latency for transferring messages from one TTCAN to another TTCAN by synchronizing its matrix cycles. The Time Mark Interrupt (TMI) signal is set whenever a selected time equals the value in the internal timer register. This signal can be connected to the Stopwatch Trigger (SWT) input of another TTCAN to calculate and reduce the time difference between the matrix cycles of both controllers. It is always possible to synchronize a time master to another time source, such as another time slave.

Data Integration Unit If additional functions beyond a simple message transfer are required, special modules that implement these features can be inserted into the cascade ring. One example of these features is a merge of several messages into a new one. Therefore a Data Integration Unit (DIU) was developed as shown in figure 4.3. This unit allows the extraction of one or more bits of a message and the arbitrary insertion of these bits into a new message. The complete message can then be copied via the cascade ring to all connected nodes.

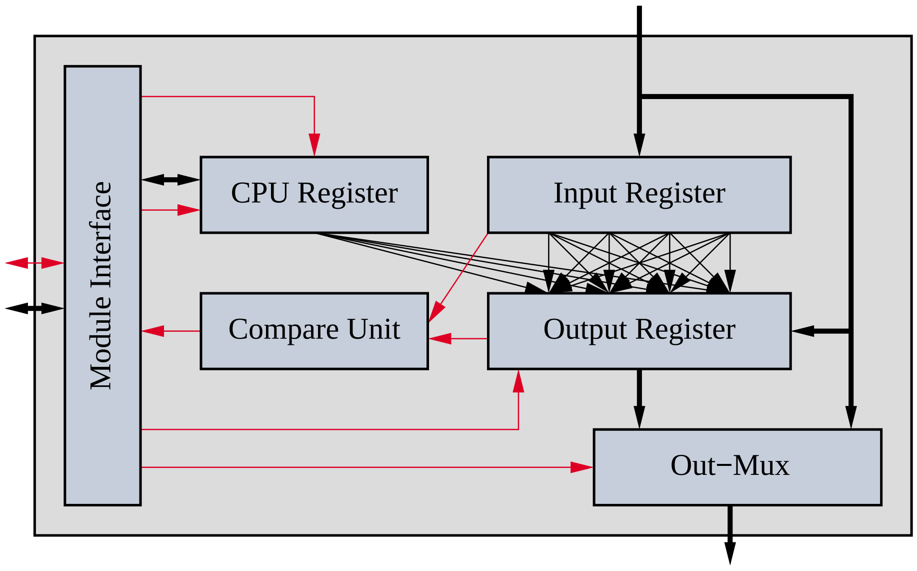


Figure 4.3: Data Integration Unit

Gateway Control Unit Several instances of these adapted single channel modules may be combined and turned into a gateway. The data flow between the (TT)CAN modules and the DIU is controlled by an application specific Gateway Control Unit (GCU) as shown in figure 4.4. It provides the control signals for the input and output multiplexers, the information to load or store a message from or to the internal message RAMs of the communication controllers and the control signals for special modules.

Possible realizations are a set of special function registers (SFR) or a finite state machine (FSM). The implemented Gateway Control Unit combines the special function registers and the finite state machine in one module.

Special Function Registers The SFR block consists of a set of registers for the DIU and each (TT)CAN module connected to the cascade ring. The registers are written by the CPU. This means that the routing algorithm and

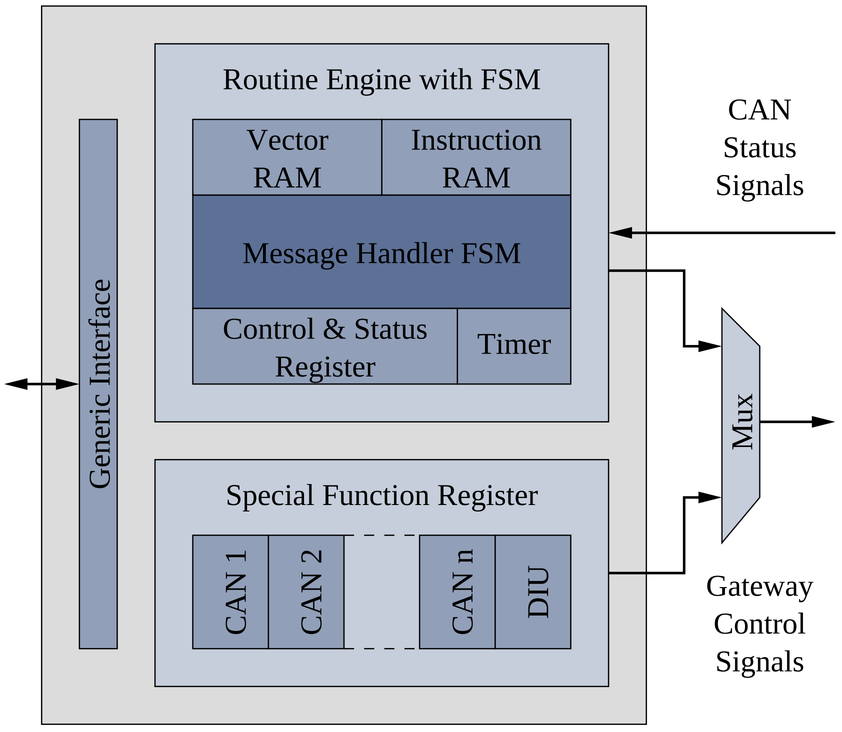


Figure 4.4: Gateway Control Unit

the routing data are stored on the host CPU. Upon the reception of a message the CPU examines the number of the affected message buffer and determines if the message buffer has to be transferred to another network. If it has to be transferred, the CPU writes the appropriate registers. The direct data path minimizes the number of read and write accesses from the CPU to the communication controllers, but a high number of CPU cycles is still necessary to check for received messages by interrupt handling or polling. This load can be further reduced by an FSM based solution.

Finite State Machine The FSM based solution is a small autonomous control unit. It consists of the message handler, control and status registers, a timer and configuration memory. The configuration memory is divided into two parts, the Vector RAM (VRAM) and the Instruction RAM (IRAM). The instruction memory contains the transfer directives for the message handler, whereas the vector memory contains event triggers and the event handler start address in the instruction memory. Possible event triggers can be the reception of a message, a receive timeout or a cyclic transmission. The implemented timer is used to realize the receive timeouts or the periodic transmission. Several control and status registers are implemented to configure the used message buffers and to show timeouts or pending interrupts. All functions concerning the handling of messages are implemented in the message handler. This includes the transfer of messages between the (TT)CAN modules, (de-)fragmentation of data and the handling of transmission requests, timeouts and interrupts. Therefore the message handler only needs a minimal instruction set.

Vector RAM The Vector RAM contains event triggers, such as a message receive, receive timeouts and periodic sending and an instruction start address for each CAN message buffer in case of an event. An interrupt bit can be set to additionally interrupt the CPU in case of an event. This makes special treatment possible, e.g. gateway functions between different communication controllers (FlexRay, MOST). A set of Vector Registers reflects the Rx/Tx and message valid configuration and holds timeout flags and interrupt pending bits.

Event Detection The algorithm to detect events differs in the software (SFR) and hardware (FSM) implementations of the CAN gateway. Both solutions use an event loop, which loops through all CANs and all message buffers to find an event (receive, receive timeout, time to send). The maximum jitter results from the event loop and is the time for a complete event loop cycle, as the worst case is an event that happened just after checking for that event. Then a complete cycle has to take place to detect that event. Therefore the hardware solution is optimized by interrupting the loop when an receive event occurs. As result the jitter is kept to a minimum. However, the software solution checks for the ranges of configured CANs and message buffers and it loops only through these. This also reduces latency and jitter. Another component of jitter is also a result of impreciseness when reading the timer in case of an Rx timeout detection or period sending of messages. Timer delays are always shorter than desired. The worst case is a timer reading shortly before and then shortly after the increment, then almost no time passed.

Instruction RAM The Instruction RAM of the FSM gateway, defines actions in case of an event. There are three groups of functions:

- The first group is for flow control and contains instructions for (un-)conditional branches and the finish instruction.
- The second group contains instructions for reading, writing and sending of CAN messages and to transfer messages from one CAN to another.
- The third group controls the Data Integration Unit for rearrangement of message data.

Gateway mode The modular structure allows a flexible programming of the gateway function. Even when the gateway function is controlled by the Gateway Control Unit, the host CPU keeps full access to all functions of each (TT)CAN cell, excluding the second interface register. This enables the parallel evaluation and processing of messages. Concurrent requests of the CPU and the FSM to the same cell are solved in a deterministic way.

Software mode If the direct data path is not used, the Gateway Control Unit can be deactivated. Then the software has full access to both interface registers and all functions of the communication controllers. The complex gateway functionality can then be completely processed by the host CPU.

Assembler An assembler has been implemented to ease the configuration of the gateway. It consists of four stages. The first stage evaluates preprocessor macros. This is especially helpful to assign a CAN network number and arbitration ID to a CAN message buffer. The second stage normalizes the remaining code, e.g. removes comments, puts labels in own lines and replaces leading and trailing and multiple spaces and tabs to get code with only one space between each argument. The third stage evaluates labels in code and branch instructions. The last stage is for writing the configuration of the gateway, represented as an array with 32-Bit hexadecimal values in a C-like fashion. In this way it is easy to parse and integrate the configuration directly into the gateway program and it permits comments, which are already done automatically for labels.

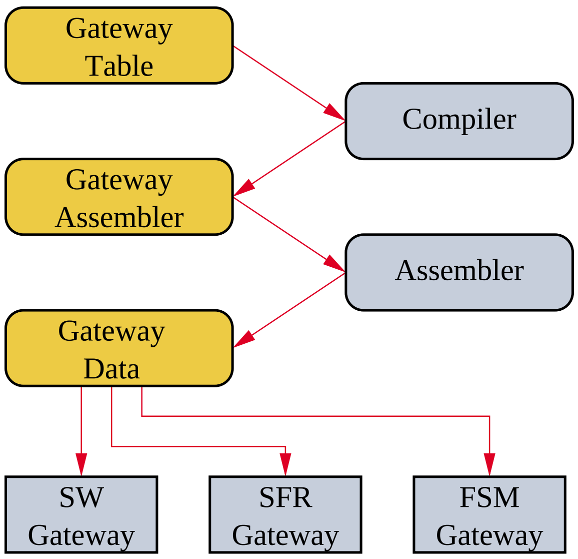


Figure 4.5: Data Generation Flow

Compiler Writing a complete CAN gateway matrix to assembler code compiler is possible as shown in figure 4.5. In this way an optimization of instructions can be done, e.g. a periodically transmitted message can get its data from other CANs at send time or the other CANs can write the data to the transmit message at their receive time, so that the transmission can then instantly take place. The first solution uses less instructions at send time, but could run unnecessary instructions at receive times. The second solution only transfers messages once at send time, but takes more time to send.

4.1.3 Multi-Protocol Gateway

The Multi-Protocol Gateway is the second implementation of an advanced gateway. It is based on the previous implementation of the CAN-CAN Gateway.

Enhancements Multiple enhancements have been made resulting from extended requirements and the gained knowledge and experiences with the CAN-CAN Gateway.

New requirements for the second gateway implementation were further protocols, e.g. FlexRay and MOST. This leads to the concept of a layered architecture containing protocol domains.

Wrappers for every protocol domain allows protocol specific operations, e.g. control functions for a CAN cascade bus can be integrated here.

A dual-bus concept was implemented to speed up the communication between the different protocol domains and the Gateway Control Unit. The cascade ring bus of the CAN domain was removed for simplicity, but can be implemented again using the Domain Wrappers.

To increase the scalability and efficiency, a partitioning of the VRAM and the FSMs working on it was necessary. The Special Function Registers have been removed in favor of the partitioned FSM concept. Interaction with the CPU is still possible while the gateway is in operation.

An automated configuration toolchain was necessary to ease configuration of the gateway. The implementation is described in the following chapter 4.2. For fully supporting FIBEX as the input format for the toolchain, further modifications were necessary for use all possible features.

FIBEX supports message debouncing, which means suppression of message bursts. This made additional entries in the VRAM necessary to define the debounce time in addition to the cyclic sending time.

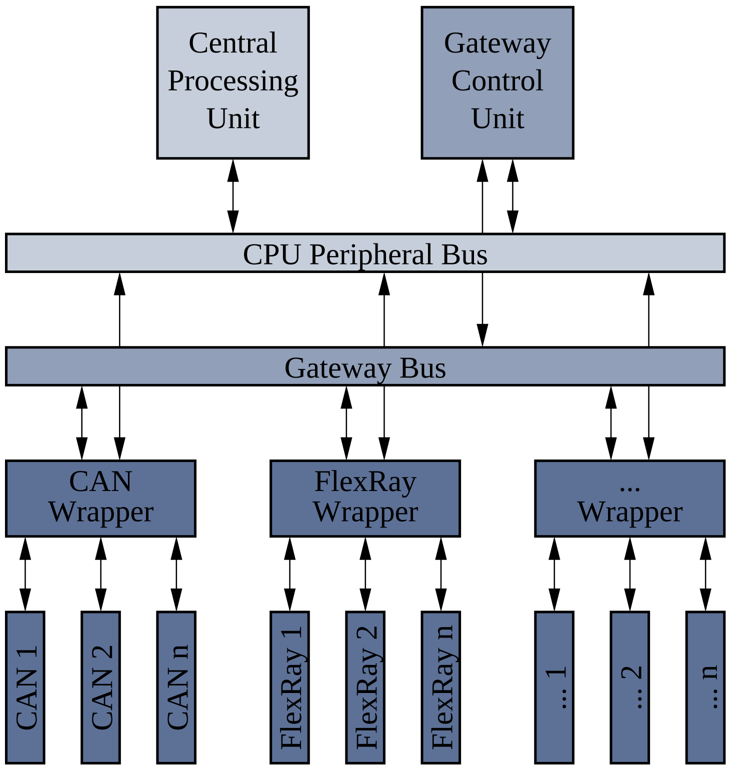


Figure 4.6: Gateway Layer Architecture

Gateway Layer Architecture The multi-protocol gateway is a dedicated hardware structure optimized for gateway operations. It can be described in four layers as shown in figure 4.6.

Processing Layer The Gateway Control Unit (GCU) in the processing layer contains two configuration RAMs as shown in figure 4.7. The Vector RAM (VRAM) and Instruction RAM (IRAM) contents are processed by a Finite State Machine (FSM). A further description of the implementation is provided in section 5.3.

The configuration is minimized and very efficient. The CPU is only needed for the configuration of the GCU and to handle exceptional transfers that are too complex to be processed in the Gateway Control Unit, e.g. advanced transport protocols or extensive arithmetic functions. It has been estimated that with an average gateway configuration less than 20% of the traffic must be handled in software running on the CPU.

Bus Layer The bus layer contains the usual CPU system and peripheral bus with all CPU-accessible peripherals used in the microcontroller, e.g. AMBA

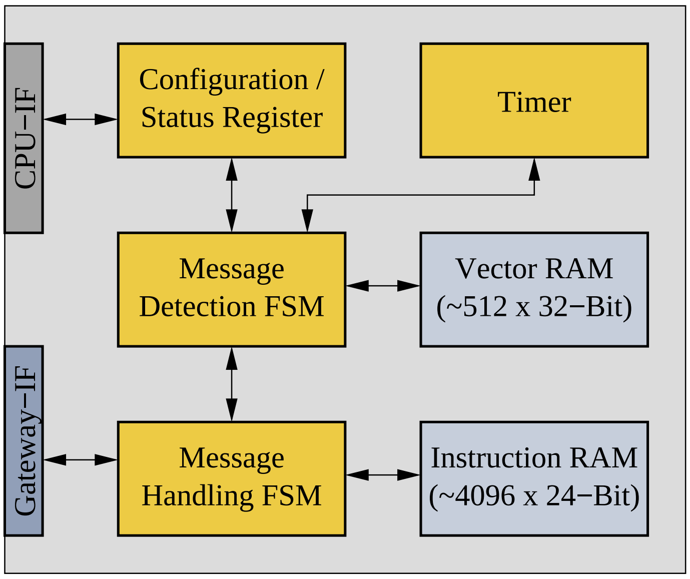

CPU−IF
Configuration / Status Register
Timer
Message Detection FSM
Vector RAM (~512 x 32−Bit)
Gateway−IF
Message Handling FSM
Instruction RAM (~4096 x 24−Bit)

Figure 4.7: Gateway Control Unit

bus family [80]. An additional gateway bus is connected to the GCU and to each wrapper for a set of equal communication controllers. Both buses are used simultaneously and without interfering with each other.

Wrapper Layer The wrapper layer monitors incoming messages and combines information for groups of 32 message buffers. A selection can be made by masking message buffers meaningless for the gateway. The wrapper is necessary to provide an abstraction of the different types of communication controllers and their signaling of receive events to the GCU. Additional features could be implemented here, for example direct data paths between communication controllers of the same type, as already done in the CAN-CAN-gateway described in the previous section 4.1.2.

Communication Controller Layer The communication controller layer contains CC IP modules, which should be used in slightly modified variants to better adapt to the gateway.

All communication controllers are accessible by the CPU and the gateway buses in parallel. Depending on the implementation of the CC, different adaptations can be made to optimize the gateway capabilities.

A communication controller with multiple interface buffers for reading and writing message objects is essential to avoid access conflicts between CPU and Gateway Control Unit. This is shown in figure 4.9. If the CC does neither

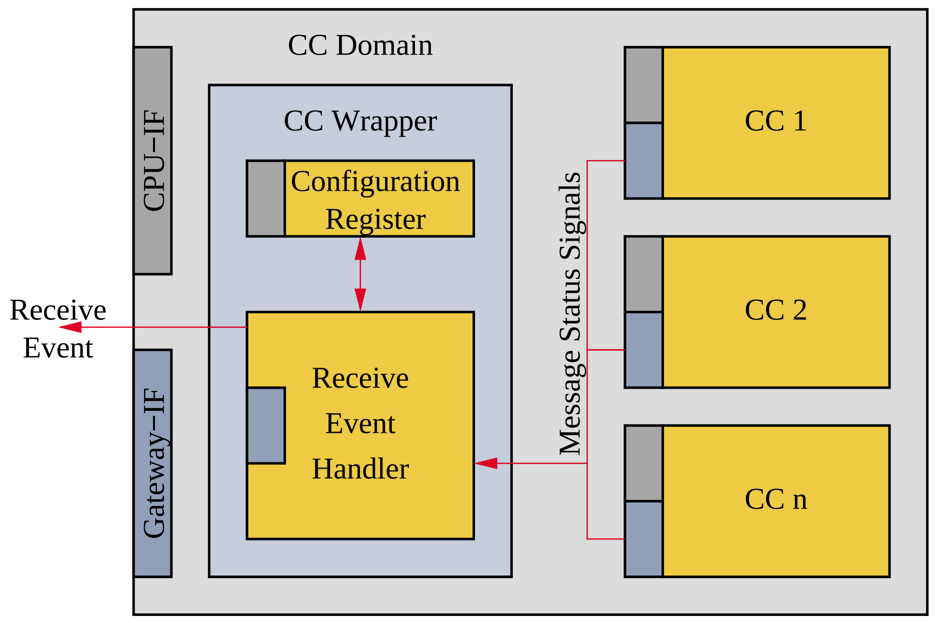


Figure 4.8: Communication Controller Wrapper

support multiple buffers, nor a modification is possible, both CPU and GCU have to arbitrate their buffer accesses using mutexes.

For even better performance it is valuable to have two different and logically independent interface ports for both CPU peripheral bus and gateway bus, called a dual bus interface as shown in figure 4.10. If not possible, arbitration has to be used on a per-access base.

Whereas Message RAM based communication controllers have message buffers to distinguish incoming messages, FIFO based communication controllers have to evaluate the incoming messages by instructions in the IRAM. Specialized commands have been implemented to do the filtering between messages intended for the Gateway Control Unit and the CPU in an efficient way.

Configuration RAMs As described above, the Gateway Control Unit contains two configuration RAMs. A block diagram of the implementation is shown in figure 5.7.

The VRAM configuration selects the communication controller buffers to be used by the Gateway Control Unit. It also contains additional processing information, like a vector to the event handling functions in the IRAM. The Gateway Control Unit detects received messages in the communication controllers and time events like timeouts, bouncing messages and transmit cycles. Three partitions constitute the VRAM.

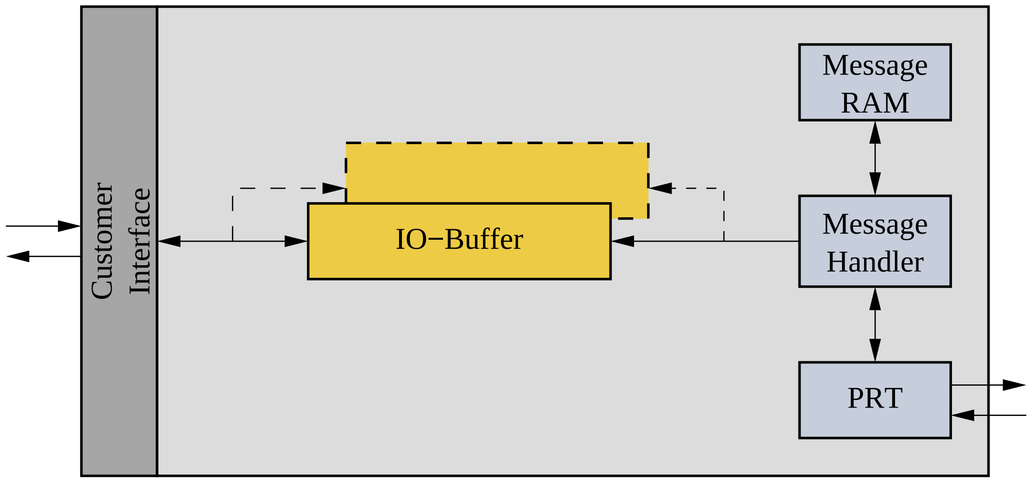


Figure 4.9: Standard Interface Concept

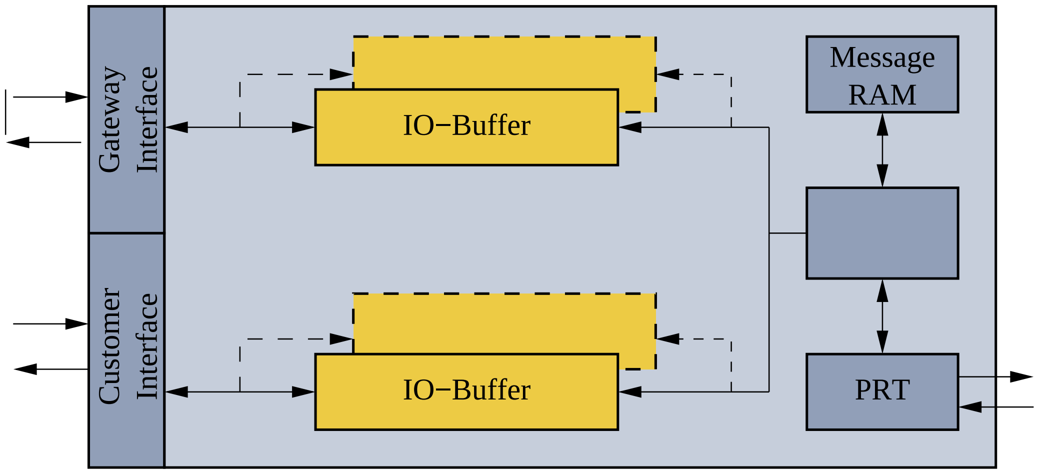


Figure 4.10: Extended Interface Concept

The VRAM Communication Controller partition (VRAM-CC) contains a specific entry for each group of message buffers. When a receive event or a time event occurred in a group, the FSM is triggered to process a table in the VRAM partition addressed by the MO-Vector.

VRAM Message Object partitions (VRAM-MO) have a variable length and contain in tabular form detailed information for each message buffer/object in the corresponding group of the VRAM-CC partition, like rx/tx configuration, timing conditions and the instruction vector to the event handling function in the IRAM.

The partitioning between the VRAM-CC and the VRAM-MO reduces the memory usage, as only the message buffers used by the gateway need to have an entry for the message object. It also reduces the time needed to look up a message object.

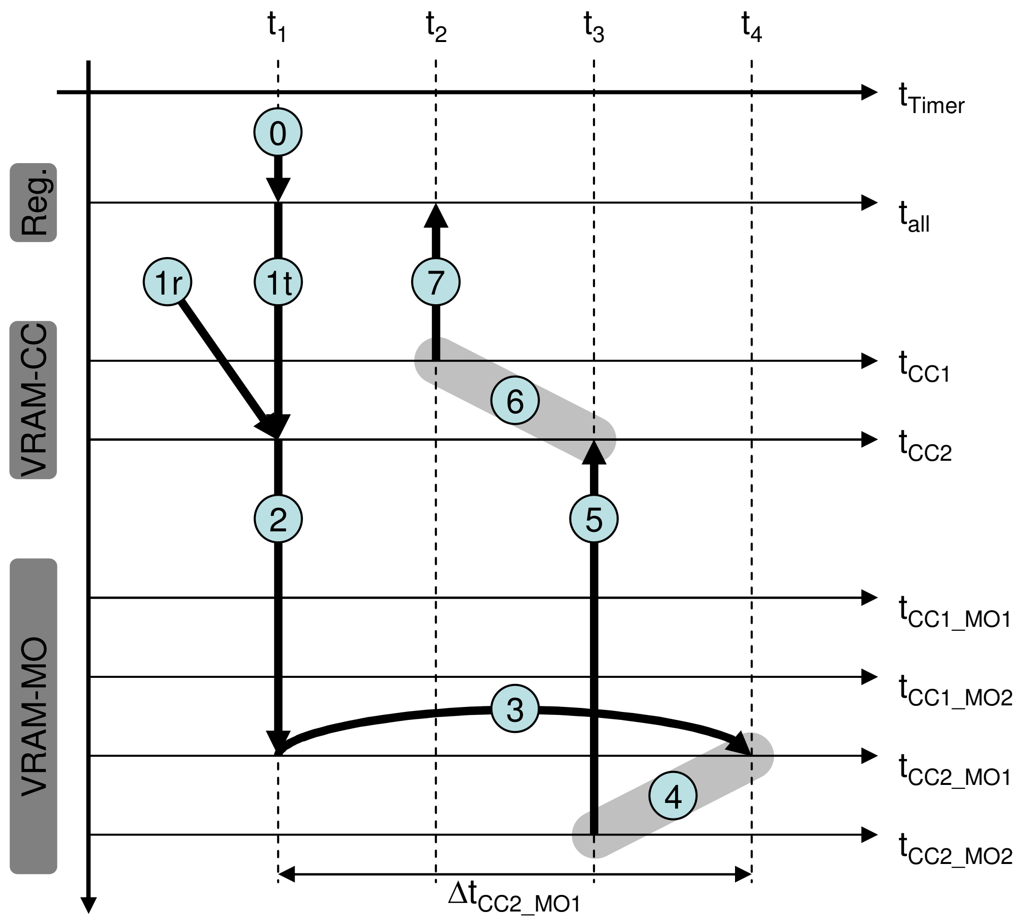
Remaining VRAM can be used as data storage and is therefore called VRAM-Data.

A state machine processes the events by executing the procedures in the Instruction RAM. The instruction set of this processing unit contains specific functions to access the communication controllers, to transfer data, to handle transport protocols and to interact with the host CPU.

Time Behavior The figure 4.11 shows the next time events of a simple gateway with two communication controllers and two gateway relevant message buffers each. The four lines on the bottom of the diagram show the two message buffers of both communication controllers. In the middle one line for every communication controller shows the next time events over all message objects of the respective controller. On the top the global next time event register over all communication controllers is displayed.

Time Event ⓪ A time event t_1 is triggered, when the continuously running timer (t_{Timer}) reaches a value greater or equal to the global next time event register (t_{all}).
$\textcircled{1t}$ When this happens, the communication controllers are scanned until the controller having the next time event is found.
② On the next level, the message buffers of this communication controller are scanned for their next time event.
③ If the triggering buffer is found, the next time event is calculated as a sum of the current time event and a stored difference to the next time event (t_4). Only then, the handling function of the instruction RAM is started.
④ When the execution is finished, the next time events (t_3 and t_4) within the scanned message buffers are searched and
⑤ the entry for the communication controller is updated with this time (t_3).
⑥ This search for the next time event is then done for all communication controllers (t_2 and t_3) simultaneous.
⑦ With this time the global next time event register is updated (t_2).

Receive Event $\textcircled{1r}$-⑦ The execution of a receive event is similar, but in place of the global timer comparison with the next time event register and the scan for communication controllers, the process directly starts for the triggering communication controller.



↓	evaluation of time vectors	t_{CC2_MO2}:
↷	calculation of next time event	next time event of 2. MO of 2. CC
▬	comparison for next time event	Δt_{CC2_MO1}:
↑	update of time vectors	time difference between time events

Figure 4.11: Time Behavior

Specialization The described process for time events is only done for cyclic sending.

In case of timeout handling, the process happens twice. $\textcircled{1r}$-⑦ The first part is equal to the described receive event process, but the calculated next time event is when the timeout triggers.
⓪-② When no further message are received and therefore the next time event is not advanced, the timeout triggers and a handling function is executed.
⓪-⑦ The timeout handler can also be executed in a cyclic manner, every time the next time event is updated.

For debounce handling, the process must be three-parted. $\textcircled{1r}$-⑦ The first part is a normal receive event.
② A debounce bit in the message buffer entry is set and the next time event is set to the time, where this bit is removed again.
$\textcircled{1r}$-② Until then no further receive events are processed.
⓪-② When the next time event is reached, the debounce bit is removed again.

Message Group Size The size of the groups of message buffers ($n_{MBperGrp}$) in the VRAM-CC partition should be dimensioned in a way that the average search time ($t_{AvgSearchTime}$) to find a particular message buffer is minimized. The terms in the following formula describe the search time through the VRAM-CC and VRAM-MO partitions under the assumption that only a part (p_{used}) of all message buffers (n_{MB}) is used:

$$t_{AvgSearchTime} = \frac{n_{MB}}{2 \cdot n_{MBperGrp}} + \frac{p_{used} \cdot n_{MBperGrp}}{2}$$

The figure 4.12 shows, that because the VRAM-CC partition acts as a hash table for the VRAM-MO entries, there is a logarithmic dependency between the number of message buffers (n_{MB}) and the number of message groups ($n_{MBperGrp}$). The average usage of the message buffers has been assumed as 80%.

As more than 250 message buffers have to be expected for a high-end gateway, a group size of 24 seems to be appropriate. For the hardware implementation the size should be a power of 2, therefore every group should contain 16 or 32 message buffers. The implementation will use 32 message buffers per group.

4.1.4 Software Gateway

Interaction with the multi-protocol gateway is possible between the host CPU and the gateway itself. Using the following interaction types, all functionality

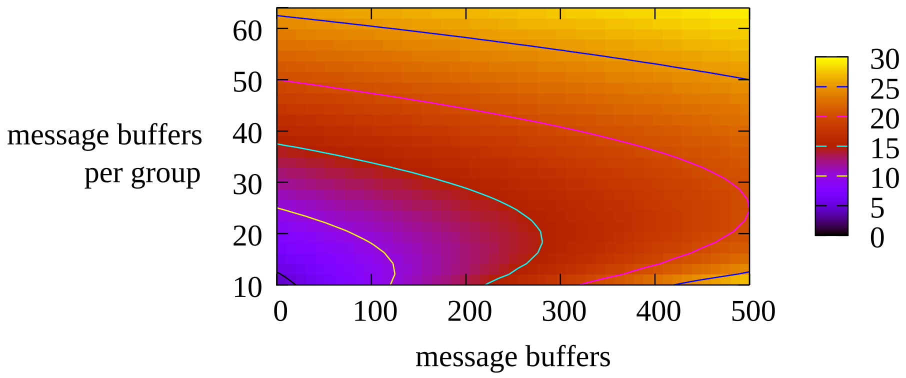


Figure 4.12: Average Search Time

that cannot be provided or are not useful to implement in hardware, can still be implemented as software components running on the host CPU.

VRAM-Data The VRAM-Data partition can be used as shared RAM to exchange data between the CPU and the gateway. For example, it is possible that the host CPU takes and processes sensor data and store this data as signals in the VRAM-data partition. The gateway can then use these data and integrate it in a frame that is send out periodically. In the other direction it is also possible that the gateway already preprocesses data for the CPU by disassembling received frames and storing its information in the VRAM-Data partition. This is known as process data unit (PDU) acceleration.

CPU Trigger Register With the usage of the CPU trigger register, the CPU is able to start instruction execution on the gateway. This is useful if the gateway should be informed about new data stored in the VRAM-Data partition and to immediately start processing of these data, i.e. to send a frame with these data.

CPU Software Interrupts The gateway is also able to inform the CPU about certain events, such as received frames, timeouts or stored signals in the VRAM-Data partitions.

Conditions Certain conditions are known in the configuration format, e.g. start-condition, stop-condition or active-condition. Depending on the conditions of the gateway system or the complete network, the software needs to activate or deactivate certain mappings or change its behavior in the hardware gateway.

Special Timing Conditions Also when special timing conditions are required beyond the possibilities provided in hardware, the software can evaluate these conditions and inform the gateway about the results or directly reconfigure its mappings. This is especially true, when multiple message debounce times or timeouts are defined on one mapping. As the number of hardware timers per mapping is limited, the software can evaluate the timers instead.

Transport and Diagnostic Protocols The usage of transport and diagnostic protocols is mostly limited to the maintenance of the car. Also different protocols are used by every manufacturer. As this requires a high flexibility, this functionality is not efficient to implement in hardware. Transport protocols also require huge state machines that can better be implemented in software.

Configuration Formats As one part of the gateway is running in hardware and the other part is running in software, a shared configuration database is necessary. The modern configuration format FIBEX provides exactly this functionality.

4.2 Configuration Toolchain

A toolchain has been developed to generate configuration images for the new gateway architecture.

Being able to use both FIBEX and DBC files as an input, it provides a standardized interface to most multi-network configuration tools, e.g. the DeComSys Designer Pro [81].

The toolchain outputs information for the hard- and software parts of the gateway as shown in figure 4.13.

The hardware information is provided by RAM configuration images for the Gateway Control Unit. Therefore the development was based on the specific RAM layout of the GCU. Residual routing information is generated for the software part.

In the future, the toolchain may easily be expanded to support AUTOSAR by an appropriate input filter or converter.

The processing stages are comparable to that of a typical compiler, apart from the different input data and optimization options as shown in figure 4.14.

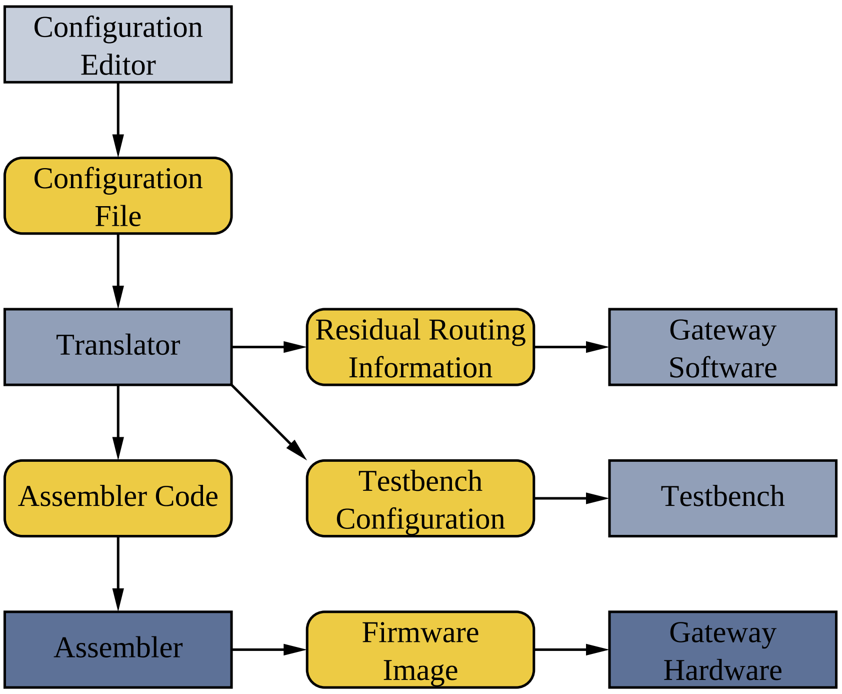


Figure 4.13: Gateway Toolchain Overview

4.2.1 FIBEX Engine

Section 3.5.1 describes FIBEX to contain all information to configure nodes on a network including gateways.

The FIBEX engine provides modules for any kind of element provided. It is possible to load and store ID-referenced objects and to change or add certain values.

Manufacturer Extensions Minor things still have an imprecise definition in FIBEX. Required information for the described gateways is also not available. As described in section 3.5.1 FIBEX allows to define manufacturer extensions to expand the descriptive capability even further.

A Manufacturer Identifier Extension has been defined to clarify certain protocol-specific aspects, like the CAN frame format (distinction between 11-Bit or 29-Bit identifiers).

Further a Manufacturer Frame Mapping Extension allows the description of gateway-specific features, like routing signals depending on certain conditions.

The Manufacturer Controller Extension is defined to enable the interaction with a software gateway on top of the hardware gateway, providing coordina-

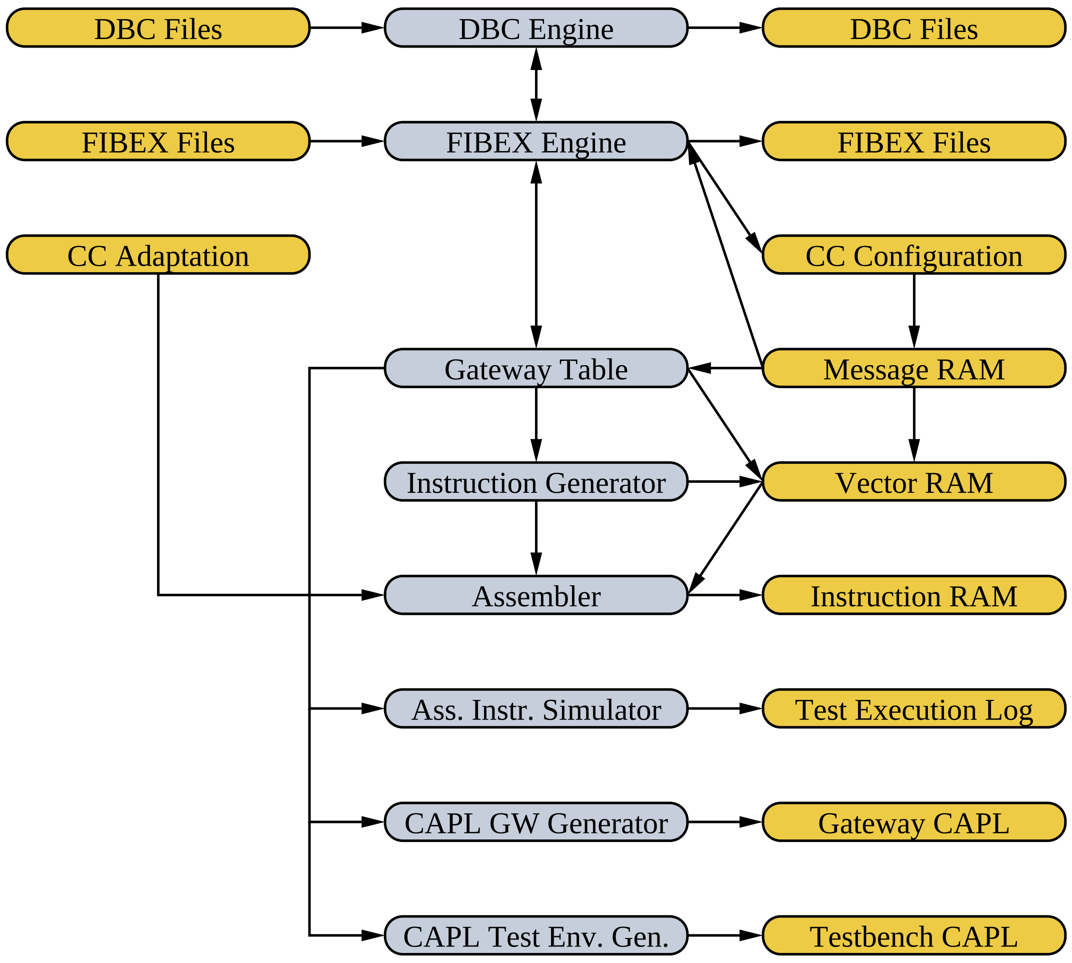


Figure 4.14: Gateway Processing and Data Flow

tion between hardware and software by predefining buffer configurations and partitioning of the gateway tasks.

4.2.2 Import/Export of CANdb++ DBC databases

DBC Databases can be used to configure CAN channels as described in section 2.2.4.

The DBC engine is a module that can generate DBC channel databases based on the information from the FIBEX file. It also allows to import DBC databases including routing definitions into the loaded FIBEX file.

4.2.3 Configuration of Communication Controllers

The first processing step is to generate configuration register sets and Message RAM contents for all involved communication controllers. This can be done for any communication controller and runs independently from the rest of the gateway toolchain. The output of this module is presented as C-style header file.

4.2.4 Gateway Configuration

The gateway configuration involves multiple data processing modules.

Gateway Table The gateway object in FIBEX contains all connector, frame-triggering and signal-instance mappings. All required information is read, sorted and stored in an internal gateway table data structure. At this state the gateway table only contains symbolic references to the message buffers of the communication controllers. By searching the generated Message RAM contents, the symbolic references can be resolved and the correct locations added to the gateway table. Aside from using the gateway table as a debug point, it also provides an abstraction of different input formats in the future, e.g. AUTOSAR.

Timer Configuration The gateway table also contains the timing- and trigger-conditions of every mapping. From this data the frequency of the global FSM timer can be computed. The frequency must be the least common denominator, that multiplied fits most times used in the gateway, e.g. cyclic sending or timeouts.

Vector RAM The gateway table and timer configuration allows generating the Vector RAM contents. However, the address vectors to the actual event handling functions in the Instruction RAM are still symbolic and need to be resolved after the generation of the Instruction RAM contents.

Instruction RAM The Instruction RAM contains the functions to handle the receive and time events processed by the FSM. If a message cannot be processed completely in hardware, a special function is generated that can be triggered by the CPU. This information is stored in the filtered FIBEX output by a manufacturer extension of the FIBEX controller object. A software layer has to use this information to interact with the Gateway Control Unit.

Depending on the timing- and trigger-conditions a high-level assembler code is generated.

Runtime and Memory Optimization By choosing different optimization settings of the instruction generator, runtime-optimized or memory-optimized codes can be generated.

4.2.5 Assembler

The assembler contains several processing steps to generate machine code out of the generated assembler code. The complete flow is shown in figure 4.15.

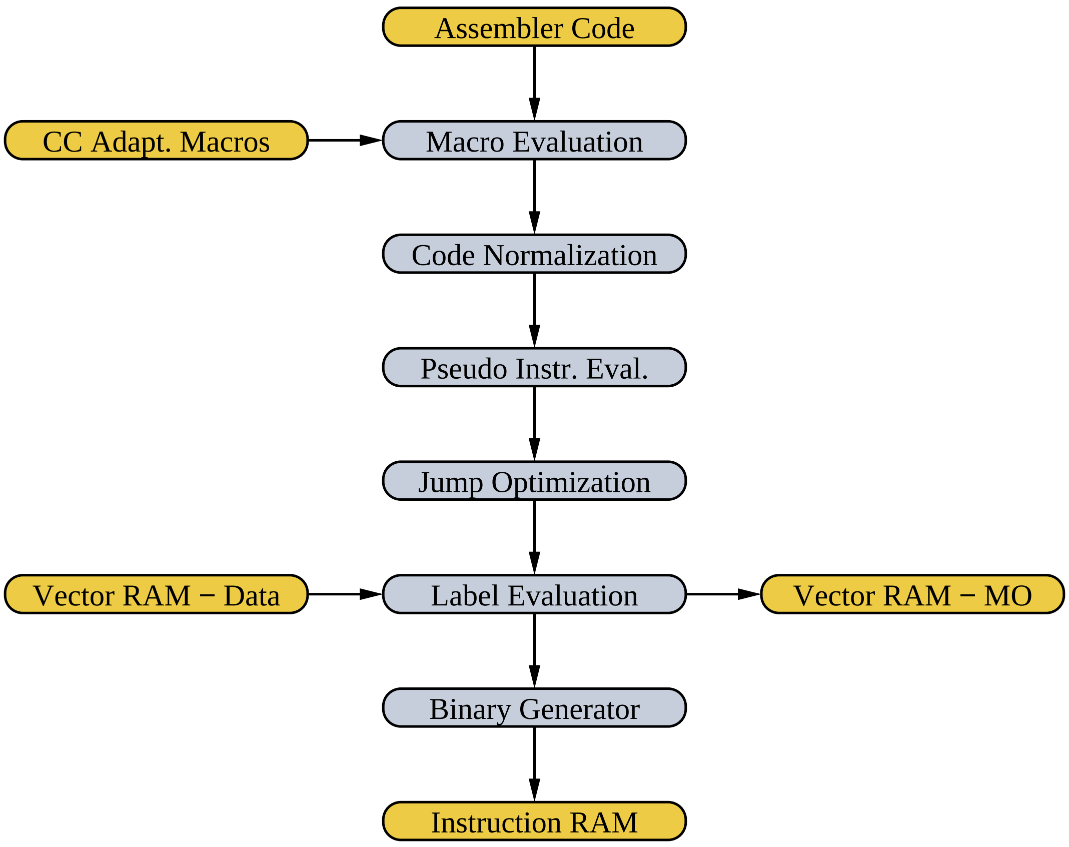
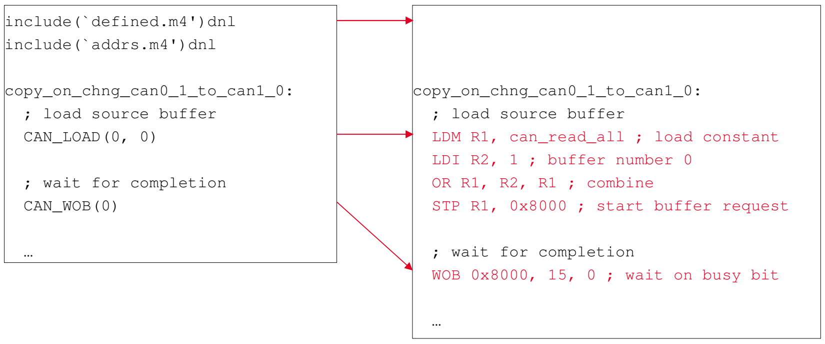


Figure 4.15: Gateway Assembler Processing Flow

Macro Evaluation The high-level assembler code contains mostly macro functions and is therefore almost independent of the target gateway. By evaluating the macros in the assembler code, it becomes more specific toward the communication controllers and their base addresses in the gateway as shown in figure 4.16. The macro expansion is done by calling the GNU m4 macro processor [82]. It interprets the macros and replaces them by the actual assembler code.



```
include(`defined.m4')dnl
include(`addrs.m4')dnl

copy_on_chng_can0_1_to_can1_0:
  ; load source buffer
  CAN_LOAD(0, 0)

  ; wait for completion
  CAN_WOB(0)

  …
```

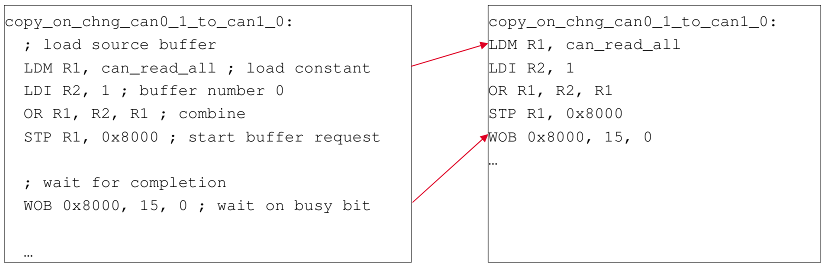
```
copy_on_chng_can0_1_to_can1_0:
  ; load source buffer
  LDM R1, can_read_all ; load constant
  LDI R2, 1 ; buffer number 0
  OR R1, R2, R1 ; combine
  STP R1, 0x8000 ; start buffer request

  ; wait for completion
  WOB 0x8000, 15, 0 ; wait on busy bit

  …
```

Figure 4.16: Macro Evaluation

Code Normalization During the normalization, the code is cleaned of any comments, all labels put to extra lines and an unified indentation is made as shown in figure 4.17.



```
copy_on_chng_can0_1_to_can1_0:
  ; load source buffer
  LDM R1, can_read_all ; load constant
  LDI R2, 1 ; buffer number 0
  OR R1, R2, R1 ; combine
  STP R1, 0x8000 ; start buffer request

  ; wait for completion
  WOB 0x8000, 15, 0 ; wait on busy bit

  …
```

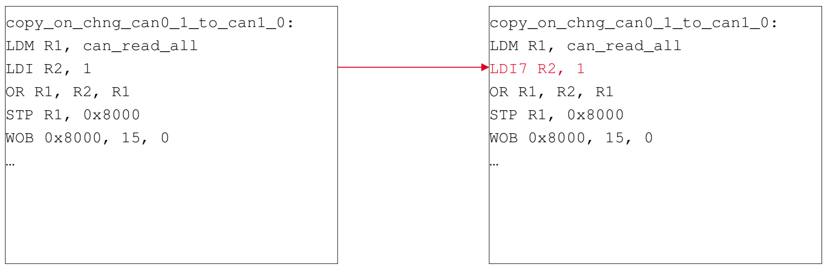
```
copy_on_chng_can0_1_to_can1_0:
LDM R1, can_read_all
LDI R2, 1
OR R1, R2, R1
STP R1, 0x8000
WOB 0x8000, 15, 0
…
```

Figure 4.17: Code Normalization

Pseudo Instruction Evaluation Some pseudo instructions are symbols representing other instructions and need to be translated in the pseudo instruction evaluation stage as shown in figure 4.18.

Jump Optimization The JMP command is 32-Bit long and takes two cycles to execute. It is used for long jumps as absolute addresses are used. Opposite to JMP the BRA command is 16-Bit long and takes only one cycle to execute. It is therefore used for short jumps using relative addresses. Sometimes the generated code contains BRA commands to addresses impossible to reach.

Therefore this optimization stage first changes every BRA to a JMP. Then it changes iteratively every JMP back to a BRA whenever it is possible, which



```
copy_on_chng_can0_1_to_can1_0:
LDM R1, can_read_all
LDI R2, 1
OR R1, R2, R1
STP R1, 0x8000
WOB 0x8000, 15, 0
…
```

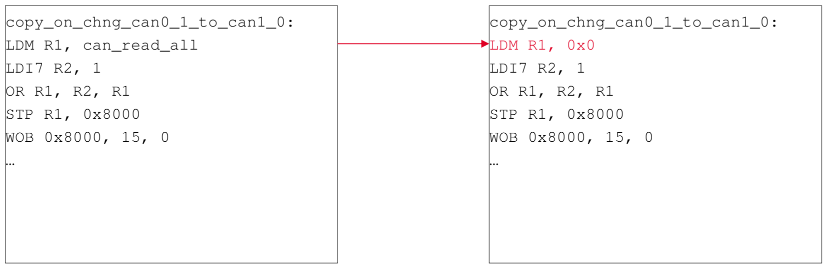
```
copy_on_chng_can0_1_to_can1_0:
LDM R1, can_read_all
LDI7 R2, 1
OR R1, R2, R1
STP R1, 0x8000
WOB 0x8000, 15, 0
…
```

Figure 4.18: Pseudo Instruction Evaluation

shortens the code until no further optimizations can be done. This order is necessary, as it is impossible to just change a BRA to a JMP without affecting surrounding branch instructions.

After this optimization no BRA instructions remain with too distant addresses. As many JMPs are changed to BRAs the code is shorter and faster to execute.

Label Evaluation The data and instruction labels are evaluated in the label evaluation stage as shown in figure 4.19. Data labels point to entries in the VRAM-Data partition and instruction labels point to locations in the assembler code. After the evaluation, the actual address of every function in the IRAM is reported back to the VRAM-MO entries to resolve the symbolic references.



```
copy_on_chng_can0_1_to_can1_0:
LDM R1, can_read_all
LDI7 R2, 1
OR R1, R2, R1
STP R1, 0x8000
WOB 0x8000, 15, 0
…
```

```
copy_on_chng_can0_1_to_can1_0:
LDM R1, 0x0
LDI7 R2, 1
OR R1, R2, R1
STP R1, 0x8000
WOB 0x8000, 15, 0
…
```

Figure 4.19: Label Evaluation

Output Filters Different output filters can generate C-style header files or binary images containing the contents of the VRAM and IRAM.

The C-code output filter generates images containing C structures, thus can be compiled statically for the configuration function of the gateway.

Binary images can be generated and distributed as part of a firmware package containing the static part of the gateway software and the configuration image.

The ModelSim output filter can be used to generate VHDL or testbench code for RTL simulations with specific gateway configuration.

Assembler Frontend An assembler frontend can directly read assembler code from a text file and generate different output formats, like C-style header files or binary images. This allows assembling manually edited files.

4.2.6 Verification and Tests

Input and output data of every module in the toolchain can be loaded from and saved to external text files. This allows intermediate white box testing of every module and manual editing and viewing. There exist different test environments around the complete toolchain and single modules.

Test cases A set of test cases provides input data for specified modules and functions and provides expected output data, which is then compared to the actual output. This makes testing easier, as only a limited set of functions need to be tested. The testbench also works as regression test against the changed software to ensure that the changes made in the current software do not affect the functionality of the existing software.

Coverage Analysis This same set of tests is also used for code coverage tests [84, 85]. A subroutine, statement/line, branch and condition coverage analysis is made. By looking at these tabular reports, many unnecessary condition checks and dead code fragments can be prevented. It also shows exactly, which code fragments or conditions are still only partially tested and need additional test cases.

Static Code Checker Static code checking is done by a lint tool [83]. It shows unused variables, imprecise descriptions and coding mistakes.

C-code Integration Tests Finally integration tests are made by compiling the generated C-style output files with parts of the actual software gateway environment. The outputs produced when running these executables, are then again compared to the expected outputs. This prevents errors in the source codes generated by the output filter, especially when parts of the software gateway change.

4.2.7 Assembler Instruction Simulator

An assembler instruction simulator has been developed. It simulates the function of the IRAM-FSM, the communication controllers and the Message RAMs. A block diagram is shown in figure 4.20.

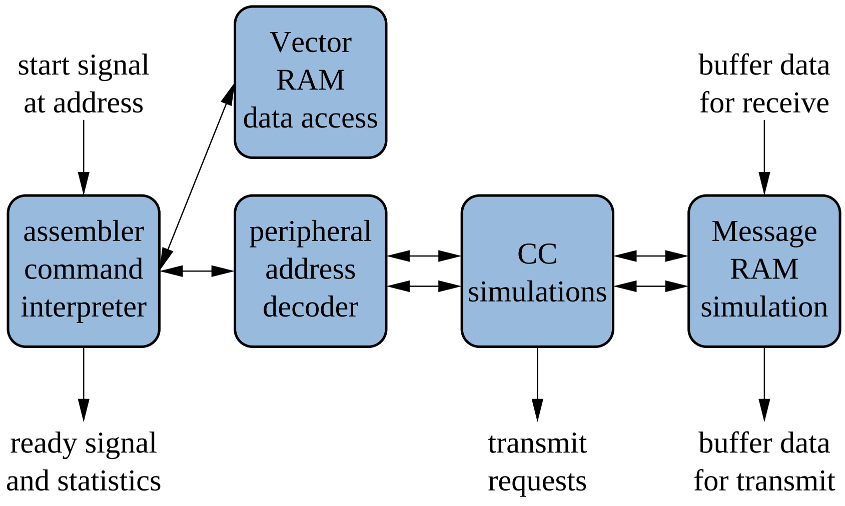


Figure 4.20: Assembler Simulator

Assembler Instruction Command Interpreter A Assembler instruction command interpreter simulates the behavior of the IRAM-FSM. It holds the current state of the registers, flags and current execution location. With each executed instruction the state is altered and counters for instruction cycles and read memory are increased accordingly. One function group is provided to simulate data accesses to the Vector RAM.

Communication Controllers Simulations All read and write accesses in the peripheral address space are first processed by a peripheral address decoder and second by a very simple simulation of the communication controllers. Only transfers between virtual Message RAMs and the interface buffers are possible. Write access that leads to message transmits are logged.

Message RAM Simulation Virtual Message RAMs used by the simulated CCs are directly accessible. Receiving messages can be simulated by directly manipulating the data therein and triggering the corresponding IRAM receive function. As message transmits are logged and the corresponding data can be directly read out of the virtual Message RAM, sending of frames can be simulated.

Equality Checks The assembler instruction simulator can be used to check the equality of the transactions made by a function under different optimization levels. With such an environment each transaction for a specified timing- and trigger-condition can be tested for every function in the Instruction RAM.

Function Checks Also the assembler simulation of the gateway allows to test the generated lines of assembler code under all possible time conditions, e.g. timeout, message debounce. To check each code segment, the test results are compared to the reactions of the executive model.

4.3 Verification Environment

Safe operation of a critical system can only be guaranteed with an intensive test system. The usual approach is to manually implement a test environment using a remaining bus simulation. Manual programming has several disadvantages. Checking the complete range of possible gateway mappings and test parameters is very expensive. Also manual programming can lead to errors not only in the implementation of the gateway function in hardware and software, but also in the test environment. This is especially true when changes to the gateway configuration also affect the test environment. Manual changes are always error-prone and often result in inconsistencies between gateway and test environment. These disadvantages can be eliminated by using tools to automate most of these tasks and by referring to a common source of information for gateway and test configuration.

White/Black Box Tests Tests can be divided into two main categories. The first category are white box tests, where information about the device under test are required to create and execute them. This is especially true, when only parts of the gateway should be examined. On the other side black box tests can be used on any device without any prior knowledge required. If black box tests include measurements, they can also be used to compare different implementation.

4.3.1 Gateway Executive Model

A base for multiple tests is provided by the implementation of an executive model. One possible usage is the comparison of the actual gateway implementation with the calculated behavior of the executive model under different stimuli. This comparison can either be implemented as static test cases or dynamic in form of a test program.

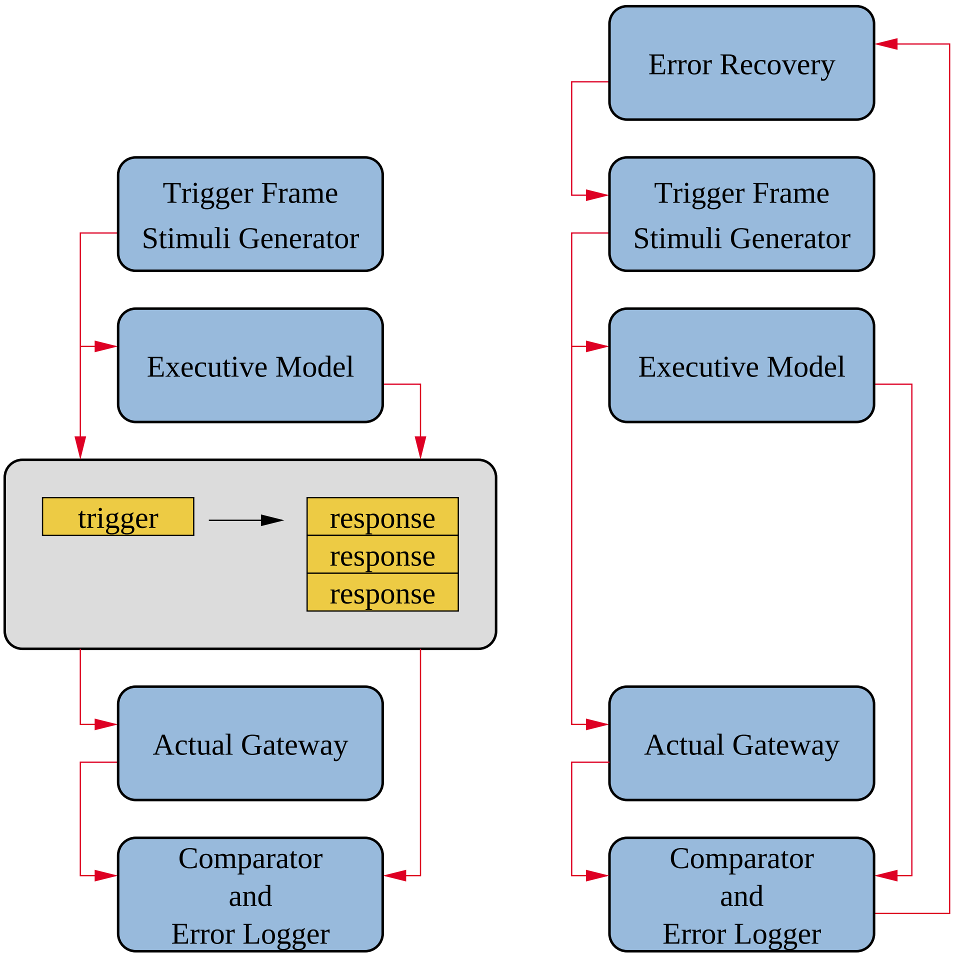

Error Recovery
Trigger Frame Stimuli Generator
Executive Model
trigger
response
response
response
Trigger Frame Stimuli Generator
Executive Model
Actual Gateway
Comparator and Error Logger
Actual Gateway
Comparator and Error Logger

Figure 4.21: Static and Dynamic Testing

Static Tests Every static test contains trigger frames and calculated response frames. The comparison can either be made while the test case runs or the response can be logged for later comparison against the calculated response frames.

Dynamic Tests The executive model can also be implemented in a test program. Thereby the trigger frames are dynamically generated and compared to the response frames on every message reception. Erroneous data can be logged to find certain patterns that can give hints in finding implementation bugs. This environment can also be used for long time regression tests, when as soon as an error is detected, the gateway is forced back to a known state by appropriate trigger frames.

Implementation This Executive Model (EM) of the gateway as shown in figure 4.22 is derived from the FIBEX Object Model. It is currently implemented as part of the toolchain, but can easily be ported to other environments to be used for dynamic testing.

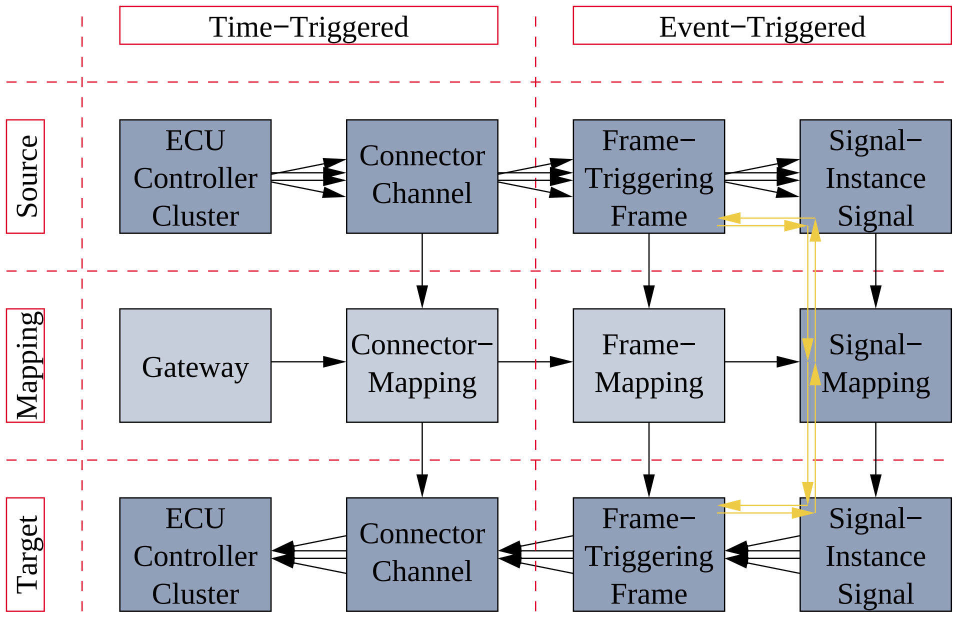


Figure 4.22: Gateway Model

Event-Triggered Communication A channel contains frames and a frame contains signals. The task of a gateway is to receive source frames and to generate and send target frames based on the defined mappings. A frame mapping can be substituted by a signal mapping with one signal definition over the complete bit range. Despite the delay times introduced by the reception mapping and transmissions, this part of the model can be used to simulate event-triggered communication.

Time-Triggered Communication Time-Triggered communication introduces additional delays between the message send request and the actual transmission time, so that each message is sent within an assigned time slot. For a simulation of time-triggered communication additional objects are necessary, that introduce the behavior of every frame in a communication channel and cluster. Implementing the time-triggered part of the model can be circumvented, when the trigger frames are send and the results are compared at specific times, e.g. sending before and receiving after the desired slot.

Backward Calculation of Trigger Frames The event-triggered part of the model can be used to generate trigger frames. These trigger frames are source frames that trigger a desired mapping under specific conditions, e.g. timeout, bouncing or data change. The backward calculation is necessary to find the trigger frames that are relevant to test the desired signal mapping. To simulate changed data the source frames or source signals can be randomized.

Forward Generation of Trigger Response Frames When the trigger frame is sent out, the contained source signals can trigger multiple mappings. Therefore multiple target signals can be produced leading to multiple target frames. The forward calculation is used to generate all response frames to a given trigger frame.

Response Filter In order to avoid checking all response frames of a trigger frame, a filter can be used to only check the relevant information regarding the desired mapping, e.g. target signals.

4.3.2 Automatic Test Generator

With the described backward and forward calculations the executive model can be used to automatically generate static test cases containing trigger frames and expected response frames for each mapping in the gateway configuration. This is called Automatic Test Generation (ATG). These static test cases can be generated for different environments.

Network Analyzer can be used to generate and monitor the traffic on physical bus systems. If they support multiple different bus systems and a simple scripting language, it is possible to use them as gateway test environment. Connected to the gateway, they can simulate real traffic and monitor the gateways responses. The simulated traffic can be described by static test cases. The correctness of the replies can be verified by condition checks in the script language, e.g. CAPL in Vector CANoe, or by examining the log files.

RTL Simulation An essential part of any hardware development is implementing test cases on the register-transfer-level (RTL) using a simulation tool, such as ModelSim. A new approach is that the test generator can also be used for automatic test case generation of RTL code in the different languages, e.g. VHDL, Verilog or SystemC. With this approach the gateway can be tested with an ideal time behavior and identical test patterns.

When ModelSim is executed with these input files, it can generate trace files in an application specific format. These log files can then be parsed and filtered to find the trigger response frames. Differences of the response frames to that in the static test cases indicate errors.

Assembler Simulation As mentioned in section 4.2.7 the executive model is also used to check the generated assembler code of each mapping under every possible timing and trigger-condition. The trigger frames contained in the test case are used to execute the simulator. Response frames from the simulator are logged and compared to those from the test case. As this test environment does not need any external hard- or software, it can be used as internal self test running after each gateway configuration run.

Chapter 5

Implementation and Results

In this chapter some of the results of both the CAN-CAN and multi-protocol gateways will be presented, along with the configuration toolchain and verification environment.

5.1 Gateway Development Hardware

The hardware development required an FPGA[93] device with enough logic elements for the gateway IP and the communication controllers. It must also provide connections for the physical layer drivers and must be fast enough to run the gateway system with the target clock frequency.

5.1.1 Development Board

Altera offers such a development board, the Altera Nios II Development Kit, Stratix II Edition as shown in figure 5.1.

The development board contains an Altera Stratix II EP2S60 (EP2S60F672C3) FPGA, 16 MByte SDRAM, 16 MB Flash and an Ethernet MAC.

A hardware toolchain is provided with Quartus II for synthesis and place-and-route of the VHDL code of the gateway. Several IP cores are provided as add-ons including an ethernet IP and a highly configurable and FPGA-optimized Nios II processor core. The SOPC Builder (System-On-a-Programmable-Chip) allows automatic connection of several of these modules using the Avalon Switch Matrix, similar to a system bus.

Eclipse as graphical IDE integrates a specialized GNU Compiler Toolchain for the software development and compilation of sample programs and gateway initialization programs. The environment comes with a development license for

Figure 5.1: Altera Nios II Development Board [92]

the MicroC/OS-II real-time operating system and a license for the lightweight TCP/IP Network Stack.

5.1.2 Physical Layer Board

Network specific boards containing the physical layer drivers can be connected on top of the board using the two expansion/prototype headers.

CAN Physical Layer Board For the connection to four physical CAN channels an expansion board has been designed to contain four Bosch CF151 CAN Physical Layer Transceivers [31, 38] or compatibles.

FlexRay Physical Layer Board Another board was manufactured containing two TJA1080 Physical Layer Transceivers from NXP (former Philips Semiconductors) or compatibles for the connection to a FlexRay network with two channels [28].

MOST Physical Layer Board For later integration of MOST communication into the gateway, a module containing the OS81050 Network Transceiver from SMSC (former OASIS SiliconSystems) with optical connection header has been purchased [42, 46]. Further description of the OS81050 is provided in section 2.5.2.

5.1.3 Clock Frequencies

When connecting modules with different clock frequencies to the Avalon Switch Matrix, clock domain crossings are automatically generated by the SOPC Builder.

The figure 5.2 shows the clock frequencies of the multi-protocol gateway. Different from this figure, the CAN-CAN Gateway omits the FlexRay Domain and the Domain Wrappers. The Gateway Bus is replaced by the cascade ring bus between the communication controllers. This is shown in figure 5.4.

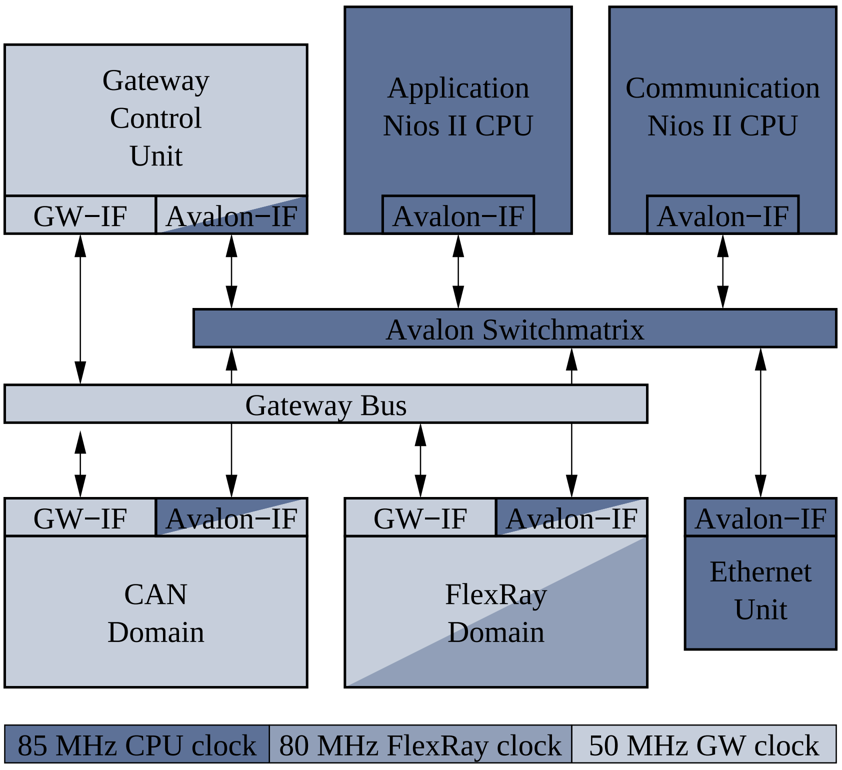


Figure 5.2: Clock Frequencies

The Application Nios CPU is implemented as the primary CPU of the ECU and the Communication Nios CPU is for the communication with the development host PC. Both CPUs and the Avalon Switch Matrix run at 85 MHz.

Most parts of the gateway, including the Gateway Control Unit, the Gateway Bus, the complete CAN Domain and the interface parts of the FlexRay Domain run at a clock frequency of 50 MHz.

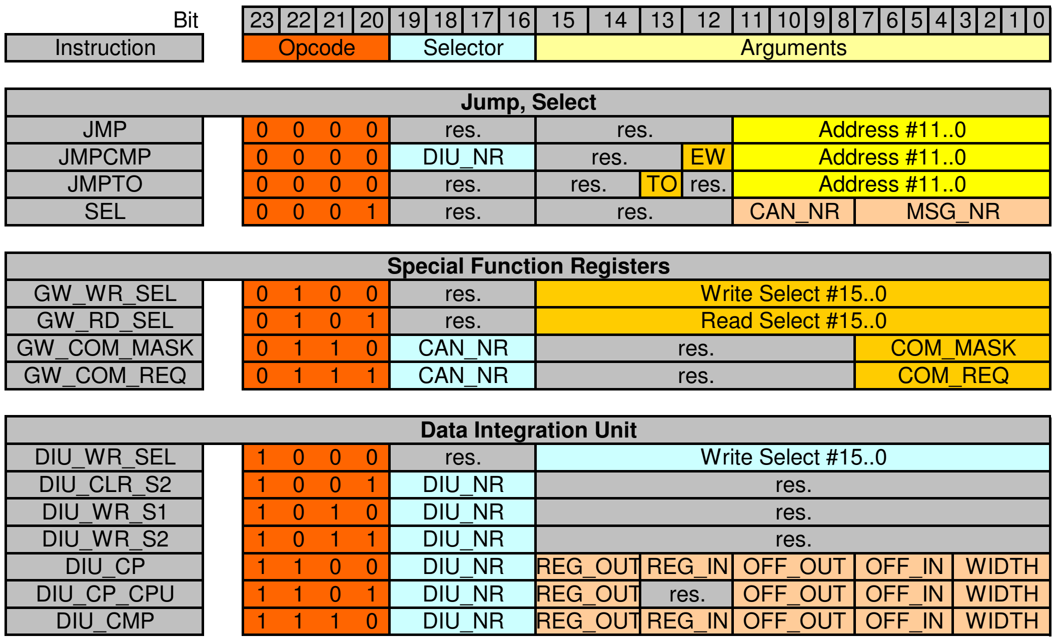
Specific for the two-clock design of the E-Ray controller, the bus side of the controller has a frequency of 80 MHz.

5.2 CAN-CAN Gateway Comparisons

This section describes some of the implementation details and measurement results of the CAN-CAN gateway.

5.2.1 Instruction Set

As described in section 4.1.2, the Gateway Control Unit has two configuration RAMs. The Vector RAM configures event triggers (message receive or time events) and the Instruction RAM of the FSM gateway defines actions to handle these events. Figure 5.3 shows the three instruction groups for flow control, communication controller access using the Special Function Registers (SFR) and Data Integration Unit (DIU) control.



Bit	23	22	21	20	19 18 17 16	15	14	13	12	11 10 9 8	7 6 5 4	3 2 1 0
Instruction	Opcode				Selector	Arguments						
Jump, Select												
JMP	0	0	0	0	res.	res.				Address #11..0		
JMPCMP	0	0	0	0	DIU_NR	res.			EW	Address #11..0		
JMPTO	0	0	0	0	res.	res.		TO	res.	Address #11..0		
SEL	0	0	0	1	res.	res.				CAN_NR	MSG_NR	
Special Function Registers												
GW_WR_SEL	0	1	0	0	res.	Write Select #15..0						
GW_RD_SEL	0	1	0	1	res.	Read Select #15..0						
GW_COM_MASK	0	1	1	0	CAN_NR	res.					COM_MASK	
GW_COM_REQ	0	1	1	1	CAN_NR	res.					COM_REQ	
Data Integration Unit												
DIU_WR_SEL	1	0	0	0	res.	Write Select #15..0						
DIU_CLR_S2	1	0	0	1	DIU_NR	res.						
DIU_WR_S1	1	0	1	0	DIU_NR	res.						
DIU_WR_S2	1	0	1	1	DIU_NR	res.						
DIU_CP	1	1	0	0	DIU_NR	REG_OUT		REG_IN		OFF_OUT	OFF_IN	WIDTH
DIU_CP_CPU	1	1	0	1	DIU_NR	REG_OUT		res.		OFF_OUT	OFF_IN	WIDTH
DIU_CMP	1	1	1	0	DIU_NR	REG_OUT		REG_IN		OFF_OUT	OFF_IN	WIDTH

Figure 5.3: Gateway Instruction Set

This limited but although efficient instruction set allows all operations to receive and send frames and to copy and compare certain signals therein.

5.2.2 Comparison of Implementations

The efficiencies of the three gateway expansion stages (software solution, software solution combined with a separate data path using the Special Function Registers, CPU independent Finite State Machine) have been compared. This includes resulting code sizes, latencies and jitters for basic functions that reflect the workload of real gateways.

Software Gateway In the first stage the gateway functionality is conventionally done by software. The software needs to monitor the receive bits of all communication controllers and regularly check for receive timeouts and times to send a periodic message. In case of an event the software gateway has to get the data out of a CAN, rearrange it and write or send it from another CAN.

The advantage of the software gateway to competing ones is that not the gateway matrix is held in memory, but the FSM instructions, which are directly interpreted by the software gateway. This reduces latency and jitter and provides a common means of configuring the three gateway solutions.

Special Function Registers The second stage of the software gateway functionality is improved by the usage of the special functions registers to directly transfer messages from one CAN to another and to make rearrangements of data by the DIU.

Finite State Machine As third stage the software only configures the FSM to provide the functionality completely in hardware.

5.2.3 Description of Test System

The test system is synthesized on the development board as described in section 5.1. It contains the (TT)CAN Gateway with two Bosch C_CAN and two TTCAN modules, controlled by an Altera NIOS II Application CPU. A second NIOS II Communication/Traffic CPU controls two additional C_CAN modules, intended for testing the CAN Gateway. This is shown in figure 5.4.

CAN Connection Each CAN of the Test Environment is connected to a CAN and a TTCAN (running in CAN mode) of the Gateway Demonstrator by an AND gate, so that communication without a CAN bus is possible. All of the six Tx pins and the two CAN channels itself, can be monitored on FPGA pins. The test CPU and the host CPU are clocked with 80 MHz, the remote CANs and the CAN Gateway at 20 MHz.

Software on the CPUs The software application is based on Micrium μC-OS RTOS. On the test CPU a software task provides communication with a controlling Linux host using an Ethernet MAC on the development board. A

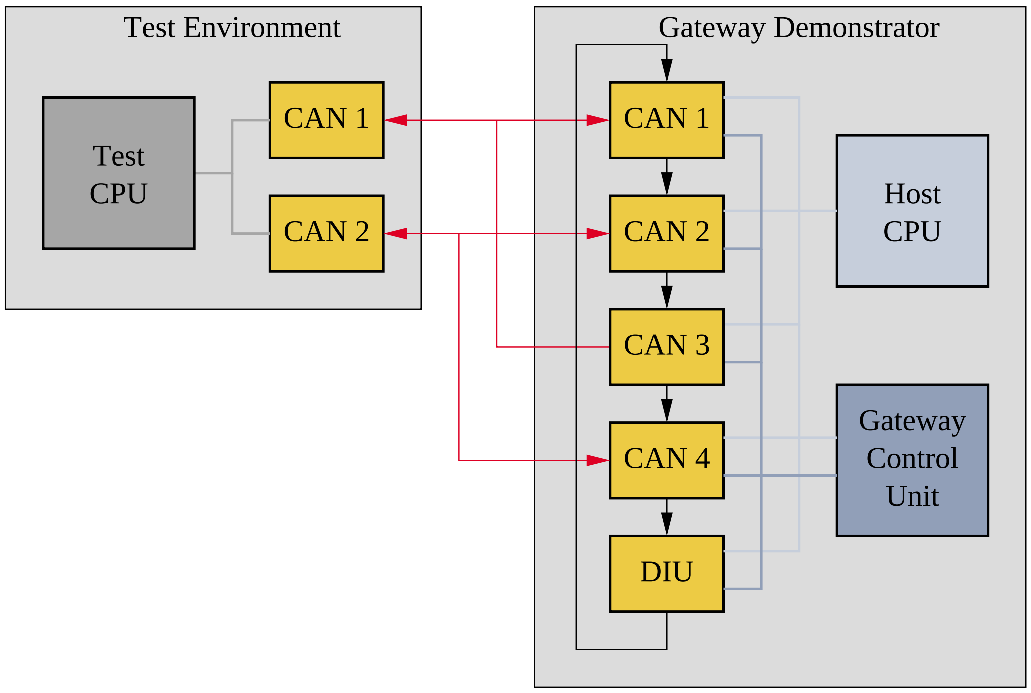


Figure 5.4: Structure of test system

second task contains several functions for testing different aspects of the CAN Gateway. This partitioning is shown in figure 5.5. User interaction is provided by buttons and LEDs on the development board.

FIFO Communication A packet based communication protocol has been defined for communication from the Linux host via Ethernet to the test CPU and via mailboxes to the host CPU and back. Two FIFO ring buffers in an on-chip shared memory provide this mailbox communication between the test CPU and the host CPU. Apart from this the host CPU is strictly limited to the gateway functionality, so that tests have no impact on the test results.

Scope For the tests an oscilloscope monitors the Tx pins from all six CANs and the two CAN buses previously described. So the CAN message (length of data, acknowledge, interframe space) and message transfer (latency, jitter) can be exactly measured. The latency is measured by subtracting the time for the acknowledge delimiter and end of frame space (EOF), before that a CAN message cannot be processed. A picture of the scope's output is shown in figure 5.6.

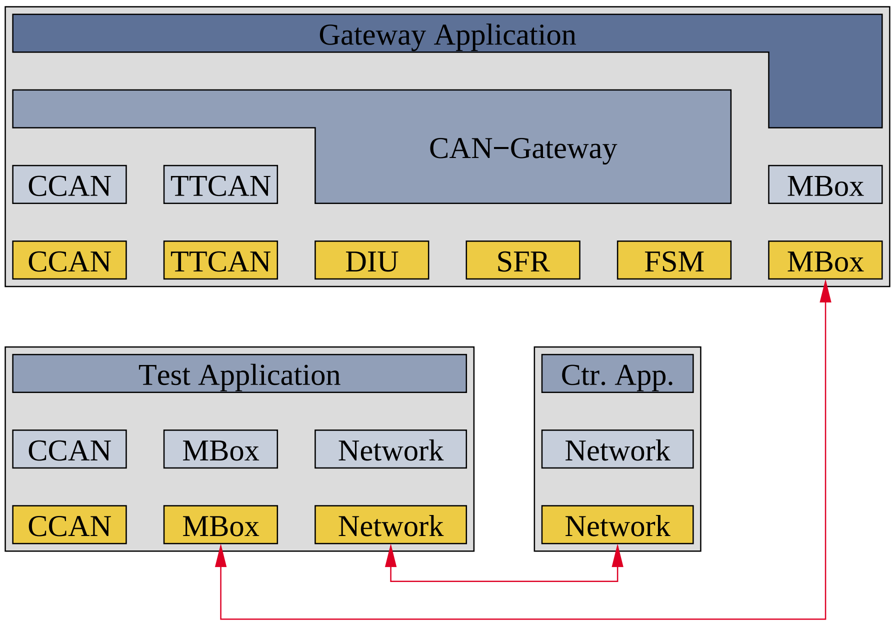


Figure 5.5: Software structure

Event Detection Loop Additionally the software triggers a signal to monitor the event detection loop, so that the maximum jitter resulting from the loop can be directly read as described above. The time to execute the event loop is static as long as no event occurs.

Instruction Execution A second signal is raised during execution of FSM instructions in case of an event. When executing instructions the event detection loop time lengthens too. The more instructions are executed, the less is the influence of the event detection loop to the overall performance. Theoretically

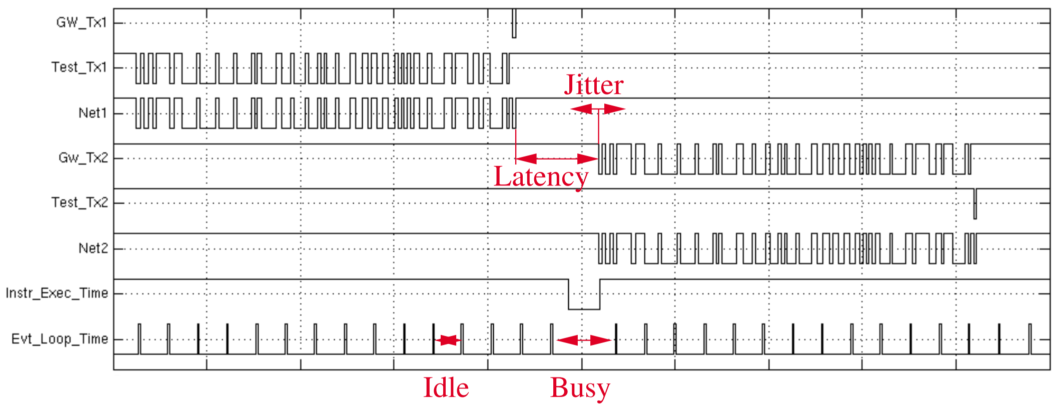


Figure 5.6: Time behavior of the CAN communication

the maximum performance is possible, when every message buffer has an event and instructions are permanently executed. So by measuring the instruction execution time, maximum performance can be calculated from the idle event loop executing time and instruction execution time.

The time to process instructions (and as such jitter and latency) is independent of the message length or the CAN bus utilization as the cascade ring bus has a constant transmission time independent of the message length. Thus the utilization only depends on the rate of messages.

5.2.4 Implemented Tests

Simple tests for the basic functionality and measurement have been implemented and more complex ones to reflect the workload of a real gateway. All other scenarios can be calculated from the following representative tests. Obviously when a process takes in average half the processors free time, then all results will have doubled durations.

Immediate Tx after Rx The "Immediate Tx after Rx" test is for measuring the latency and jitter of a transfer of a message that is received on one CAN controller and transferred to and transmitted by another CAN. Two FSM instructions are needed to read the message out of the CAN message RAM, two to put into and get it from the cascade ring and two to send it on the second CAN.

	SW	SFR	FSM
Idle event loop	5.2 μs	4.7 μs	–
Instruction time	24.7 μs	9.1 μs	–
Average latency	28.7 μs	12.1 μs	1.6 μs
Jitter +/-	2.7 μs	2.4 μs	0.0 μs

Table 5.1: Test immediate Tx after Rx

The FSM has no jitter, which is a result of the different algorithms. The latency of the software solution is high, as the data needs to be copied out of the CAN and copied into the next. Whereas the SFR supported solution only switches multiplexers to directly transfer the data onto and from the cascade ring.

Tx Cyclic Sending The "Tx cyclic sending" test is for measuring the accuracy and jitter of a periodically transmitted message at an interval of 1 ms. Accuracy and jitter can be calculated by the software triggered pins. There are only two instructions required to transmit a message. The first one is required to load the value to transmit the specific message buffer and the second to store the value in the specific CAN command register.

	SW	SFR	FSM
Idle event loop	5 μs	4.5 μs	–
Instruction time	9.3 μs	6.8 μs	–
Accuracy	994.1 μs	992.6 μs	996.2 μs
Jitter +/-	4.5 μs	3 μs	3.4 μs

Table 5.2: Tx cyclic sending

The software jitter is almost identical to the hardware jitter, as the software benefits from the limitation of message buffers to check. The latency and the shortened period as a result of inaccurate timer readings as mentioned above, results in an almost exact compliance of the time period.

Tx after Rx Timeout This test is for measuring the accuracy and jitter of a CAN message which is transmitted as a result of a receive timeout after 3 ms. It needs a conditional branch instruction which tests for a timeout. Only in case of a timeout a message transfer takes place and three instructions need to be executed.

	SW	SFR	FSM
Idle event loop	5.5 μs	5 μs	–
Instruction time	26.2 μs	9.3 μs	–
Latency	3080 μs	3060 μs	3043.6 μs
Jitter +/-	35.4 μs	35.4 μs	34 μs

Table 5.3: Tx after Rx Timeout

Even though the latency is not given, it can be read out of the accuracy, e.g. the latency of the SW solutions differs by 20 μs. Compared to the previous example the latency timings are always longer then the desired period. This is a result of the different measurements. The time difference in this case is measured from the end of the last message, instead of the beginning as in the previous test.

Swap Message A more complex test is the reception of a message and the swap of each of the four 16-Bit words before transmitting the message again.

The latency of the FSM solution is almost constant over all tests and there is again no jitter as the instructions are triggered by a received message and not by the timer.

	SW	SFR	FSM
Idle event loop	5.2 μs	4.7 μs	–
Instruction time	39.2 μs	16.7 μs	–
Latency	44.3 μs	21.2 μs	1.6 μs
Jitter +/-	3.1 μs	2.4 μs	0 μs

Table 5.4: Swap Message

Split Message Another complex test is receiving and splitting of a message and the transfer to different CAN message buffers. Again, two instructions are needed to get or send a message and one to get a message from and onto the cascade ring, altogether 15 instructions.

	SW	SFR	FSM
Idle event loop	5.2 μs	4.7 μs	–
Instruction time	50.2 μs	22.7 μs	–
Latency	30.7 μs	18.4 μs	1.8 μs
Jitter +/-	3.2 μs	2.9 μs	0 μs

Table 5.5: Split Message

These results are almost identical to the previous one, as the test has an almost identical function and uses just a few more instructions.

Code and RAM Sizes The following tables show the code and RAM sizes for the three solutions, whereby the code necessary for the FSM is only for the configuration function.

	Code sizes			RAM sizes		
	SW	SFR	FSM	SW	SFR	FSM
Init	1316	2732	516	12	12	12
Event loop	836	2304	0	50	50	0
Instructions	1828	548	0	79	9	0
Sum	3980	5584	516	141	71	12

Table 5.6: Code and RAM sizes (bytes)

High code sizes of the SFR solution are a result of different optimization results. In addition to the RAM size sums, the configuration RAMs and registers must be added. With 1024 possible instructions and 32 message buffers, additional 5632 bytes must be calculated.

5.2.5 Results

The last section describes the implementation of the CAN-CAN gateway solution providing the gateway functionality partially (SFR) or completely (FSM) in hardware. Efficiency and the routing capabilities of both solutions were compared to a gateway completely running in software.

It was shown that a partial hardware support can reduce the demand on CPU performance, because of the minimized read/write accesses to the peripheral modules. Latency and jitter are reduced, but the check for received messages by interrupt handling or polling wastes computing capacity.

Compared to the conventional software solution the FSM based solution has an almost constant latency and almost no jitter. It is usually more than seventy times faster, when taking the different clock frequencies in account without causing any CPU load. Despite this fact, the FSM solution is not very complex and is still interface compatible to the standard C_CAN.

An enhancement of the (TT)CAN gateway to support further automotive communication architectures, such as FlexRay or MOST, is only possible with huge difficulties. The corresponding protocol converters must be integrated in the cascade ring bus, which was originally implemented to support CAN communication.

5.3 Multi-Protocol Gateway Implementation

The first step in the implementation of the multi-protocol gateway described in section 4.1.3 was the definition of the RAM layouts and an instruction set. After the implementation, the final design is a result of repetitive optimizations of the finite state machines using gate count and RAM usage values from the synthesis tools.

5.3.1 RAM Configuration

As described before the configuration is stored in two RAMs, the Instruction RAM (IRAM) and the Vector RAM (VRAM), whereas the VRAM contains three partitions for the Communication Controllers (VRAM-CC), the Message Objects (VRAM-MO) and one to store shared data (VRAM-Data). The figure 5.7 shows the IRAM and the three partitions of the VRAM.

Buffer Configuration In the Vector RAM Message Object Partition, additional timing information to each message buffer used by the gateway is stored.

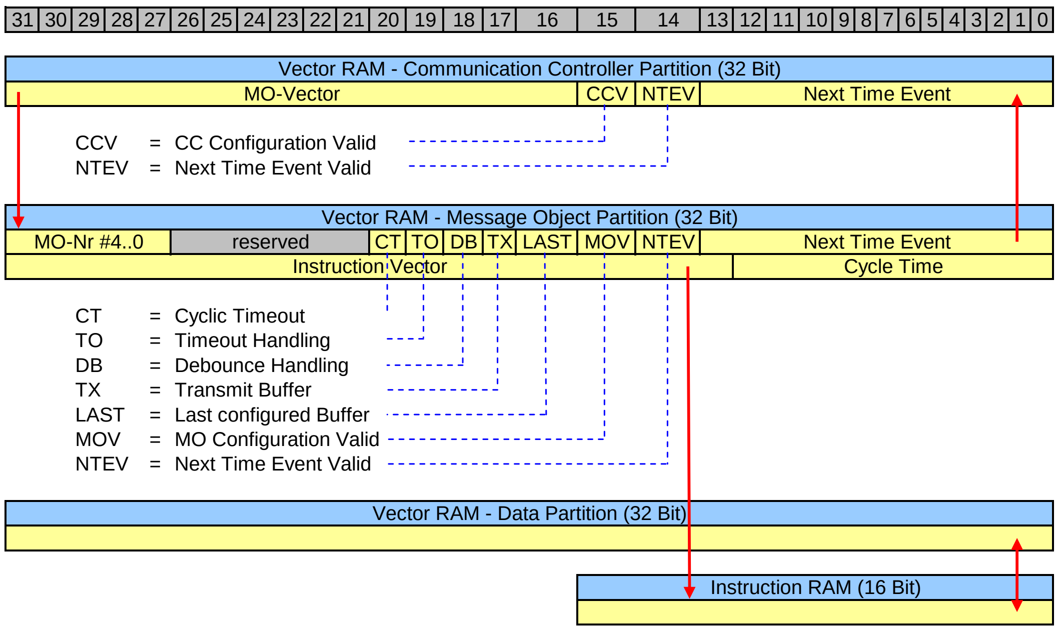


Figure 5.7: Gateway Vector and Instruction RAM

Generally a buffer can be configured as transmit or receive buffer depending on the TX bit. As the only timing option for transmit buffers is cyclic sending, the TX automatically configured the buffer for cyclic transmission. A receive buffer can additionally be configured to react to single timeouts, periodic timeouts or to debounce incoming frames. Each of these entries can be temporarily deactivated by resetting the MOV bit. The possible buffer configurations are shown in the following table 5.7.

Buffer Configuration	MOV	TX	TO	CT	DB	NTEV
Deactivated	0	-	-	-	-	-
Rx Immediate	1	0	0	0	0	0
Tx Periodic	1	1	0	0	0	1
Rx Timeout (single)	1	0	1	0	0	1/0
Rx Timeout (periodic)	1	0	1	1	0	1/0
Rx Debouncing	1	0	0	0	1	0

Table 5.7: Buffer Configuration

5.3.2 Instruction Set

It contains four groups of instructions. The first group contains all instructions for long jumps and short branches, as well as instructions to signal software interrupts to the CPU and to set internal markers for jump and branch decisions.

Two groups are related to load and store operations from and to internal memory and external communication controller addresses. Beside the normal arithmetic instructions, two groups were introduced especially for gateway operations. The block transfer group allows unaligned transfers and comparisons of one or more 32-Bit values directly on the external memory. Bit Field Operations are for unaligned copies of bit fields with one or up to 32-Bit. Also one instruction in this group is used for changing 32-Bit values between different endianess storage formats. The instruction set and codes are shown in figure 5.8 and in figure 5.9.

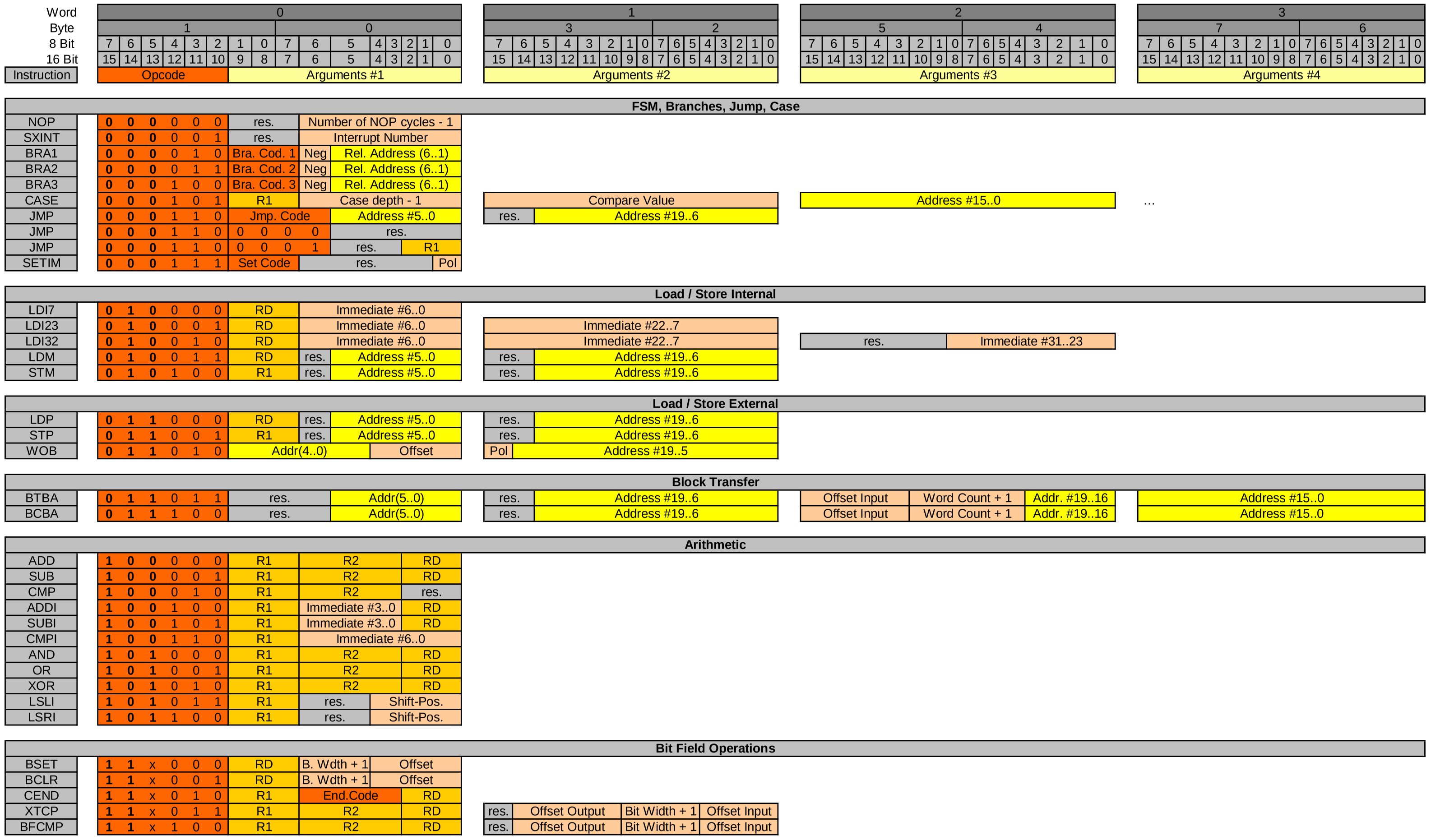
5.3.3 Finite State Machines

Based on the defined RAM layout and functional description, multiple parallel working FSMs have been implemented as shown in figure 5.10.

Main Message Handler FSM This FSM controls the startup and shutdown of the Gateway Control Unit. After powering up, the RAMs are automatically cleared. When finished, the FSM does switch from the "wait on clear ready" state to the "wait on init" state. While the initialization bit is set, the RAMs remain configurable. After clearing the init bit the Gateway Control Unit starts operation and the CPU is limited to only access the VRAM-Data partition in the "FSM is active" state.

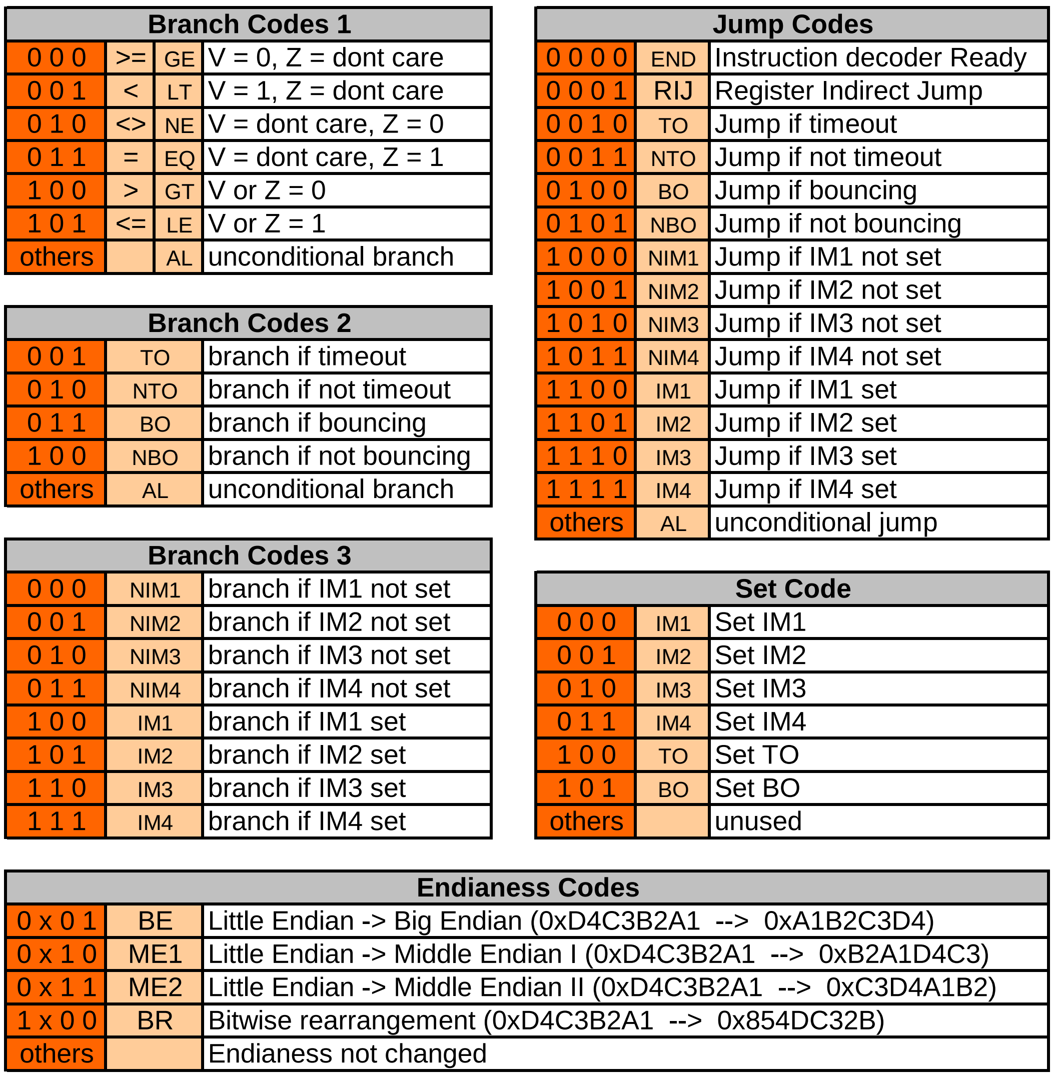
Vector RAM - Communication Controller Partition - FSM The FSM for the VRAM-CC starts handling receive events or time events, as well as CPU trigger executions.

Until a new message or time event occurs, the FSM remains in the "idle" state. When an event happens, it executes a new read cycle through the VRAM-CC partition. After reading the location of the triggering communication controller in the "scan VRAM CC" state and a new message or time event is detected, the wrapper register is read out to find the correct message object in the "read wrapper" state, otherwise it goes back to the "idle" state. With that information, the processing can be passed over to the VRAM-MO FSM to process the message object. Until this is done, the VRAM-CC FSM remains in the "wait on VRAM MO" state. When the VRAM-MO FSM has finished, the time information is written back to the VRAM-CC partition in the "write new values" state. When there are no further groups of message buffers as checked in the "check next time event" state, the FSM writes back the next time event and becomes "idle" again, otherwise scanning is continued in the "scan VRAM CC" state.



Word	0	0	1	2	3
Byte	1	1 / 0	3 / 2	5 / 4	7 / 6
8 Bit	7 6 5 4 3 2	1 0 / 7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0 / 7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0 / 7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0 / 7 6 5 4 3 2 1 0
16 Bit	15 14 13 12 11 10	9 8 7 6 5 4 3 2 1 0	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Instruction	Opcode	Arguments #1	Arguments #2	Arguments #3	Arguments #4
FSM, Branches, Jump, Case					
NOP	0 0 0 0 0 0	res. | Number of NOP cycles - 1			
SXINT	0 0 0 0 0 1	res. | Interrupt Number			
BRA1	0 0 0 0 1 0	Bra. Cod. 1 | Neg | Rel. Address (6..1)			
BRA2	0 0 0 0 1 1	Bra. Cod. 2 | Neg | Rel. Address (6..1)			
BRA3	0 0 0 1 0 0	Bra. Cod. 3 | Neg | Rel. Address (6..1)			
CASE	0 0 0 1 0 1	R1 | Case depth - 1	Compare Value	Address #15..0	...
JMP	0 0 0 1 1 0	Jmp. Code | Address #5..0	res. | Address #19..6		
JMP	0 0 0 1 1 0	0 0 0 0 | res.			
JMP	0 0 0 1 1 0	0 0 0 1 | res. | R1			
SETIM	0 0 0 1 1 1	Set Code | res. | Pol			
Load / Store Internal					
LDI7	0 1 0 0 0 0	RD | Immediate #6..0			
LDI23	0 1 0 0 0 1	RD | Immediate #6..0	Immediate #22..7		
LDI32	0 1 0 0 1 0	RD | Immediate #6..0	Immediate #22..7	res. | Immediate #31..23	
LDM	0 1 0 0 1 1	RD | res. | Address #5..0	res. | Address #19..6		
STM	0 1 0 1 0 0	R1 | res. | Address #5..0	res. | Address #19..6		
Load / Store External					
LDP	0 1 1 0 0 0	RD | res. | Address #5..0	res. | Address #19..6		
STP	0 1 1 0 0 1	R1 | res. | Address #5..0	res. | Address #19..6		
WOB	0 1 1 0 1 0	Addr(4..0) | Offset	Pol | Address #19..5		
Block Transfer					
BTBA	0 1 1 0 1 1	res. | Addr(5..0)	res. | Address #19..6	Offset Input | Word Count + 1 | Addr. #19..16	Address #15..0
BCBA	0 1 1 1 0 0	res. | Addr(5..0)	res. | Address #19..6	Offset Input | Word Count + 1 | Addr. #19..16	Address #15..0
Arithmetic					
ADD	1 0 0 0 0 0	R1 | R2 | RD			
SUB	1 0 0 0 0 1	R1 | R2 | RD			
CMP	1 0 0 0 1 0	R1 | R2 | res.			
ADDI	1 0 0 1 0 0	R1 | Immediate #3..0 | RD			
SUBI	1 0 0 1 0 1	R1 | Immediate #3..0 | RD			
CMPI	1 0 0 1 1 0	R1 | Immediate #6..0			
AND	1 0 1 0 0 0	R1 | R2 | RD			
OR	1 0 1 0 0 1	R1 | R2 | RD			
XOR	1 0 1 0 1 0	R1 | R2 | RD			
LSLI	1 0 1 0 1 1	R1 | res. | Shift-Pos.			
LSRI	1 0 1 1 0 0	R1 | res. | Shift-Pos.			
Bit Field Operations					
BSET	1 1 x 0 0 0	RD | B. Wdth + 1 | Offset			
BCLR	1 1 x 0 0 1	RD | B. Wdth + 1 | Offset			
CEND	1 1 x 0 1 0	R1 | End.Code | RD			
XTCP	1 1 x 0 1 1	R1 | R2 | RD	res. | Offset Output | Bit Width + 1 | Offset Input		
BFCMP	1 1 x 1 0 0	R1 | R2 | RD	res. | Offset Output | Bit Width + 1 | Offset Input		

Figure 5.8: Gateway Instruction Set



Branch Codes 1			
0 0 0	>=	GE	V = 0, Z = dont care
0 0 1	<	LT	V = 1, Z = dont care
0 1 0	<>	NE	V = dont care, Z = 0
0 1 1	=	EQ	V = dont care, Z = 1
1 0 0	>	GT	V or Z = 0
1 0 1	<=	LE	V or Z = 1
others		AL	unconditional branch

Branch Codes 2		
0 0 1	TO	branch if timeout
0 1 0	NTO	branch if not timeout
0 1 1	BO	branch if bouncing
1 0 0	NBO	branch if not bouncing
others	AL	unconditional branch

Branch Codes 3		
0 0 0	NIM1	branch if IM1 not set
0 0 1	NIM2	branch if IM2 not set
0 1 0	NIM3	branch if IM3 not set
0 1 1	NIM4	branch if IM4 not set
1 0 0	IM1	branch if IM1 set
1 0 1	IM2	branch if IM2 set
1 1 0	IM3	branch if IM3 set
1 1 1	IM4	branch if IM4 set

Jump Codes		
0 0 0 0	END	Instruction decoder Ready
0 0 0 1	RIJ	Register Indirect Jump
0 0 1 0	TO	Jump if timeout
0 0 1 1	NTO	Jump if not timeout
0 1 0 0	BO	Jump if bouncing
0 1 0 1	NBO	Jump if not bouncing
1 0 0 0	NIM1	Jump if IM1 not set
1 0 0 1	NIM2	Jump if IM2 not set
1 0 1 0	NIM3	Jump if IM3 not set
1 0 1 1	NIM4	Jump if IM4 not set
1 1 0 0	IM1	Jump if IM1 set
1 1 0 1	IM2	Jump if IM2 set
1 1 1 0	IM3	Jump if IM3 set
1 1 1 1	IM4	Jump if IM4 set
others	AL	unconditional jump

Set Code		
0 0 0	IM1	Set IM1
0 0 1	IM2	Set IM2
0 1 0	IM3	Set IM3
0 1 1	IM4	Set IM4
1 0 0	TO	Set TO
1 0 1	BO	Set BO
others		unused

Endianess Codes		
0 x 0 1	BE	Little Endian -> Big Endian (0xD4C3B2A1 --> 0xA1B2C3D4)
0 x 1 0	ME1	Little Endian -> Middle Endian I (0xD4C3B2A1 --> 0xB2A1D4C3)
0 x 1 1	ME2	Little Endian -> Middle Endian II (0xD4C3B2A1 --> 0xC3D4A1B2)
1 x 0 0	BR	Bitwise rearrangement (0xD4C3B2A1 --> 0x854DC32B)
others		Endianess not changed

Figure 5.9: Gateway Instruction Set Codes

For the interaction with the software, the CPU is able to execute functions in the IRAM. Therefore the VRAM-CC FSM can directly start the IRAM FSM and wait until the execution is finished in the "wait on IFSM" state.

Vector RAM - Message Object Partition - FSM Finally the VRAM-MO FSM works similar to the VRAM-CC FSM, except it works on the VRAM-MO partition.

Starting in the "idle" state, the FSM is triggered from the VRAM-CC FSM to read a group of message objects. This happens in the "scan VRAM MO" state. When a relevant message object is found, additional information is read in the "read second entry" state. Before the execution of the event handler using

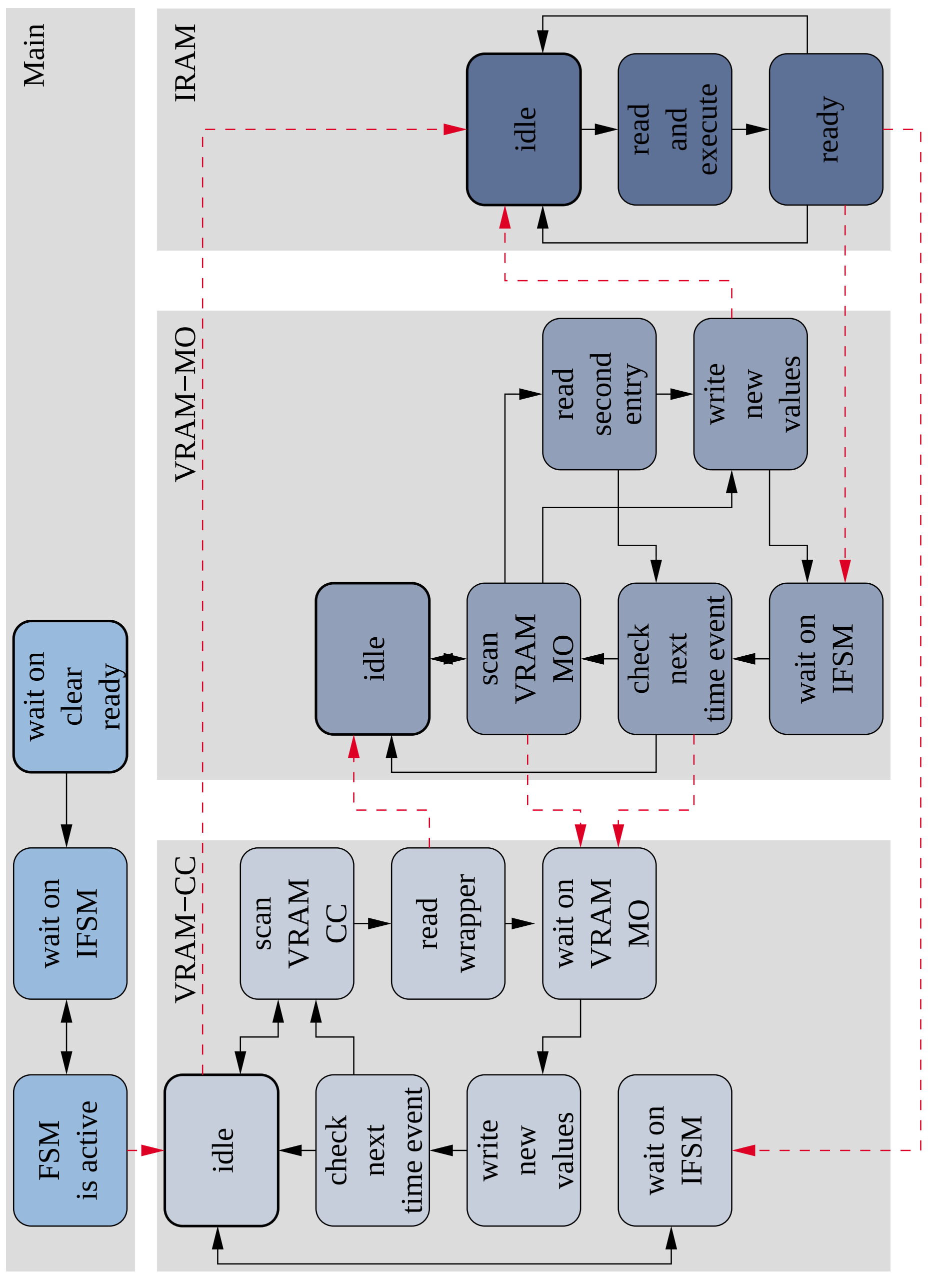


Figure 5.10: Finite State Machines

the IRAM FSM, the next time event for the message object is updated in the "write new values" state. The FSM remains in the state "wait on IFSM" until the IRAM FSM has finished execution. When no further message buffers are in the group as checked in the "check next time event" state, the FSM writes back the next time event and becomes "idle" again.

Instruction RAM - FSM Where the VRAM FSMs handle event detections, the IRAM FSM handles events by executing functions contained in the IRAM.

The IRAM FSM remains "idle" until it is triggered by one of the previous FSMs. Then execution is started in the "read and execute" state, where it remains until a special instruction signals the end of the function execution. Afterwards it switches over to the "ready" state and optionally waits there for on a signal of the VRAM FSMs, before it becomes "idle" again.

Vector RAM Arbiter The Vector RAM is accessible by the CPU, the VRAM FSMs and the IRAM FSM. Therefore an arbitration is necessary for the access to the Vector RAM. This task is done by the VRAM Arbiter as shown in figure 5.11.

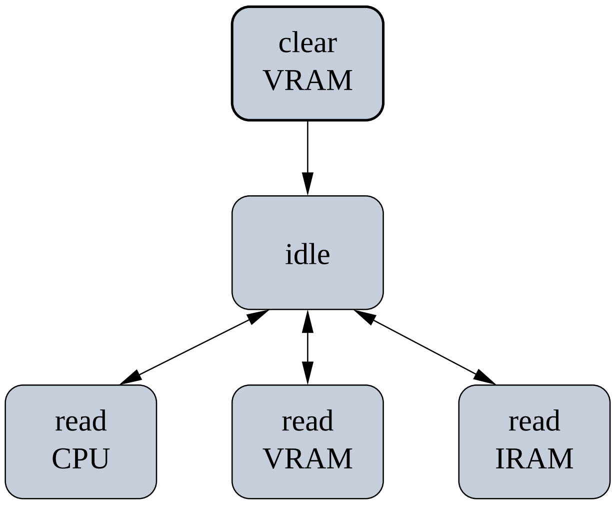


Figure 5.11: VRAM Arbiter

As the VRAM Arbiter has direct access to the VRAM, the RAM is cleared by the FSM in the "clear VRAM" state. It then remains in the "idle" state.

Depending on the origin of an IRAM access, the Arbiter switches to the states "read CPU", "read VRAM FSM", "read IRAM FSM" until the access is finished. Prioritization is implemented to fasten up the data transfers from and to the CPU, then from and to the VRAM FSMs and finally from the IRAM FSM.

5.3.4 Gate Count

Each module of the gateway has been synthesized separately for an ASIC and for a FPGA. The FPGA synthesis was done for the Altera Stratix II EP2S60 using Quartus II Version 5.1. For the ASIC synthesis Synopsys Design Compiler v2003.03 was used for a 0.13 μm fabrication process. 50 MHz was used as target frequency for all synthesis.

Regarding the communication controllers multiple synthesis with different message buffer count and before and after the interface modification for the gateway were done.

The table 5.8 shows the results, whereas the exact values for the gate counts may vary with each synthesis and with the used optimization settings.

		ASIC (kGates)		FPGA (ALUTs)	
Module	Configuration	pre GW	post GW	pre GW	post GW
C_CAN	32 MOs	15	16	2100	2300
C_CAN	64 MOs	19	20	2660	2860
CAN Wr.	3x32xCAN-MOs		2		300
CAN Wr.	4x64xCAN-MOs		6		860
E-Ray	128 MOs	105	115	14700	16200
E-Ray Wr.	1x128xFR-MOs		2		270
GCU			16		2450

Table 5.8: Gate count of CCs, Wrapper and Gateway Control Unit

The table shows, that a gateway with three CAN controllers and 32 message buffers each and one E-Ray controller with 128 message buffers has a gate count of 183000 ASIC gates or 26120 FPGA ALUTs. The same gateway with four CAN controllers and 64 message buffers each would have a gate count of 219000 ASIC gates or 31220 FPGA ALUTs.

5.4 Toolchain Optimization Results

This section shows the results of the toolchain under different optimization levels and configurations.

5.4.1 Optimization Levels

As mentioned in section 4.2.7 the assembler instruction simulator can be used to test each transaction for a specified timing- and trigger-condition for each function in the Instruction RAM. The generated statistics about execution time and read memory can be used to compare different optimization levels. Currently three optimization levels are implemented: Runtime, balanced and memory optimized. The results are shown in figure 5.12.

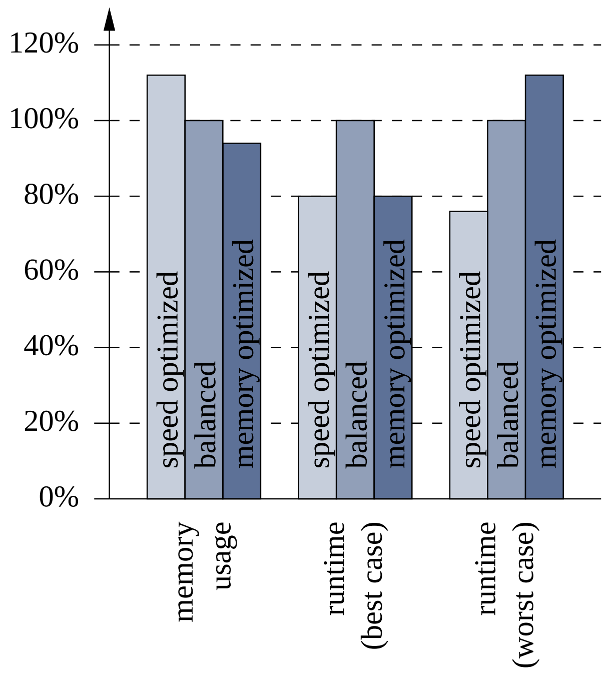


Figure 5.12: Runtime and Memory Optimization

5.4.2 Code Size and Execution Time

A configuration was implemented to demonstrate the available trigger- and timing-conditions. The Gateway Control Unit was synthesized at 50 MHz on an Altera Stratix II EP2S60 FPGA. One FlexRay and three CAN controllers are part of the gateway as described in section 5.1.3.

Instruction Count and Code Size Figure 5.13 shows the number of instructions and the corresponding code sizes for each mapping. The results show an average number of 15 instructions and an average code size of 50 bytes.

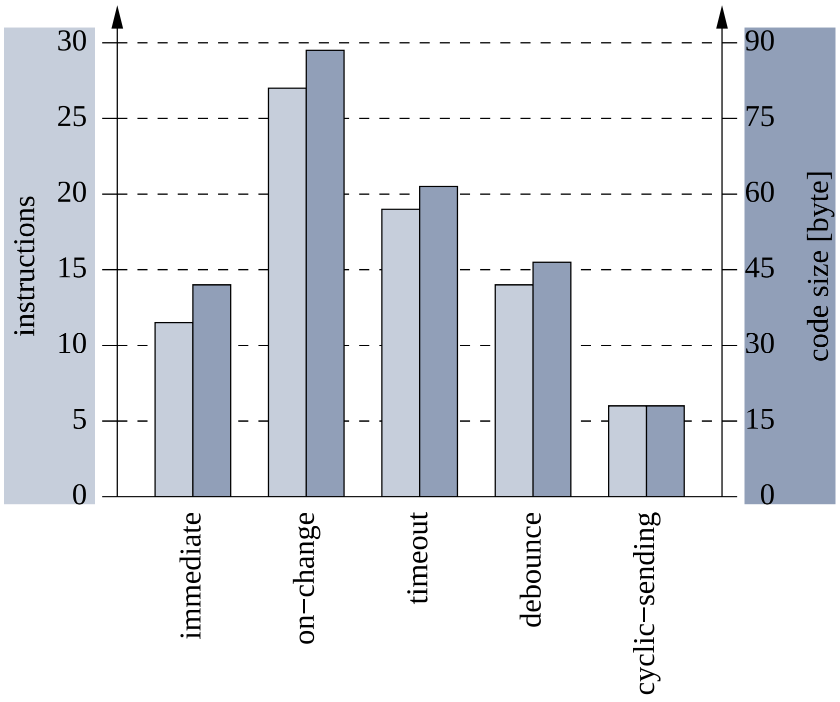


Figure 5.13: Instruction Count and Code Size

Execution Times The best and worst case execution times of the Gateway Unit are shown in figure 5.14. Additionally the FSM generates an overhead of 0.4 μs for the preparation and post processing of the instruction execution.

5.5 Verification Environment Implementation

A test generator has been implemented as an enhancement to the existing gateway configuration toolchain. This allows the usage of FIBEX files as a common source of information for both, the configuration of the gateway and the test environment.

Validity of Data One aspect of these tests is the validity of data. This includes that source and target frames and the signals therein (optionally transformed by the gateway mappings) contain correct data. These checks are required independent of the network and timing specifications.

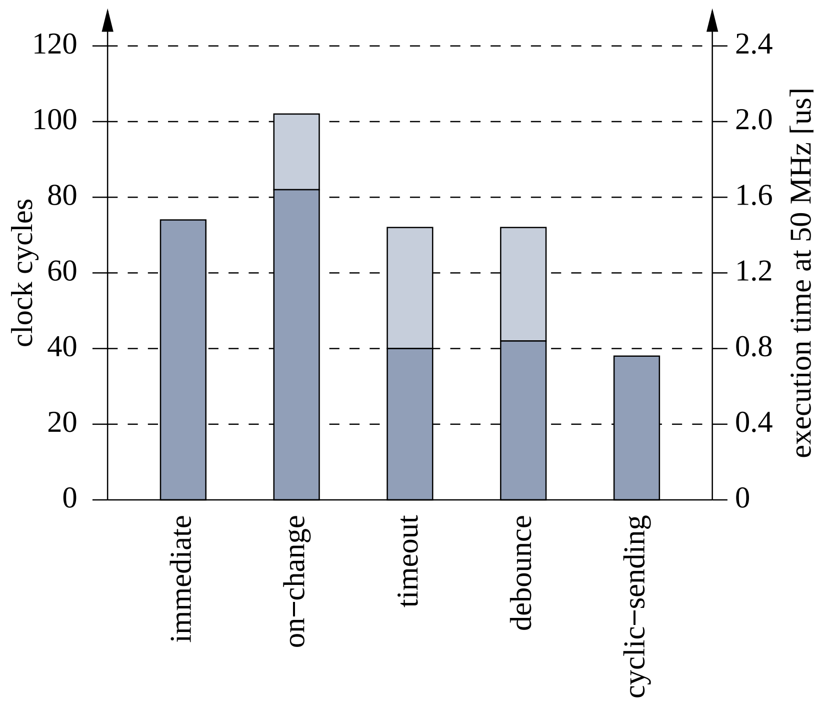


Figure 5.14: Execution time (best and worst case)

Timing Behavior Another aspect is to measure and test whether these mappings are done inside the specified FIBEX time constraints, e.g. timeouts, that are defined as attributes of frame or signal mappings. Maximum latency for the immediate reaction is defined in the specification of the gateway. They are provided using a manufacturer extension to the FIBEX gateway object. Cyclic sending or message debounce times can be defined for target frames as attributes of FIBEX frame triggerings. Both aspects are checked within each test case.

Complexity Regarding the time behavior, testing even a single frame mapping can be difficult, if all the previously named time attributes are defined. In the worst case a target frame received by the test environment cannot clearly be assigned to a cyclic sending, timeout or regular mapping. An assignment is only possible, when the time windows are known in which certain trigger conditions and therefore target frames are expected.

Main Focus Measuring and checking the time behavior is necessary especially in event-triggered networks and is in the focus of the following test methods. Time-triggered networks have static frame schedules instead of cyclic sending and do not use message debounce configurations. Besides measuring the frame schedule, only the mappings and timeout handling have to be checked.

5.5.1 Test Description

A test run is split into two phases. The first phase synchronizes the test environment to the gateway internal timers and allows simple time measurements and checks. This is called the initial test phase. After the timers of the test environment are synchronized to the gateway the main test phase starts.

Cyclic Sending In the initial test phase four tests are processed on every relevant mapping. The first test stage measures the time windows in which cyclic sending frames are expected (figure 5.15). As preparation the test stage waits until the possibly configured timeout conditions of all mappings are met. The only frames transmitted by the gateway are now the cyclic sending frames. They can be measured and checked against limitations. The test environment starts for each of these frames two timers with the same period. Their phase difference defines time windows in which to expect cyclic sending frames. The frame reception times (with tolerances) are specified in the cyclic-timing element of the FIBEX file. Calibration of the time windows and identification of all frames is a necessary preparation for the next test stages.

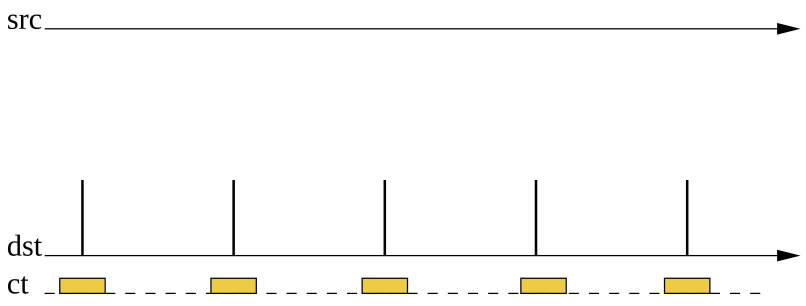


Figure 5.15: Cyclic Sending: Cycle Time

Latency The second test stage checks the latency of event controlled frame forwarding. A repeated source frame with interval duration between debounce time and timeout time (figure 5.16) stimulates the gateway. The trigger condition attribute of the configured mapping changes the behavior of frame receptions. When the trigger condition is "immediate", the target frames are sent out immediately. If the trigger condition is "on-change", then the target frames are sent only when source signals or frames have changed, additionally to cyclic sending frames. When the trigger condition is "none", then the target frames are prepared for cyclic transmission. The test stage checks the data and measures the latency between source frame and target frames. The measured latencies are used to configure timers on each mapping, so that in case of an event controlled frame forwarding, the forwarded frames can be identified in preparation for the next test stages.

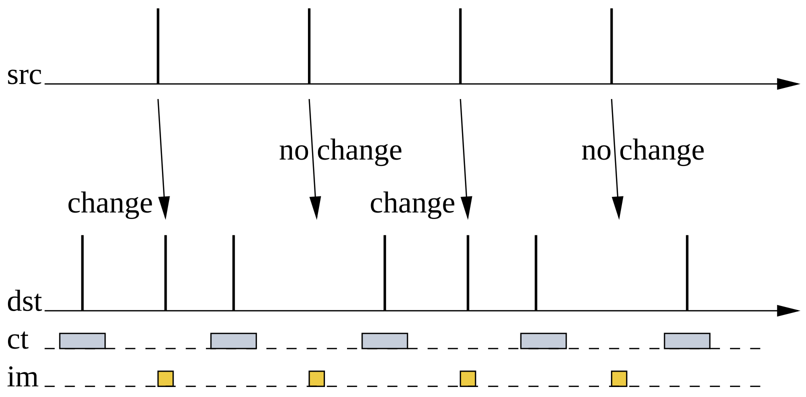


Figure 5.16: Event Controlled Sending: Forwarding Latency

Timeout Timeout measurement is the third test stage (figure 5.17). This is done by stimulating the gateway with periodic source frames with interval duration longer than the maximum timeout time. The gateway reacts to the received source frames as specified by the mappings and their trigger condition. The long interval between the source frames causes the execution of the mapping's timeout reaction. The reaction is to send a target frame with the newest signals or with default values. In either case the delay between the source frame and the target frame is measured and compared with the specification and tolerances defined in the FIBEX file. Two timers for each mapping are adjusted for the expected timeout frame window, similar as done in the previous test stages.

Debouncing The final test stage of the initial test phase is a check of the debounce time (figure 5.18), which is defined in FIBEX as attribute of the target frame triggering. The test environment provides a stimulus with alternating interval times. One source frame is sent and the gateway reacts with an immediate processing of the relevant mappings. The next source frame follows earlier than the debounce time and the gateway must not react immediately to the second frame. As there is no frame sent, the debounce time cannot be measured directly. It is only possible to check that no target frames are sent out in the specified debounce time window. A timer is used to define a debounce time window for each target frame. A specific target frame may not be transmitted inside its debounce window with the possible exception that cyclic sending may still trigger the transmission of this frame.

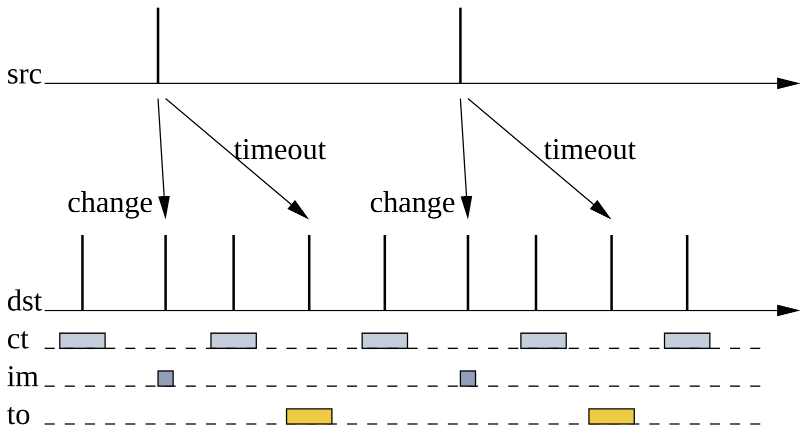


Figure 5.17: Timeout

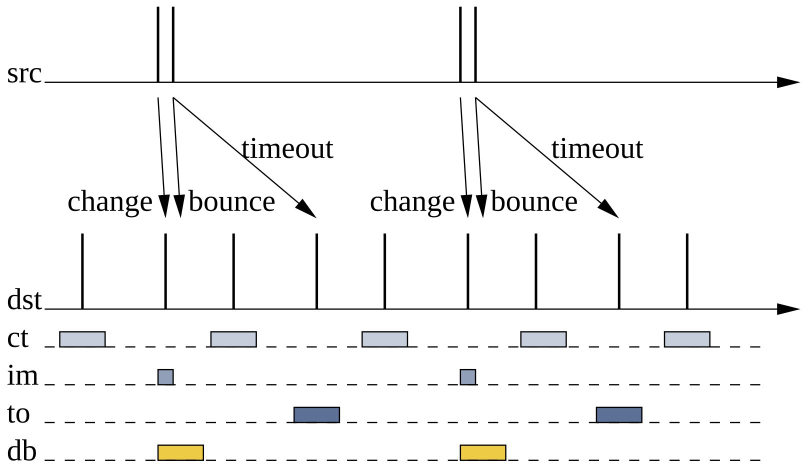


Figure 5.18: Debouncing

Post-Processing After running these test stages for each mapping separately, the time windows in which the gateway's reaction can be expected are known. This calibration allows concurrent observation and checking in parallel to further testing in the main test phase.

5.5.2 Test Scenarios

In the initial test phase all tests were run with minimal bus load. In the main test phase test cases provide stimuli to check the gateway's behavior in several bus load situations with different interactions between the configured mappings. This can be done by connecting actual ECUs or by connecting emulated network nodes as part of a remaining bus simulation.

Complex Scenario One test scenario uses configuration data from an actual implementation. For this scenario the user has to provide load information to be used as stimuli. This information should be integrated into the FIBEX files, as the common database. This can either be done by setting the cyclic sending attributes of frames sent to the gateway or by a manufacturer extension to the FIBEX standard, which defines the frame rates without using the cyclic sending attribute.

Bus Utilization The second test scenario increases the bus load up to the desired bus utilization. This is done by increasing specific source frame rates, which can grow up to worst case scenarios or burst situations. The results from this test show how latency and jitter is related to the bus utilization. This can also show the limits within the gateway keeps the specified tolerances.

Further Scenarios Other scenarios are easy to implement by introducing new stimuli or load environments. The measurement and check environment always stays the same.

5.5.3 Implementation Requirements

A programmable hardware environment is required to implement the described test methods. This environment should allow time stamping of all frames received or transmitted by the gateway using a common time base. The test software is partitioned in a way that most of the code is kept static and only a small part depends on the gateway's configuration. This part is generated automatically by the toolchain. Two possible hardware platforms have been evaluated.

Test System One environment is a high performance processor with enough communication controllers to connect to all required networks. It needs enough memory to store log files or needs an interface to transfer the log files to external storage, e.g. via Ethernet. The platform should allow the software direct access to the communication controllers. Advantageous would be timestamps generated in hardware.

Remaining Bus Simulation An easier way is using a tool for remaining bus simulations. Using network interfaces such an environment can be connected to the physical networks and allows the programming of simulated ECUs, usually in a high-level scripting language. Currently only few such tools are available which support CAN, FlexRay and further network types and that also have a synchronized timer for time stamping frames on these networks.

Vector CANoe The implemented toolchain currently generates test case output for using the remaining bus simulation features of Vector CANoe. At this time network support is implemented for CAN and FlexRay only.

This tool allows programming of ECUs in a C-like language called CAPL, which stands for Communication Access Programming Language. CANoe also has a sophisticated test management system, which allows the description of test cases without the need for simulated ECUs. This is done in CAPL using an extended function set based on the test service library.

Additionally XML test specification files can be used to provide a simpler and more restricted description of test cases. A Document Type Definition (DTD) for the test specification file is provided for this.

The CANdb output filter in the toolchain can be used to generate CANdb databases for the required description of CAN networks in CANoe.

CAPL Generation Several CAPL templates are combined to generate complete test cases. There are templates for source and target frame handling and for the mapping functions between them. Currently the generated output has the initial test phase implemented and provides different stimuli, while monitoring the gateway's behavior in parallel. After a specified time the monitored data is evaluated and a direct success or failure for every test case is generated along with the measured timings for each mapping.

5.5.4 Results

The described initial test phase has been implemented in a test case generator based on the already existing FIBEX gateway configuration toolchain as shown in figure 5.19. As the focus was to measure event-triggered networks, the generator currently supports CAN networks only.

Proof of Concept As a proof of concept, the multi-protocol gateway connected to two CAN channels has been used as device under test (DUT) to demonstrate the gateway verification environment. The first channel provides input to the gateway. One mapping was configured to send on change, debounce frames arriving earlier than 100 ms, timeout at 2 s, and a cyclic sending with an interval of 1 s. The second channel was used as target for this mapping.

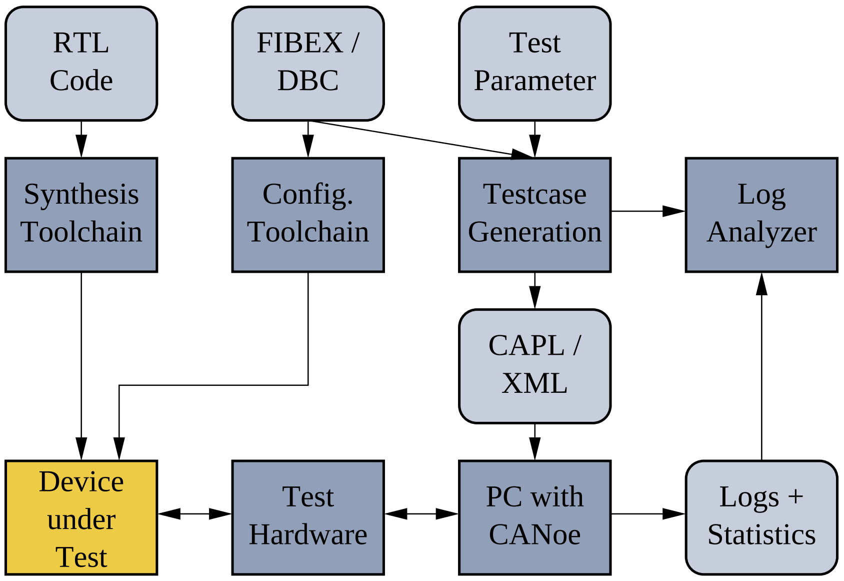


Figure 5.19: CANoe Test Environment

Execution Time The complete test has been executed on the DUT. 10 measurements per test and per mapping were done in the initial test phase to calibrate the test environment. This takes 10 s for the cyclic sending test. The immediate test with alternating changed and unchanged data takes 20 s, the timeout test takes 40 s, and the debounce test takes 2.5 s. A cumulated time of 72.5 s is required to process the initial test phase.

	Minimum time	Average time	Maximum time
Cycle time	1000.0058 ms	1000.0076 ms	1000.0078 ms
Immediate	0.1287 ms	0.1287 ms	0.1290 ms
Timeout	1999.9024 ms	2000.0115 ms	2000.1262 ms

Table 5.9: Gateway measurement results

All timings are within the tolerances specified in the FIBEX configuration. The initial tests show no significant latency and jitter created by the device under test with the provided timer resolutions as shown in table 5.9. Most of the immediate transmission time of 0.1287 ms is introduced by the transmission time of the CAN frame, which itself is already longer than 125 μs.

Main Test Scenario A simple load test was created manually. It sends arbitrary frames with randomized payload in a specified interval. The current DUT shows no impact on latency and jitter times introduced by high bus loads. Using more complex configurations or different gateways will probably lead to other results.

5.5.5 Future Improvements

Extrapolating from the time of the complete example test run, testing a gateway with several hundred frames and several thousand mappings may take a very long time. Further optimizations, e.g. more parallelization and less initial measurements, are necessary to reduce the execution time.

Error Recovery The initial test phase calibrates the test environment to the gateway, whereas the main test phase takes measurements under different load scenarios. Currently errors may falsify all following test cases till the end of the complete test run. This could be prevented when the test environment interrupts the test run and brings the gateway and test environment back to a known state if an error is detected.

More Scenarios In the main test phase several load scenarios are missing. The target should be to create and execute further load tests. Special focus lies in scenarios that reflect realistic environments.

Comparison of Different Gateways As FIBEX is a relatively new standard, most gateway configurations are not yet available in this format, but in specification documents of other form. These specifications must be converted to FIBEX format to be able to execute authentic test scenarios. These gateway architectures and implementations can then be tested and compared with each other, especially regarding latency and jitter times.

Further Networks Support for further networks needs to be implemented. Adding FlexRay connectivity will be the next step. LIN and MOST networks could follow, as these are used by many gateway implementations.

5.6 Gateway Evaluation in OEM Environments

Different OEM environments have been evaluated for the use of the multi-protocol gateway. The main focus beside the general network utilization was the RAM usage and GCU utilization of the gateway. As most OEMs provide the specification of their networks in different formats and not necessarily in FIBEX or DBC format, the results were calculated based on the average code sizes and execution times of similar configurations in the testbench.

5.6.1 Vector RAM Usage Estimation

The RAM usage estimation is based on several estimated parameters. One limitation is that the current gateway implementation only allows the controllers of one type to have the same number of message buffers. Using these parameters, the following formula allows to calculate the Vector RAM usage per controller type. For the complete gateway, these results must be summarized across all controller types (multiple CAN controllers or FlexRay controllers).

In the following variables the following naming scheme is used. Variables with $n_{...}$ stand for a number or count. A rate is marked with $r_{...}$. And a percentage or part is named as $p_{...}$.

The correct number of message buffers (n_{MB}) must be greater than the number of all receive and transmit messages (n_{RX} and n_{TX}) divided by the number of controllers (n_{CC}) under a balanced distribution of message buffers:

$$n_{MB} \geq \frac{n_{RX} + n_{TX}}{n_{CC}}$$

As one 32-Bit entry in the VRAM-CC partition per 32 message buffers is necessary, the RAM usage for this partition can be calculated as:

$$V_{CC} = 4\ Byte \cdot \frac{n_{CC} \cdot n_{MB}}{32}$$

Each receive and each cyclic transmit message buffer used by the gateway requires two 32-Bit entries in the VRAM-MO partition. The complete number of cyclic transmit message buffer can be calculated based on the percentage of cyclic timing messages over all transmit messages (p_{CT}). Then the following formula calculates the VRAM-MO partition usage:

$$V_{MO} = 4\ Byte \cdot 2 \cdot (n_{RX} + n_{TX} \cdot p_{CT})$$

Four 32-Bit entries in the VRAM-Data partition store constants for loading, storing, transmitting and a combination of storing and transmitting of a message buffer. Therefore the VRAM-Data partition usage (V_{Data}) is always constant:

$$V_{DATA} = 4\ Byte \cdot 4$$

To calculate the Vector RAM size in byte, the previous number of 32-Bit entries in the VRAM partitions must be summarized:

$$V = V_{CC} + V_{MO} + V_{DATA}$$

These formulas must be repeated for each controller type to calculate the complete VRAM usage.

5.6.2 Instruction RAM Usage Estimation

The calculation of the 16-Bit wide instruction RAM usage is based on several more estimations about the gateways configuration. As the input and output buffers of the C_CAN and E-Ray communication controllers are very similar, the following formulas applies to both types of controllers.

The average number of instructions per signal copy (I_{SgCp}) is calculated based on three classes of signal copy operations. The first class contains copy operations within 32-Bit boundaries and of less than 32-Bit (n_{cp1}). The second class contains copy operations within 32-Bit boundaries, but with at least 32-Bit signals (n_{cp2}). And the third class contains copy operations across 32-Bit boundaries (n_{cp3}). The average number of instructions per signal copy (I_{SgCp}) can then be calculated as follows:

$$I_{SgCp} = 2\ Byte \cdot (8 \cdot n_{cp1} + 4 \cdot n_{cp2} + 16 \cdot n_{cp3})$$

If an estimated percentage of target frames should only be sent on change ($p_{TxOnChg}$), a second execution path for the comparison must be generated parallel to the signal copy path. This second path is usually four instructions longer:

$$I_{SgCpCmp} = 2 \cdot I_{SgCp} + 2\ Byte \cdot 4 \cdot p_{TxOnChg}$$

Before the signal copy operations can be executed at least the source message buffers must be loaded. The target message buffer must only be loaded when it is partially overwritten. After the copy operations, the target message buffers must be stored or transmitted. To calculate the number of load, store and transmit operations (I_{LdStTx}), which usually require eight instructions each, the average number of transmit messages per receive message must be known ($n_{TxPerRx}$) as well as the percentage of partially overwritten messages thereof ($p_{PartTxLd}$).

$$I_{LdStTx} = 2\ Byte \cdot 8 \cdot n_{TxPerRx} \cdot (1 + p_{PartTxLd})$$

Together with some leading and trailing instructions, the complete number of instructions to forward received frames (I_{Frwd}) can be calculated based on the average number of signals per message ($p_{SgPerMsg}$):

$$I_{Frwd} = n_{RX} \cdot (8 + I_{LdStTx} + p_{SgPerMsg} \cdot I_{SgCpCmp} + 1)$$

The amount of configured timeout handling per receive message (p_{RxTo}) allows the calculation of timeout handling instructions ($I_{RxToHndl}$):

$$I_{RxToHndl} = 2\ Byte \cdot 10 \cdot n_{RX} \cdot p_{RxTo}$$

The number of cyclic sending instructions ($I_{TxCtHndl}$) can be calculated based on the average number of cyclic sending configurations per transmit message (n_{CT}):

$$I_{TxCtHndl} \quad = \quad 2\ Byte \cdot 9 \cdot n_{CT}$$

Now the instruction RAM usage can be calculated as:

$$I \quad = \quad n_{RX} \cdot (I_{Frwd} + I_{RxToHndl}) + n_{TX} \cdot I_{TxCtHndl}$$

5.6.3 GCU Utilization Estimation

Based on the CAN frame format described in section 2.2 the following formula can be derived. It calculates the bit length of a standard CAN frame (l) based on the payload ($l_{payload}$):

$$l = 47 + l_{payload} * 8 + \left(\frac{47 + l_{payload} * 8 - 13 - 5}{4} + 1 \right) /2$$

The calculation of the Gateway Control Unit utilization makes a categorization of the mapping types necessary:

- cyclic sending
- receiving and partial forwarding
- receiving and complete forwarding
- send-on-change with no change
- send-on-change with partial forwarding
- send-on-change with complete forwarding

After calculating the rates of messages for every of these mappings, the GCU utilization can be calculated based on the number of instructions and the frequency the GCU is operating at.

Every mapping containing a cycle time (t_{ct}) must be included in the calculation of cyclic transmitted frames per second (r_{fps_ct}):

$$r_{fps\_ct}(i) = \frac{1000ms}{t_{ct}(i)}$$

Additionally to cyclic sending frames, a specified amount is send on-change (p_{soc}). This allows the calculation of frames per second over both transmit types:

$$r_{fps}(i) = r_{fps\_\_ct}(i) \cdot (1 + p_{soc})$$

Using these formula and the knowledge of the gateway's configuration, the following categories can be summarized:

$$\begin{aligned} r_{gw\_\_tx} &= \textstyle\sum_{\forall i} r_{fps}(i) && \text{if frame is send by the gateway} \\ r_{gw\_\_rx} &= \textstyle\sum_{\forall j} r_{fps}(j) && \text{if frame is received by the gateway} \\ r_{no\_\_gw} &= \textstyle\sum_{\forall k} r_{fps}(k) && \text{if frame is not relevant for the gateway} \end{aligned}$$

The sum over all cyclic transmitted frames per second is:

$$r_{ct\_\_tx} = \sum_{\forall i} r_{fps\_\_ct}(i)$$

Cyclic received frames, which are partially or completely forwarded, can be summarized as follows:

$$\begin{aligned} r_{ct\_\_rx\_\_p}(i) &= \textstyle\sum_{\forall i} r_{fps}(i) && \text{if frame is received cyclic and part. forw.} \\ r_{ct\_\_rx\_\_c}(i) &= \textstyle\sum_{\forall i} r_{fps}(i) && \text{if frame is received cyclic and compl. forw.} \end{aligned}$$

Frames additionally forwarded on change can also be summarized in three categories:

$$\begin{aligned} n_{soc\_\_n} &= \textstyle\sum_{\forall i} \frac{1000ms}{t_{ct}(i)} && \text{on no change} \\ n_{soc\_\_p} &= \textstyle\sum_{\forall i} p_{soc} \cdot \frac{1000ms}{t_{ct}(i)} && \text{on partially forward} \\ n_{soc\_\_c} &= \textstyle\sum_{\forall i} p_{soc} \cdot \frac{1000ms}{t_{ct}(i)} && \text{on completely forward} \end{aligned}$$

Now as the frame rates of each type of mapping are known, the number of instructions for each of these mappings as shown in table 5.10 can be multiplied to get the number of instructions per second. For this calculation the average number of signals per message ($n_{SgPerMsg}$), the average number of transmit messages per receive message ($n_{TxPerRx}$) and the average message length (n_{AvgLen}) must be estimated. The utilization of the Gateway Control Unit (U_{GW}) is the number of instruction per second (n_{ips}) divided by the frequency the Gateway Unit (f_{GW}) is running:

$$U_{GW} = \frac{n_{ips}}{f_{GW}}$$

The frame rates can also be used to calculate the network utilization U_{Net} based on its bitrate (s):

$$U_{Net} = \frac{1}{s} \cdot \sum_{\forall i} s \cdot r_{fps}(i)$$

	send cyclic	receive partial frwd.	receive complete frwd.	send on chng. no change	send on chng. partial fwrd.	send on chng. complete fwrd.
	$r_{ct\_\_tx}$	$r_{ct\_\_rx\_\_p}$	$r_{ct\_\_rx\_\_c}$	$n_{soc\_\_n}$	$n_{soc\_\_p}$	$n_{soc\_\_c}$
pre/post processing	21	21	21	21	21	21
source buffer load	0	12	12	12	12	12
target buffer load	0	$12*n_{TxPerRx}$	0	$12*n_{TxPerRx}$	$12*n_{TxPerRx}$	$12*n_{TxPerRx}$
compare	0	0	0	$19*n_{SgPerMsg}$	$2.5*n_{SgPerMsg}$	$(7.5+3*n_{AvgLen}/8) *n_{TxPerRx}$
copy	0	$14*n_{SgPerMsg}$	$(10+3*n_{AvgLen}/4) *n_{TxPerRx}$	0	$7*n_{SgPerMsg}$	$(5+3*n_{AvgLen}/8) *n_{TxPerRx}$
target buffer store+transmit	12	$12*n_{TxPerRx}$	$12*n_{TxPerRx}$	0	$12*n_{TxPerRx}$	$12*n_{TxPerRx}$

Table 5.10: Instructions per Mapping Category

5.6.4 Routing Latency

Maximum routing latency occurs, when the currently processed mapping is configured to have the maximum number of transmit messages per receive message ($max\,(n_{TxPerRx})$) and the maximum number of signals per message ($max\,(n_{SgPerMsg})$). If the number of instructions ($max\,(n_i)$)to handle this worst case routing is divided by the frequency the Gateway Control Unit (f_{GW}) is enabled, then the maximum routing latency is the result:

$$t_{MaxRoutLat} = \frac{max(n_i)}{f_{GW}}$$

5.6.5 Worst Case Execution Time

Worst case execution time (WCET) is the time to handle a situation where all possible events on all interfaces (n_{CC}) happen at the same time and require the longest time to process them. For the receive events, the maximum routing latency is already given. For the time events, the maximum processing time can be calculated based on the frequency of the gateway control unit (f_{GW}) and the number of instructions required to send a cyclic message ($I_{TxCtHdnl}$). Therefore the worst case execution time (t_{WCET}) can be calculated as follows:

$$t_{WCET} = n_{CC} \cdot \left(t_{MaxRoutLat} + \frac{I_{TxCtHndl}}{f_{GW}} \right)$$

5.6.6 Evaluation of a High-Class Automotive Network

The first OEM configuration was provided for a high-class vehicle. Its network transfers about 3400 periodic messages per second across four CAN channels (Body CAN, Chassis CAN, Diagnostic CAN and Impact CAN) as seen in table 5.11. A Body CAN controls most of the indoor functions, such as seat and door/window control. Tire sensors (pressure and RPM) and further control functions regarding the chassis movements are connected to the Chassis CAN. The Impact CAN handles all safety relevant functions, e.g. airbag, acceleration sensors. In maintenance the Diagnostic CAN gives central access to monitor the other CANs and control most ECUs.

Gateway Tasks As the table 5.11 shows, one of the main tasks of the gateway is to transfer frames between the Body CAN and the Chassis CAN. While in maintenance most messages are also routed to the Diagnose CAN. The following results in table 5.12 and table 5.13 are calculated including these diagnostic messages.

	Body	Chassis	Diagnostics	Impact	sum	unit
baud rate	125	500	500	500	1625	kBit/s
GW Tx cyclic	175	156	899	606	1836	frm/s
GW Rx cyclic	370	1068	5	110	1553	frm/s
GW Tx sporadic	59	37	107	8	211	frm/s
GW Rx sporadic	80	58	98	9	245	frm/s

Table 5.11: Frame rates of a High-Class Car

RAM Usage The gateway has to handle 1552 messages per second and transmits 1836 periodic messages per second in addition to sporadic messages. As an average of 85 message buffers per CAN controller are required, all four CAN controllers $n_{CC} = 4$ have $n_{MB} = 128$ message buffers. $p_{RxTo} = 10\%$ of the $n_{RX} = 168$ receive messages are configured to have timeouts. $p_{CT} = 87\%$ of the $n_{TX} = 172$ transmit messages are configured for cyclic sending in addition to sporadic transmissions. The copy operations have a 40% : 40% : 20% distribution across the four copy classes $(n_{cp1}, n_{cp2}, n_{cp3})$. Only a few of these copy operations must be duplicated for send on change $p_{TxOnChg} = 23\%$. The average number of signals per frame is $n_{SgPerMsg} = 4$ and about two new messages are generated on receive $n_{TxPerRx} = 2$, whereof $p_{PartTxLd} = 40\%$ are only partially overwritten.

GCU Utilization As average message length the following assumption of $n_{AvgLen} = 6$ was made.

Vector RAM	2.6 kByte
Instruction RAM	27.2 kByte

Table 5.12: RAM Usage of a High-Class Car

System Clock	20 MHz	32 MHz	50 MHz
Max. Routing Latency	24.8 μs	15.4 μs	10.0 μs
Theoretical WCET	104.0 μs	65.0 μs	41.6 μs
Total GCU Load	1.7%	1.1%	0.7%

Table 5.13: GCU Utilization of a High-Class Car

5.6.7 Evaluation of a Mid-Class Automotive Network

The second OEM configuration is provided for a medium-class vehicle. Its network does not have less messages than the high-class vehicle, mainly because of more intense fragmentation of redundant information across the CAN networks (Comfort CAN, Combi CAN, Powertrain CAN, Info CAN) as seen in table 5.14. The Comfort CAN has a similar function as the Body CAN from the previous OEM configuration, it handles most data for controlling indoor components. An own Combi CAN exchanges information with the combi instrument. Engine management, transmission and chassis functions, such as ESP, are managed over the Powertrain CAN. The Info CAN handles the infotainment and multimedia applications.

	Comfort	Combi	Powertrain	Info	sum	unit
baud rate	125	500	500	125	1250	kBit/s
utilization	49%	27%	50%	19%		
frame rate	717	1259	2230	256	4462	frm/s
	distribution of frame rates					
GW Tx	195	1101	220	209	1725	frm/s
GW Rx	247	158	1210	24	1638	frm/s
other	275	0	800	23	1098	frm/s
	distribution of GW Rx frame rates					
partial transfer	31	60	793	0	884	frm/s
complete transfer	61	50	1035	10	1156	frm/s
ECU relevant	69	25	0	12	106	frm/s

Table 5.14: Frame rates of a Mid-Class Car

Gateway Tasks Most messages are transferred from the Powertrain CAN to the other three networks. Much less messages are transferred in other directions.

RAM Usage The gateway transmits about 1638 periodic messages and needs to handle 1725 receive messages per second. All four CAN controllers $n_{CC} = 4$ are configured to have $n_{MB} = 64$ message buffers to receive $n_{RX} = 78$ messages and transmit $n_{TX} = 122$ messages. $p_{RxTo} = 10\%$ of the receive messages are checked for timeouts and all transmit messages $p_{CT} = 100\%$ are sent on change. An average of $n_{SgPerMsg} = 2$ signals are part of each message and each message usually generates $n_{TxPerRx} = 2$ other messages, whereof $n_{PartTxLd} = 50\%$ is only partially overwritten. The distribution of copy operations is identical to the previous OEM configuration and most of the copy operations are duplicated for send on change $p_{TxOnChg} = 90\%$.

GCU Utilization The average frame length is identical to the previous OEM configuration $n_{AvgLen} = 6$.

Vector RAM	1.6 kByte
Instruction RAM	13.0 kByte

Table 5.15: RAM Usage of a Mid-Class Car

System Clock	20 MHz	32 MHz	50 MHz
Max. Routing Latency	16 μs	10.0 μs	6.5 μs
Theoretical WCET	70.8 μs	44.0 μs	28.3 μs
Total GCU Load	1.4%	0.9%	0.6%

Table 5.16: GCU Utilization of a Mid-Class Car

5.6.8 Conclusion

The evaluation of Mid-Class and High-Class vehicles shows that both have similar communication requirements. As the RAM usages and GCU utilizations show, the multi-protocol gateway can be used in both scenarios.

Chapter 6

Summary and Outlook

Present gateway solutions may lead to high latencies and jitter that are caused by interrupt handling routines, complex gateway functions and the amount of data. The current trend to increase the clock frequency to face these problems results in high power consumption and high electro-magnetic radiation.

Two CAN-CAN gateway solutions providing the gateway functionality partially (SFR) or completely (FSM) in hardware were presented. The efficiency and the routing capabilities of both solutions were compared to a gateway solution completely realized in software.

It was demonstrated that a partial hardware support (SFR) can reduce the demand on CPU performance, because of the minimized read/write accesses to the peripheral modules. Latency and jitter are reduced, but the check for received messages by interrupt handling or polling still wastes computing capacity.

Compared to the conventional software solution, the FSM-based solution has an almost constant latency and almost no jitter. It is in average more than seventy times faster when you take the different clock frequencies into account and does not cause any CPU load. Despite this fact, the FSM solution is not very complex and is still interface-compatible to the standard C_CAN except the second interface register.

The requirement to support further automotive field buses in addition to CAN resulted in the concept and implementation of the multi-protocol gateway. To unify the event signaling mechanisms of every network domain to the controlling gateway unit, wrapper units for C_CAN and E-Ray communication controllers have been implemented.

Further timing options and faster message event detections lead to the reimplementation of the gateway unit. The configuration RAM has been enhanced by a hash table in hardware to handle groups of messages and for faster access to the message objects.

As not every function is intended to run on the gateway unit without CPU interaction, the interface to the software layer of the gateway has been improved. This allows integration in existing ECU software projects including AUTOSAR. The hardware gateway as coprocessor can reduce the load on the CPU significantly and provides faster data transfers. The assumptions of very small code size and high execution speed have been verified by a demonstration environment.

A complete toolchain has been developed for generating gateway configurations based on FIBEX files. The generated configurations include data for the specialized gateway hardware and several communication controllers.

The combination of toolchain, hardware and software gateway provides one of the first solutions, fully configurable by a FIBEX file.

The hardware gateway is able to process about 80% of all transfers of an average gateway configuration. Complex transport protocols and extensive arithmetic operations are processed in cooperation with the software layer of the gateway ECU.

A verification environment has been implemented to allow black box testing of arbitrary gateways. This includes checks for data corruption, timing constraints and timing precision.

Extrapolating from the duration of the complete example test run, testing a gateway with several hundred frames and several thousand mappings may take a very long time. Further optimization, e.g. parallelization and less initial measurements, are possibilities to reduce the execution time.

The current initial test phase calibrates the test environment to the gateway, whereas the main test phase takes measurements under different load scenarios. Currently errors may falsify all following testcases till the end of the complete test run. This could be prevented if the test environment interrupts the test run and brings the gateway back to a known state as soon as an error is detected.

In the current main test phase several high load scenarios are missing. The target should be to create and execute further load tests, focused on scenarios that reflect realistic environments. As FIBEX is a relatively new standard, few gateway configurations are available in this format. These specification documents must be converted to FIBEX format to be able to execute authentic test scenarios.

Support for further networks needs to be implemented. Adding FlexRay connectivity will be the next step. LIN and MOST networks could follow, as these are used by some gateway implementations. Support of additional network protocols, such as transport and diagnostic protocols, will be necessary in the future.

Glossary

Notation	Description
4B5B	4 Bit-5 Bit Signal Codec
8B10B	8 Bit-10 Bit Signal Codec
API	Application Programming Interface
ASAM	Association for Standardisation of Automation- and Measurement Systems
ASIC	Application Specific Integrated Circuit
ATG	Automatic Test Generator
AUTOSAR	Automotive Open System Architecture
C2H	C-to-Hardware
CAN	Controller Area Network
CC	Communication Controller
CiA	CAN in Automation
CIF	Customer Interface
CISC	Complex Instruction Set Computer
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
CSMA	Carrier Sense Multiple Access
CSMA/BA	Carrier Sense Multiple Access with Bitwise Arbitration
CSMA/CA	Carrier Sense Multiple Access with Collision Avoidance
CSMA/CD	Carrier Sense Multiple Access with Collision Detection
CSMA/CD+CR	Carrier Sense Multiple Access with Collision Detection and Collision Resolution
DC	Direct Current
DIU	Data Integration Unit
DLL	Dynamic Link Library
DMA	Direct Memory Access

Notation	**Description**
DSP	Digital Signal Processor
ECU	Electronic Control Unit
EOF	End Of Frame
ePHY	Electrical Physical Interface
ESP	Electronic Stability Program
FIBEX	Field Bus Exchange Format
FIFO	First In First Out
FPGA	Field Programmable Gate Array
FPU	Floating Point Unit
FSM	Finite State Machine
FSP	Frame and Symbol Processing
FTDMA	Flexible Time Division Multiple Access
FTMA	Fault Tolerant Midpoint Algorithm
GCU	Gateway Control Unit
GIF	Generic Interface
GTU	Global Time Unit
IBF	Input Buffer
ICM	INIC Control Message
ID	Identification/Identifier
IDE	Integrated Development Environment
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
IFC	Interface Control (Register)
INIC	Intelligent Network Interface Controller
INT	Interrupt Control
IP 1	Intellectual Property
IP 2	Internet Protocol
IRAM	Instruction RAM
ISO	International Organization for Standardization
LAN	Local Area Network
LIN	Local Interconnect Network
LLC	Logical Link Control
MAC	Media Access Control(ler)
MCM	MOST Control Message
MDP	MOST Data Packet

Notation	Description
MHD	Message Handler
MIPS	Million Instructions Per Second
MLB	Media Local Bus
MMU	Memory Management Unit
MOST	Media Oriented System Transport
MRAM	Message RAM
Mutex	Mutual Exclusion
NEM	Network Management
NIC	Network Interface Controller
NMEA	National Marine Electronic Association
NRZ	Non-Return-to-Zero
OBF	Output Buffer
ODVA	Open DeviceNet Vendor Association
OEM	Original Equipment Manufacturer
oPHY	Optical Physical Interface
OSI	Open Systems Interconnection
PCM	Parallel Combined Mode
PDU	Process Data Unit
PHY	Physical Layer Transceiver
POC	Plastic Optical Fiber
PRT	FlexRay Channel Protocol Controller
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
RTE	Run Time Environment
RTL	Register Transfer Level
RTOS	Real-Time Operating System
SAE	Society of Automotive Engineers
SDK	Software Development Kit
SFR	Special Function Register
SIMD	Single Instruction, Multiple Data
SOPC	System-On-a-Programmable-Chip
SPE	Signal Processing Extension
SUC	System Universal Control
SWT	Stopwatch Trigger of Bosch TTCAN
TBF	Transient Buffer RAM
TCP	Transmission Control Protocol

Notation	**Description**
TDMA	Time Division Multiple Access
TMI	Time Mark Interrupt of Bosch TTCAN
TTCAN	Time-Triggered CAN
TTP	Time-Triggered Protocol
TTP/C	Time-Triggered Protocol SAE Class C
UDS	Unified Diagnostic Services
UTP	Unshielded Twisted Pair
VHDL	Very High Speed Integrated Circuit Hardware Description Language
VRAM	Vector RAM
VRAM-CC	Vector RAM - Communication Controller Partition
VRAM-MO	Vector RAM - Message Object Partition
WCET	Worst Case Execution Time
WLAN	Wireless Local Area Network
xCDEF	XML Cluster Definition File
XML	Extensible Markup Language

List of Figures

2.1 FIFO- / MRAM-based Communication Controller 17
2.2 CAN Standard and Extended Frame Format 19
2.3 C_CAN Block Diagram 20
2.4 TTCAN Matrix Cycle 22
2.5 TTCAN Block Diagram 23
2.6 FlexRay Frame Format 24
2.7 E-Ray Block Diagram 26
2.8 MOST Frame Format 29
2.9 Central Gateway Architecture 31
2.10 Backbone Architecture 31

3.1 Latency between two networks 33
3.2 Freescale MPC5567 Block Diagram 36
3.3 Infineon TriCore TC1130 Block Diagram 37
3.4 NEC V850E/PH3 Phoenix-FS Block Diagram 38
3.5 AUTOSAR Components 40
3.6 FIBEX Object Model 43

4.1 C_CAN Controller with Gateway Interface 46
4.2 Gateway Module with cascaded CAN/TTCAN cells 47
4.3 Data Integration Unit 48
4.4 Gateway Control Unit 49
4.5 Data Generation Flow 51
4.6 Gateway Layer Architecture 53

4.7 Gateway Control Unit 54
4.8 Communication Controller Wrapper 55
4.9 Standard Interface Concept 56
4.10 Extended Interface Concept 56
4.11 Time Behavior 58
4.12 Average Search Time 60
4.13 Gateway Toolchain Overview 62
4.14 Gateway Processing and Data Flow 63
4.15 Gateway Assembler Processing Flow 65
4.16 Macro Evaluation 66
4.17 Code Normalization 66
4.18 Pseudo Instruction Evaluation 67
4.19 Label Evaluation 67
4.20 Assembler Simulator 69
4.21 Static and Dynamic Testing 71
4.22 Gateway Model 72

5.1 Altera Nios II Development Board [92] 76
5.2 Clock Frequencies 77
5.3 Gateway Instruction Set 78
5.4 Structure of test system 80
5.5 Software structure 81
5.6 Time behavior of the CAN communication 81
5.7 Gateway Vector and Instruction RAM 86
5.8 Gateway Instruction Set 88
5.9 Gateway Instruction Set Codes 89
5.10 Finite State Machines 90
5.11 VRAM Arbiter 91
5.12 Runtime and Memory Optimization 93
5.13 Instruction Count and Code Size 94
5.14 Execution time (best and worst case) 95

5.15 Cyclic Sending: Cycle Time 96
5.16 Event Controlled Sending: Forwarding Latency 97
5.17 Timeout 98
5.18 Debouncing 98
5.19 CANoe Test Environment 101

List of Tables

2.1 ISO/OSI Reference Model 13

5.1 Test immediate Tx after Rx 82
5.2 Tx cyclic sending 83
5.3 Tx after Rx Timeout 83
5.4 Swap Message 84
5.5 Split Message 84
5.6 Code and RAM sizes (bytes) 84
5.7 Buffer Configuration 86
5.8 Gate count of CCs, Wrapper and Gateway Control Unit 92
5.9 Gateway measurement results 101
5.10 Instructions per Mapping Category 107
5.11 Frame rates of a High-Class Car 109
5.12 RAM Usage of a High-Class Car 109
5.13 GCU Utilization of a High-Class Car 109
5.14 Frame rates of a Mid-Class Car 110
5.15 RAM Usage of a Mid-Class Car 111
5.16 GCU Utilization of a Mid-Class Car 111

Bibliography

Introduction

[1] Jürgen Leohold. Communication Requirements for Automotive Systems. *5th IEEE Workshop on Factory Communication Systems*, sep 2004. http://iestcfa.org/presentations/wfcs04/keynote_leohold.pdf (accessed June 04th, 2007).

Communication Networks

[2] CAN in Automation. *CANopen*, 2006. http://www.can-cia.org/canopen/ (accessed June 04th, 2007).

[3] Konrad Etschberger. *Controller Area Network.* IXXAT Automation GmbH, 2001. ISBN 3-00-007376-0.

[4] Konrad Etschberger. *Controller-Area-Network. Grundlagen, Protokolle, Bausteine, Anwendungen.* Fachbuchverlag Leipzig, third edition, nov 2006. ISBN 3-446-21776-2.

[5] FlexRay Consortium. *FlexRay Communication System. Protocol Specification. Version 2.1 Revision A*, dec 2005. http://www.flexray.com (accessed June 04th, 2007).

[6] Lars-Berno Frederikson. *A CAN Kingdom. Revision 3.01.* KVASER AB, 1995. http://www.cankingdom.org/ck301.zip (accessed June 04th, 2007).

[7] Prof. Harald Heinecke, Dr. Anton Schedl, Josef Berwanger, Martin Peller, Volker Nieten, Dr. Ralf Belschner, Dr. Bernd Hedenetz, Peter Lohmann, and Claas Bracklo. FlexRay - ein Kommunikationssystem für das Automobil der Zukunft. *Design & Elektronik*, sep 2002. http://www2.elektroniknet.de/topics/kommunikation/fachthemen/2003/0002/ (accessed June 04th, 2007).

[8] Robert Huber. *Konzept, Realisierung und Bewertung des Sensor/Aktorbusses TTP/A*. PhD thesis, Technical University Munich, mar 2003. `http://mediatum.ub.tum.de/mediatum/servlets/TUMDistributionServlet?id=mediaTUM_disshab_000000000001588` (accessed June 04th, 2007).

[9] Infineon Technologies. *Single Wire CAN-Transceiver - TLE 6255 G*, nov 2003. `http://www.infineon.com/upload/Document/cmc_upload/documents/010/6284/tle6255finalDSV25.pdf` (accessed June 04th, 2007).

[10] ISO/IEC. *Information technology - Open Systems Interconnection - Basic Reference Model: The Basic Model*, 1994. `http://standards.iso.org/ittf/PubliclyAvailableStandards/s020269_ISO_IEC_7498-1_1994(E).zip` (accessed June 04th, 2007).

[11] Ursula Kelling. The MultiCAN module -Two CAN were not enough. *CAN Newsletter*, pages 16–18, jun 2004.

[12] Hermann Kopetz. *Real-Time Systems. Design Principles for Distributed Embedded Applications*. Springer Netherlands, first edition, 1997. ISBN 0-7923-9894-7.

[13] Wolfhard Lawrenz. *CAN Controller Area Network. Grundlagen und Praxis*. Hüthig, 2000. ISBN 3-7785-2780-0.

[14] Wolfhard Lawrenz. *CAN System Engineering. From Theory to Practical Applications*. Springer Verlag, Berlin, 2001. ISBN 0-387-94939-9.

[15] LIN Consortium. *LIN Specification Package. Revision 1.3*, dec 2002. `http://www.lin-subbus.org/` (accessed June 04th, 2007).

[16] LIN Consortium. *LIN Specification Package. Revision 2.0*, sep 2003. `http://www.lin-subbus.org/` (accessed June 04th, 2007).

[17] LIN Consortium. *LIN Specification Package. Revision 2.1*, nov 2006. `http://www.lin-subbus.org/` (accessed June 04th, 2007).

[18] G. Maus. Stand-alone CAN controller with eight channels. *CAN Newsletter*, pages 22–24, jun 2004.

[19] Michael Stock Flight Systems. *CANaerospace*, 2006. `http://www.stockflightsystems.com/html/canaerospace.html` (accessed June 04th, 2007).

[20] MOST Cooperation. *MOST Specification Framework*, 1999. Revision 1.1. Version 1.1-07.

[21] MOST Cooperation. *MOST Specification*, jun 2005. Revision 2.4.

[22] NMEA 2000®Standard Committee. *NMEA 2000®Standard*, 2003. `http://www.nmea.org/pub/2000/index.html` (accessed June 04th, 2007).

[23] OASIS SiliconSystems. *OS8104 MOST Network Transceiver. Final Product Data Sheet*, jan 2003. `http://www.smsc-ais.com/files/ics/DS8104FP4.pdf` (accessed June 04th, 2007).

[24] OASIS SiliconSystems. *Errata: OS8104, Revision D*, 2006. `http://www.smsc-ais.com/files/ics/ER8104ZZ0D8.pdf` (accessed June 04th, 2007).

[25] OASIS SiliconSystems. *Media Local Bus Specification. Physical Layer and Link Layer. Version 3.0*, 2006. `http://www.smsc-ais.com/files/MediaLB_Spec_3v0.pdf` (accessed June 04th, 2007).

[26] ODVA. *DeviceNet Technical Overview*, 2004. `http://www.odva.org/10_2/05_tech/PUB00026R1.pdf` (accessed June 04th, 2007).

[27] M. Peller, J. Berwanger, and R. Grießbach. *byteflight specification*. BMW AG, oct 1999. `http://www.byteflight.com` (accessed June 04th, 2007).

[28] Philips Electronics. *TJA1080 FlexRay Bus Driver. High-speed time-triggered communication system transceiver chip*, jun 2004. `http://www.semiconductors.philips.com/acrobat_download/literature/9397/75013048.pdf` (accessed June 04th, 2007).

[29] G. Reichart, R. Constapel, P. Hansson, H. Hönninger, K. Lange, R. Makowitz, J.-M. Schneider, and W. Streit. Flexray - eine vision wird realität. In *12ter internationaler Kongress - Elektronik im Kraftfahrzeug*. VDI Fahrzeug- und Verkehrstechnik, oct 2005.

[30] Robert Bosch GmbH. *CAN Specification. Version 2.0*, 1991.

[31] Robert Bosch GmbH. *Datenblatt CF151 (UB62)*, jul 1998. `http://www.semiconductors.bosch.de/pdf/DB_CF151.pdf` (accessed June 04th, 2007).

[32] Robert Bosch GmbH. *C_CAN User's Manual. Revision 1.2*, jun 2000.

[33] Robert Bosch GmbH. *TTCAN User's Manual. Revision 1.6*, nov 2002.

[34] Robert Bosch GmbH. *TTCAN Module Integration Guide. Revision 1.6*, mar 2003.

[35] Robert Bosch GmbH. *C_CAN Module Integration Guide. Revision 1.2.1*, sep 2004.

[36] Robert Bosch GmbH. *E-Ray. FlexRay IP Module. Module Integration Guide. Revision 1.0.2*, dec 2006.

[37] Robert Bosch GmbH. *E-Ray. FlexRay IP Module. User's Manual. Revision 1.2.5*, dec 2006.

[38] Robert Bosch GmbH. *Product Information CAN Bus Transceiver - CF151*, feb 2006. `http://www.semiconductors.bosch.de/pdf/CF151_Product_Info.pdf` (accessed June 04th, 2007).

[39] Stephan Rohr and Herbert Kabza. High Speed Optical Controller Area Network (CAN). In *10th iCC Proceedings*, pages 04–1–04–8. CAN in Automation, mar 2005.

[40] SAE International. *SAE J1939 Standards Collection*, 2006. `http://www.sae.org/standardsdev/groundvehicle/j1939a.htm` (accessed June 04th, 2007).

[41] SMSC. *OS8104A MOST Network Transceiver. Final Product Data Sheet*, jul 2005.

[42] SMSC. *OS81050 INIC. Priliminary Product Data Sheet*, oct 2005.

[43] SMSC. *ePHY Overview and Implementation*, 2006. `http://www.smsc-ais.com/files/ePHY/ePHY_Overview_TB0812AN1A1.pdf` (accessed June 04th, 2007).

[44] SMSC. *MOST Network Interface Controller. Multimedia Network Protocol Chip OS8104A*, 2006. `http://www.smsc-ais.com/files/OS8104A/PFL_OS8104A_V01_00_XX-1.pdf` (accessed June 04th, 2007).

[45] SMSC. *MOST Network Interface Controller OS8104. Multimedia Network Protocol Chip*, 2006. `http://www.smsc-ais.com/files/ics/PFL_OS8104-03.pdf` (accessed June 04th, 2007).

[46] SMSC. *OS81050 INIC25 oPHY*, 2006. `http://www.smsc-ais.com/files/ics/PFL_OS81050_V01_00_XX-6.pdf` (accessed June 04th, 2007).

[47] SMSC. *OS81082 INIC50 ePHY*, 2006. `http://www.smsc-ais.com/files/OS81082/PFL_OS81082_V01_00_XX-2.pdf` (accessed June 04th, 2007).

[48] Andrew S. Tanenbaum. *Computernetzwerke*. Pearson Studium - Prentice Hall, fourth edition, 2003. ISBN 3-8273-7046-9.

[49] TTTech Computertechnik GmbH. *Time-Triggered Protocol TTP/C, High-Level Specification Document, Protocol Version 1.1*, nov 2003. `http://www.tttech.com` (accessed June 04th, 2007), `http://www.ttagroup.org` (accessed June 04th, 2007).

[50] Vector Informatik. *CANdb++ und CANdb++ Admin. The Distributed System's Backbone*, may 2005. `http://www.vector-worldwide.com/portal/medien/cmc/datasheets/CANdb_DataSheet_DE.pdf` (accessed June 04th, 2007).

[51] Vector Informatik. *CANdb++ und CANdb++ Admin. The Distributed System's Backbone*, may 2005. `http://www.vector-worldwide.com/portal/medien/cmc/datasheets/CANdb_DataSheet_EN.pdf` (accessed June 04th, 2007).

[52] Vector Informatik. *CANdbLib. Programmierschnittstelle für den Zugriff auf CAN-Datenbanken*, may 2005. `http://www.vector-worldwide.com/portal/medien/cmc/datasheets/CANdbLib_DataSheet_DE.pdf` (accessed June 04th, 2007).

[53] Vector Informatik. *CANdbLib. Programming Interface for Access to CAN Databases*, may 2005. `http://www.vector-worldwide.com/portal/medien/cmc/datasheets/CANdbLib_DataSheet_EN.pdf` (accessed June 04th, 2007).

[54] Werner Zimmermann and Ralf Schmidgall. *Bussysteme in der Fahrzeugtechnik*. Vieweg, sep 2007. ISBN 3-8348-0235-2.

Current Gateways

[55] ASAM e.V. *FIBEX. Field Bus Exchange Format. Version 1.2*, sep 2005.

[56] ASAM e.V. *FIBEX. Field Bus Exchange Format. Version 2.0*, jun 2006.

[57] AUTOSAR GbR. *Specification of System Template. Version 1.0.0*, 2005.

[58] AUTOSAR GbR. *AUTOSAR - Layered Software Architecture*, feb 2007. Version 2.1.0. `https://svn.autosar.org/repos/10Releases/internal/release2.1/Standard/AUTOSAR_LayeredSoftwareArchitecture.pdf` (accessed June 04th, 2007).

[59] AUTOSAR GbR. *AUTOSAR - Technical Overview*, jan 2007. Version 2.1.0. `https://svn.autosar.org/repos/10Releases/internal/release2.1/auxiliary/AUTOSAR_TechnicalOverview.pdf` (accessed June 04th, 2007).

[60] Thomas Barthel. Austauschformat der Zukunft - XML-basiertes Datenformat zur Beschreibung von Fahrzeugnetzwerken. *Automotive*, 2006. `http://www.elektroniknet.de/home/automotive/fachwissen/uebersicht/l/bussysteme/austauschformat-der-zukunft/` (accessed June 04th, 2007).

[61] G. Bernat. Analyse der Worst Case Execution Time in Automobilanwendungen. *Elektronik Praxis*, pages 45–55, 9 2005.

[62] Oliver Falkner and Christiane Picard. Quo vadis Kfz-Elektronik? - Optimierungsvorschläge für die Elektronikentwicklung. *Elektronik Automotive*, pages 82–85, 1 2005.

[63] Freescale. *XGate Library: Signal Gateway - Implementing CAN and LIN Signal Level Gateway*, nov 2006. `http://www.freescale.com/files/microcontrollers/doc/app_note/AN3333.pdf` (accessed June 04th, 2007).

[64] Freescale Semiconductor. *16-bit Microcontrollers - The XGATE Coprocessor*, jan 2005. `http://www.freescale.com/files/microcontrollers/doc/fact_sheet/XGATECOPROCFS.pdf` (accessed June 04th, 2007).

[65] Freescale Semiconductor. *High-Performance 16-bit Microcontrollers - HCS12X*, jun 2005. `http://www.freescale.com/files/microcontrollers/doc/fact_sheet/HC9S12XFAMFS.pdf` (accessed June 04th, 2007).

[66] Freescale Semiconductor. *MPC5567 32-Bit Embedded Controller*, 2006. `http://www.freescale.com/files/32bit/doc/fact_sheet/MPC5567FS.pdf` (accessed June 04th, 2007).

[67] Josef Fuchs. Robuster Rechenkünstler - Optimierung der Systemleistung am Beispiel des MPC5554: Hohe Taktfrequenzen allein reichen nicht aus. *Elektronik Automotive*, 5 2005. `http://www.elektroniknet.de/home/bauelemente/fachwissen/uebersicht/aktive-bauelemente/mikrocontroller-prozessoren-dsps/robuster-rechenkuenstler/` (accessed June 04th, 2007).

[68] Dr. Robert Huber. Vom einfachen Bussystem zum anspruchsvollen Datennetz - Teil 2. In *Automotive 9-10.2004*, pages 35–40. Carl Hanser Verlag GmbH, 2004. `http://www.hanser-automotive.de/fileadmin/heftarchiv/2004/3025.pdf` (accessed June 06th, 2007).

[69] Infineon Technologies AG. *TC1130 32-Bit Single-Chip Microcontroller. Volume 1 (of 2): System Units*, nov 2004. `http://www.infineon.com/upload/Document/cmc_upload/documents/`

010/1702/tc1130_um_v1.3_2004_11_sys.pdf (accessed June 04th, 2007).

[70] Infineon Technologies AG. *TC1130 32-Bit Single-Chip Microcontroller. Volume 2 (of 2): Peripheral Units*, nov 2004. http://www.infineon.com/upload/Document/cmc_upload/documents/010/1703/tc1130_um_v1.3_2004_11_per.pdf (accessed June 04th, 2007).

[71] Infineon Technologies AG. *TC1130 32-bit Superscalar TriCore^{TM}Architecture. Product Brief*, feb 2004. http://www.infineon.com/upload/Document/cmc_upload/documents/098/690/tc1130-pb.pdf (accessed June 04th, 2007).

[72] Infineon Technologies AG. *TC1130 32-Bit Single-Chip Microcontroller Advance Information*, feb 2005. http://www.infineon.com/upload/Document/cmc_upload/documents/098/695/TC1130_DS_V1.0.pdf (accessed June 04th, 2007).

[73] ISO/IEC. *Road vehicles – Open interface for embedded automotive applications*, 2005. http://www.iso.org/iso/en/StandardsQueryFormHandler.StandardsQueryFormHandler?scope=ALL\&keyword=\&isoNumber=17356\&sortOrder=ISO\&title=true\&search_type=ISO\&search_term=17356\&languageCode=en (accessed June 04th, 2007).

[74] K2L Software. *K2L MoCCa Vario II*, 2005. http://k2lgmbh.de/website/products/moccavario2/MoCCaVario2.pdf (accessed June 04th, 2007).

[75] Ata Khan. Weniger Verlustleistung ist gefordert! - 90-nm-Mikrocontroller verbessern Integrationsgrad, Energiebilanz, Rechenleistung und Produktionskosten. *Elektronik SoC*, 1 2006. http://www.elektroniknet.de/home/bauelemente/fachwissen/uebersicht/aktive-bauelemente/mikrocontroller-prozessoren-dsps/weniger-verlustleistung-ist-gefordert/ (accessed June 04th, 2007).

[76] NEC Electronics (Europe) GmbH. *V850 Phoenix-FS 32-bit RISC Microcontroller*, nov 2005. http://www.eu.necel.com/_pdf/EPMC-PU-0002-3.0.PDF (accessed June 04th, 2007).

[77] OSEK/VDX. *OSEK/VDX - Communication*, jul 2004. Version 3.0.3. http://portal.osek-vdx.org/files/pdf/specs/osekcom303.pdf (accessed June 04th, 2007).

[78] OSEK/VDX. *OSEK/VDX - Network Management*, jul 2004. Version 2.5.3. `http://portal.osek-vdx.org/files/pdf/specs/nm253.pdf` (accessed June 04th, 2007).

[79] OSEK/VDX. *OSEK/VDX - Operating System*, feb 2005. Version 2.2.3. `http://portal.osek-vdx.org/files/pdf/specs/os223.pdf` (accessed June 04th, 2007).

Advanced Gateway Architectures

[80] ARM. *AMBA Specification, Revision 2.0*, may 1999.

[81] DECOMSYS GmbH. *DECOMSYS::DESIGNER PRO*, feb 2007. Version 4.2. `http://www.decomsys.com/flyer/Datasheet_DESIGNER_PRO.pdf` (accessed June 04th, 2007).

[82] Free Software Foundation. *GNU M4 - A powerful macro processor*, aug 2006. Version 1.4.6. `http://www.gnu.org/software/m4/manual/m4.pdf` (accessed June 04th, 2007).

[83] Stephen Johnson. Lint, a C program checker. In *Computer Science Technical Report 65*. Bell Laboratories, jul 1998. `http://citeseer.ist.psu.edu/johnson78lint.html` (accessed June 04th, 2007).

[84] Cem Kaner, James Bach, and Bret Pettichord. *Lessons Learned in Software Testing. A Context-Driven Approach.* Wiley, first edition, dec 2001. ISBN 0-471-08112-4.

[85] Peter Liggesmeyer. *Software Qualität. Testen, Analysieren und Verifizieren von Software.* Spektrum Akademischer Verlag, aug 2002. ISBN 3-827-41118-1.

Implementation and Results

[86] Altera Corporation. *Nios II Custom Instruction - User Guide*, dec 2004. `http://www.altera.com/literature/ug/ug_nios2_custom_instruction.pdf` (accessed June 04th, 2007).

[87] Altera Corporation. *Avalon Memory-Mapped Interface - Specification*, nov 2006. `http://www.altera.com/literature/manual/mnl_avalon_spec.pdf` (accessed June 04th, 2007).

[88] Altera Corporation. *Nios II C2H Compiler - User Guide*, aug 2006. `http://www.altera.com/literature/ug/ug_nios2_c2h_compiler.pdf` (accessed June 04th, 2007).

[89] Altera Corporation. *Nios II Processor Reference Handbook*, nov 2006. `http://www.altera.com/literature/hb/nios2/n2cpu_nii5v1.pdf` (accessed June 04th, 2007).

[90] Altera Corporation. *Quartus II Version 6.1 Handbook*, nov 2006. `http://www.altera.com/literature/hb/qts/quartusii_handbook.pdf` (accessed June 04th, 2007).

[91] Altera Corporation. *Stratix II Device Handbook*, apr 2006. `http://www.altera.com/literature/hb/stx2/stratix2_handbook.pdf` (accessed June 04th, 2007).

[92] Altera Corporation. *Nios II Development Kit, Stratix II Edition*, 2007. `http://www.altera.com/products/devkits/altera/kit-niosii-2S60.html` (accessed June 04th, 2007).

[93] Wikipedia. *Field-Programmable gate array*, 2007. `http://en.wikipedia.org/wiki/FPGA` (accessed August 28th, 2007).

Gateway-related Bosch Publications

[94] Stefan Bleeck. Evaluierung von Gateway-Konzepten für automobile Bussysteme. Master's thesis, Universität Rostock, Fakultät für Informatik und Elektrotechnik, jul 2006.

[95] Florian Hartwich and Christian Horst. Message Handling Concept for a FlexRay Communication Controller. In *Automotive 2004 - Special Edition FlexRay*, pages 32–35. Carl Hanser Verlag GmbH, 2004. `http://www.hanser-automotive.de/fileadmin/heftarchiv/2004/flex32-35.pdf` (accessed June 04th, 2007).

[96] Thomas Lindenkreuz and Florian Hartwich. Integration of Car Communication. In *electronica automotive conference 2006*, 2006.

[97] Tobias Lorenz, Jan Taube, and Markus Ihle. Evaluation of CAN gateway implementations. In *CAN Newsletter*, pages 24–26. CAN in Automation, mar 2007.

[98] Tobias Lorenz, Jan Taube, Markus Ihle, Otto Manck, and Helmut Beikirch. FIBEX Gateway Configuration Tool Chain. In *11th international CAN Conference Proceedings*, pages 04–13–04–19. CAN in Automation, sep 2006.

[99] Tobias Lorenz, Jan Taube, Markus Ihle, Otto Manck, and Helmut Beikirch. Verification Environment for Automotive Gateways. In *Proceedings embedded world Conference*. WEKA Fachzeitschriften-Verlag GmbH, feb 2007.

[100] Jan Taube. *Entwicklung und Konzeptionierung von Hardwarearchitekturen für Gateways im Automotive-Bereich*. PhD thesis, Universität Rostock, Fakultät für Informatik und Elektrotechnik, may 2007.

[101] Jan Taube and Helmut Beikirch. Self-Synchronizing Time-Triggered Networks in Automotive and Industrial Gateways. In *4th International Symposium on AUTOMATIC CONTROL*. Hochschule Wismar, sep 2005. ISBN 3-910102-79-4.

[102] Jan Taube and Kay Hammer. CAN Gateway and FlexRay IP Solutions for Automotive Networks, 2005. Flyer FlexRay Product Day 2005.

[103] Jan Taube, Florian Hartwich, and Helmut Beikirch. C_CAN Gateway Module - A New Approach for CAN Gateways. In *Proceedings embedded world Conference*, pages 80–87. WEKA Fachzeitschriften-Verlag GmbH, feb 2005.

[104] Jan Taube, Florian Hartwich, and Helmut Beikirch. Comparison of CAN Gateway Modules for Automotive and Industrial Control Applications. In *10th international CAN Conference Proceedings*, pages 06–1–06–8. CAN in Automation, mar 2005.

[105] Jan Taube, Florian Hartwich, and Helmut Beikirch. Gateway Concepts for Automotive Networks. In *automotive 2005 - Special Edition FlexRay*, pages 10–12. Carl Hanser Verlag GmbH, dec 2005.

[106] Jan Taube, Tobias Lorenz, Helmut Beikirch, and Otto Manck. Performance evaluation of different CAN Gateway implementations. In *Proceedings embedded world Conference 2006*, volume 1, pages 573–581. Franzis Verlag GmbH Poing, feb 2006. ISBN 3-7723-0143-6.

[107] Jan Taube, Tobias Lorenz, Markus Ihle, Helmut Beikirch, and Otto Manck. Kommunikationsarchitekturen zur Verbindung von Netzwerken im Automobil. In *KFZ-Elektronik*. WEKA Fachzeitschriften-Verlag GmbH, may 2007.

Gateway-related Bosch Patents

[108] Jan Taube Florian Hartwich. Patent: Verfahren und Vorrichtung zur Synchronisation zweier Bussysteme sowie Anordnung aus zwei Bussystemen, 2005. German Patent Nr.: 102005018837.0.

[109] Florian Hartwich and Jan Taube. Patent: Kommunikationsbausteinanordnung mit einem Schnittstellenmodul und Schnittstellenmodul, 2004/2005. German Patent Nr.: 102004057410.3. European Patent Nr.: 05817026.7-2212.

[110] Tobias Lorenz, Jan Taube, and Markus Ihle. Patent: Teilnehmer und Kommunikationscontroller eines Kommunikationssystems und Verfahren zur Realisierung einer Gateway-Funktionalität in einem Teilnehmer eines Kommunikationssystems, 2005. German Patent Nr.: 102005048585.5.

[111] Jan Taube, Markus Ihle, and Tobias Lorenz. Patent: Gateway zum automatischen Routen von Nachrichten zwischen Bussen, 2007. German Patent Nr.: 102007001137.9.

[112] Jan Taube, Tobias Lorenz, and Markus Ihle. Patent: Gateway zum Datentransfer zwischen seriellen Bussen, 2006. German Patent Nr.: 102006055514.7.

[113] Jan Taube, Tobias Lorenz, and Markus Ihle. Patent: Kommunikationsbaustein, 2006. German Patent Nr.: 102006055513.9.

[114] Jan Taube, Tobias Lorenz, Markus Ihle, and Stefan Bleeck. Patent: Mehrprozessor-Gateway, 2006. German Patent Nr.: 102006055512.0.

Curriculum Vitae

Name:	Tobias Lorenz
Birth:	April 26th, 1980 in Berlin, Germany
High school:	Aug 1997 - Jun 1999: OSZ Energietechnik I, Berlin-Spandau
Civilian service:	Jul 1999 - Jul 2000: Bethel Krankenhaus, Berlin-Steglitz Tasks: Administration, Warehouse, Engineering
University:	Oct 2000 - Sep 2004: Technical University of Berlin Degree: Dipl.-Ing. Computer Engineering (Grade A) Duration: 8 semester (regular 10 semester)
Diploma thesis:	uCLinux 2.6 for ARM: Kernel porting and Network bootloader in cooperation with Mikroelektronik Anwendungszentrum Brandenburg
Doctorate:	since May 2005: Industrial doctorate at **Robert Bosch GmbH** in Reutlingen in cooperation with **Technical University of Berlin**

Internships:	Jan 2004 - Mar 2004: Itellium (KarstadtQuelle), Essen Tasks: Linux terminal servers
	Apr 2004 - May 2004: Mikroelektronik Anwendungszentrum Brandenburg Tasks: Development of GDB-JTAG-ARM debugger
Part-time jobs:	Jun 2004 - May 2005: Mikroelektronik Anwendungszentrum Brandenburg Tasks: Industrial automation
	Sep 2003 - Dec 2003: Bethel Krankenhaus, Administration Tasks: Conception and implementation of databases

Berlin, May 31, 2007