Document Title	Technical Overview
Document Owner	AUTOSAR
Document Responsibility	AUTOSAR
Document Identification No	067
Document Classification	Auxiliary

Document Version	2.2.2
Document Status	Final
Part of Release	3.2
Revision	1

Document Change History

Date	Version	Changed by	Change Description
27.04.2011	2.2.2	AUTOSAR Administration	Legal disclaimer revised
23.06.2008	2.2.1	AUTOSAR Administration	Legal disclaimer revised
28.11.2007	2.2.0	AUTOSAR Administration	• Subchapter “limitations of the current version of this document” added • Document meta information extended • Small layout adaptations made
31.01.2007	2.1.0	AUTOSAR Administration	Removed CCU as self-contained module in MCAL • Legal disclaimer revised • Release Notes added • “Advice for users” revised • “Revision Information” added
14.03.2006	2.0.0	AUTOSAR Administration	Major update of Methodology chapter due to integration with metamodel. New RTE section
15.07.2005	1.0.1	AUTOSAR Administration	References updated. Minor Typos corrected
08.07.2005	1.0.0	AUTOSAR Administration	Initial Release

Disclaimer

This specification and the material contained in it, as released by AUTOSAR is for the purpose of information only. AUTOSAR and the companies that have contributed to it shall not be liable for any use of the specification.

The material contained in this specification is protected by copyright and other types of Intellectual Property Rights. The commercial exploitation of the material contained in this specification requires a license to such Intellectual Property Rights.

This specification may be utilized or reproduced without any modification, in any form or by any means, for informational purposes only.
For any other purpose, no part of the specification may be utilized or reproduced, in any form or by any means, without permission in writing from the publisher.

The AUTOSAR specifications have been developed for automotive applications only. They have neither been developed, nor tested for non-automotive applications.

The word AUTOSAR and the AUTOSAR logo are registered trademarks.

Advice for users

AUTOSAR Specification Documents may contain exemplary items (exemplary reference models, "use cases", and/or references to exemplary technical solutions, devices, processes or software).

Any such exemplary items are contained in the Specification Documents for illustration purposes only, and they themselves are not part of the AUTOSAR Standard. Neither their presence in such Specification Documents, nor any later documentation of AUTOSAR conformance of products actually implementing such exemplary items, imply that intellectual property rights covering such exemplary items are licensed under the same rules as applicable to the AUTOSAR Standard.

Table of Contents

1 Motivation 5
1.1 Introduction 5
1.2 Limitations to this version of the document 5
1.3 Why do we need AUTOSAR? 5
1.4 Objectives and Technical Benefits of AUTOSAR 5
2 Concept 8
2.1 Introduction to the AUTOSAR concept 8
2.1.1 Basic AUTOSAR approach 8
2.2 Components 9
2.2.1 The AUTOSAR Software Component 9
2.2.2 The AUTOSAR Software Component is an "Atomic Software Component" 10
2.2.3 Implementing and shipping an AUTOSAR Software Component 10
2.2.4 The AUTOSAR Software Component Description 11
2.2.5 The AUTOSAR Software Component implementation is independent from the infrastructure 11
2.2.6 Sensor/Actuator Software Components 12
2.2.7 The generic “Component” concept 13
2.2.8 Summary 13
2.3 VFB 13
2.3.1 General 13
2.3.2 VFB Context 13
2.3.3 Communication mechanisms 14
2.4 AUTOSAR ECU Software Architecture 17
2.4.1 Overview 17
2.4.2 AUTOSAR Software 17
2.4.3 AUTOSAR Runtime Environment 18
2.4.4 AUTOSAR Basic Software 18
2.4.5 Classification of interfaces 19
3 AUTOSAR Methodology 21
3.1 Introduction 21
3.1.1 Describing Notation – SPEM 21
3.2 Methodology Overview 22
3.3 System Configuration 24
3.4 ECU Design and Configuration Methodology 25
3.4.1 Extract ECU-Specific Information 26
3.4.2 Configure ECU 26
3.4.3 Generate Executable 27
3.5 Component Implementation 27
4 Detailed ECU Architecture 30
4.1 Layered Software Architecture 30
4.1.1 The Layered Architecture 30
4.1.2 Refinement of the Layered Architecture 32
4.1.3 Related Documents 35

4.2 The Runtime Environment (RTE) 35
4.2.1 Overview 35
4.2.2 Access to ports from a software component implementation 36
4.2.3 Implementation of connectors 36
4.2.4 Lifecycle management 37
4.2.5 Access to Basic Software 37
4.2.6 Multiple Instantiations of software components 37
4.3 Complex Device Driver 37
4.3.1 General 37
4.3.2 Complex Sensor and Actuator Control 38
4.3.3 Non-Standardized Drivers 38
4.3.4 Migration Mechanism 38
5 Functional Interfaces 39
5.1 Overview 39
5.2 Functional domains 40
6 References 41

1 Motivation

1.1 Introduction

This Technical Overview is a brief description of the basic technical concepts of AUTOSAR. It assumes no previous knowledge of the project. Starting with the motivation for AUTOSAR the system architecture and the development methodology will be presented. This document does not focus on details. For a more comprehensive introduction to certain topics further AUTOSAR specifications have to be considered.

With this scope the emphasis is on an introduction for new AUTOSAR stakeholders, e.g. new work package members. Besides of this, it shall be a compact and efficient overview of the AUTOSAR technology. Hence it shall also help experienced AUTOSAR members to keep an overview on the technological context of their detailed work or it could provide them with technical background for business decisions.

1.2 Limitations to this version of the document

Over the past releases in AUTOSAR, detailed work on both template and software specifications has progressed. Hence, one might experience inconsistencies between this top-level document and detailed specifications as referred to in Chapter 6 References. In the unlikely case of a conflict between this document and one of the referenced detailed specifications, the latter shall take precedence.

1.3 Why do we need AUTOSAR?

AUTOSAR (AUTomotive Open System ARchitecture) is a partnership of automotive manufacturers and suppliers working together to develop and establish a de-facto open industry standard for automotive E/E architectures.

From a technical point of view, we can summarize the driving forces for the intended standardization as follows:

- Manage increasing E/E *complexity* associated with growth in functional scope
- Improve *flexibility* for product modification, upgrade and update
- Improve *scalability* of solutions within and across product lines
- Improve *quality and reliability* of E/E systems
- Enable detection of errors in *early design phases.*

1.4 Objectives and Technical Benefits of AUTOSAR

The primary project objectives are shown on the left side of
Figure 1. The figure also shows the functional domains which are in focus of AUTOSAR. From these primary objectives a list of main requirements has been derived, see [MainReq].

How are these objectives reached from a technical point of view? To explain the basic ideas, the following table shows some principal challenges and the solutions suggested by AUTOSAR together with the benefits achieved by these solutions (from [Conv2004]). The table does not include process and business aspects.

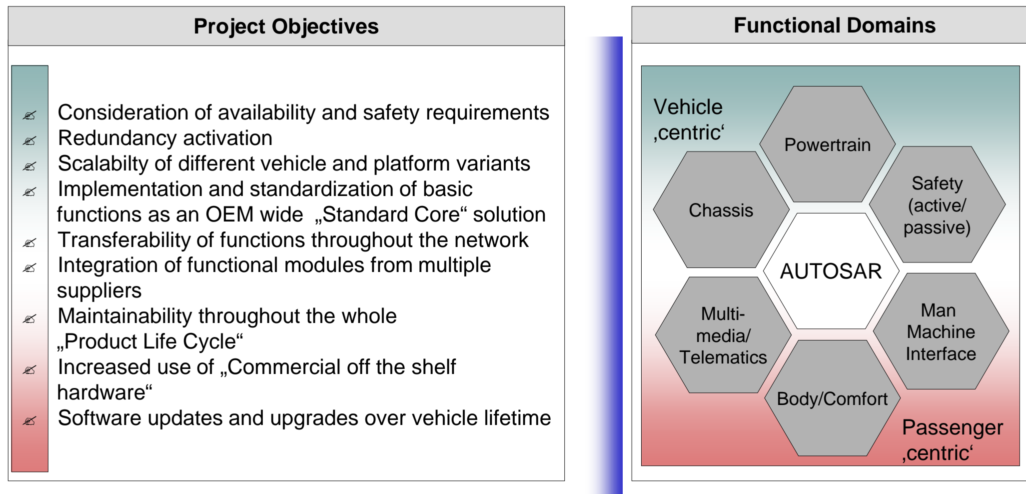


Figure 1: AUTOSAR high-level project objectives

Challenge	*Solution*	*Benefit*
• Immature processes because of acting in ad-hoc mode/ missing traceability of functional requirements. • Lack of compatible tooling (supplier, OEM).	**Standardization of specification exchange formats**	• Improvement in specification (format and content). • Opportunity for a seamless tool chain.
• Effort wasted on implementation and optimization of components, which add no value recognized by the customer.	**Basic Software core**	• Enhancement of software quality. • Concentration on functions with competitive value.
• Stop of availability of microcontroller models causes huge efforts in adapting existing software. • Extended needs for microcontroller performance (caused by new functions) cause need for upgrade, i.e. re-design effort.	**Microcontroller abstraction**	• Microcontroller can be exchanged without need for adaptations of higher software layers.
• Large effort when relocating functions between ECUs. • Large effort when reusing functions.	**Runtime Environment (RTE)**	• Encapsulation of functions creates independence of communication technology. • Communication easier through standardized mechanisms. • Partitioning and relocatability of functions possible.
• Non-competitive functions have to be adapted to OEM specific environments. • Tiny little innovations cannot be implemented at reasonable effort as provision of interfaces from other components requires a lot of effort. • Missing clear interfaces between basic software and code generated from models.	**Standardization of interfaces**	• Reduction/avoidance of interface proliferation within and across OEMs and suppliers. • Eased implementation of hardware independent software functionality by using generic interface catalogues. • Simplifies the model-based development and allows the use of standardized AUTOSAR code generation tools. • Reusability of modules cross-OEM. • Exchangeability of components from different suppliers.

2 Concept

2.1 Introduction to the AUTOSAR concept

2.1.1 Basic AUTOSAR approach

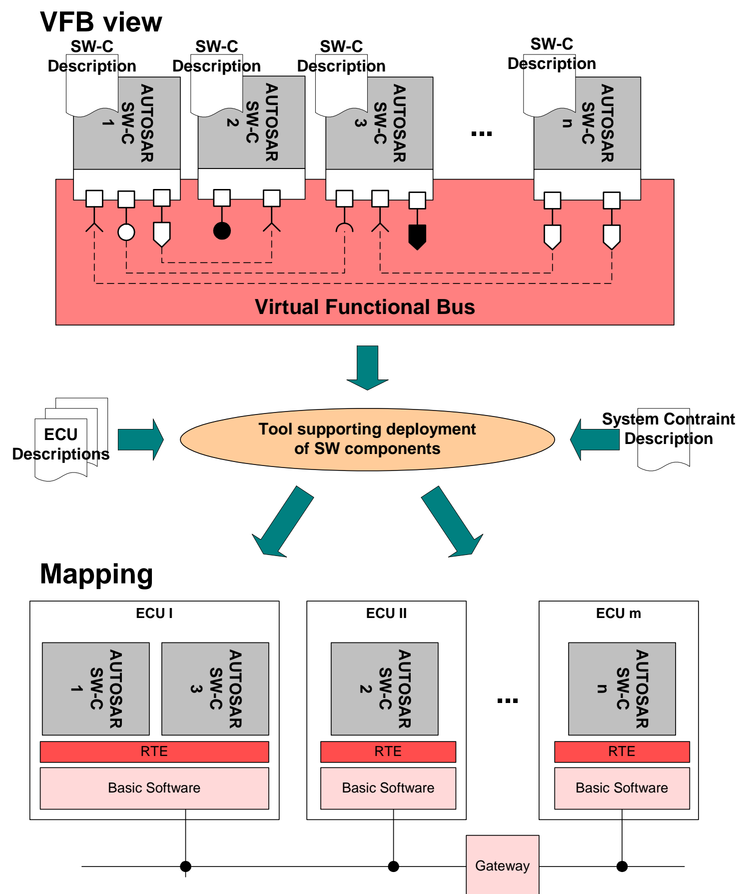


Figure 2: Basic AUTOSAR approach.

Figure 2 shows a very condensed view of the AUTOSAR approach. The basic concepts, which will be further explained throughout this document, are:

AUTOSAR SW-C
The AUTOSAR Software Components (AUTOSAR SW-C) encapsulate an application which runs on the AUTOSAR infrastructure. The AUTOSAR Software Components have well-defined interfaces, which are described and standardized within AUTOSAR. Chapter 2.2 defines the component concept in more detail.

SW-C Description
For the interfaces as well as other aspects needed for the integration of the AUTOSAR Software Components, AUTOSAR provides a standard description format, i.e. the Software Component Description (SW-C Description).

Virtual Functional Bus (VFB)
The VFB is the sum of all communication mechanisms (plus some interfaces to the basic software) provided by AUTOSAR on an abstract, i.e. technology independent, level. When the connections between AUTOSAR Software Components for a concrete system are defined, the VFB will allow a virtual integration of them in a quite early development phase.

System Constraint and ECU Descriptions
In order to integrate AUTOSAR Software Components into a network of ECUs, AUTOSAR provides description formats for the complete system as well as for the resources and configuration of the single ECUs. These descriptions are kept independent of the Software Component Descriptions.

Mapping on ECUs
AUTOSAR defines the methodology and tool support needed to bring the information of the various description elements together in order to build a concrete system of ECUs. This includes especially the configuration and generation of the Runtime Environment and the Basic Software on each ECU.

Runtime Environment (RTE)
From the viewpoint of the AUTOSAR Software Component, the RTE implements the VFB functionality on a specific ECU. The RTE can however delegate this task to the Basic Software as far as possible.

Basic Software
The Basic Software provides the infrastructural functionality on an ECU.

2.2 Components

2.2.1 The AUTOSAR Software Component

A fundamental design concept of AUTOSAR is the separation between:

- application and
- infrastructure.

An application in AUTOSAR consists of interconnected "AUTOSAR Software Components".

- AUTOSAR Confidential -

Figure 3 shows an application consisting of three AUTOSAR Software Components which are interconnected by several "connectors".

Each AUTOSAR Software Component encapsulates part of the functionality of the application. AUTOSAR does not prescribe how large the AUTOSAR Software Components are. Depending on the requirements of the application domain an AUTOSAR Software Component might be a small, reusable piece of functionality (such as a filter) or a larger block encapsulating an entire automotive functionality.

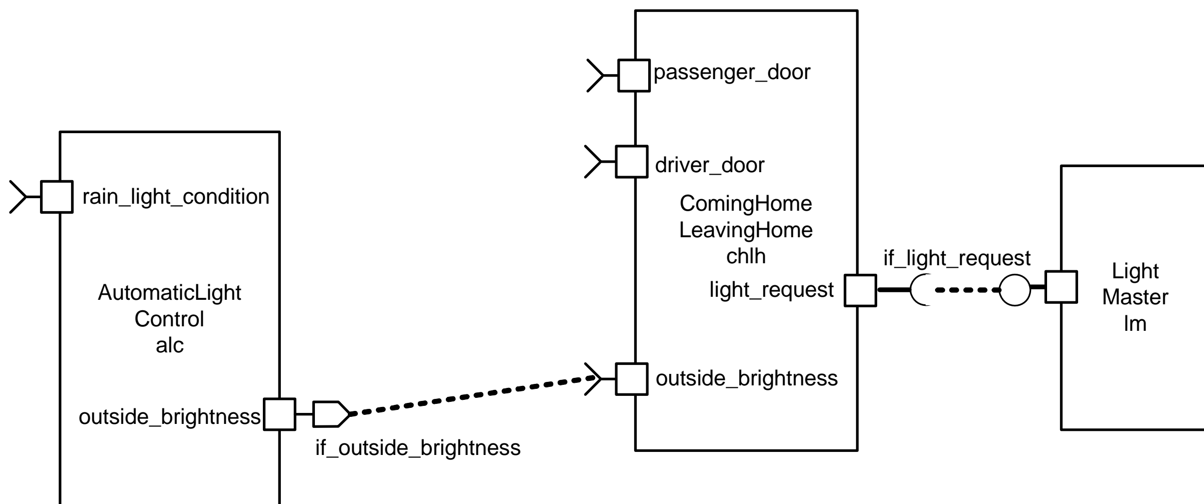


Figure 3: Example of interconnected AUTOSAR Software Components

2.2.2 The AUTOSAR Software Component is an "Atomic Software Component"

The AUTOSAR Software Component is a so-called "*Atomic* Software Component". *Atomic* here means that the AUTOSAR Software Component cannot be distributed over several AUTOSAR ECUs. Each instance of an AUTOSAR Software Component that should be present in a vehicle is assigned to one ECU.

2.2.3 Implementing and shipping an AUTOSAR Software Component

AUTOSAR does not prescribe HOW an AUTOSAR Software Component should be implemented (for example, whether the component is generated from a model or written "by hand"). Rather AUTOSAR prescribes everything that is needed to allow several AUTOSAR Software Components to be integrated correctly in an infrastructure consisting of networked ECUs.

A shipment of an AUTOSAR Software Component therefore consists of

- a complete and formal *Software Component Description* which specifies how the infrastructure must be configured for the component (see [SWCTempl]), and

- an implementation of the component, which could be provided as "object code" or "source code"[1].

2.2.4 The AUTOSAR Software Component Description

The AUTOSAR Software Component Description contains

- the operations and data elements that the software component provides and requires (this is described using the AUTOSAR PortInterface concept),
- the requirements that software component has on the infrastructure,
- the resources needed by the software component (memory, CPU-time, etc.), and
- information regarding the specific implementation of the software component.

The structure and format of this AUTOSAR Software Component description is called the “software component template”.

2.2.5 The AUTOSAR Software Component implementation is independent from the infrastructure

An implementation of an AUTOSAR Software Component (in source code[2]) fundamentally is independent from

- the type of microcontroller of the ECU on which the AUTOSAR Software Component is mapped[3],
- the type of ECU on which the AUTOSAR Software Component is mapped (The AUTOSAR infrastructure takes care of providing the software component with a standardized view on the ECU hardware – such as the ECU input/output periphery.),
- the location of the other AUTOSAR Software Components with which the software component interacts (The software component description precisely describes the data or services that the component provides or requires. The component does not need to know if these services or data elements are provided from components on the same ECU or are coming from components that run on a different ECU[4]. The implementation of the software component therefore also is network technology independent.), and
- the number of times a software component is instantiated in a system or within one ECU[5].

[1] There are important trade-offs to be made in deciding whether an AUTOSAR Software Component is shipped as object code or source code. AUTOSAR fundamentally allows both integration approaches.
[2] When an AUTOSAR Software Component is shipped as object-code, this object-code is of course highly dependent on the ECU architecture.
[3] In exceptional optimization cases, the implementation might contain dependencies on a specific microcontroller (for example: in-line assembly code); such limitations to the mapping of the software component are described in the Software Component Description.
[4] This flexibility does NOT imply that an *arbitrary* distribution of software components over ECUs is possible. The AUTOSAR Software Component Descriptions contain requirements on the performance of the connectors between the software components which might force closely interacting components to be mapped on the same ECU. System constraints related to issues of security or safety might also reduce the freedom in mapping components on ECUs.
[5] There might also be exceptions – described in the Software Component Descriptions.

2.2.6 Sensor/Actuator Software Components

Sensor/Actuator Software Components are special AUTOSAR Software Components which encapsulate the dependencies of the application on specific sensors or actuators.

Figure 4 shows the typical conversion process from physical signals to software signals (e.g. car velocity) and back (e.g. car light). As described previously, the AUTOSAR infrastructure takes care of hiding the specifics of the microcontroller (this is done in the MCAL, the microcontroller abstraction layer, which is part of the AUTOSAR infrastructure running on the ECU) and the ECU electronics (this is handled by the ECU-Abstraction which is also part of the AUTOSAR Basic Software).

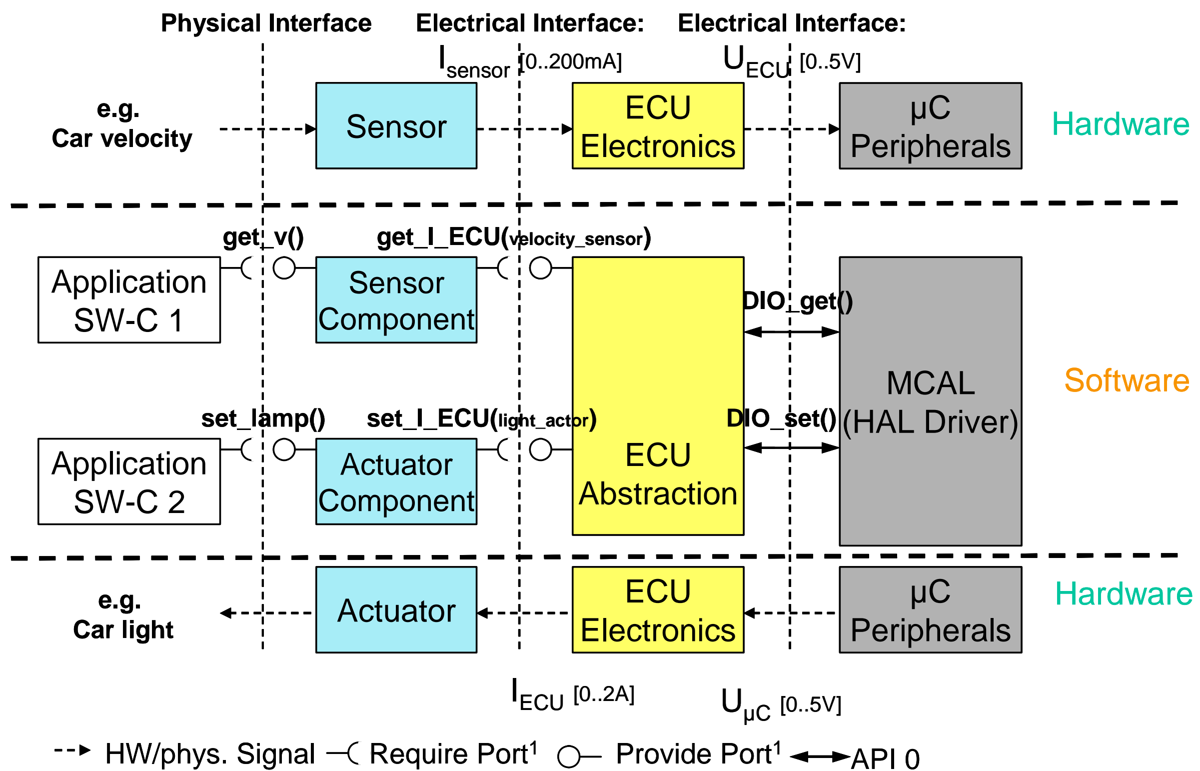


Figure 4: Hardware Interaction

The AUTOSAR infrastructure does NOT hide the specifics of sensors and actuators. The dependencies on a specific sensor and/or actuator are dealt with in "Sensor/Actuator Software Component", which is a special kind of "AUTOSAR Software Component". Such a component is independent of the ECU on which it is mapped but is dependent on a specific sensor and/or actuator for which it is designed. For example, a "Sensor Component" typically inputs a software representation of the electrical signal present at an input-pin of the ECU (e.g. a current produced by the sensor) and outputs the physical quantity measured by the sensor (e.g. the current car speed). Typically, because of performance issues, such components will need to run on the ECU to which the sensor/actuator is physically connected.

2.2.7 The generic "Component" concept

This section has so far clarified the "AUTOSAR Software Component" concept and the "Sensor/Actuator Software Component" as a special case of an AUTOSAR Software Component.

AUTOSAR uses the "component" concept in a more general and powerful way.
When modeling a system with AUTOSAR, a logical interconnection of components can be packaged as a component. Such a component is called a "composition". In contrast to the Atomic Software Components, the components inside a composition can be distributed over several ECUs.

In addition to the "composition" and the "AUTOSAR Software Component", the following entities in an AUTOSAR system are also considered "components": the "ECU Abstraction", the "Complex Device Driver" and the "AUTOSAR Services". These concepts will be explained in later sections.

2.2.8 Summary

This section has explained the AUTOSAR "Component" concept. The "AUTOSAR Software Component" is an atomic piece of software that implements part of an application, is independent of the infrastructure, and can be mapped on an ECU. The "Sensor/Actuator Software Component" is a special kind of AUTOSAR Software Component which encapsulates the dependencies on specific sensors and/or actuators.

AUTOSAR also allows the usage of the "Component" concept in a more generic way. A component can also be a logical interconnection of other components (called a "composition") which can be distributed over several ECUs.

2.3 VFB

2.3.1 General

In order to fulfill the goal of relocatability, AUTOSAR Software Components are implemented independently from the underlying hardware. The independence is achieved by providing the virtual functional bus as a means for a virtual hardware and mapping independent system integration. This enables a virtual integration of AUTOSAR Software Components so that parts of the integration process of automotive software can be done in much earlier design phases compared to today's development processes.

2.3.2 VFB Context

The virtual functional bus is the abstraction of the AUTOSAR Software Components interconnections of the entire vehicle. The communication between different software components and between software components and its environment (e.g. hardware

driver, OS, services, etc.) can be specified independently of any underlying hardware (e.g. communication system). The functionality of the VFB provided by communication patterns is defined in section 2.3.3.

From the VFB view ports of AUTOSAR Software Components, Complex Device Drivers, the ECU Abstraction and AUTOSAR Services can be connected. Complex Device Drivers, the ECU Abstraction and AUTOSAR Services are part of the Basic Software (Figure 5). Whereas the AUTOSAR Service Interfaces are standardized, the Complex Device Drivers and the ECU Abstraction are ECU specific.

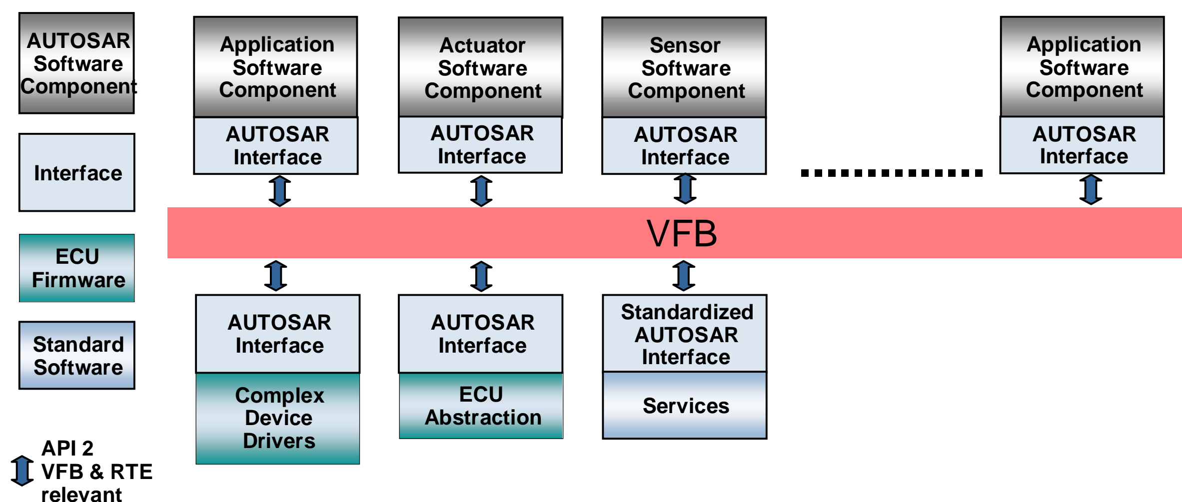


Figure 5: Representations of Atomic Software Components and AUTOSAR Services connected to the Virtual Functional Bus (see definition of Standard Software [Glossary]).

2.3.3 Communication mechanisms

2.3.3.1 Components, Ports and AUTOSAR Interfaces

The central structural element in AUTOSAR is the component, see section 2.2. A component has well-defined ports, through which the component can interact with other components. A port always belongs to exactly one component and represents a point of interaction between a component and other components. There could be several instances[6] of the same component in a car.

To define the services or data that are provided on or required by a port of a component, the AUTOSAR Interface concept is introduced. The AUTOSAR Interface can either be a Client-Server Interface (defining a set of operations that can be invoked) or a Sender-Receiver Interface, which allows the usage of data-oriented communication mechanisms over the VFB.

A port is either a PPort or an RPort. A PPort provides an AUTOSAR Interface. An RPort requires such an interface. When a PPort of a component *provides* an

[6] Dynamic instantiation at runtime is not scope of the present release of AUTOSAR.

interface, the component to which the port belongs provides an implementation of the operations defined in the Client-Server Interface respectively generates the data described in a data-oriented Sender-Receiver Interface.

When an RPort of a component *requires* an AUTOSAR Interface, the component can either invoke the operations when the interface is a Client-Server Interface or alternatively read the data elements described in the Sender-Receiver Interface.

2.3.3.2 AUTOSAR Communication Patterns

Both widely used elementary communication patterns Client-Server and Sender-Receiver[7] are supported by AUTOSAR. The specification what information sender-receiver communication transports and which services with which arguments can be called by client-server communication is done by interfaces. For a formal description of interfaces see software component template. There, also data types that can be used, interface compatibility, etc. are defined.

The detailed behavior of a basic communication pattern is specified by attributes. With those attributes e.g. the length of data queues and the behavior of receivers (blocking, non-blocking, etc.) and senders (send cyclic, etc.) can be defined.

2.3.3.2.1 Client-Server Communication

A widely used communication pattern in distributed systems is the client-server pattern, in which the server is a provider of a service[8] and the client is a user of a service.

The client initiates the communication, requesting that the server performs a service, transferring a parameter set if necessary. The server waits for incoming communication requests from a client, performs the requested service and dispatches a response to the client's request. So, the direction of initiation is used to categorize whether an AUTOSAR Software Component is a client or a server. A single component can be both a client and a server depending on the software realization.

The client can be blocked (synchronous communication) respectively non-blocked (asynchronous communication) after the service request is initiated until the response of the server is received. Figure 6 gives an example how client-server communication for a composition of three software components and two connections is modeled in the VFB view.

[7] In the context of AUTOSAR, sending, receiving and distributing of events is seen as part of the sender-receiver communication pattern.

[8] Service in this chapter is a functionality which is offered by a certain AUTOSAR Software Component (the server), and which can be used by other AUTOSAR Software Components (the clients). It is not to be mixed up with an AUTOSAR Service.

- AUTOSAR Confidential -

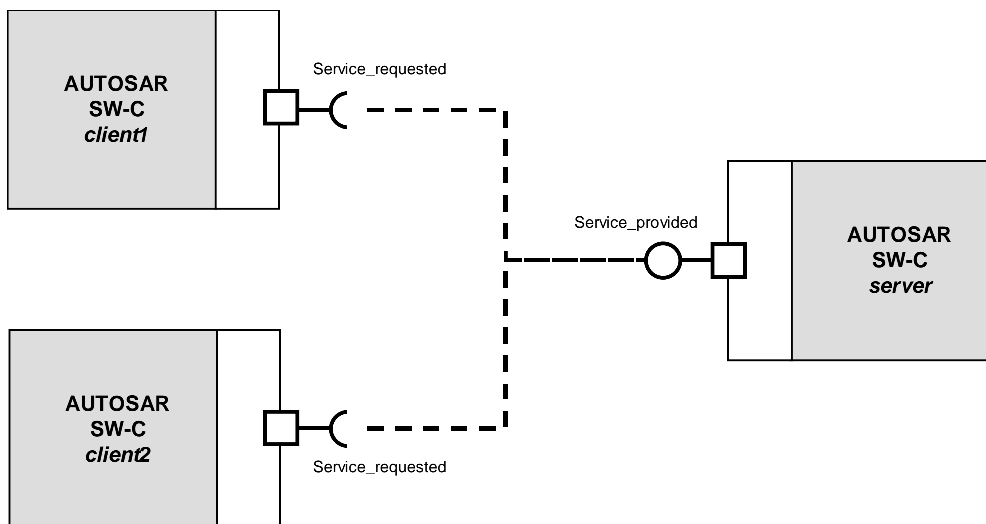


Figure 6: Client-server communication in the VFB view

2.3.3.2.2 Sender-Receiver Communication

The sender-receiver pattern gives solution to the asynchronous distribution of information where a sender distributes information to one or several receivers.

The sender is not blocked (asynchronous communication) and neither expects nor gets a response from the receivers (data or control flow), i.e. the sender just provides the information and the receivers decides autonomously when and how to use this information. It is the responsibility of the communication infrastructure to distribute the information.

The sender component does not know the identity or the number of receivers to support transferability and exchange of AUTOSAR Software Components. Figure 7 gives an example how sender-receiver communication is modeled in the AUTOSAR VFB view.

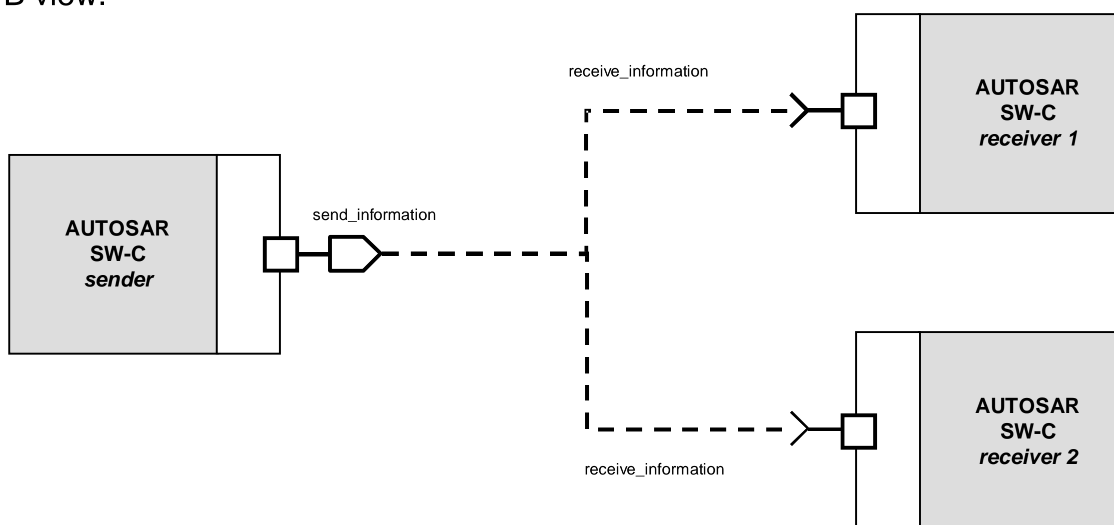


Figure 7: Data distribution by asynchronous non-blocking communication in the VFB view

2.4 AUTOSAR ECU Software Architecture

2.4.1 Overview

Figure 8 shows the structure of the software for an ECU. The layers and its main elements are described below.

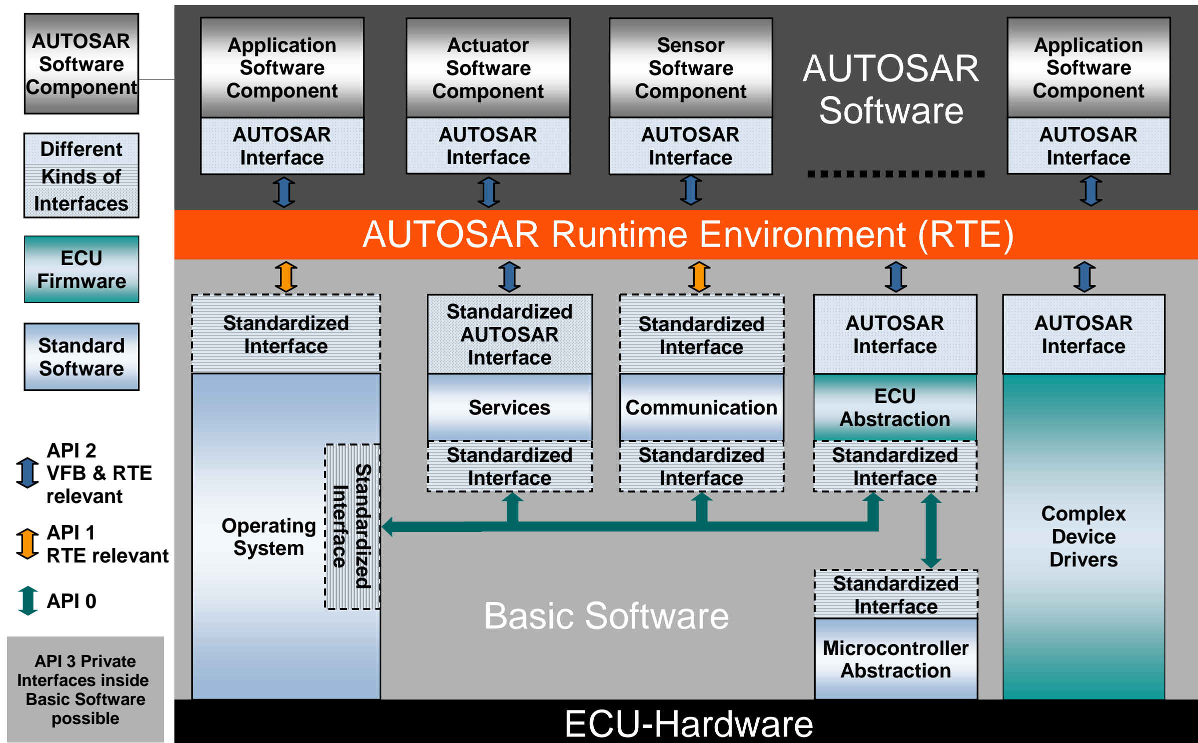


Figure 8 Schematic view of AUTOSAR ECU software architecture

2.4.2 AUTOSAR Software

The AUTOSAR Software (the layer above AUTOSAR Runtime Environment) consists of AUTOSAR Software Components that are mapped on the ECU.

All interaction between AUTOSAR Software Components and Atomic Software Components[9] is routed through the AUTOSAR Runtime Environment. The AUTOSAR Interface assures the connectivity of software elements surrounding the AUTOSAR Runtime Environment.

[9] Atomic Software Components can be: other AUTOSAR Software Components, AUTOSAR Services, the ECU Abstraction, or Complex Device Drivers (see section 2.2.7).

2.4.3 AUTOSAR Runtime Environment

At system design level (i.e. then drafting a logical view of the entire system irrespective of hardware) the AUTOSAR Runtime Environment (RTE) acts as a communication center for inter- and intra-ECU information exchange. The RTE provides a communication abstraction to AUTOSAR Software Components attached to it by providing the same interface and services whether inter-ECU communication channels are used (such as CAN, LIN, Flexray, MOST, etc.) or communication stays intra-ECU.

As the communication requirements of the software components running on top of the RTE are application dependent, the RTE needs to be tailored. It is therefore very likely, that the main parts of RTE will be generated to provide desired communication services while still being resource-efficient. In addition some parts of the RTE software may be tailored via configuration. Thus, the resulting RTE will likely differ between one ECU and another.

2.4.4 AUTOSAR Basic Software

Basic Software is the standardized software layer, which provides services to the AUTOSAR Software Components and is necessary to run the functional part of the software. It does not fulfill any functional job itself and is situated below the AUTOSAR Runtime Environment. The Basic Software contains standardized and ECU specific components. The earlier include:

Services
System services cover e.g. diagnostic protocols; NVRAM, flash and memory management.

Communication
This topic addresses the communication framework (e.g. CAN, LIN, FlexRay...), the I/O management, and the network management.

Operating System
As AUTOSAR aims at an architecture that is common for all vehicle domains it will specify the requirements for an AUTOSAR Operating System. The following requirements shall be seen as examples of such: The OS

- is configured and scaled statically,
- is amenable to reasoning of real-time performance,
- provides a priority-based scheduling,
- provides protective functions at run-time, and
- is hostable on low-end controllers and without external resources.

It is assumed that some domains (e.g. telematic/infotainment) will continue to use proprietary OSs. In these cases the interfaces to AUTOSAR components must still be AUTOSAR compliant. I.e. the proprietary OS must be abstracted to an AUTOSAR OS.
The standard OSEK OS (ISO 17356-3) is used as the basis for the AUTOSAR OS.

Microcontroller Abstraction

Access to the hardware is routed through the Microcontroller Abstraction layer (MCAL) to avoid direct access to microcontroller registers from higher-level software.
MCAL is a hardware specific layer that ensures a standard interface to the components of the Basic Software. It manages the microcontroller peripherals and provides the components of the Basic Software with microcontroller independent values. MCAL implements notification mechanisms to support the distribution of commands, responses and information to different processes. Among others it can include

- Digital I/O (DIO),
- Analog/Digital Converter (ADC),
- Pulse Width (De)Modulator (PWM, PWD),
- EEPROM (EEP),
- Flash (FLS),
- Watchdog Timer (WDT),
- Serial Peripheral Interface (SPI), and
- I^2C Bus (IIC).

ECU specific components are:

ECU Abstraction

The ECU Abstraction provides a software interface to the electrical values of any specific ECU in order to decouple higher-level software from all underlying hardware dependencies.

Complex Device Driver (CDD)

The CDD allows a direct access to the hardware in particular for resource critical applications.
For more detailed info see section 4.3

2.4.5 Classification of interfaces

In the Figure 8 above there exist three different types of interfaces, "AUTOSAR Interface", "Standardized AUTOSAR Interface" and "Standardized Interface". Note that the boxes define the classification of the interfaces for the different modules, i.e. the interface boxes in the Figure 8 shall not be regarded as separate modules or layers. The semantics of these classifications are as follows. For further details see the [Glossary].

AUTOSAR Interface

An "AUTOSAR Interface" defines the information exchanged between software components. This description is independent of a specific programming language, ECU or network technology.
AUTOSAR Interfaces are used in defining the ports of software-components. Through these ports software-components can communicate with each other (send or receive information or invoke services). AUTOSAR makes it possible to implement this communication between software-components either locally or via a network.

In other words: "AUTOSAR Interfaces" are to be used when it should be possible to route the information flowing through the interface through a network.

Standardized AUTOSAR Interface

A "Standardized AUTOSAR Interface" is an "AUTOSAR Interface" whose syntax and semantics are standardized in AUTOSAR. The "Standardized AUTOSAR Interfaces" are typically used to define AUTOSAR Services, which are standardized services provided by the AUTOSAR Basic Software to the software-components.

Standardized Interface

A "Standardized Interface" is an API which is standardized within AUTOSAR without using the "AUTOSAR Interface" technique. These "Standardized Interfaces" are typically defined for a specific programming language (like "C"). Because of this, "standardized interfaces" are typically used between software-modules which are always on the same ECU. When software modules communicate through a "standardized interface", it is NOT possible any more to route the communication between the software-modules through a network.

- AUTOSAR Confidential -

3 AUTOSAR Methodology

3.1 Introduction

AUTOSAR requires a common technical approach for some steps of system development. This approach is called the "AUTOSAR Methodology". This chapter describes all major steps of the development of a system with AUTOSAR: from the system-level configuration to the generation of an ECU executable.

The AUTOSAR Methodology is neither a complete process description nor a business model and "roles" and "responsibilities" are not defined in this methodology. Furthermore, it does not prescribe a precise order in which activities should be carried out. The methodology is a mere work-product flow: it defines the dependencies of activities on work-products.

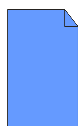
Basically this chapter is an extract of the detailed AUTOSAR Methodology [Meth].

3.1.1 Describing Notation – SPEM

AUTOSAR describes the methodology using the Software Process Engineering meta-model, or SPEM for short. SPEM is a standard [SPEM] defined by the Object Management Group (OMG) and is designed to describe software development processes.

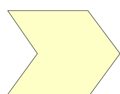
In the context of the AUTOSAR Methodology only a very small subset of SPEM is actually used. The following subsections describe the appropriate graphical notation.

Work-Product



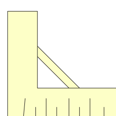
A «Work-Product» is a piece of information or physical entity produced by or used by an activity. For the AUTOSAR Methodology several specific kinds of «Work-Product» are defined, e.g. XML-Document, c-Document (for files containing sources in the language C), obj-Document (for object files), or h-Document (for files containing header files that are included in c-files).

Activity

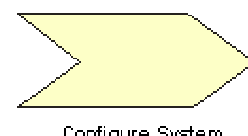


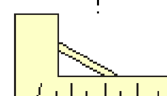
An «Activity» describes a piece of work performed by one or a group of persons: the tasks, operations, and actions that are performed by a role or with which the role may assist[10].

Guidance



«Guidance» elements are associated with activities and represent additional information or tools that are to be used to perform the activity.
The example on the right shows that the activity `Configure System` is associated with the «Guidance» `AUTOSAR System Configuration Generator`, which means the tool

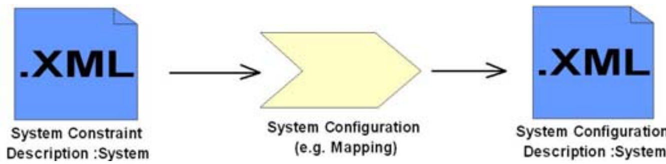




[10] Note that the AUTOSAR methodology does NOT define roles.

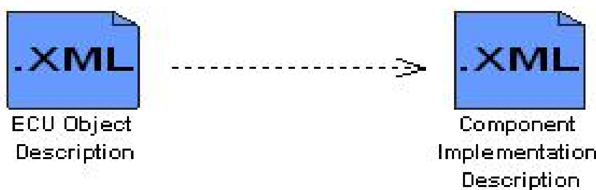
AUTOSAR System Configuration Generator is used to perform the activity.

Flow of Work-Products



The flow of work-products is graphically represented by a line with an arrowhead. It is always directed from its source to its destination and is used to identify the input and output of an activity. In this example the activity System Configuration uses the work-product System Constraint Description as input and outputs the System Configuration Description.

Dependency



A «Dependency» is a dotted line with an arrowhead that indicates that one work-product depends on another work-product. It is a unidirectional «Dependency» and the direction of the line clarifies who depends on whom. The example shows that the XML-Document ECU Object Description depends on the XML-Document Component Implementation Description. In this context the «Dependency» can also be interpreted as a reference: the XML-document ECU Object Description contains references to information contained in the XML-document Component Implementation Description.

3.2 Methodology Overview

Figure 9 shows a rough overview of the AUTOSAR Methodology. It sketches the design steps from the system-level configuration to the generation of an ECU executable.

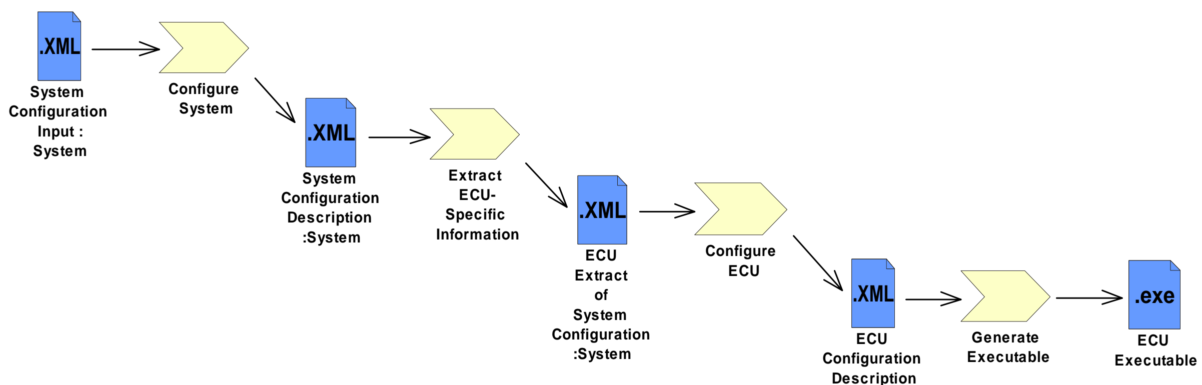


Figure 9: Overview AUTOSAR Methodology

Firstly the System Configuration Input has to be defined. This is a system design or architecture task. The software components and the hardware have to be selected, and overall system constraints have to be identified. AUTOSAR intends to

ease the formal description of these initial system design decisions via the information exchange format and the use of templates. So defining the `System Configuration Input` means filling out or editing the appropriate templates.

This addresses information of the following packages:

- Software Components: each software component requires a description of the software API e.g. data types, ports, interfaces, etc. [SWCTempl].
- ECU Resources: each ECU requires specifications regarding e.g. the processor unit, memory, peripherals, sensors and actuators. [ECURes].
- System Constraints: this contains e.g. constraints regarding the bus signals, topology and mapping of belonging together software components. [SysTempl].

It depends on the use case whether a template has to be filled out from scratch or whether a reuse – probably with some editing – is possible. Basically the AUTOSAR methodology allows for a high degree of reuse in this context. In any case this editing is assumed to be supported by editing tools.

The activity of the `Configure System` mainly maps the software components to the ECUs with regard to resources and timing requirements. The output of this activity is the `System Configuration Description`. This description includes all system information (e.g. bus mapping, topology) and the mapping of which software component is located on which ECU.

The further steps (as depicted in Figure 9) have to be performed for each ECU in the system. The step `Extract ECU-Specific Information` extracts the information from the `System Configuration Description` needed for a specific ECU. This is then placed in the `ECU Extract of System Configuration`.

The activity `Configure ECU` adds all necessary information for implementation like task scheduling, necessary Basic Software modules, configuration of the Basic Software, assignment of runnable entities to tasks, etc. The result of the activity `Configure ECU` is included in the `ECU Configuration Description`, which collects all information that is local to a specific ECU. The runnable software to this specific ECU can be built from this information.

In the last step `Build Executable` an executable is generated based on the configuration of the ECU described in the `ECU Configuration Description`. This step typically involves generating code (e.g. for the RTE and the Basic Software), compiling code (compiling generated code or compiling software-components available as source-code) and linking everything together into an executable.

Parallel to these briefly described steps of the methodology there are several steps required to integrate the software components into the whole system, e.g. generating the components API, and implementing the components functionality. For clarity they are not depicted in Figure 9. Nevertheless the implementation of a software

component is more or less independent from ECU configuration. This is a key feature of the AUTOSAR methodology.

The following sections describe the various parts of the AUTOSAR methodology in more detail. To reflect the parallelism of the several activities we don't follow the simplified sequential structure of Figure 9, but we distinguish parts of the methodology that are necessary at least once per system, per ECU, and per component.

3.3 System Configuration

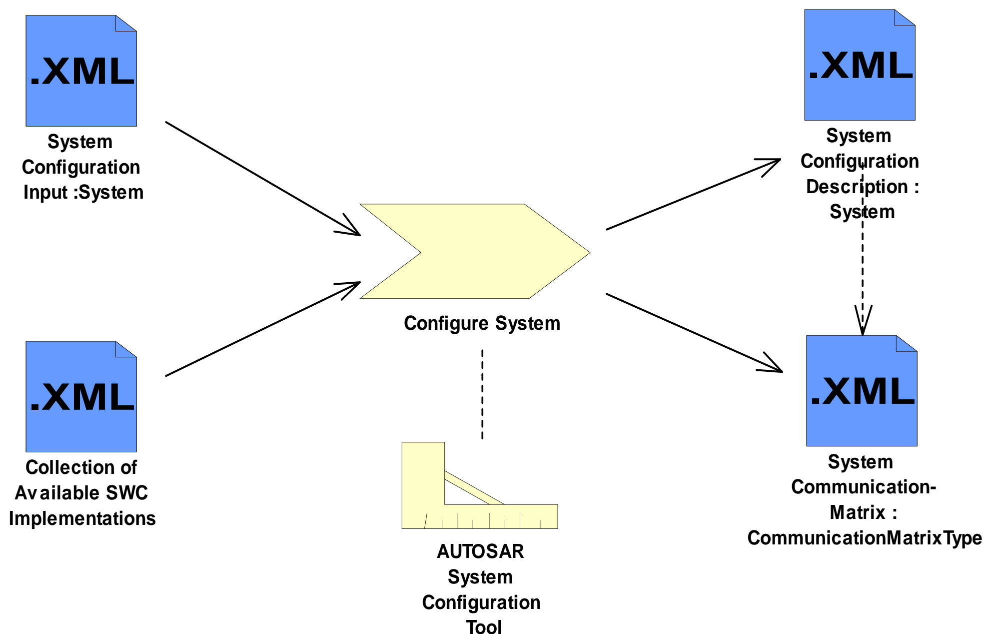


Figure 10: System configuration overview

The activity `Configure System` takes engineering decisions at system level. These decisions are based on the `System Configuration Input` and the `Collection of Available SWC Implementations`[11], and the `AUTOSAR System Configuration Tool` supports the decisions. Output of this activity is a complete `System Configuration Description` and an associated `System Communication-Matrix`.

One of the most important decisions that are taken during the `Configure System` activity is the “mapping”: for each component a decision must be taken on what ECU in the component runs. As part of the mapping decisions, the `Configure System` activity might decide on the use of specific implementations for certain software components. These implementations are chosen from the `Collection of Available SWC Implementations`. Choosing an implementation at system-

[11] This collection describes possible implementation variants of the components. Hence it should not be mixed up with the Software Component Descriptions.

level might enable a more precise analysis of required and provided resources and allows the system-designer to influence more precisely what happens inside the ECU. In many cases however, such choices are not made at system-level, but are left over to the configuration of the specific ECU.

The `System Configuration Input` includes or references various constraints that should be considered during the `Configure System` activity. These constraints can force or forbid certain components to be mapped to certain ECUs or requires certain implementations to be used for components. In addition, these constraints can contain resource estimations describing the net availability of resources on ECUs and thereby limiting the possible mappings.

Finally, an important aspect of the activity is the design of the `System Communication-Matrix`. This `System Communication-Matrix` completely describes the frames running on the networks described in the topology and the contents and timing of those frames.

The activity `Configure System` contains complex algorithms and/or engineering work. There is a strong need for experience in system architecture to map all the software components to the ECUs. The tool `AUTOSAR System Configuration Tool` supports the configuration. It should help to take the aforementioned engineering decisions (e.g. via clear graphical representation), to store the results, and to change them later if necessary.

3.4 ECU Design and Configuration Methodology

Figure 11 shows an overview about the design steps to build an ECU with the AUTOSAR technology. That means these steps have to be performed for each ECU in the system.

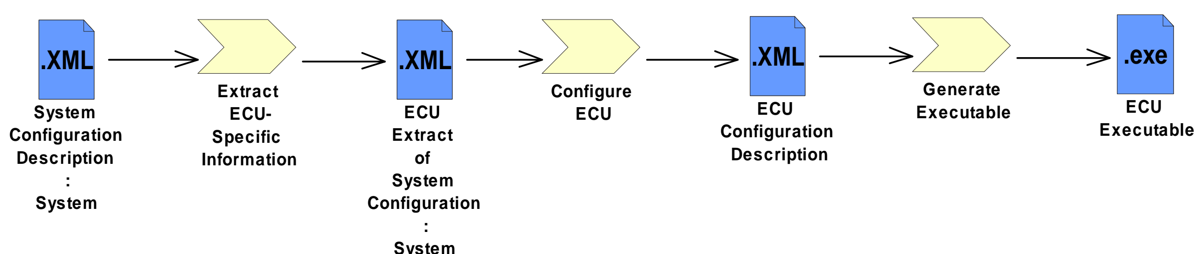


Figure 11: Overview about ECU part of the AUTOSAR methodology

The input to this phase is the `System Configuration Description`, which is created during the system configuration phase. The output of this phase is the executable ECU software[12]. The major activities in this phase are the extraction of

[12] The `ECU Executable` described in the methodology is not always the executable which will be used finally in the production line. In practice the executable likely will change during development, e.g. due to optimizations or to consider calibration.

- AUTOSAR Confidential -

ECU-specific information from the System Configuration Description, the configuration of the ECU and the generation of the executable ECU software. The following sections will describe these activities in more detail.

3.4.1 Extract ECU-Specific Information

The tool AUTOSAR ECU Configuration Extractor extracts the information from the System Configuration Description needed for a specific ECU. This is a one to one copy of all elements of the System Configuration Description that are appointed to this specific ECU. The result is the ECU Extract of System Configuration. Hence Extract ECU-Specific Information is a step that can be completely automated.

3.4.2 Configure ECU

The ECU configuration mainly deals with the configuration of the RTE and the Basic Software modules. As depicted in Figure 12, the configuration is based on the information that is available from the ECU Extract of System Configuration, Collection of Available SWC Implementations, and BSW Module Description. The latter contains the Vendor Specific ECU Configuration Parameter Definition which defines all possible configuration parameters and their structure. This is necessary because the output ECU Configuration Description has a flexible structure which does not define a fixed number of configuration parameters a priori. The BSW Module Description is assumed to consist of single descriptions delivered together with the appropriate used BSW module. The ECU Configuration Description needs to reference several inputs for configuration technicalities.

An important part of the configuration: the detailed scheduling information or the configuration data for e.g. the communication module, the operating system, or AUTOSAR services have to be defined in this activity. Moreover at the latest here an implementation is selected for each Atomic Software Component.

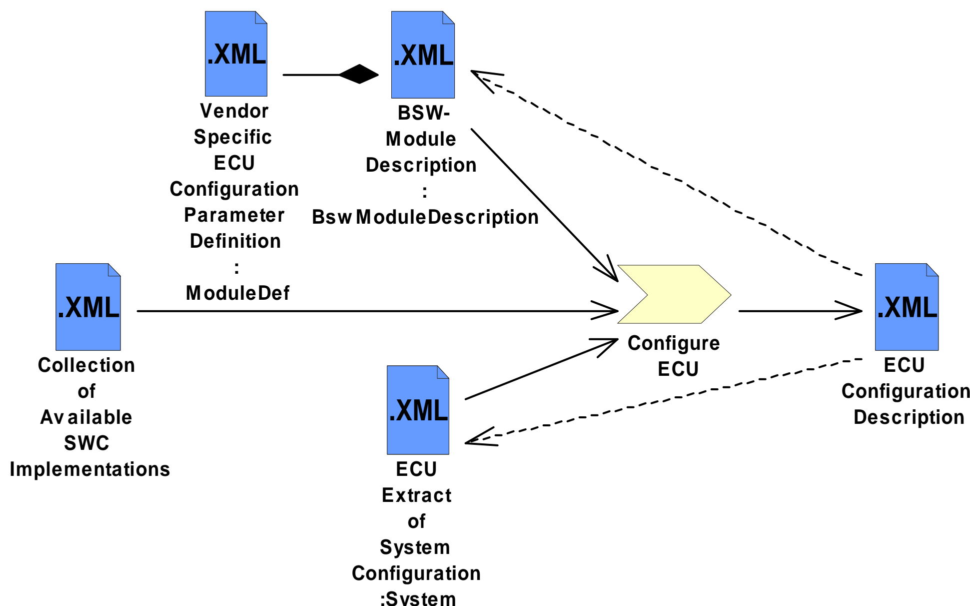


Figure 12: Overview about ECU configuration

In contrast to the extraction of ECU-specific information, the configuration activity is a non-trivial design step, which requires complex design algorithms and engineering knowledge.

3.4.3 Generate Executable

After the ECU has been configured, software for several parts of the ECU can be generated. This refers to the Basic Software (e.g. communication module, operating system, etc.), and the RTE. All these generation steps are assumed to be tool supported with a high degree of automation.

The remaining steps to generate the executable ECU code resemble today's development practice. This is mainly a linking of all components. But it is important to note that there are additional steps involved. E.g. information from the `ECU Configuration Description` might be used to generate specially configured executable software. Furthermore it has to be synchronized with the component implementation (as described in section 3.5).

3.5 Component Implementation

This section describes the workflow and the necessary activities in terms of the AUTOSAR methodology to start the development of an application software component and to integrate it later into the system. The workflow shall allow a more or less independent development of the software component's core functionality. These activities have to be performed for every application software component.

Figure 13 depicts the per component part of the AUTOSAR methodology. For clarity and easier understandability, this addresses only a basic workflow without any ECU-

configuration-specific optimizations. However, it is assumed that such optimizations will be rather the default case in practice.

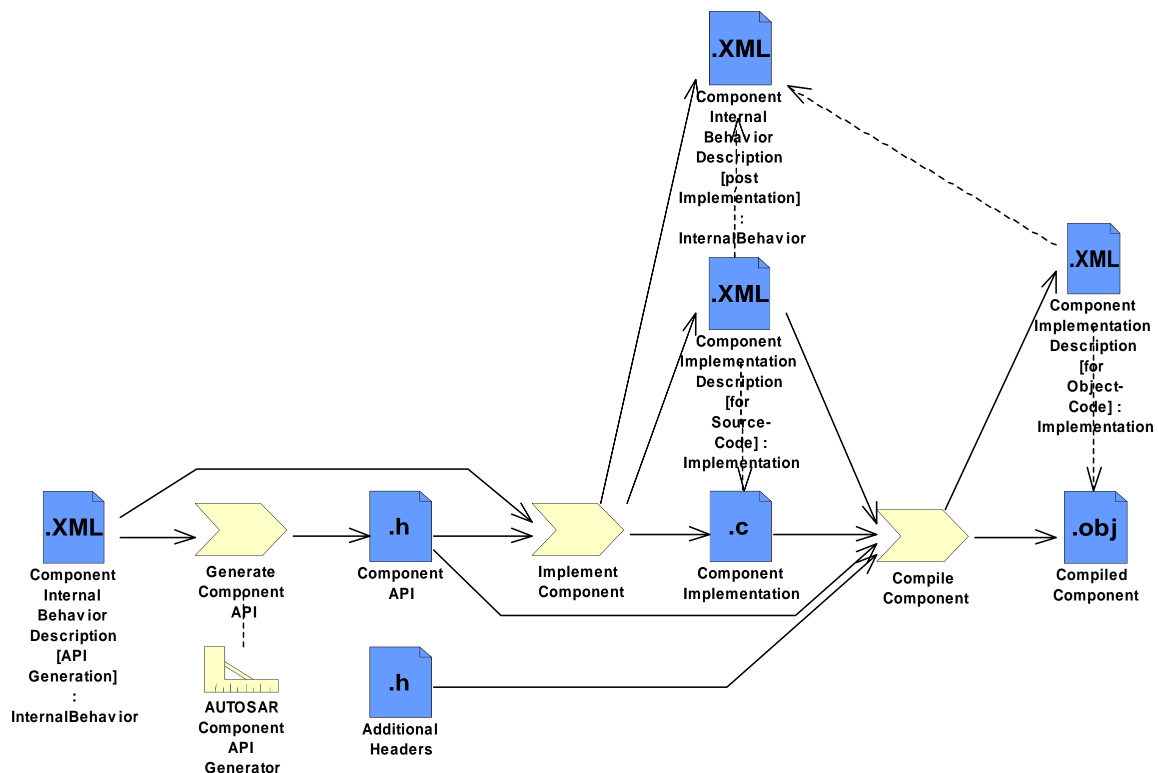


Figure 13: Component implementation part of the AUTOSAR Methodology

The main workflow in Figure 13 runs from the left to the right. The initial work in this context starts with providing the necessary parts of the software component description [SWCTempl]. That means at least the `Component Internal Behavior Description` as part of the software component related templates has to be filled out. The internal behavior describes the scheduling relevant aspects of a component, i.e. the runnable entities and the events they respond to. Furthermore, the behavior specifies how a component (or more precisely which runnable) responds to events like received data elements. However, it does not describe the detailed functional behavior of the component.

Afterwards `Generate Component API` has to be performed. This is a tool-based activity. The `AUTOSAR Component API Generator`[13] reads the `Component Internal Behavior Description` of the appropriate software component and generates the `Component API` accordingly. The `Component API` contains all header declarations for the RTE communication. There isn't any further engineering or configuration expected in this activity.

[13] The AUTOSAR Component API Generator does not have to be a stand-alone tool. The functionality propably is included in the `AUTOSAR RTE Generator`.

- AUTOSAR Confidential -

Next `Implement Component` means the functional development of the component. With the `Component Internal Behavior Description` and the `Component API` a software developer can implement (i.e. developing, programming, testing) the component vastly independent from the other system design. This implementation basically is outside the scope of AUTOSAR. The results of the implementation will be the `Component Implementation` (i.e. typically the C-sources), a refined `Component Internal Behavior Description`, which contains now additional implementation specific information, and a `Component Implementation Description`, which contains information about the further build process (e.g. compiler settings, optimizations, etc.).

The following activities address the integration of the previously provided component. `Compile Component` uses the `Component Implementation Description` for compiling the `Component Implementation` together with the `Component API` and the `Additional Headers`. This yields the `Compiled Component` and again a refined `Component Implementation Description`. This contains additional new build process information (mainly linker settings) and the entry points.

4 Detailed ECU Architecture

4.1 Layered Software Architecture

4.1.1 The Layered Architecture

In chapter 2.4 the ECU software architecture has been introduced. Based on this architecture a layered architecture has been developed within AUTOSAR to enable a clear and structured interface definition and a well defined abstraction of the hardware. In this chapter the layer concept will be introduced.

The layered software architecture is structured in 5 layers plus the possibility to implement Complex Device Drivers which cannot be mapped into a single layer (see Figure 14).

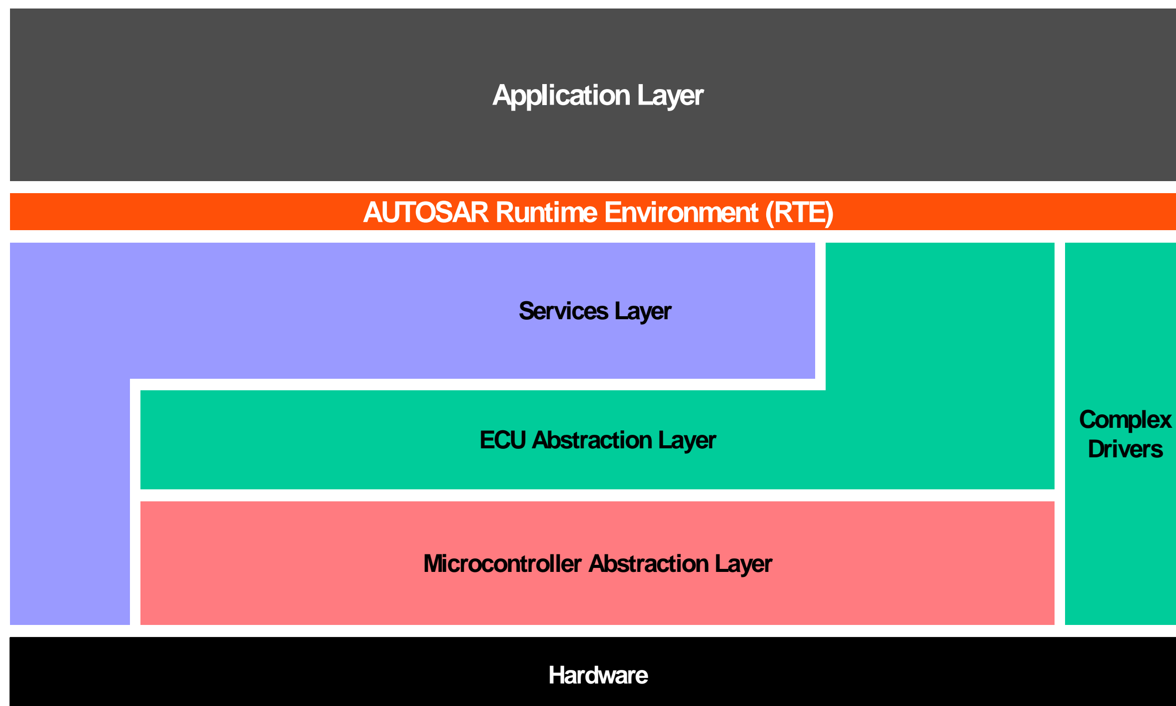


Figure 14: Overall layered architecture

Application Layer:	
Mapping from ECU SW-Architecture:	The application layer of the ECU software architecture has been left untouched. A layering within the application layer is not the focus of AUTOSAR.
Purpose:	In the application layer all AUTOSAR Software Components, the Application Software Components as well as the Sensor/Actuator Software Components, are located.
Responsibility:	The software components of the application layer shall communicate internally via RTE and access all ECU resources via RTE.
Properties:	Implementation: µC independent, ECU independent, HW independent Interface: standardizable, syntactics pre-defined by AUTOSAR

AUTOSAR Runtime Environment (RTE):	
Mapping from ECU SW-Architecture:	The RTE has also not been changed compared to the ECU software architecture.
Purpose:	The purpose of the RTE is to enable the implementation of AUTOSAR Software Components independent from the mapping to a specific hardware/ECU
Responsibility:	The RTE is a hardware independent layer providing communication services for the application containing Application Software Components and Sensor/Actuator Software Components. Above the RTE the software architecture style changes from „layered“ to „component style“. The AUTOSAR Software Components communicate with other components (inter and/or intra ECU) via the RTE.
Properties:	Implementation: ECU and application specific (generated individually for each ECU) Upper Interface: completely ECU independent

Services Layer:	
Mapping from ECU SW-Architecture:	The service layer is a layer which consists of the communication, services and operating system blocks shown in the ECU software architecture.
Purpose:	It provides basic services for application and Basic Software modules to abstract the microcontroller as well as the ECU hardware from the layers above.
Responsibility:	It is the highest layer of the Basic Software which also applies for its relevance for the application software: while access to I/O signals is covered by the ECU abstraction layer, the services layer offers • Operating system services • Vehicle network communication and management services • Memory services (NVRAM management) • Diagnosis Services (including diagnosis communication interface and error memory) • ECU state management
Properties:	Implementation: partly µC, ECU hardware and application specific Upper Interface: µC and ECU hardware independent

- AUTOSAR Confidential -

ECU Abstraction Layer:	
Mapping from ECU SW-Architecture:	The ECU abstraction layer is the most obvious change of the layered software architecture compared to the ECU software architecture. Nearly all standardized interfaces within the Basic Software shown in the ECU software architecture have been mapped into that layer.
Purpose:	To abstract the ECU layout from the above layer.
Responsibility:	It contains handlers, which are software modules which abstract how peripherals are connected to the CPU (e.g. on chip, implemented in ASIC and connected via SPI, …).
Properties:	Implementation: µC independent, ECU hardware dependent Upper Interface: µC and ECU hardware independent, dependent on signal type

Complex Drivers:	
Mapping from ECU SW-Architecture:	The Complex Device Driver could not be mapped to a specific layer and has therefore been taken over from the ECU software architecture with no change.
Purpose:	The Complex Device Drivers fulfill the special functional and timing requirements for handling complex sensors and actuators.
Responsibility:	A Complex Driver implements complex sensor evaluation and actuator control with direct access to the µC using specific interrupts and/or complex µC peripherals (like PCP, TPU), e.g. • injection control • electric valve control • incremental position detection
Properties:	Implementation: highly µC, ECU and application dependent Upper Interface: specified and implemented as AUTOSAR Interfaces.

A more detailed description of the Complex Device Driver can be found in chapter 4.3.

Microcontroller Abstraction Layer (MCAL):	
Mapping from ECU SW-Architecture:	According to the ECU software architecture the microcontroller abstraction layer has been placed between the ECU abstraction and the real hardware.
Purpose:	The MCAL abstracts the microcontroller from the above layer to make upper layers µC independent.
Responsibility:	The MCAL layer is the lowest software layer of the Basic Software. It contains drivers, which are software modules with direct access to the µC internal peripherals and memory mapped µC external devices.
Properties:	Implementation: µC dependent, Upper Interface: standardizable and µC independent

4.1.2 Refinement of the Layered Architecture

To get a more modularized view of the software architecture the layered architecture introduced in the chapter above has been further refined in the area of Basic Software. Around 80 Basic Software modules have been defined. Describing all these modules would go beyond the scope of this overview. Therefore a more abstract view will be introduced, see Figure 15.

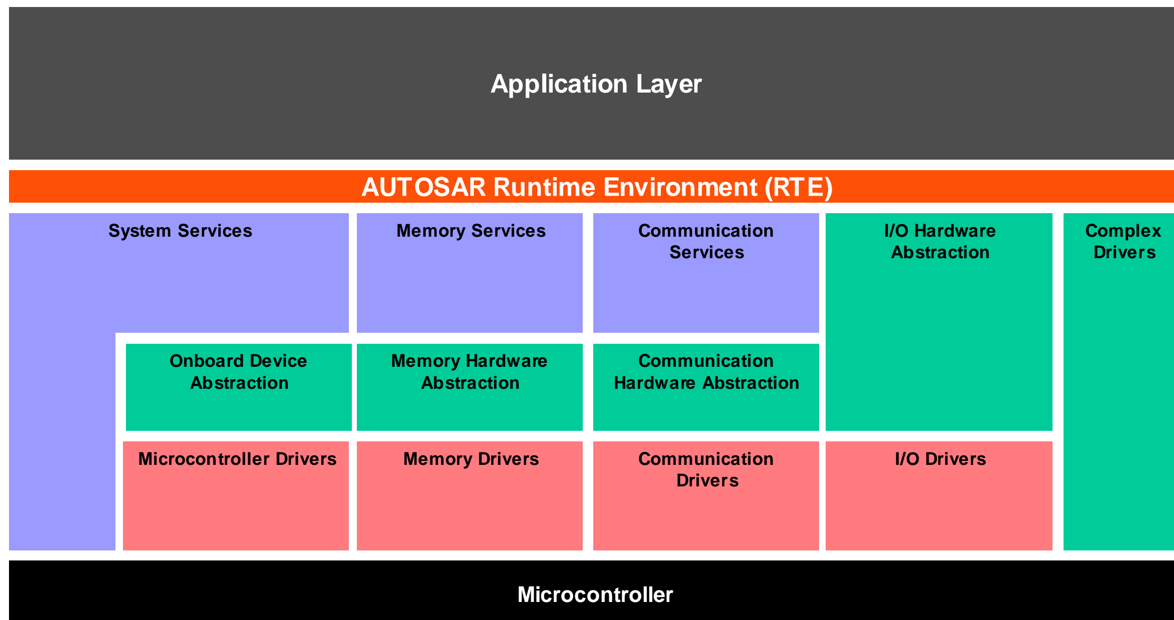


Figure 15: Refined layered software architecture

The Basic Software consists out of 11 blocks (plus Complex Device Drivers) which will now be explained.

4.1.2.1 Microcontroller Abstraction Layer

The Microcontroller abstraction layer has been subdivided into 4 parts:

1. I/O Drivers
   Drivers for analog and digital I/O (e.g. ADC, PWM, DIO).
2. Communication Drivers
   Drivers for ECU onboard (e.g. SPI, I2C) and vehicle communication (e.g. CAN). OSI-Layer: Part of Data Link Layer.
3. Memory Drivers
   Drivers for on-chip memory devices (e.g. internal Flash, internal EEPROM) and memory mapped external memory devices (e.g. external Flash).
4. Microcontroller Drivers
   Drivers for internal peripherals (e.g. Watchdog, Clock Unit) and functions with direct µC access (e.g. RAM test, Core test).

4.1.2.2 ECU Abstraction Layer

Also the ECU abstraction layer has been subdivided into 4 parts:

I/O Hardware Abstraction:
The I/O hardware abstraction is a group of modules which abstracts from the location of peripheral I/O devices (on-chip or on-board) and the ECU hardware layout (e.g. µC pin connections and signal level inversions). The I/O hardware abstraction does not abstract from the sensors/actuators!
The different I/O devices are accessed via an I/O signal interface.

The task of this group of modules is

- to represent I/O signals as they are connected to the ECU hardware (e.g. current, voltage, frequency), and
- to hide ECU hardware and layout properties from higher software layers.

Communication Hardware Abstraction

The communication hardware abstraction is a group of modules which abstracts from the location of communication controllers and the ECU hardware layout. For all communication systems a specific communication hardware abstraction is required (e.g. for LIN, CAN, MOST, FlexRay). Example: An ECU has a microcontroller with 2 internal CAN channels and an additional on-board ASIC with 4 CAN controllers. The CAN-ASIC is connected to the microcontroller via SPI.

The communication drivers are accessed via bus specific interfaces (e.g. CAN Interface). That means the access to the CAN controller should be regardless of whether it is located inside the microcontroller, externally to it, or whether it is connected via SPI.

The task of this group of modules is

- to provide equal mechanisms to access a bus channel regardless of it‘s location (on-chip / on-board).

Memory Hardware Abstraction

The memory hardware abstraction is a group of modules which abstracts from the location of peripheral memory devices (on-chip or on-board) and the ECU hardware layout. Example: on-chip EEPROM and external EEPROM devices should be accessible via an equal mechanism.

The memory drivers are accessed via memory specific interfaces (e.g. EEPROM Interface).

The task of this group of modules is

- to provide equal mechanisms to access internal (on-chip) and external (on-board) memory devices.

Onboard Device Abstraction

The onboard device abstraction contains drivers for ECU onboard devices which cannot be seen as sensors or actuators like system basic chip, external watchdog etc. Those drivers access the ECU onboard devices via the µC abstraction layer.

The task of this group of modules is

- to abstract from ECU specific onboard devices.

4.1.2.3 Service Layer

The service layer consists out of 3 different parts:

Communication Services

The communication services are a group of modules for vehicle network communication (CAN, LIN, FlexRay and MOST). They are interfacing with the communication drivers via the communication hardware abstraction.

The task of this group of modules is

- to provide a uniform interface to the vehicle network for communication between different applications,
- to provide uniform services for network management,
- to provide a uniform interface to the vehicle network for diagnostic communication, and
- to hide protocol and message properties from the application.

Memory Services
The memory services are a group of modules responsible for the management of non volatile data (read/write from different memory drivers). The NVRAM manager provides a RAM mirror as data interface to the application for fast read access.

The task of this group of modules is

- to provide non volatile data to the application in a uniform way,
- to abstract from memory locations and properties, and
- to provide mechanisms for non volatile data management like saving, loading, checksum protection and verification, reliable storage etc.

System Services
The system services are a group of modules and functions which can be used by modules of all layers. Examples are real-time operating system, error manager and library functions (like CRC, interpolation etc.). Some of these services are µC dependent (like OS), ECU hardware and/or application dependent (like ECU state manager, DCM) or hardware and µC independent.

The task of this group of modules is

- to provide basic services for application and Basic Software modules.

4.1.3 Related Documents

A detailed definition of the layered architecture can be found in [BSWArch].

4.2 The Runtime Environment (RTE)

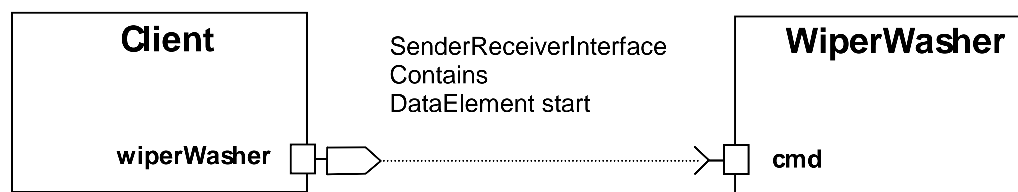
4.2.1 Overview

The AUTOSAR Runtime Environment (RTE) is the runtime representation of the Virtual Function Bus for a specific ECU. It has the responsibility to provide a uniform environment to AUTOSAR Software Components. The purpose of this uniform environment is to make the implementation of the software components independent from the communication mechanisms and channels. The RTE achieves this by mapping the communication relationships between components, that are specified in the different templates, to a specific intra-ECU communication mechanism, such as a function call, or an inter-ECU communication mechanism, such as a COM message which leads to CAN communication.

To reduce resource requirements of the RTE layer and to improve its performance some parts of the RTE will be generated. The resulting RTE will likely differ between one ECU and another.

4.2.2 Access to ports from a software component implementation

The implementation of an AUTOSAR Software Component is not allowed to use the communication layer, for example OSEK COM, directly. To communicate with other software components it uses ports and client-server communication or sender-receiver communication. The RTE generator is responsible for creating the appropriate language-dependant APIs based on the definition of the interface of the component in the Software Component Template. The API has to be the same independent from the mapping of the components, i.e. the component's code must not be changed when the mapping is changed. The API names are derived from the XML files and conform to a naming convention.



Implementation of Client Runnable run1:

```
Rte_Runnable_run1() {
        …
        Rte_Write_wiperWasher_start(…);
        …
}
```

Implementation of WiperWasher Runnable run1:

```
Rte_Runnable_run1() {
        …
        v = Rte_Read_cmd_start(…);
        …
}
```

Figure 16: Mapping from ports to APIs

4.2.3 Implementation of connectors

The RTE generator is also responsible for generating code, which implements the connectors between the ports, such as the AssemblyConnector. This generated code is dependant on the mapping of the software components to ECUs. If the connector connects two components on the same ECU a local communication stub can be generated. Otherwise, a stub that uses network communication must be generated.

Intra-ECU connector

```
Rte_Write_Client_wiperWasher_start(…) {
        modify variable
}
```

Inter-ECU connector

```
Rte_Write_Client_wiperWasher_start(…) {
        access COM
}
```

Figure 17: Mapping from connectors to stubs

The mapping from a connector to a communication stub must conserve the semantics of this connector independent from the used communication medium.

The communication stub is also responsible for parameter marshalling. This includes serializing complex data to a byte stream. But endian conversion (if any is necessary) is delegated to the communication module of the Basic Software.

4.2.4 Lifecycle management

The RTE is responsible for the lifecycle management of AUTOSAR Software Components. It has to invoke startup and shutdown functions of the software component.

4.2.5 Access to Basic Software

An AUTOSAR Software Component is not allowed to access Basic Software directly. Firstly the access to services, to the ECU abstraction, or to Complex Device Drivers is abstracted via ports and AUTOSAR interfaces. With respect to the component implementation, the RTE provides appropriately generated APIs for Basic Software access.

4.2.6 Multiple Instantiations of software components

The RTE shall support multiple instantiations of software components, but in AUTOSAR R2.0 this feature is cancelled and hence, only the enabling/preparation for future releases will be fulfilled in R2.0.

The basic intention of multiple instantiation is to avoid code duplication if possible. Furthermore different private states of multiple instances shall be supported.

4.3 Complex Device Driver

4.3.1 General

The Complex Device Driver is a loosely coupled container, where specific software implementations can be placed. The only requirement to the software parts is that the interface to the AUTOSAR world has to be implemented according to the AUTOSAR port and interface specifications.

The reason to define Complex Device Drivers will be explained in the following chapters.

4.3.2 Complex Sensor and Actuator Control

The main task of the complex drivers is to implement complex sensor evaluation and actuator control with direct access to the µC using specific interrupts and/or complex µC peripherals (like PCP, TPU), e.g.

- injection control
- electric valve control
- incremental position detection

4.3.3 Non-Standardized Drivers

Further on the Complex Device Drivers will be used to implement drivers for hardware which is not supported by AUTOSAR.

If for example a new communication system will be introduced in general no AUTOSAR driver will be available controlling the communication controller. To enable the communication via this media, the driver will be implemented proprietarily inside the Complex Device Drivers. In case of a communication request via that media the communication services will call the Complex Device Driver instead of the communication hardware abstraction to communicate.

Another example where non-standard drivers are needed is to support ASICs that implement a non-standardized functionality.

4.3.4 Migration Mechanism

Last but not least the Complex Device Drivers are to some extend intended as a migration mechanism. Due to the fact that direct hardware access is possible within the Complex Device Drivers already existing applications can be defined as Complex Device Drivers. If interfaces for extensions are defined according to the AUTOSAR standards new extensions can be implemented according to the AUTOSAR standards, which will not force the OEM nor the supplier to reengineer all existing applications.

5 Functional Interfaces

5.1 Overview

The software implementing the automotive functionality is mainly encapsulated in software components. Standardization of the interfaces for AUTOSAR Software Components is a central element to support scalability and transferability of functions across electronic control units of different vehicle platforms.

All specified functions shall satisfy the AUTOSAR Software Component template. The functional interfaces shall show clear semantics of the interfaces and be published in function catalogues of the AUTOSAR partnership.

The objective is that any standard-conformant implementation of a software component can be integrated with substantially reduced effort in a system. Being conformant to the standard in this sense would entail that the component provides a specific, precisely defined functionality through completely defined AUTOSAR Interfaces.

Nevertheless, the standardization could be developed incrementally where possible entities for standardization could be:

Level of abstraction
- Functional aspects
- Behavior and implementation aspects

Level of decomposition
- Low degree of decomposition of the functional domain
- High degree of decomposition of the functional domain

Level of architecture definition

- Terminology
- Standardized data-types
- Partial description of interfaces (without semantics)
- Complete description of interfaces (without semantics)
- Complete description of interfaces (with semantics)
- Partial definition of the functional domain
- Complete definition of the functional domain

5.2 Functional domains

The specification of functional interfaces is divided into 6 domains:

- Body/Comfort
- Powertrain
- Chassis
- Safety
- Multimedia/Telematics
- Man-machine-interface

The domains could be differently handled due to intellectual property rights issues and decomposition levels. In the first phase of AUTOSAR only in the domains body/comfort, chassis, and powertrain results can be expected. All others have lower priority in the first phase.

6 References

[BSWArch] Layered Software Architecture
AUTOSAR_LayeredSoftwareArchitecture.pdf

[Conv2004] AUTomotive Open System ARchitecture – An Industry-Wide Initiative to Manage the Complexity of Emerging Automotive E/E-Architectures,
AUTOSAR_Paper_Convergence_2004.pdf

[ECURes] Specification of ECU Resource Template
AUTOSAR_ECU_ResourceTemplate.pdf

[Glossary] AUTOSAR Glossary
AUTOSAR_Glossary.pdf

[MainReq] Main Requirements
AUTOSAR_MainRequirements.pdf

[Meth] Methodology
AUTOSAR_Methodology.pdf

[SPEM] OMG Object Management Group: Software Process Engineering Metamodel Specification,
http://www.omg.org/technology/documents/modeling_spec_catalog.htm

[SWCTempl] Specification of Software Component Template,
AUTOSAR_SoftwareComponentTemplate.pdf

[SysTempl] Specification of System Template
AUTOSAR_SystemTemplate.pdf

[VFBSpec] Specification of Virtual Function Bus
AUTOSAR_SWS_VFB.doc.pdf