# Standardized Basic System Software for Automotive Embedded Applications

Thomas M. Galla
Elektrobit Austria GmbH
Kaiserstraße 45/2, A-1070 Vienna, Austria
phone: +43 1 59983 5015
fax: +43 1 59983 5018
email: `thomas.galla@elektrobit.com`

May 31, 2014

**Abstract**

The number of electronic systems in modern cars is continuously growing. Currently the development of electronic systems, consisting of so-called electronic control units (ECUs) interconnected by a communication network, account for up to 40% of the overall development costs of a modern car. 50 - 70% of the development costs for a single ECU are related to software. Consequently, software plays an ever more important role, both for the implementation of customer related functions and the infrastructure.

In order to benefit from software reuse, the major automotive companies have standardized (in the context of different consortia) this software infrastructure in form of layered architectures of software modules to a large degree.

In this chapter we will present the most important consortia dealing with the specification of automotive standard system software, and we will describe their output, the standardized software architectures/modules.

## 1 Introduction

In the last decade the percentage of electronic components in today's cars has been ever increasing. Premium cars for example use up to 70 electronic control units (ECUs) which are connected via five system networks and realize over 800 different functions [15].

Since 1993 major automotive companies have been striving for the deployment of standard software modules in their applications as the potential benefits of using standard software modules are huge [17]. While the functional software heavily depends on the actual system and is a discriminating factor of competitive importance, this does not apply to the software infrastructure. Furthermore with continuously shortened development cycles, especially in the electronics area, requirements arise concerning compatibility, reusability, and increased test coverage that can only be fulfilled by setting standards for the various system levels.

This trend has been a key motivation for the formation of several consortia like the OSEK/VDX ("Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug

– Vehicle Distributed Executive")[1] consortium [6] in 1993/94, the ASAM ("Association for Standardization of Automation and Measuring Systems") initiative [1], the HIS ("Herstellerinitiative Software")[2] group [4], the JasPar ("Japan Automotive Software Platform Architecture") consortium [5] in 2004, the EASIS ("Electronic Architecture and System Engineering for Integrated Safety Systems") project consortium [2] in 2003, the AUTOSAR ("Automotive Open System Architecture") consortium [12] in 2003, and the GENIVI Alliance [3] in 2009.

This chapter provides an overview of today's state-of-the-art in standardization of automotive software infrastructures. The chapter is structured as follows: Section 2 provides a short overview of the automotive hardware architecture. Section 3 provides information on the software modules specified by the German working groups OSEK/VDX. Section 4 illustrates the software modules standardized by the ISO. Section 5 targets the software modules defined by ASAM. Section 6 deals with the AUTOSAR initiative. Section 7 provides a short summary, concluding the chapter.

## 2 Hardware Architecture

The hardware architecture of automotive systems can be viewed at different levels of abstraction. On the highest level of abstraction, the *system level*, an automotive system consists of a number of networks interconnected via gateways (see Figure 1). In general, these networks correspond to the different functional domains that can be found in today's cars (i.e., chassis domain, power train domain, body domain).

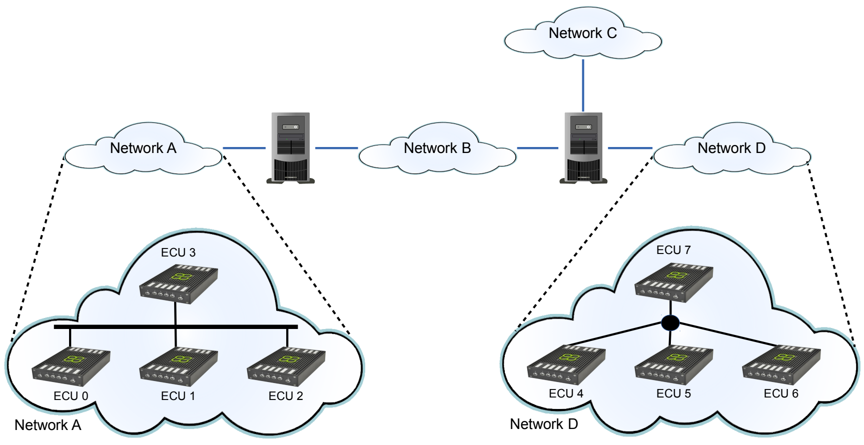


Figure 1: Hardware Architecture – System and Network Level

The networks themselves comprise a number of electronic control units (ECUs) which are interconnected via a communication media (see zoom-in on network A and D in Figure 1). The physical topology used for the interconnection is basically arbitrary; however, bus, star, and ring topologies are the most common topologies in today's cars. – This *network level* represents the medium level of abstraction.

On the lowest level of abstraction, the *ECU level* (Figure 2), the major parts of an ECU are of interest. An ECU comprises one or more micro controller units (MCUs) as well as one or more communication controllers (CCs). In the past in most cases, exactly one (single-core)

[1]Translated into English: "Open Systems and the Corresponding Interfaces for Automotive Electronics"
[2]Translated into English: "Manufacturers' Software Initiative"

MCU and one CC were used to build up an ECU. Today however there is a trend towards multi-core MCUs and even ECUs consisting of multiple MCUs for performance and faul-tolerance reasons. In order to be able to control physical processes in the car (e.g., control the injection pump of an engine), the ECU's MCU is connected to actuators via the MCU's analog or digital output ports. To provide means to obtain environmental information, sensors are connected to the MCUs analogue or digital input ports. We call this interface the ECU's environmental interface. The CC(s) facilitate(s) the physical connectivity of the ECU to the respective network(s). We call this interface of an ECU the ECU's network interface.

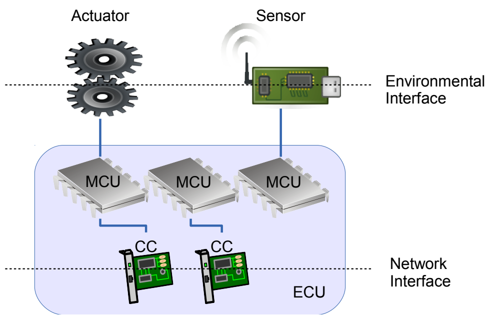


Figure 2: Hardware Architecture – ECU Level

## 3 OSEK/VDX

The OSEK/VDX standard was the result of the endeavors of major german and french car manufacturers and their suppliers to create a standardized software infrastructure for automotive electronics. This standard was initially designed for applications in the area of automotive body electronics or for the power train where autonomous control units build up a loosely-coupled network. It comprises the following standardized components:

**OSEK OS:** The OSEK operating system (OS) [33] is an event-driven operating system intended for hard real-time applications. OSEK OS provides services for task management, task activation by means of events and alarms, inter-task communication via messages, mutual exclusion by means of resources (implementing the *priority ceiling protocol* [16]), and interrupt handling. Hereby OSEK OS distinguishes between *basic tasks* (which do not use any blocking inter-process communication (IPC) or synchronization constructs) and *extended tasks* (which are allowed to use blocking IPC or synchronization constructs).

**OSEK NM:** OSEK network management (NM) [19] provides network management facilities which take care of a *controlled coordinated shutdown of the communication* of multiple ECUs within a network. OSEK NM establishes a *logical ring* among all ECUs participating in OSEK NM. Along this ring OSEK NM *ring messages* are passed between the participating ECUs. These ring messages contain information on whether the sending ECUs desires to perform a transition into a low-power sleep mode. In case all ECUs along the logical ring agree on this transition (i.e., no ECU objects), a coordinated transition into the sleep mode is performed. In case any ECU objects to this decision because it still requires network communication, a transition into the sleep mode is prevented.

**OSEK COM:** OSEK communication (COM) [10] offers services to transfer data between different tasks and/or interrupt service routines (ISRs) residing on the same ECU (internal communication) or possibly being distributed over several ECUs (external communication). OSEK COM supports both *cyclic time-driven communication* as well as *on-demand event-driven communication.* In case of external communication, the *interaction sub-layer* of OSEK COM takes care of the representational issues of signals like byte ordering and alignment. On the sender's side, this layer converts signals from the byte order of the sending ECU into the network byte order. Furthermore, the interaction layer packs multiple signals into a single communication frame in order to reduce communication bandwidth consumption. On the receiver's side, this layer extracts multiple signals from a single communication frame and performs a byte order conversion from the network byte order to the byte order of the receiving ECU.

To support the detection of lost communication frames at the sender and the receiver side, OSEK COM provides mechanisms for transmission and reception timeout monitoring.

**OSEKtime OS:** OSEKtime OS [7] is a time-driven operating system designed for minimal operating system footprint and deployment in safety-related applications. Tasks are activate by a dispatcher based on pre-defined activation times stored in a *dispatcher table* before compile time. The processing of the dispatcher table during run-time is done in a cyclical fashion.

**OSEKtime FTCom:** OSEKtime FTCom [31] is the fault-tolerant communication layer accompanying OSEKtime OS that only supports *cyclic time-driven communication.* Similar to OSEK COM, OSEKtime FTCom can be used for both external as well as internal communication.

In case of external communication, the *fault tolerance sub-layer* of OSEKtime FTCom manages all fault tolerance issues, namely signal replication and signal reduction. On the sender's side, this layer replicates a single application signal and thus produces multiple signal instances. These signal instances are handed over to the interaction sub-layer for byte order conversion and packing. Afterwards, the packed signal instances are transmitted via redundant communication paths. Considering redundant transmission paths temporal redundancy (multiple transmissions on a single communication channel) and spatial redundancy (transmission on multiple communication channels) can be distinguished. On the receiver's side, multiple signal instances are collected from the interaction sub-layer and reduced to obtain a single application signal which is handed on to the application layer.

In order to minimize the memory footprint as well as the required execution time, each of these components is configured upon design time via a configuration file in OSEK implementation language (OIL) syntax [11]. Using a generation tool, the appropriate data structures in the used programming language are created based on this configuration file. Thus, none of the components provides services for dynamic resource allocation (like task creation, memory allocation, a.s.o.).

## 4 ISO

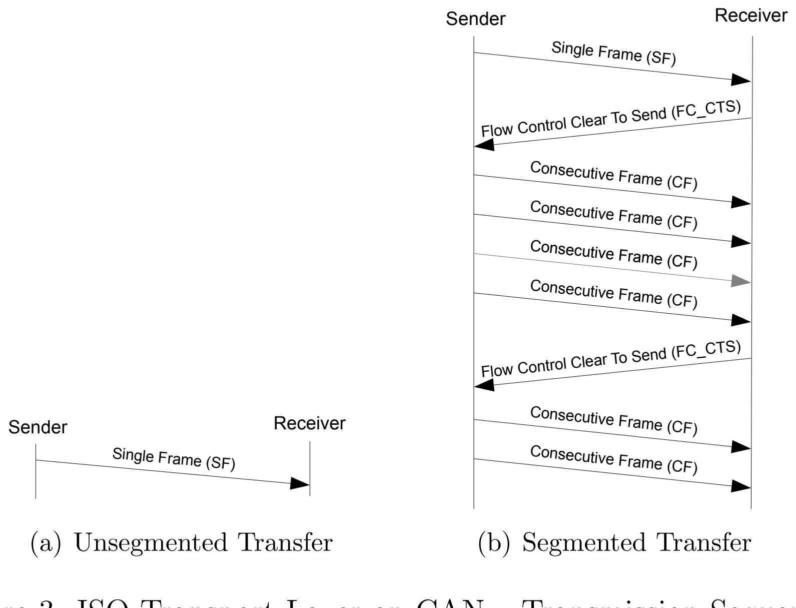
Some of the software layers used for automotive networks in today's cars have been standardized by the International Organization for Standardization (ISO). – The most prominent of these software layers are the following:

**Transport Layer – CAN and FlexRay:** This layer provides *segmented data transfer* to all higher layers (e.g., the diagnostics layer) and thus facilitates exchange of data amounts larger than the maximum transfer unit (MTU) of the underlying communication system.

**Diagnostics Layer – UDS and OBD:** The Unified Diagnostic Services (UDS) specifications define manufacturer specific enhanced diagnostic services which *allow an external tester device to control diagnostic functions* in an ECU via a (serial) data link. The On Board Diagnostics (OBD) specifications define a set of mandatory diagnostic services facilitating the *retrieval of emission-related diagnostic information* from an ECU by an *external tester device.*

## 4.1 Transport Layer – CAN and FlexRay

The ISO transport layer on CAN [23] provides services for *unacknowledged segmented data transfer (USDT) of known length* to higher software layers (e.g., ISO diagnostics (see Section 4.2)) by facilitating the transmission of messages whose length is greater than the MTU of the underlying communication system. On the sender's side, the transport layer will split such long messages into multiple segments, each small enough for the underlying communication system. On the receiver's side, the transport layer reassembles these segments again. As far as the frame format is concerned, the ISO transport layer distinguishes between single frames (SF), first frames (FF), consecutive frames (CF), and flow control frames (FC), e.g., the clear-to-send flow control frame (FC_CTS) depicted in Figure 3(b). Hereby single frames are used for reduction of protocol overhead in case the amount of data to be transmitted does not exceed the MTU. – In that case only a single frame is used in order to exchange this data. Otherwise a first frame, which contains the total number of data bytes to be sent as well as the first few data bytes, is transmitted followed by one or more consecutive frames, which then contain the remaining data bytes. For flow control reasons (i.e., to prevent the sender from outpacing the receiver) the receiver is allowed to send flow control frames at defined points in time. Figure 3 illustrates the frame exchange in an unsegmented (Figure 3(a)) and a segmented transmission (Figure 3(b)).



(a) Unsegmented Transfer (b) Segmented Transfer

Figure 3: ISO Transport Layer on CAN – Transmission Sequence

Hereby the frame format depicted in Figure 4 is used by the ISO transport protocol (TP) on CAN. The *source* and *destination address* fields are used to identify the sender and the receiver(s) of the transport layer frame. The *address type* field is used to define the semantics of the destination address. Here ISO TP on CAN provides support for single ECU addresses (called *physical addresses* by the ISO TP on CAN specification) and multicast addresses (called *functional addresses* by the ISO TP on CAN specification). In case of multicast transmissions, only unsegmented data transfer is allowed in ISO TP on CAN.

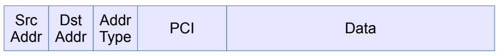


Figure 4: ISO Transport Layer on CAN – Frame Format

The *protocol control information (PCI)* field is used to distinguish between the different frame types (i.e., SF, FF, CF, FC). Furthermore the PCI field carries the *total number of data bytes* to be transmitted in case of a single or a first frame, a *sequence number* in case of a consecutive frame (in order to facilitate early detection of a loss of a consecutive frame), and more detailed information regarding the *type of flow control* in case of a flow control frame.

In addition to the service of the ISO transport layer on CAN, the ISO transport layer on FlexRay [22] provides an *acknowledge and retry mechanism* facilitating a positive and a negative acknowledgement by the receiver to indicated successful reception or the request for (early) retransmission to the sender. To this end a new flow control frame for explicit acknowledgement (FC_ACK) is introduced. Furthermore the ISO transport layer on FlexRay supports the *transmission of messages with unknown but finite data length* by adding a dedicated last frame (LF) that marks the end of the transmission. By combining these two features, the ISO transport layer on FlexRay facilitates *acknowledged segmented data transfer (SDT) of unknown length.*

## 4.2 Diagnostics – UDS and OBD

The Unified Diagnostic Services (UDS) [25] define client/server diagnostic services which allow a tester device (client) to control diagnostic functions in an ECU (server) via a (serial) data link. Hereby the provided services, the encoding of the service identifiers (SIDs), and the encoding of the parameters are standardized by the UDS specification. The services provided by UDS can be grouped into the following six functional units:

**Diagnostic Management:** This functional unit covers all services realizing diagnostic management functions between the client and the server. – Examples for these kind of services are the services `startDiagnosticSession` for initiating a diagnostic session, `stopDiagnosticSession` for terminating a diagnostic session, `ecuReset` for resetting an ECU, and `readECUIdentification` for reading ECU identification information like the serial number or the programming date of the ECU.

**Data Transmission:** This functional unit covers all services dealing with data exchange between the client and the server. – Prominent examples for these services are the services `readMemoryByAddress` and `writeMemoryByAddress`, which are used to read and write a memory range (defined by start address and the number of bytes to read/write) of the ECU as well as `readDataByIdentifier` and `writeDataByIdentifier` which read and write a data item that is uniquely identified via an ID.

**Stored Data Transmission:** This functional unit covers all services which are used to perform exchange of data which is stored within the ECU (in a non-volatile way).

– Examples for these kind of services are the service `readDiagnosticTroubleCodes`, which is used to retrieve stored trouble codes from the ECU's error log as well as the service `clearDiagnosticInformation`, which is used to remove all entries from the ECU's error log.

**Input/Output Control:** This functional unit covers all services which deal with input and output control functions. – An example for a service contained in this group is the service `inputOutputControlByLocalIdentifier`, which can be used by the client to substitute a value for an input signal of the server.

**Remote Activation of Routine:** This functional unit covers all services which provide remote activation of routines within an ECU. – Examples for services in this functional unit are the services `startRoutineByAddress` and `stopRoutineByAddress`, which are used to remotely invoke a routine in the server (ECU) and to terminate an executing routine in the server (ECU). Hereby the routine to invoke or stop is identified by the routine's address.

**Upload/Download:** This functional unit covers all services which realize upload/download functionality between the client and the server. – Examples for services within this unit are the `requestDownload` service, which gives the client the possibility to request the negotiation of a data transfer from client to server, the `requestUpload` service, which gives the client the possibility to request the negotiation of a data transfer from server to client, and the `transferData` service, which actually takes care of the data transmission between client and server.

In UDS services are identified by the so-called *service identifier (SID)* field which is the first byte in a diagnostic message[3]. Based on this field the layout of the remainder of the diagnostic message is completely different. The diagnostic server uses the SID to select the proper service requested by the client and interprets the remainder of the diagnostic message as parameters to the service request according to the SID.

The On Board Diagnostics (OBD) specify a set of mandatory diagnostic services focussing on the *retrieval of emission-related information* from an ECU by an *external tester device*. For this purpose OBD defines services for the *retrieval of emission-related diagnostic trouble codes*, the *clearing/resetting of emission-related diagnostic trouble codes*, the *controlling of on-board diagnostic monitoring functionality* like the invocation of defined tests (e.g., a test for leakage in the evaporative system of the vehicle), the *request of results of conducted on-board diagnostic monitoring tests*, and the *retrieval of vehicle information* like the vehicle identification number (VIN).

Just like in UDS, OBD encodes the diagnostic service requested by the client in a *service identifier (SID)* located in the first byte in a diagnostic message. – Hereby the SID ranges of ODB and UDS are disjunct to prevent interference between the two standards.

For the exchange of information between the client and the server (and vice-versa) both diagnostic protocols use transport layer described in Section 4.1.

## 4.3 Diagnostics over IP – DoIP

In order to facilitate diagnostic communication between external tester device and ECUs using the internet protocol (IP) as well as the transmission control protocol (TCP) and the user datagram protocol (UDP), the diagnostics over IP (DoIP) specification [24] defines

[3]This diagnostic message is payload from ISO TP's point of view.

features that can be used to *detect a vehicle in a network* and to *enable communication with the vehicle's central diagnostic gateway* as well as with *other ECUs not directly connected to the tester device.*

ECUs which are capable of adhering to the DoIP protocol are termed *DoIP entities* by the DoIP specification. DoIP entities are further distinguished into *DoIP nodes*, which do not forward diagnostic communication requests towards a non-Ethernet in-vehicle network, and *DoIP gateways*, which perform this kind of forwarding and thus relay information between the external tester and non-DoIP capable ECUs.

**Vehicle Identification Request and Vehicle Announcement:** These two types of messages are required to identify the DoIP entities in a network and to retrieve their IP and logical addresses. The *vehicle identification request* contains either the *vehicle identification number (VIN)* or a *unique hardware ID* (e.g., the medium access control (MAC) address) of the addressed DoIP entity and is broadcasted in the network. The DoIP entity with a matching VIN or hardware ID answers this request with a *vehicle announcement* containing its own IP address (as part of the IP header), its logical address (which is the ECU's physical ISO TP address (see Section 4.1)), and port numbers of TCP and UDP ports to be used for subsequent communication with the DoIP entity.

**Routing Activation Request and Response:** These two message types are required to activate routing in DoIP gateways between the vehicle's external network and the vehicle internal network in order to facilitate subsequent diagnostic data exchange between the external tester device and non-DoIP capable ECUs connected to the vehicle internal network.

**Diagnostic Message and Diagnostic Message Acknowledgement:** After a successful routing activation diagnostic message exchange between an external tester device and a non-DoIP capable ECU connected to the vehicle internal network can take place. The diagnostic message used for the data exchange between the tester and the DoIP gateway hereby contains the ISO TP source address followed by the ISO TP destination address followed by the UDS/OBD message. If the DoIP gateway receives such a diagnostic message, it extracts the two ISO TP addresses and the UDS/OBD message and forwards the UDS/OBD message to the ISO TP of the target network which is selected based on the destination address. Hereby the ISO TP source address and the ISO TP destination address contained in the DoIP diagnostic message are used as respective source and destination addresses on the target network. A successful or a failed forwarding to the target network is signalled to external tester device by means of positive or negative diagnostic message acknowledgment.

# 5 ASAM

The Association for Standardization of Automation and Measuring Systems (ASAM) started as an initiative of German car manufacturers with the goal to define standards for data models, interfaces and syntax specifications for testing, evaluation, and simulation applications.

Apart from several data exchange formats like Open Diagnostic Data Exchange Format (ODX), the Functional Specification Exchange Format (FSX), the Meta Data Exchange Format for Software Module Sharing (MDX), and the Fieldbus Exchange Format (FIBEX), ASAM defines the Universal Measurement and Calibration Protocol Family (XCP) which is described in the next section.

## 5.1 XCP – The Universal Measurement and Calibration Protocol Family

The Universal Measurement and Calibration Protocol Family XCP [32] is used for the following main purposes:

- Synchronous data transfer (acquisition and stimulation)
- On-line calibration
- Flash programming for development purposes

Prior to describing these main operations of XCP though, we will focus on the protocol's internal structure. XCP itself consists of two main parts, namely the *XCP protocol layer* and several *XCP transport layers*, one dedicated transport layer for each underlying communication protocol (currently CAN, FlexRay, universal serial bus (USB), TCP/IP, UDP/IP, and standard peripheral interface (SPI) are supported). – Figure 5 illustrates this XCP protocol stack.

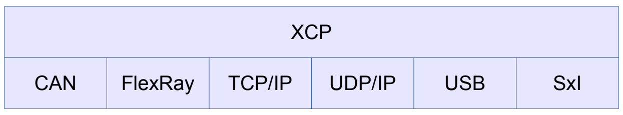


Figure 5: XCP Protocol Stack

### 5.1.1 XCP Protocol Layer

The XCP protocol layer is the higher layer of the XCP protocol family. This layer implements the main operations of XCP which are described in detail in Sections 5.1.3, 5.1.4, and 5.1.5. The XCP protocol layer itself is independent of a concrete communication protocol (e.g., CAN, FlexRay, . . . ). Data exchange on the XCP protocol layer level is performed by data objects called *XCP packets*. – Figure 6 illustrates the structure of an XCP packet.

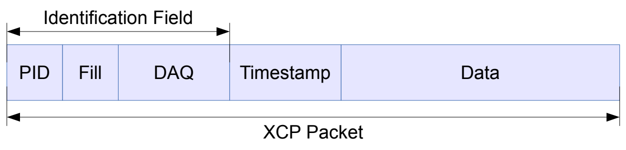


Figure 6: Structure of an XCP Packet

An XCP packet starts with a *Identification Field* containing a *Packet Identifier (PID)* which is used to establish a common understanding about the semantics of the packet's data between the XCP master and the XCP slave. The PID is thus used to uniquely identify each of the following two basic packet types and their respective sub-types:

**Command Transfer Object (CTO) Packets:** This packet type is used for the transfer of generic control commands from the XCP master to the XCP slave and vice-versa. It is used for carrying out *protocol commands (CMD)*, transferring *command responses (RES)*, *errors (ERR)*, *events (EV)*, and for issuing *service requests (SERV)*.

**Data Transfer Object (DTO) Packets:** This packet type is used for transferring synchronous data between the XCP master and the XCP slave device. *Synchronous data acquisition (DAQ) data* is transferred from the XCP slave to the XCP master whereas *synchronous data stimulation (STIM) data* is transported from the master to the slave.

The *DAQ Field* is used to uniquely identify the DAQ list (see Section 5.1.3) to be used for data acquisition or stimulation if the XCP packet is of DTO packet type. In order to have the DAQ Field aligned to 16-bit boundaries, a Fill Byte is introduced between the PID Field and the DAQ Field. – In case the packet is a CTO packet, the DAQ field is omitted.

The *Time-Stamp Field* is used in DTO packets to carry a time-stamp provided by the XCP slave for the respective data acquisition. – The length of the time-stamp field may vary between one to four bytes depending on the configuration. In case the packet is of CTO type, the time-stamp field is omitted.

Command packets (CMD) are explicitly acknowledged on the XCP protocol layer by sending either a command response packet (RES) or an error packet (ERR). Event (EV) packets (i.e., packets informing the XCP master that a specific event has occurred in the XCP slave (e.g., an overload situation has occurred during data acquisition)), service request (SERV) packets (i.e., packets used by the slave to request certain services from the master (e.g., a packet containing text that is to be printed by the master)) and data acquisition packets (DAQ, STIM) are sent asynchronously and unacknowledged at the XCP protocol layer. Therefore, it may not be guaranteed that the master device will receive these packets when using a non-acknowledged transportation link like e.g., UDP/IP.

### 5.1.2 XCP Transport Layers

The protocol layer described in the previous sections is independent from the underlying communication protocol. – In order to be able to use XCP on-top of different communication protocols, the XCP specification defines multiple XCP transport layers that perform the packing of the protocol independent XCP packets into frames of the respective communication protocol, by adding an XCP header containing a *node address field* used to identify the destination ECU, a *counter field* used for sequence numbering of the XCP packets and a *length field* defining the length of the XCP packet. Depending on the actual communication protocol used, some of these header fields might be missing (e.g., in case TCP/IP is used as communication protocol, the node address is omitted, since IP's addressing scheme is used).

### 5.1.3 Synchronous Data Transfer

The synchronous data transfer feature of XCP allows for a data exchange between XCP master and XCP slave that is performed synchronous to the XCP slave's execution. This exchange is carried out by using *data transfer objects (DTOs)* which are transfered via DTO packets. The memory regions of the XCP slave's memory that are the source or the destination of the transfer are linked to the DTO by so-called *object description tables (ODTs)*. A sequence of one or more ODTs are grouped into a so-called *data acquisition (DAQ) list* (see Figure 7).

Hereby the DAQ-DTO contains a packet identification field (PID), which is used to link the DAQ-DTO to the respective ODT (PID field matches ODT number). For each element within a DAQ-DTO, a corresponding ODT entry is present in the ODT which references a specific part in the ECU's memory by the attributes address and length. Upon processing of the ODT, the ODT entries are used to transfer element from the ECU's memory into the

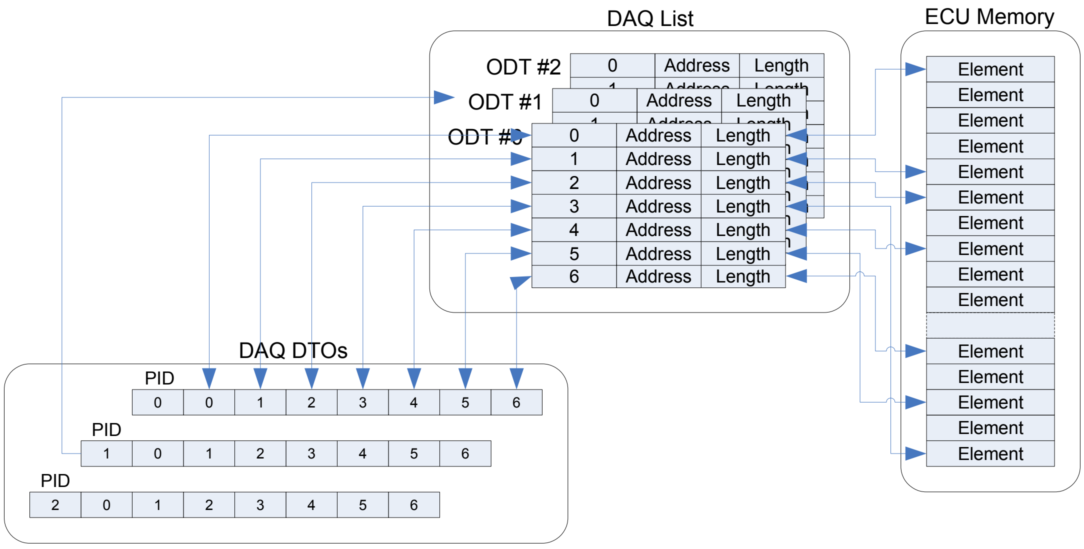


Figure 7: Structure of a DAQ List

corresponding element in the DAQ-DTO (in case of data acquisition) and vice-versa (in case of data stimulation).

DAQ lists can either be statically stored in an XCP slave ECU's memory or dynamically allocated by the XCP master via special protocol command packets.

### 5.1.4 On-line Calibration

For the on-line calibration feature of XCP, the slave's physical memory is divided into so-called *sectors* which reflect the memory's size and limit constraints when reprogramming/erasing parts of the memory. This division into sectors thus describes the physical layout of the XCP slave's memory.

The logical layout is described by dividing the memory into *segments*. – This division does not need to adhere to the physical limitations of the division into sectors. Each segment can further consist of one or multiple pages where at any given instance in time only one page of a segment is accessible to the ECU. – This page is called the active page for the ECU's application. The same holds true for XCP itself as well (active XCP page). XCP ensures that concurrent access to the same page by the ECU's application and XCP itself is prevented.

Via XCP CMD packets, the XCP master can instruct the XCP slave to switch active pages of the same segment (i.e., to make a page which has previously not been the active one the new active page) and to copy data between pages of the same or of different memory segments.

This way, the calibration data used for the slave ECU's control loops, for example, can be altered upon run-time under control of the XCP master.

### 5.1.5 Flash Programming

In order to facilitate the exchange of the current image of the application program in the slave ECU's non-volatile memory, XCP defines special commands (exchanged via XCP command packets) for performing flash (re-)programming. The following list contains a short description of the main commands used in flash programming:

**PROGRAM_START:** Indicate the beginning of a programming sequence of the slave ECU's non-volatile memory.

**PROGRAM_CLEAR:** Clear a part of the slave ECU's non-volatile memory (required before programming new data into this part of memory).

**PROGRAM_FORMAT:** Used to specify the format for the data that will be transferred to the slave ECU (e.g., used compression format, used encryption algorithm, ...).

**PROGRAM:** This command, which is issued in a loop as long as there is still data available that needs to be programmed, is used to program a given number of data bytes to a given address within the slave ECU's non-volatile memory.

**PROGRAM_VERIFY:** Verify that the programming process has completed successfully by, for example, calculating a checksum over the programmed memory area.

**PROGRAM_RESET:** Indicates the end of the programming sequence and optionally resets the slave ECU.

By means of these flash programming XCP commands, a flash download process for the ECU's development stage can be implemented. – In-system flash programming in series cars however is usually performed via the diagnostics protocol (see Section 4.2).

## 6 AUTOSAR

The development partnership AUTomotive Open System ARchitecture (AUTOSAR) is an alliance of car manufacturers, major supplier companies, and tool vendors working together to develop and establish an open industry standard for an automotive electronic architecture together with a development methodology and an XML-based data exchange format which is intended to serve as a basic infrastructure for the management of (soft and hard real-time) functions within automotive applications.

At the time being, AUTOSAR versions 1.0, 2.0, 2.1, 3.0, 3.1, 3.2, 4.0, and quite recently 4.1 as a major update of the standard which, for example, for the first time included support for Ethernet as an in-vehicle communication network have been released. Active development of the AUTOSAR standard is limited to the AUTOSAR 3.2 and 4.1 versions though. – All other versions are in maintenance mode.

While series projects completed by today[4] are based on AUTOSAR 2.1 and 3.1, the next generation vehicles currently under development will mainly be based on AUTOSAR 4.0 [26].

The AUTOSAR software architecture makes a rather strict distinction between application software and basic or system software. While the *basic (or system) software* provides functionality like communication protocol stacks for automotive communication protocols (e.g., FlexRay [13, 27]), a real-time operating system, and diagnostic modules, the *application software* comprises all application specific software items (i.e., control loops, interaction with sensor and actuators etc.). This way, the basic or system software provides the foundation the application software is built upon.

The so-called *Runtime Environment* (`Rte`) acts as an interface between application software components and the system software modules as well as the infrastructure services that enables communication to occur between application software components.

[4]The resulting vehicles are already on the road.

## 6.1 Application Software Architecture

The application software in AUTOSAR is structured into *software components* which communicate via *ports* and are interconnected by means of *connectors*. In the following subsections these concepts will be explained in detail.

### 6.1.1 Software Components

Application software in AUTOSAR is composed of reusable *software components (SWCs)*. For structuring reasons AUTOSAR facilitates a hierarchical decomposition of the application software by distinguishing between *atomic SWCs* which represent the bottom level of this decomposition and thus cannot be divided any further and *composition SWCs*, which can be divided into multiple atomic SWCs and further composition SWCs.

The following types of atomic SWCs are defined by AUTOSAR:

**Application SWCs:** An application SWC implements (part of) the application. Since the application SWC does not directly interact with any particular hardware resources it is hardware- and location independent.

**Parameter SWCs:** A parameter SWC provides read access to parameter values that can either be based on constant data or on variable data that can be modified by means of calibration (see Section 5.1.4). Like application SWCs the parameter SWCs do not directly interact with any particular hardware resources and are thus hardware- and location independent as well.

**Service SWCs:** The service SWC makes the services of a local (i.e., located on the same ECU) system software module (see Section 6.2) available to other SWCs.

**Service Proxy SWCs:** This type of SWC is responsible for distribution of mode information throughout the system. Since AUTOSAR does not allow non-service proxy SWCs to directly access to mode information provided by other ECUs, each ECU requiring such remote mode information requires a service proxy SWC.

**Complex Driver SWCs:** The complex driver SWC facilitates direct access to the MCU's hardware in particular for resource critical applications performing complex sensor evaluation and actuator control.

**ECU-Abstraction SWCs:** The ECU-abstraction SWC is a specialization of a complex driver SWC that makes the ECU's specific I/O capabilities available to the other SWCs (especially to the sensor-actuator SWCs).

**Sensor-Actuator SWCs:** The sensor-actuator SWC handles the specifics of a particular sensor and/or actuator. To do so, the sensor-actuator SWC directly interacts with the ECU abstraction SWCs.

**NV Block SWCs:** The non-volatile (NV) block SWCs provide fine granular (i.e., smaller than the block-based access provided by the Non-Volatile RAM Manager (`NvM`) (see Section 6.2.2)) read and write access to non-volatile data to other SWCs by combining these fine granular accesses and mapping them to the blocks provided by the `NvM`.

While parameter SWCs and application SWCs can be relocated among ECUs (deployed on different ECUs), the other types of SWCs are bound to a specific ECU, since they require a particular piece of hardware (e.g., a particular sensor in case of sensor-actuator SWCs and

ECU-abstraction SWCs), provide access to the ECU's local non-volatile memory (in case of NV block SWCs), or provide access to the ECU's local system software or complex drivers (in case of service SWCs and complex driver SWCs). Since the sole purpose of service proxy SWCs is to act as a local proxy for the provision of remote mode information, service proxy SWCs are bound to the specific ECU as well.

### 6.1.2 Ports and Interfaces

All SWCs interact with their environment through *ports*. Depending on whether the SWC requires or provides specific services or particular data elements, a distinction between *require ports (R-ports)* and *provide ports (P-ports)* is made.

Ports are typed by a specific *interface*, where AUTOSAR defines the following types of interfaces:

**Sender-Receiver (S/R) Interfaces:** Via a sender-receiver interface, a single *sender* distributes information to one or more *receivers* (1:n communication), or one receiver gets information from one or several senders (n:1 communication). The sender-receiver interface consists of *data elements* that define the data that is exchanged. The type of a data-element can either be a simple data type (e.g., an integral value) or a complex data type (e.g., an array or a structure). The transfer of a single data element is hereby performed in an atomic fashion.

**Client-Server (C/S) Interfaces:** Via a client-server interface one or more *clients* can invoke a remote operation provided by a *server* (n:1 communication). The client-server interface consists of the *operations* that are provided by the server and can be invoked by the client. This operation invocation can either be *synchronous* (i.e., the client is blocked until the result of the operation is available) or *asynchronous* (the client is not blocked but asynchronously notified as soon as the result is available).

**Mode-Switch (M/S) Interfaces:** Via a mode-switch interface a *mode manager* publishes the current mode to one or more *mode users* (1:n communication) in order to have these mode users adjust their behavior according to the current mode or to synchronize activities to mode switches. The mode-switch interface consists of a so-called *mode declaration group* that is the aggregation of all distinct *mode declarations* (i.e., all different modes the mode manager can reside in).

**Parameter Interface:** Via a parameter interface a parameter SWC *publishes constant data or calibration data*. Similar to the S/R interface, the parameter interface consists of *data elements* that define the data that is provided.

**Non-Volatile (NV) Data Interface:** Via an NV data interface an NV block SWC provides *access to non-volatile data* on a fine-grained level. Similar to the S/R interface, the NV data interface consists of *data elements* that define the NV data that is provided.

**Trigger Interface:** By means of a trigger interface an SWC is capable of *directly triggering* the execution of the *runnable entities* (see Section 6.1.4) of other software components in order to facilitate fast responses to certain events.

### 6.1.3 Connectors

The ports of different SWCs can be connected by means of *connectors*. Based on the type of interface of the ports these connectors represent the exchange of data elements (in case of S/R interfaces), the invocation of a (remote) operation (in case of C/S interfaces), the information about a performed mode switch (in case of M/S interfaces), the retrieval of constant or calibration data (in case of parameter interfaces), the access to non-volatile data (in case of NV data interfaces), and the direct triggering of the execution of a runnable entity (in case of trigger interfaces). Depending on whether a connector connects two P-ports or two R-ports or whether the connector connects a P-port with an R-port, AUTOSAR distinguishes between *delegation connectors* and *assembly connectors*. Whereas the former are used for connection of ports located on *different levels of the SWC hierarchy* (i.e., connection of a port of an SWC with the corresponding port of its enclosing composition SWC) the latter connect ports within the *same level of the SWC hierarchy* (i.e., connection of corresponding ports of SWCs within the same composition SWC).

### 6.1.4 Runnable Entities

The software components themselves are composed of one or more *runnable entities (REs)* which are executed in the context of operating system tasks. Runnable entities are the basic unit of execution in AUTOSAR and basically correspond to a function in common programming languages. Depending on whether or not runnable entities have internal wait-points (i.e., they use blocking inter-process communication (IPC) or synchronization constructs), AUTOSAR distinguishes between *category 1 runnable entities* (without wait-points) and *category 2 runnable entities* (with wait-points). – The former can be assigned to basic tasks of the operating system (which are not allowed to invoke any blocking system calls) whereas the latter have to be mapped to extended tasks (which are allowed to use blocking system calls).

Runnable entities are invoked by *`Rte` events*. The following list names the most important `Rte` events that are defined by AUTOSAR:

**Timing Event:** This kind of event facilitates the periodic activation a runnable entity in an SWC.

**Data Received Event:** This kind of event is used to activate a runnable entity in a receiver SWC upon the reception of a data element at an S/R port.

**Data Send Completed Event:** This kind of event is used to activate a runnable entity in a sender SWC upon the successful transmission of a data element at an S/R port.

**Operation Invoked Event:** This kind of event is used to activate the runnable entity implementing a C/S operation in a server SWC upon the request of the client SWC.

**Asynchronous Server Call Returns Event:** This kind of event is used to activate a runnable entity in a client SWC as a result of a notification that the server SWC has completed an asynchronous C/S operation.

**Mode Switch Event:** This kind of event is used to activate a runnable entity in a mode user SWC as a result of a mode switch performed by a mode manager SWC.

**Mode Switch Acknowledge Event:** This kind of event is used to activate a runnable entity in a mode manager SWC as a result of a notification that all mode users have reacted to the mode switch.

**(Internal/External) Trigger Occured Event:** This kind of event is used to activate a runnable entity in a SWC as a result of an explicit trigger by a runnable entity of the same SWC (internal) or of some other SWC (external).

**Init Event:** This kind of event is used to activate a runnable entity in an SWC as a result of the initialization of the `Rte`.

### 6.1.5 Communication Scopes

When looking at the the different possible scopes of communication, AUTOSAR distinguishes between the following scopes:

**Task:** Depending on whether or not the communicating runnables are allocated to the same task, a distinction between *intra-task communication* and *inter-task communication* is made.

**Partition:** Depending on whether or not the tasks of the communicating runnables are allocated to the same memory partition, a distinction between *intra-partition communication* and *inter-partition communication* is made.

**Core:** Depending on whether or not the tasks of the communicating runnables are allocated to the same MCU core, a distinction between *intra-core communication* and *inter-core communication* is made.

**ECU:** Finally depending on whether or not the SWCs of the communicating runnables are deployed to the same ECU, a distinction between *intra-ECU communication* and *inter-ECU communication* is made.

## 6.2 System Software Architecture

In addition to the application software components, AUTOSAR also defines a layered architecture of system software modules [8] which provides the basic platform for the execution of the application software components. Figure 8 provides a coarse grained overview of the major layers of these system software modules.

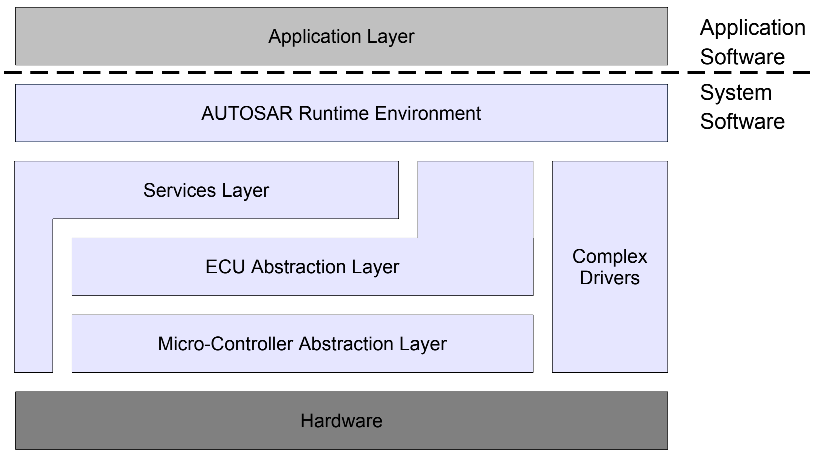


Figure 8: AUTOSAR – System Software Layered Architecture

**Micro-Controller Abstraction Layer:** This is the lowest software layer of the system software in AUTOSAR. This layer contains software modules which directly access the micro-controller's internal peripheral devices as well as memory mapped external devices. – The task of this layer is to make higher software layers independent of the micro-controller type.

**ECU Abstraction Layer:** This layer interfaces with the software modules of the micro-controller abstraction layer and additionally contains drivers for external devices. The layer offers a standardized API to access the peripheral devices of an ECU regardless of their location (i.e., whether the peripherals are internal or external with respect to the ECU's micro-controller) and their connection (e.g., via port pins, via a serial peripheral interface (SPI), a.s.o.). – The task of this layer is to make higher software layers independent of the ECU's hardware layout.

**Services Layer:** This layer is mainly located on-top of the ECU abstraction layer and provides operating system services, vehicle network communication and management services, memory services (e.g., non-volatile random access memory (NVRAM) management), diagnostic services (e.g., error logger), and state management services of the whole ECU. – The task of this layer is the provision of basic services to other system software modules and the application software. In the latter case the respective service layer module acts as a service SWC towards the `Rte`.

**Complex Drivers:** The concept of complex drivers somehow intentionally violates the AUTOSAR layered architecture in order to provide the possibility to deploy legacy device drivers in an AUTOSAR system software module stack. Additionally, the complex device driver module concept has been introduced to facilitate the integration of highly application and ECU dependent drivers that require complex sensor evaluation and actuator control with direct access to the micro-controller itself and complex micro-controller peripherals for performance reasons. Complex drivers may provide an interface to (application) SWCs. In that case, the complex drivers act as complex driver SWCs towards the `Rte`.

**Runtime Environment (`Rte`):** The AUTOSAR Runtime Environment provides the interface between application software components and the system software modules as well as the infrastructure services that enable communication between application software components.

**Application Layer:** Actually this layer is not part of the AUTOSAR system software modules layered architecture, since this layer contains the AUTOSAR application software components described in Section 6.1.

**Libraries:** AUTOSAR libraries are *not* assigned to any particular software layer. Instead *all* layers (including the SWCs of the application layer) are allowed to make use of these libraries.

In addition to this mainly vertical structuring AUTOSAR further horizontally subdivides the system software modules into different sub-stacks. This subdivision is depicted in Figure 9.

**Input/Output Sub-Stack:** The input/output sub-stack comprises software modules that provide standardized access to sensors, actuators, and ECU on-board peripherals (e.g., D/A or A/D converters etc.).

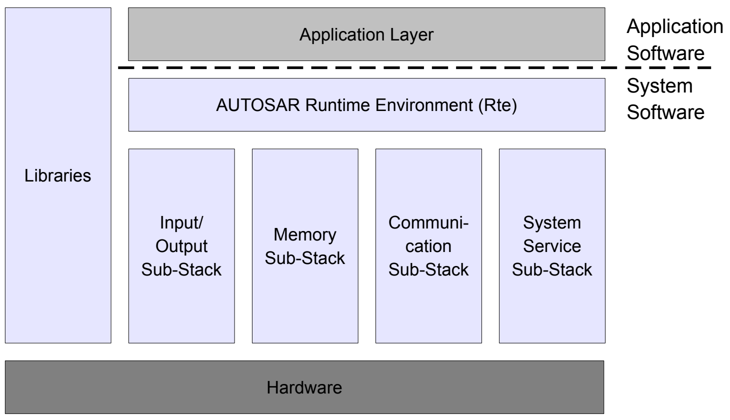


Figure 9: AUTOSAR – System Software Sub-Stacks Overview

**Memory Sub-Stack:** The memory sub-stack comprises software modules that facilitate the standardized access to internal and external non-volatile memory for means of persistent data storage.

**Communication Sub-Stack:** The communication sub-stack contains software modules that provide standardized access to vehicle networks (i.e., the Local Interconnect Network (LIN) [9], the Controller Area Network (CAN) [20, 21], FlexRay [13, 27], and Ethernet).

**System Service Sub-Stack:** Last but not least, the system service sub-stack encompasses all software modules that provide standardized (e.g., operating system, timer support, error loggers) and ECU specific (ECU state management, watchdog management) system services, and library functions.

**Libraries:** Since the libraries are not part of a particular sub-stack, they are depicted separately in Figure 9.

Regardless of the vertical and horizontal structuring, the following classification can be applied to the AUTOSAR system software modules:

**Drivers:** A driver contains the functionality to *control and access an internal or an external device.* Hereby, internal devices, which are located within the micro-controller, are controlled by internal drivers whereas external devices, which are located on ECU hardware outside the micro-controller, are controlled by external drivers. Internal drivers can usually be found within the micro-controller abstraction layer, whereas external drivers are situated in the ECU abstraction layer. Drivers do not change the content of the data handed to them.

**Interfaces:** An interface contains the functionality to *abstract from the hardware realization of a specific device* and to provide a generic API to access a specific type of device independent of the number of existing devices of that type and independent of the hardware realization of the different devices. Interfaces are generally located within the ECU abstraction layer. Interfaces do not change the content of the data handed to them.

**Handlers:** A handler *controls the concurrent, multiple, and asynchronous accesses of one or more clients* to one or more driver or interface modules. Thus a handler performs buffering, queuing, arbitration, and multiplexing. Handlers do not change the content of the data handed to them.

**Managers:** A manger offers specific services for multiple clients. Managers are required whenever pure handler functionality is insufficient for accessing and using interface and driver modules. Managers furthermore are allowed to *evaluate and change or adapt the content of the data* handed to them. Managers are usually located in the services layer.

In the following sections, the different sub-stacks of AUTOSAR system software modules are described in detail.

### 6.2.1 Communication Sub-Stack

The communication sub-stack contains a group of modules that facilitate communication among the different ECUs in a vehicle via automotive communication network (CAN, LIN, FlexRay, and Ethernet). The structure of the communication sub-stack is depicted in Figure 10.

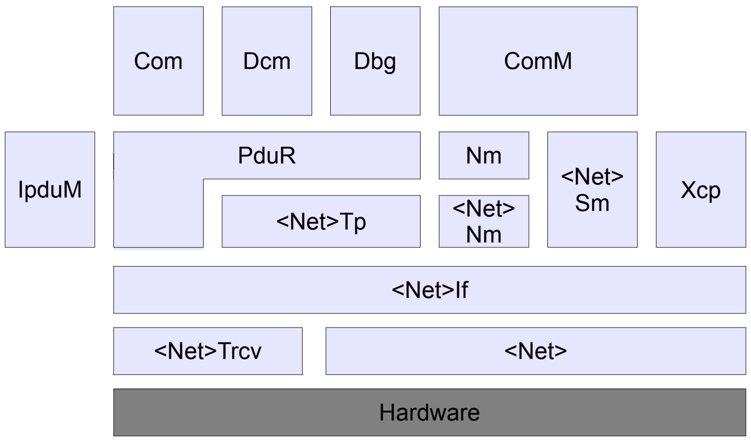


Figure 10: AUTOSAR – Communication Sub-Stack

Hereby, `<Net>` is used as a placeholder for the respective communication network (i.e., CAN, LIN, FlexRay, and Ethernet). Thus the AUTOSAR communication sub-stack contains communication network specific instances of the Transport Protocol (`Tp`), Network Management (`Nm`), Interface (`If`), State Manger (`Sm`), Transceiver Driver (`Trcv`), and Driver (no suffix) modules.

The structure of the communication sub-stack for Ethernet networks and the communication sub-stack for CAN networks for heavy duty vehicles according to the SAE J1939 standard however slightly deviate from the structure depicted in Figure 10. For Ethernet networks the Transport Protocol functionality is distributed among two modules, namely the `TcpIp` modules and the the `SoAd` module, and two additional modules, namely the `DoIp` module and the `Sd` module are introduced. The structure of the communication sub-stack for Ethernet networks is depicted in Figure 11.

For CAN networks for heavy duty vehicles according to the SAE J1939 standard dedicated modules for Transport Protocol (`J1939Tp`) and Network Management (`J1939Nm`) replace the respective modules of the communication sub-stack for standard CAN networks, and an additional Diagnostic Communication Manager module (`J1939Dcm`) and a Request

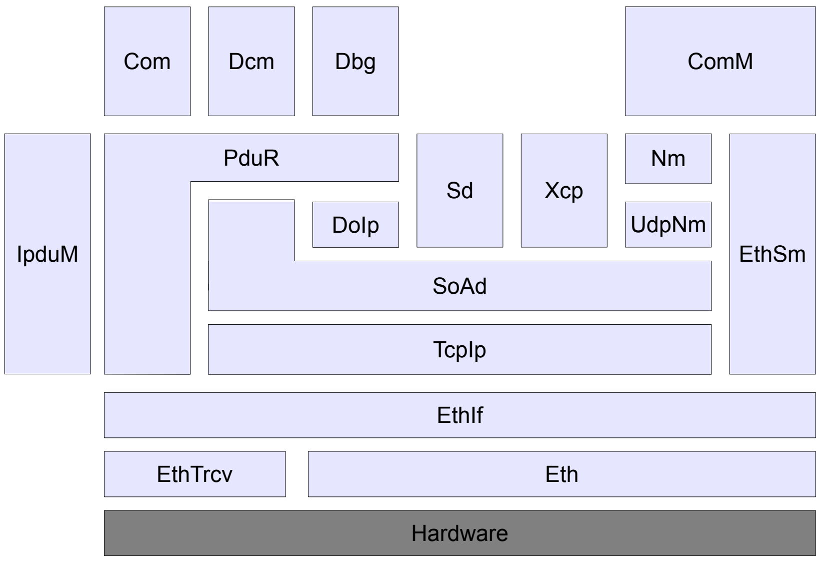


Figure 11: AUTOSAR – Communication Sub-Stack for Ethernet Networks

Management module (`J1939Rm`) are introduced which implement J1939 specific functionality. The structure of the communication sub-stack for CAN networks for heavy duty vehicles according to the SAE J1939 standard is depicted in Figure 12.

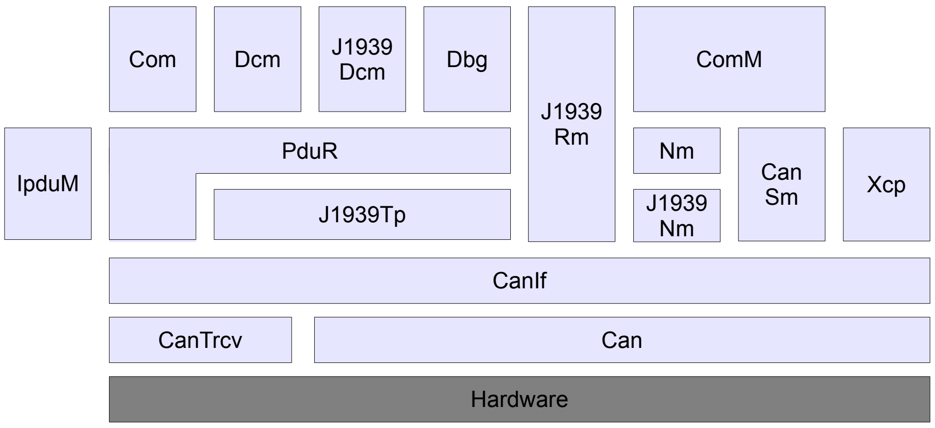


Figure 12: AUTOSAR – Communication Sub-Stack for CAN Networks for Heavy Duty Vehicles

In the following, the different modules of the communication sub-stack are described in detail.

**Driver (`<Net>`):** The Driver module (`Fr`, `Can`, `Lin`, and `Eth`) provides the basis for the respective Interface module by facilitating the *transmission and the reception of frames* via the respective CC. Hereby, the Driver is designed to handle multiple CCs of the same type. Thus, if an ECU contains for example FlexRay CCs of two different types, two different FlexRay Driver modules are required.

In case of CAN, the Driver module (`Can`) optionally provides the required support for TTCAN [14] and CAN FD [18] as well.

**Transceiver Driver (`<Net>Trcv`):** The Transceiver Driver module (i.e., `FrTrcv`, `CanTrcv`, `LinTrcv`, and `EthTrcv`) provides API functions for *controlling the transceiver hardware* (i.e., switching the transceivers into special modes (e.g., listen only or low power mode)) and for obtaining diagnostic information from the transceiver hardware (e.g., information about short circuits of the different bus-lines of CAN or information about wake-up events on the network).

**Interface (`<Net>If`):** Using the frame-based services provided by the Driver module, the Interface module (`FrIf`, `CanIf`, `LinIf`, and `EthIf`) facilitates *the sending and the reception of protocol data units (PDUs).* Hereby, multiple PDUs can be packed into a single frame at the sending ECU and have to be extracted again at the receiving ECU[5]. The point in time when this packing and extracting of PDUs takes place is governed by the temporal scheduling of so-called *communication jobs* of the FlexRay and the LIN Interface. The instant when the frames containing the packed PDUs are handed over to the Driver module for transmission or retrieved from the Driver module upon reception is triggered by communication jobs of the Interface module as well. In FlexRay, the schedule of these communication jobs is aligned with the communication schedule, in LIN, the schedule of the LIN Interface module governs the communication schedule on the LIN bus. In contrast to this in CAN, the temporal schedule of the PDU transmission is governed by the `Com` module (see below). In FlexRay, each communication job can consist of one or more *communication operations*, each of these communication operations handling exactly one communication frame (including the PDUs contained in this frame).

In case of CAN, the Interface module (`CanIf`) optionally provides the required support for TTCAN [14] and CAN FD [18] as well.

In case of Ethernet, the Interface module (`EthIf`) additionally provides support for virtual local area networks (VLANs) by taking care of the handling of the VLAN tags (i.e., tag protocol identifier (TPID) and tag control information (TCI)) and thus providing an abstraction from the distinction between VLAN and "normal" LANs to the upper layers.

All interface modules are designed to be able to deal with multiple different drivers for different types of CCs (e.g., freescale's MFR4300 or FlexRay CCs based on the BOSCH E-Ray core in the FlexRay case). Furthermore, the Interface module wraps the API provided by the Transceiver Driver module and provides support for multiple different Transceiver Driver modules (similar to the support for multiple different Driver modules).

**Transport Protocol (`<Net>Tp`):** The transport protocol is used to perform *segmentation and reassembly of large protocol data units (PDUs).* On CAN (`CanTp`) and FlexRay (`FrTp`) the used protocols are compatible (in certain configuration settings) to their respective ISO TP counterparts (see Section 4.1).

For FlexRay networks AUTOSAR specifies an alternative transport protocol (`FrArTp`) which is backwards compatible to the FlexRay transport protocol used in AUTOSAR 3.1 (but incompatible to the ISO TP for FlexRay).

In order to support the SAE J1939 standard and thus make AUTOSAR applicable for the use in heavy duty vehicles, an alternative transport protocol for CAN (`J1939Tp`) is provided as well, that adheres to the respective SAE J1939 specification [29].

[5]Currently only the FlexRay Interface module supports the packing of multiple PDUs into a single frame. – For the CAN, the LIN, and the Ethernet interface modules there is a 1:1 relationship between PDUs and frames.

A combination of `FrTp` and `FrArTp` or `CanTp` and `J1939Tp` on a single physical network is hereby not supported.

For the LIN communication network, AUTOSAR does not define a separate transport protocol module, but instead integrates the transport protocol functionality (which adheres to the LIN 2.1 specification [9]) into the LIN Interface (`LinIf`) module.

For the Ethernet communication network AUTOSAR decided to re-use already well-proven protocols, namely the internet protocol (IP), the internet control message protocol (ICMP), the address resolution protocol (ARP) for IP address resolution, the user datagram protocol (UDP) for unreliable connectionless communication, the transmission control protocol (TCP) for reliable connection oriented communication, and the dynamic host configuration protocol (DHCP) for automated IP address assignment. The functionality of these protocols is implemented in the `TcpIp` module. The Socket Adapter (`SoAd`) module, which is located on top of the `TcpIp` module maps AUTOSAR PDUs (which are identified by a unique PDU identifier) to the network endpoints of TCP connections and/or to the UDP datagrams (identified by a 4-tuple of local/remote IP address and port number) by means of static configuration tables. Hereby the `SoAd` abstracts from TCP specifics like the different methods for local IP address assignment (i.e., stateless address auto-configuration according to RFC 3927, DHCP according to RFC 2131, or simply by means of static configuration) and details regarding the connection setup and tear-down in order to provide an AUTOSAR PDU-based interface to the `PduR`. Similar to the packing of multiple PDUs into a single FlexRay frame in the `FrIf` for efficiency reasons, the `SoAd` allows for packing multiple PDUs into a single UDP datagram or TCP segment in order to achieve a decent header/payload ratio and to reduce the overhead for connection setup and tear-down. Especially for the exchange and relaying of diagnostic data between an external tester device the Diagnostics over IP (`DoIp`) module which is located above the `SoAd` implements the services specified by the ISO DoIP standard (see Section 4.3) and presents itself as conventional AUTOSAR transport protocol (i.e., like `CanTp` or `FrTp`) towards the `PduR` by providing the respective API.

Just like with ISO TP, the user of the services provided by the various transport protocol modules is the diagnostic layer, called Diagnostic Communication Manager in AUTOSAR (see below).

**Network Management (`<Net>Nm`, `Nm`):** Similar to OSEK NM (see Section 3) the AUTOSAR NM modules provide means for the coordinated transition of the ECUs in a network into and out of a low-power (or even power down) sleep mode. Hereby the "network" can be the whole *physical* network with all its connected ECUs or only a *logical* network consisting of a statically defined subset of ECUs of the physical network, a so-called *partial network*.

The AUTOSAR NM functionality is divided into two modules: a communication protocol independent module named Generic NM (`Nm`) and communication protocol dependent modules named FlexRay NM (`FrNm`), CAN NM (`CanNm`), LIN NM (`LinNm`), and UDP NM (`UdpNm`) for Ethernet networks.

In order to support the SAE J1939 standard and thus make AUTOSAR applicable for the use in heavy duty vehicles, an alternative network management protocol for CAN (`J1939Nm`) is provided as well, that adheres to the respective SAE J1939 specification [30]. `J1939Nm` however does not take care of a coordinated transition into and out of a low-power sleep mode, but handles the assignment of unique addresses to each

ECU instead. A combination of `CanNm` and `J1939Nm` on a single physical network is hereby not supported.

**State Manager (`<Net>SM`):** The State Manager modules (`CanSM`, `LinSM`, `FrSM`, and `EthSM`) facilitate the *state management of the respective communication controllers* with respect to communication system dependent startup and shutdown features and provide a common state machine API to the upper layer (i.e., the Communication Manager (`ComM`)). This API consists of functions for requesting the communication modes `Full`, `Silent` (i.e., listen only), and `No Communication`.

**XCP (`Xcp`):** For measurement and calibration purposes via CAN, FlexRay, and Ethernet networks, AUTOSAR includes an `Xcp` module in the communication stack, which implements the ASAM XCP specification (see Section 5.1).

**PDU Router (`PduR`):** The PDU Router module provides two major services. On the one hand, it *dispatches PDUs received via the underlying modules* (i.e., Interface and Transport Layer modules) to the different higher layers (`Com`, `Dcm`). On the other hand, the PDU router *performs gateway functionalities* between different communication networks by forwarding PDUs from one interface to another of either the same (e.g., FlexRay to FlexRay) or of different type (e.g., CAN to FlexRay). Routing decisions in the PDU Router are based on a static routing table and on the identifiers of the PDUs.

**PDU Multiplexer (`IpduM`):** The PDU Multiplexer module takes care of *multiplexing parts of a PDU*. Hereby, the value of a dedicated part of the PDU (the *multiplexer switch*) is used to define the semantic content of the remainder of the PDU (just like the tag element in a variant record or a union in programming languages). In the reception case, multiplexed PDUs are forwarded from the `PduR` to the `IpduM` for demultiplexing. Once demultiplexed, the `IpduM` hands the PDUs back to the `PduR`. In the sending case, the `PduR` obtains a PDU from `Com` and hands this PDU to the `IpduM` for multiplexing. The `IpduM` returns the multiplexed PDU to the `PduR`, which routes the multiplexed PDU to its final destination.

**Communication (`Com`):** Similar to OSEK Com (see Section 3) the `Com` module in AUTOSAR provides *signal-based communication* to the upper layer (`Rte`). The signal-based communication service of `Com` can be used for intra-ECU communication as well as for inter-ECU communication. In the former case, `Com` mainly uses shared memory for this intra-ECU communication whereas for the latter case at the sender side `Com` packs multiple signals into a PDU and forwards this PDU to the `PduR` in order to issue the PDU's transmission via the respective Interface module. – On the receiver side, `Com` obtains a PDU from the PDU router, extracts the signals contained in the PDU, and forwards the extracted signals to the upper software layer (`Rte`).

**Diagnostic Communication Manager (`Dcm`):** The Diagnostic Communication Manager module is a sub-module of the AUTOSAR diagnostic module. The `Dcm` module provides *services which allow a tester device to control diagnostic functions* in an ECU via the communication network (i.e., CAN, LIN, FlexRay). Hereby the `Dcm` supports the diagnostic protocols UDS and OBD (see Section 4.2 for details).

In order to support the SAE J1939 standard and thus make AUTOSAR applicable for the use in heavy duty vehicles, an additional diagnostic communication manager module (`J1939Dcm`) is provided that implements the diagnostic services mandated by the respective SAE J1939 specification [28]. Since the services provided by the `J1939Dcm` module are additions to the functionality provided by `Dcm` module, both modules can be used within a single ECU.

Both the `Dcm` and the `J1939Dcm` module act as *service SWCs* (see Section 6.1.1) towards the `Rte` and thus make their services available to (application) SWCs via dedicated ports (see Section 6.1.2).

**Communication Manager (`ComM`):** The Communication Manager is a resource manager which *encapsulates the control of the underlying communication services.* The `ComM` collects network communication access requests from communication requesters (e.g., `Dcm`) and coordinates these requests by interacting with `Nm` and the respective State Manager (`<Net>SM`) modules. This way the `ComM` provides a simplified API to the network management where a user of the API does not require any knowledge of the particular physical communication network (or partial network) to use. Via the `ComM` API, a user simply requests a specific communication mode (i.e., `Full`, `Silent`, or `No Communication`) and the `ComM` switches (based on a statically configured table mapping users to networks) the communication capability of the corresponding physical or partial networks to `On`, `Silent`, or `Off`.

The `ComM` module acts as a *service SWC* (see Section 6.1.1) towards the `Rte` and thus makes its services available to (application) SWCs via dedicated ports (see Section 6.1.2).

**J1939 Request Manager (`J1939Rm`):** The J1939 Request Manager takes care of the SAE J1939 specific feature that the transmission of specific PDUs (termed parameter group number (PGN) in J1939) can explicitly be requested by dedicated request PDU [29]. The `J1939Rm` module receives this kind of request PDUs and based on the contained information triggers the transmission of the requested PDU.

The `J1939Rm` module acts as a *service SWC* (see Section 6.1.1) towards the `Rte` and thus makes its services available to (application) SWCs via dedicated ports (see Section 6.1.2).

**Debugging (`Dbg`):** The AUTOSAR Debugging module supports the remote debugging process of a other AUTOSAR modules. For this purpose, the `Dbg` module collects information like function calls/returns and values/value changes of important variables of other modules during the runtime of the ECU without halting the processor. This on the one hand requires proper instrumentation of the debugged modules (e.g., the `Dbg` module must be notified upon function entry and exit) and on the other hand mandates that the relevant variables are accessible to the `Dbg` module. The collected data is then transmitted (either immediately or upon request) to an external debugging host system for further inspection.

**Service Discovery (`Sd`):** The Service Discovery module provides means for a *service requester* (i.e., an ECU that requires a specific service) to detect available services which are offered by *service provides* within the vehicle network. Additionally the `Sd` module facilitates to control the send behavior of event information in a way that an *event publisher* (i.e., an ECU sending the event information) only sends this information to the dynamically registered *event subscribers* (i.e., ECUs that want to receive this event information).

### 6.2.2 Memory Sub-Stack

The memory sub-stack contains a group of modules that facilitate handling of the ECU's on-board non-volatile memory (i.e., providing API functions to store data in and retrieve data from the ECU's non-volatile memory (e.g., electrically eraseable programmable read-only memory (EEPROM) or flash EEPROM). The structure of the memory sub-stack is depicted in Figure 13.

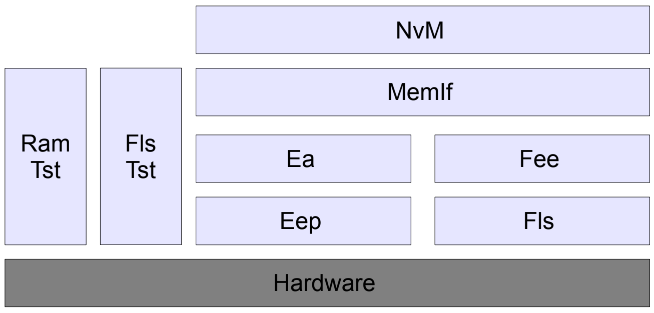


Figure 13: AUTOSAR – Memory Sub-Stack

In the following the different modules of the memory sub-stack are described in detail.

**Flash Driver (`Fls`):** The Flash Driver provides services for *reading from, writing to, and erasing parts of the ECU's flash memory.* Furthermore, the Flash Driver facilitates the setting and resetting of the write/erase protection of the flash EPPROM if such protection is supported by the underlying hardware. In addition to these basic services, the Flash Driver provides a service for comparing a data block in the flash EEPROM with a data block in the memory (e.g., RAM).

**EEPROM Driver (`Eep`):** The EEPROM Driver provides services for *reading from, writing to, and erasing parts of the ECU's EEPROM.* Additionally, similar to the Flash Driver, the EEPROM Driver provides a service for comparing a data block in the EEPROM with a data block in the memory (e.g., RAM).

**Flash EEPROM Emulation (`Fee`):** The Flash EEPROM Emulation module *emulates EEPROM functionality* using the services provided by Flash Driver module. By making use of multiple flash sectors and smart copying of the data between these sectors, the `Fee` simulates an EEPROM-like behavior, i.e., the possibility to perform program/erase operations on sub-sector granularity.

**EEPROM Abstraction (`Ea`):** The EEPROM Abstraction module *provides uniform mechanisms to access the ECU's internal and external EEPROM devices.* It abstracts from the location of peripheral EEPROM devices (including their connection to the microcontroller), the ECU hardware layout, and the number of EEPROM devices.

**Memory Abstraction Interface (`MemIf`):** The Memory Abstraction Interface module allows the Non-Volatile RAM Manager module (see below) to *access several memory abstraction modules* (`Fee` or `Ea` modules) *in a uniform way.* Hereby the `MemIf` abstracts from the number of underlying `Fee` or `Ea` modules providing a runtime translation of each block access initiated by the Non-Volatile RAM Manager module to select the corresponding driver functions which are unique for all configured EEPROM or flash EEPROM storage devices.

**Non-Volatile RAM Manager (`NvM`):** The Non-Volatile RAM Manager module provides services to *ensure the data storage and maintenance of non-volatile data* according to their individual requirements in an automotive environment, namely synchronous as well as asynchronous services for the initialization, the reading, the writing and the control of non-volatile data. The `NvM` operates on a block basis distinguishing the following types of blocks: For *native blocks* the `NvM` provides a *RAM mirror* which contains a copy of the data stored in the non-volatile memory block. This RAM mirror is initialized with the data from the non-volatile block upon ECU power-up. Upon ECU shutdown, the data from the RAM mirror is flushed to the corresponding non-volatile memory block. Additionally, the `NvM` provides API services which can force the transfer of a memory block from non-volatile memory into the corresponding RAM mirror and vice-versa. In addition to the facilities of native blocks, *redundant blocks* provide enhanced fault tolerance, reliability, and availability. Due to replication of the redundant block in non-volatile memory, the resilience against data corruption is increased.

The `NvM` module acts as a *service SWC* (see Section 6.1.1) towards the `Rte` and thus makes its services available to (application) SWCs via dedicated ports (see Section 6.1.2).

**RAM Test (`RamTst`):** The RAM Test module provides means to perform *functional tests of the ECU's internal RAM cells. Complete* tests are performed upon ECU startup and shutdown as well as on request by special diagnostic commands. During operation *partial test* are performed in a periodic manner (e.g., block by block or cell by cell). For both types of tests several RAM test algorithms, which have been chosen according to the IEC 61508 standard, are available. Depending on the algorithms' diagnostic coverage rate, the algorithms are divided into the following categories: Group 1 (low) with a diagnostic coverage rate smaller than 60%, group 2 (medium) exhibiting a diagnostic coverage rate of 60% to 90%, and group 3 (high) with a diagnostic coverage rate of 90% to 99%.

**Flash EEPROM Test (`FlsTst`):** The Flash EEPROM Test module provdies means to perform *functional tests on the content of the ECU's non-mutable memory cells (e.g., program/data Flash EEPROM content). Foreground tests* are performed on a per-block basis upon request via the module's API functions. *Background test* are performed in a periodic manner for a set of configured blocks. Both kind of tests compute a configurable signature over the memory block (e.g., 8, 16, and 32 bit CRCs or hashes) and compare the computed signature value with a pre-computed value stored in the `FlsTst` module's configuration.

### 6.2.3 Input/Output Sub-Stack

The input/output sub-stack contains a group of modules that facilitate the handling of the ECU's input/output capabilities. The structure of the input/output sub-stack is depicted in Figure 14.

In the following the different modules of the input/output sub-stack are described in detail.

**Port Driver (`Port`):** The Port Driver module provides the service for *initializing the whole port structure* of the micro-controller, allowing for the the configuration of different functionalities for each port and port pin (e.g., analogue digital conversion (ADC),

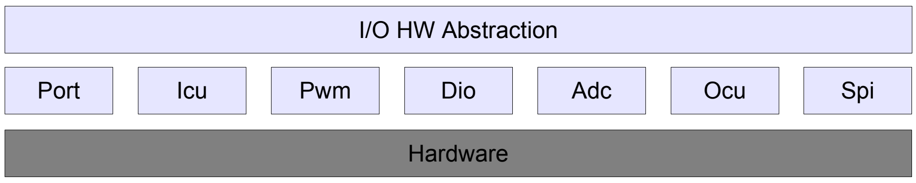


Figure 14: AUTOSAR – Input/Output Sub-Stack

digital I/O (DIO), ...). Hereby the port pin direction (input/output), the initial level of the port pin, and the fact whether the port pin direction is modifiable during runtime is part of this configuration. Other I/O drivers (e.g., `Icu`, `Pwm`, ...) rely on the configuration performed by the Port Driver.

**Interrupt Capture Unit Driver (`Icu`):** The Interrupt Capture Unit (ICU) Driver is a *module using the ICU hardware* to implement services like signal edge notification, controlling of wake-up interrupts, periodic signal time measurement, edge time stamping (usable for the acquisition of non-periodic signals), and edge counting. Hereby the `Icu` module works on pins and ports which have been properly configured by the Port Driver for this purpose.

**Pulse Width Modulation Driver (`Pwm`):** The Pulse Width Modulation (PWM) Driver module provides functions to *initialize and control the hardware PMW unit* of the micro-controller. The `Pwm` module allows for the generation of pulses with variable pulse width by facilitating the selection of the duty cycle and the signal period time. The `Pwm` module supports multiple PWM channels, where each channel is linked to a hardware PWM unit which belongs to the micro-controller. Similar to the `Icu` module the `Pwm` module relies on pins and ports which have been properly configured by the Port Driver for this purpose.

**Digital I/O Driver (`Dio`):** The Digital I/O (DIO) Driver provides services for *reading from and writing to DIO channels* (i.e., port pins), DIO ports, and groups of DIO channels. Hereby the `Dio` module works on pins and ports which have been properly configured by the Port Driver for this purpose.

**Analogue/Digital Converter Driver (`Adc`):** The Analogue to Digital Converter (ADC) Driver module *initializes and controls the internal ADC unit(s)* of the micro-controller. The module provides services to start and stop an analog to digital conversion respectively to enable and disable the trigger source for a conversion. Furthermore the module provides services to enable and disable a notification mechanism and routines to query status and result of a conversion. The `Adc` module works on so-called ADC channel groups. An ADC channel group combines an ADC channel (i.e., an analogue input pin), the needed ADC circuitry itself, and a conversion result register into an entity that can be individually controlled and accessed via the `Adc` module. The `Adc` module operates on pins and ports which have been properly configured by the Port Driver for this purpose.

**Output Compare Unit Driver (`Ocu`):** The Output Compare Unit (OCU) Driver module provides functions to *initialize and control the hardware OCU unit* of the micro-controller. The `Ocu` module allows comparing and acting automatically (e.g., by calling a notification function) when the value of a free-running counter matches a defined threshold. The module provides services to set the comparison threshold value, to start and stop a comparison, to enable and disable a notification mechanism, and to query the current counter value. The Ocu module works on so-called OCU channels. An OCU channel group combines a free-running counter and the corresponding threshold value into an entity that can be individually controlled and accessed via the Ocu module. The Ocu module relies on pins and ports which have been properly configured by the Port Driver for this purpose.

**Standard Peripheral Interface Handle/Driver (Spi):** The Standard Peripheral Interface (SPI) Handler/Driver module provides services for reading from and writing to peripheral devices connected via SPI buses. These peripheral devices are explicitly addressed by means of a chip select (CS) line. For each peripheral device a so-called *SPI job* consisting of multiple *SPI channels* can be configured in the Spi module's configuration. SPI channels hereby basically define the atomic units of data transferred between the micro-controller and a peripheral device. SPI jobs are grouped into *SPI sequences* to specify a defined order of data exchange.

Based on these SPI sequences Spi module provides services to synchronously and asynchronously conduct data transfers, to query the status of such a transfer on SPI sequence and SPI job granularity, and notification services upon completion of a particular SPI sequence and/or a particular SPI job.

**I/O Hardware Abstraction:** The I/O Hardware Abstraction module provides a signal-based interface to internal and external I/O devices of an ECU. Hereby the module abstracts from whether a certain I/O device is an MCU internal device, or whether a device is externally connected to the MCU, by performing static normalization/inversion of values according to their physical representation at the inputs/outputs of the ECU hardware (i.e., static influences, like voltage division or hardware inversion, on the path between the I/O device and the MCU port pin are compensated).

The I/O Hardware Abstraction module acts as an *ECU abstraction SWC* (see Section 6.1.1) towards the Rte and thus makes its services available to (application) SWCs via dedicated ports (see Section 6.1.2).

### 6.2.4 System Services Sub-Stack

The system services sub-stack contains a group of modules that can be used by modules of all AUTOSAR layers. Examples are real-time operating system, error handler, and watchdog management. The structure of the system services sub-stack is depicted in Figure 15.

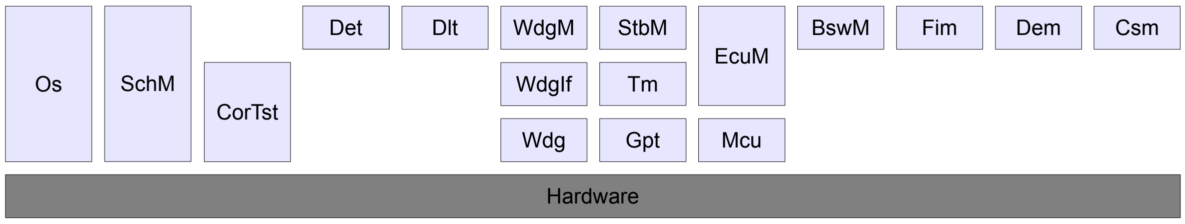


Figure 15: AUTOSAR – System Services Sub-Stack

In the following the different modules of the system services sub-stack are described in detail.

**Operating System (Os):** The AUTOSAR Operating System provides *real-time operating system services* to both the other system software modules as well as to the application software components of AUTOSAR. The `Os` module is configured and scaled statically, provides a priority-based scheduling policy and protective functions with respect to memory and timing at runtime, and is designed to be hostable on low-end controllers.

Similar to the OSEKtime dispatcher tables (see Section 3), AUTOSAR `Os` provides so-called *schedule tables* consisting of one or more *expiry points*. Hereby each expiry point is assigned an offset measured in `Os` ticks from the start of the schedule table. Once an expiry point is reached, the action corresponding to the expiry point (e.g., the activation of a task or the setting of an event) is processed. At runtime, the `Os` iterates over the schedule table, processing each expiry point in turn. The iteration is driven by an `Os` counter. In order to facilitate the execution of tasks synchronous to external events (e.g., synchronous to the FlexRay communication schedule), schedule tables can be synchronized to external time sources (e.g., FlexRay's global time).

As far as protection against timing violations is concerned, AUTOSAR `Os` does not provide deadline monitoring (as does OSEKtime OS) but provides the facility to track the execution time of each task and interrupt service routine and to raise an error in case either exceeds its statically assigned execution time budgets. Regarding memory protection, AUTOSAR `Os` uses the memory protection unit of the MCU to provide coarse grained memory protection of so-called *OS applications*, which is a grouping of related tasks, against each other.

With AUTOSAR 4.0 onward the AUTOSAR `Os` provides support for multi-core MCUs by allowing different OS applications to be statically allocated to the different cores and by supporting data exchange among these cores by means of the Inter-OS Application Communicator (`Ioc`) sub-module.

The `Os` module acts as a *service SWC* (see Section 6.1.1) towards the `Rte` , thus making its services available to (application) SWCs via dedicated ports (see Section 6.1.2).

**Basic Software (BSW) Scheduler (`SchM`):** The BSW Scheduler module provides means to embed other AUTOSAR system software module implementations into the context of an AUTOSAR `Os` task or interrupt service routine, trigger main processing functions of the system software modules, and apply data consistency mechanisms for these modules. Just like the `Rte` provides the infrastructure for software components by embedding runnable entities in a task context, the `SchM` module *provides the infrastructure for the other system software modules* by embedding their main processing functions (termed schedulable entities) in a task context[6].

Due to the rather similar functionality between `Rte` and `SchM` and to allow for global scheduling and data consistency optimizations (e.g., automatic elision of data consistency mechanisms in case the involved runnable and schedulable entities cannot preempt one another) the `SchM` module's functionality is integrated into the `Rte` with AUTOSAR 4.0 onward.

**Micro Controller Unit (MCU) Driver (`Mcu`):** The MCU Driver module provides services for *basic micro-controller initialization*, *power-down functionality*, *micro-controller reset* and micro-controller specific functions required from other system software modules. The initialization services of the MCU Driver module allow a flexible and application related MCU initialization in addition to the startup code[7]. The services of the

[6]Usually the main processing functions of multiple system software modules are embedded into a single task in order to keep the number of tasks required for execution of the whole AUTOSAR system software low.

[7]The startup code itself is not within the scope of AUTOSAR.

MCU Driver include the initialization of the MCU's clock, initialization of the MCU's phase-locked loop (PLL), the initialization of clock pre-scalers, and the configuration of the MCU's clock distribution. Furthermore the MCU Driver takes care of the initialization of the MCU's RAM sections, facilitates the activation of the MCU's reduced power modes (i.e., putting the MCU into a low-power mode), and provides a service for enforcing a reset of the MCU and a service for obtaining the reset reason from the MCU hardware.

**ECU State Manager (`EcuM`):** The ECU State Manager module manages all aspects of the ECU related to the `Off`, `Run`, and `Sleep` states of that ECU and the transitions between these states like *startup and shutdown.* In detail, the ECU state manager is responsible for the initialization and de-initialization of all system software modules including `Os` and `Rte`, cooperates with the `ComM`, and hence indirectly with `Nm`, to shut down the ECU (`Off` state) when needed, manages all wakeup events, and configures the ECU for `Sleep` state when requested. In order to fulfill all these tasks, the `EcuM` makes use of the services provided by the `Mcu` module and implements some important protocols: the *run request protocol*, which is needed to coordinate whether the ECU must be kept alive or is ready to shut down, the *wake-up validation protocol* to distinguish "real" wake-up events from "erratic" ones, and the *time-triggered increased inoperation protocol* which allows to put the ECU into an increasingly energy saving `Sleep` state depending on the duration of the ECU's inactivity.

With AUTOSAR 4.0 a distinction between an ECU State Manager module with *fixed state machine* (`EcuMfixed`) and an ECU State Manager module with *flexible state machine* (`EcuMflex`) has been made. While the former one implements all of the above services by itself in a rather fixed way, the latter one focuses on the early startup and late shutdown phases and initializes only a small number of other system software modules, delegating the rest of the functionality to the Basic Software Mode Manager (`BswM`) module.

The `EcuM` module (flex and fixed) acts as a *service SWC* (see Section 6.1.1) towards the `Rte` and thus makes its services available to (application) SWCs via dedicated ports (see Section 6.1.2).

**Basic Software Mode Manager (`BswM`):** The Basic Software Mode Manager module arbitrates requests from application SWCs or other system software modules based on statically configurable rules, and performs actions based on the arbitration result. Hereby these configurable rules comprise one or more logical conditions which are combined into a logical expression and a list of actions that are executed in case the logical expression evaluates to true or to false respectively. The logical conditions can be mode switches performed and announced by other modules and/or SWCs. The performed actions can be mode switches performed by the `BswM` itself which are announced towards other modules and/or SWCs. Thus the `BswM` on the one hand acts as a mode user and on the other hand as a mode manager.

Additionally the `BswM` is in charge of the part of initialization and shutdown process that is not covered by the `EcuMflex` module.

The `BswM` module acts as a *service SWC* (see Section 6.1.1) towards the `Rte` and thus makes its services available to (application) SWCs via dedicated ports (see Section 6.1.2).

**Diagnostic Event Manager (`Dem`):** The Diagnostic Event Manager module realizes part of the diagnostic functionality within AUTOSAR. The `Dem` is responsible for *processing and persistently storing diagnostic events/errors*[8] and associated data (so-called freeze frame data). To facilitate the persistent storage of these DTCs, the `Dem` makes use of the services provided by the `NvM`. Application software components as well as other system software modules can raise diagnostic events by means of `Dem` API calls.

The diagnostic events registered by the `Dem` serve as triggers for state updates of the `Fim` and thus might lead to the inhibition of certain runnables. Upon request of the `Dcm`, the `Dem` provides an up-to-date list of the currently stored DTCs which are then sent to a tester client by means of the `Dcm` services.

The `Dem` module acts as a *service SWC* (see Section 6.1.1) towards the `Rte` and thus makes its services available to (application) SWCs via dedicated ports (see Section 6.1.2).

**Function Inhibition Manager (`Fim`):** Like the `Dem` and the `Dcm` module, the Function Inhibition Manager realizes part of the diagnostic functionality in AUTOSAR. The `Fim` is responsible for providing an *execution control mechanism for the runnables* of application software components and system software modules. By means of the `Fim`, these runnables can be inhibited (i.e., deactivated) according to the `Fim`'s static configuration. The functionalities of the runnables are assigned to a unique function identifier (FID) along with an inhibit condition for that particular FID. The functionalities poll for the permission state of their respective FIDs before execution. If an inhibit condition is true for a particular FID, the corresponding functionality is not executed anymore.

The `Fim` is closely related to the `Dem` since diagnostic events and their status information can serve as possible inhibit conditions. Hence, functionality which needs to be stopped in case of a failure can be represented by a particular FID. If the failure is detected and the event is reported to the `Dem`, the `Fim` then inhibits the FID and therefore the corresponding functionality.

The `FiM` module acts as a *service SWC* (see Section 6.1.1) towards the `Rte` and thus makes its services available to (application) SWCs via dedicated ports (see Section 6.1.2).

**Watchdog Driver (`Wdg`):** This module provides services for initialization, changing of the operation mode (`Fast`, `Slow`, `Off`) and *triggering the ECU's watchdog device.* In case an ECU provides multiple different watchdog devices (e.g., internal and external devices), a dedicated `Wdg` module has to be present for each of the devices.

**Watchdog Interface (`WdgIf`):** In case of more than one watchdog device and corresponding Watchdog Driver (e.g., both an internal software watchdog and an external hardware watchdog) is being used on an ECU, the Watchdog Interface module allows the Watchdog Manager module (see below) to select the correct Watchdog Driver – and thus the watchdog device – via a device index, while retaining the API and functionality of the underlying driver. Thus the `WdgIf` module provides *uniform access to services of the underlying Watchdog Drivers* like mode switching and triggering.

**Watchdog Manager (`WdgM`):** The Watchdog Manager module is intended to *supervise the program execution* of application software components or other system software modules, so-called supervised entities. The `WdgM` provides services for *alive supervision* for

[8]These events can be mapped to UDS/OBD diagnostic trouble code (DTCs) by means of the `Dem`'s module configuration.

periodic software, *deadline supervision* for aperiodic software, and *logical supervision* monitoring of the program flow of a supervised entity. Hereby the `WdgM` monitors each supervised entity autonomously, derives a local status for this supervised entity. The aggregation of the local stati of all supervise entities then yields a global status. Based on this global supervision status, the `WdgM` decides whether or not to trigger the hardware watchdog via the `WdgIf`'s API. Hereby, the set of supervised clients, the desired degree of supervision, and the individual timing constraints are defined by configurable parameters of the `WdgM`.

The `WdgM` module acts as a *service SWC* (see Section 6.1.1) towards the `Rte` and thus makes its services available to (application) SWCs via dedicated ports (see Section 6.1.2).

**Development Error Tracer (`Det`):** The Development Error Tracer module is the central instance where all other system software modules *report detected development errors* to. The API parameters handed to the `Det` allow for tracing source and kind of error, namely the module and the function in which the error has been detected and the type of the error. The functionality behind the API of the `Det` is not specified in AUTOSAR. Possible functionalities could be the setting of debugger breakpoints within the error reporting API, the counting of the number of reported errors, the logging of `Det` calls together with the passed parameters to a RAM buffer for later analysis, and the sending of reported errors via some communication interface (e.g., `CanIf`) to external logger devices.

The `Det` module acts as a *service SWC* (see Section 6.1.1) towards the `Rte` and thus makes its services available to (application) SWCs via dedicated ports (see Section 6.1.2).

**Diagnostic Log and Trace (`Dlt`):** The Diagnostic Log and Trace module provides services for the logging of errors, warnings and information messages to SWCs, as well as to the `Det` and the `Dem` module. Additionally the `Dlt` modules provides trace functionality to the `Rte` to facilitate the tracing of relevant `Rte` events (see Section 6.1.4). The detail of both the logging and the tracing can hereby be controlled during runtime. The logged/traced data is then transmitted via the `PduR` or by means of the data transmission services of the `Dcm`.

The `Dlt` module acts as a *service SWC* (see Section 6.1.1) towards the `Rte` and thus makes its services available to (application) SWCs via dedicated ports (see Section 6.1.2).

**General Purpose Timer Driver (`Gpt`):** The General Purpose Timer (GPT) Driver module provides services for starting and stopping a functional timer instance within the hardware general-purpose timer module and thus *provides exact and short-term timings* for use in the `Os` or within other system software modules where an `Os` alarm service causes too much overhead. Individual timeout periods (*single shot mode*) as well as repeating timeout periods (*continuous mode*) can be generated via the `Gpt` module. The user can configure whether a notification shall be invoked when the requested timeout period has expired. These notifications can be enabled and disabled at runtime. Both, the relative time elapsed since the last notification occurred and the time remaining until the next notification will occur, can be queried via API functions of the `Gpt` module. Additionally the `Gpt` module provides a set of free-running timers together with API functions to obtain the current time value of these free-running timers.

**Time Service (`Tm`):** The Time Service module builds on the free-running timers of the `Gpt` module and provides a set of *pre-defined timers with defined tick duration in physical time units* (e.g., microseconds) together with a set of API functions to reset these timers, to (busy) wait for a guaranteed minimum waiting time, to set the timers to a reference time, and to compute the time difference between the current time and this reference time. This way compatibility of time-based functionality is ensured for all platforms which support the required pre-defined timers.

**Synchronized Time Base Manager (`StbM`):** The Synchronized Time Base Manager *provides a "global time"* to other system software modules or to the application SWCs (so-called *time base users*). Hereby the `StbM` itself however does not provide any facility (e.g., synchronization protocols) for establishing a synchronized time base across multiple ECUs, but relies on the existence of such protocols at so-called *time base providers* (e.g., the `FrIf` which provides a synchronized time base for all ECUs connected to the specific FlexRay network). To facilitate a triggering of runnable entities of SWCs or schedulable entities of other system software modules that is synchronous to this global time base, the `StbM` is capable of synchronizing the schedule tables of the `Os` to this global time base.

The `StbM` module acts as a *service SWC* (see Section 6.1.1) towards the `Rte` and thus makes its services available to (application) SWCs via dedicated ports (see Section 6.1.2).

**Crypto Service Manager (`Csm`):** The Crypto Service Manager module provides *cryptographic services* based on a software library or on a dedicated hardware module. These services include means to compute and verify checksums and hash values over a block of data elements, means to generate random numbers, means for symmetric and asymmetric en- and decryption of blocks of data, message authentication and integrity checks by means of digital signatures, derivation of one or more secret keys using a key derivation function, wrapping, serialization, and de-serialization of keys for transportation purposes, and generation and secure exchange of shared keys.

The `Csm` module acts as a *service SWC* (see Section 6.1.1) towards the `Rte` and thus makes its services available to (application) SWCs via dedicated ports (see Section 6.1.2).

**Core Test (`CorTst`):** The Core Test module provides means to perform *functional tests related to basic functionality of the MCU's core(s)*, namely tests verifying the correct functionality of the core's registers, the arithmetic logical unit (ALU), the interrupt controller and the exception handling, the memory interface, the memory protection unit (MPU) in case one is available, and the cache controller (w.r.t., cache coherency and consistency). Similar to the `RamTst` module, the `CorTst` module supports *foreground tests* which are performed upon request via the module's API functions and *background test* which are performed in a periodic manner.

### 6.2.5 Libraries

In addition to the previously described system software modules, AUTOSAR defines a set of libraries. These libraries are neither assigned to any particular software layer nor to a particular sub-stack. Instead *all* layers (including the SWCs of the application layer) as well as integration code are allowed to make use of these libraries. To facilitate this flexible use AUTOSAR defines that the code of libraries

- is executed synchronously in the context of the caller in the same protection environment
- is only allowed to call other library code (i.e., library code must not call the code of system software modules or SWCs)
- is re-entrant and does not have any internal state
- does not require any initialization

As far a the provided functionality if concerned, AUTOSAR specifies libraries for fixed point mathematics, floating point mathematics, interpolation routines for fixed point data, interpolation routines for floating point data, bit handling routines, end-to-end communication protection, CRC calculation, filtering routines, and cryptographical routines.

In the following a sub-set of the libraries provided by AUTOSAR is discussed in more detail:

**Cyclic Redundancy Check (`Crc`):** The Cyclic Redundancy Check library provides bitwise, table-based, and (if available) hardware-assisted computation of cyclic redundancy checks (CRCs) using different 8, 16, and 32 bit generator polynomials, namely the SAE-J1850 CRC8, the CCITT-FALSE CRC16, and the IEEE-802.3 CRC32 used for Ethernet.

**End-to-End Protection (`E2e`):** The End-to-End Protection library provides means to protect safety-related data exchange at runtime against the effects of faults along the communication path, like random HW faults (e.g., corrupt registers of a communication controller), interference (e.g., due to EMC), and systematic faults within the communication sub-stack. To this end additional control fields like CRCs or sequence counters are added to the transmitted data at the sender side. At the receiver side these control fields are evaluated and checked for correctness. In case this check fails, the received data is marked as corrupt. In order to support the use with different fault hypotheses, multiple profiles which differ in the added control fields (e.g., the strength of the used CRC) are supported.

**Crypto Abstraction (`Cal`):** From the functional point of view, the Crypto Abstraction library provides the same services as the `Csm` module. The only difference is the fact that the `Cal` is implemented as a library adhering to all the restrictions AUTOSAR imposes on the implementation of libraries (see above).

# 7 Summary

Since 1993 the major automotive companies are striving for the deployment of standard software modules in their applications to achieve an increased test coverage and higher reliability, requirements that can only be met if standardized modules are used at various system levels.

This chapter provided an overview of today's industry practices in standardized automotive system software. Existing standards proposed by industry partnerships like OSEK/VDX, ASAM, and AUTOSAR, and by standardization authorities like ISO have been presented. Of all presented approaches, the AUTOSAR partnership which started off in May 2003 and started putting software modules according to the AUTOSAR standard into production vehicles in 2008 (BMW being first to deploy AUTOSAR in its 7 series car followed by Daimler, Audi, VW, PSA and others) turns out to be the most promising one.

This is on the one hand due to the fact that several of already approved standards used today (like OSEK OS and COM, ASAM XCP, and ISO transport layer and diagnostics) heavily inspired the corresponding AUTOSAR standards, ensuring that the AUTOSAR standard is built on well-proven technology. On the other hand the AUTOSAR open standard has massive industrial backup: All AUTOSAR core members including seven of the world's biggest vehicle manufacturers accounting for about 55% of all vehicles produced are strongly committed to the project and released the first revision of the AUTOSAR 4.1 specifications in March 2013, which will serve as a solid basis for future automotive software development.

## References

[1] ASAM – Association for Standardization of Automation and Measuring Systems. Project Web Page.

[2] EASIS – Electronic Architecture and System Engineering for Integrated Safety Systems. Project Web Page.

[3] GENIVI Alliance. Alliance Web Page.

[4] HIS – Herstellerinitiative Software. Project Web Page.

[5] JASPAR – Japan Automotive Software Platform Architecture. Consortium Web Page.

[6] OSEK/VDX. Project Web Page.

[7] V. Barthelmann, A Schedl, E. Dilger, T. Führer, B. Hedenetz, J. Ruh, M. Kühlewein, E. Fuchs, Y. Domaratsky, A. Krüger, P. Pelcat, M. Glück, S. Poledna, T. Ringler, B. Nash, and T. Curtis. OSEK/VDX – Time-Triggered Operating System, Version 1.0. Technical report, OSEK, July 2001.

[8] AUTOSAR Consortium. AUTOSAR – Layered Software Architecture. Technical Report Version 3.3.0, Release 4.1, Rev 1, AUTOSAR Consortium, January 2013.

[9] LIN Consortium. LIN Specification Package. Technical Report Version 2.1, LIN Consortium, November 2006.

[10] J. Spohr et. al. OSEK/VDX – Communication, Version 3.0.3. Technical report, OSEK, July 2004.

[11] J. Spohr et. al. OSEK/VDX – System Generation – OIL: OSEK Implementation Language, Version 2.5. Technical report, OSEK, July 2004.

[12] H. Fennel, S. Bunzel, H. Heinecke, J. Bielefeld, S. Fürst, K.-P. Schnelle, W. Grote, N. Maldener, T. Weber, F. Wohlgemuth, J. Ruh, L. Lundh, T. Sandén, P. Heitkämper, R. Rimkus, J. Leflour, A. Gilberg, U. Virnich, S. Voget, K. Nishikawa, K. Kajio, K. Lange, T. Scharnhorst, and B. Kunkel. Achievements and Exploitation of the AUTOSAR Development Partnership. In *Proceedings of the Convergence 2006*, number SAE 2006-21-0019, Detroit, MI, USA, October 2006.

[13] T. Führer, F. Hartwich, R. Hugel, and H. Weiler. FlexRay – The Communication System for Future Control Systems in Vehicles. In *Proceedings of the SAE 2003 World Congress & Exhibition*, number SAE 2003-01-0110, Detroit, MI, USA, March 2003. Society of Automotive Engineers.

[14] T. Führer, B. Müller, W. Dieterle, F. Hartwich, R. Hugel, and M. Walther. Time Triggered Communication on CAN. In *Proceedings of the 7th International CAN Conference, CAN in Automation*, Amsterdam, Netherlands, 2000.

[15] S. Fürst. Challenges in the Design of Automotive Software. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 256–258, Leuven, Belgium, 2010. European Design and Automation Association.

[16] J. Goodenough and L. Sha. The Priority Ceiling Protocol: A Method for Minimizing the Blocking of High-Priority Ada Tasks. Technical Report SEI-SSR-4, Software Engineering Institute, Pittsburgh, Pennsylvania, USA, May 1988.

[17] P. Hansen. AUTOSAR Standard Software Architecture Partnership Takes Shape. *The Hansen Report on Automotive Electronics*, 17(8):1–3, October 2004.

[18] F. Hartwich. CAN with Flexible Data-Rate. In *Proceedings of the 13th International CAN Conference, CAN in Automation*, Hambach, Germany, 2012.

[19] C. Hoffmann, J. Minuth, J. Krammer, J. Graf, K. J. Neumann, F. Kaag, A. Maisch, W. Roche, O. Quelenis, E. Farges, P. Aberl, D. John, L. Mathieu, M. Schütze, D. Gronemann, and J. Spohr. OSEK/VDX – Network Management – Concept and Application Programming Interface, Version 2.5.3. Technical report, OSEK, July 2004.

[20] ISO. Road Vehicles – Controller Area Network (CAN) – Part 1: Data Link Layer and Physical Signalling. Technical Report ISO 11898-1:2003, ISO (International Organization for Standardization), 1, rue de Varembe, Case postale 56, CH-1211 Geneva 20, Switzerland, 2003.

[21] ISO. Road Vehicles – Controller Area Network (CAN) – Part 2: High-Speed Medium Access Unit. Technical Report ISO 11898-2:2003, ISO (International Organization for Standardization), 1, rue de Varembe, Case postale 56, CH-1211 Geneva 20, Switzerland, 2003.

[22] ISO. Road Vehicles – Communication On FlexRay – Part 2: Communication Layer Services. Technical Report ISO 10681-2:2010, ISO (International Organization for Standardization), 1, rue de Varembe, Case postale 56, CH-1211 Geneva 20, Switzerland, June 2010.

[23] ISO. Road Vehicles – Diagnostics on Controller Area Networks (DoCAN) – Part 2: Transport Protocol and Network Layer Services. Technical Report ISO 15765-2:2011, ISO (International Organization for Standardization), 1, rue de Varembe, Case postale 56, CH-1211 Geneva 20, Switzerland, 2011.

[24] ISO. Road Vehicles – Diagnostic Communication over Internet Protocol (DoIP) – Part 2: Transport Protocol and Network Layer Services. Technical Report ISO 13400-2:2012, ISO (International Organization for Standardization), 1, rue de Varembe, Case postale 56, CH-1211 Geneva 20, Switzerland, 2012.

[25] ISO. Road Vehicles – Unified Diagnostic Services (UDS) – Part 1: Specification and Requirements. Technical Report ISO 14229-1:2013, ISO (International Organization for Standardization), 1, rue de Varembe, Case postale 56, CH-1211 Geneva 20, Switzerland, 2013.

[26] L. Lundth. About AUTOSAR Achievements Phase III, Exploitation and Rollout. In *Proceedings of the 5th AUTOSAR Open Conference*, Beijing, China, November 2012.

[27] R. Mores, G. Hay, R. Belschner, J. Berwanger, C. Ebner, S. Fluhrer, E. Fuchs, B. Hedenetz, W. Kuffner, A. Krüger, P. Lohrmann, D. Millinger, M. Peller, J. Ruh, A. Schedl, and M. Sprachmann. FlexRay – The Communication System for Advanced Automotive Control Systems. In *Proceedings of the SAE 2001 World Congress*, number SAE 2006-21-0019, Detroit, MI, USA, March 2001. Society of Automotive Engineers.

[28] Society of Automotive Engineers (SAE) International. Recommended Practice for a Serial Control and Communications Vehicle Network – Application Layer – Diagnostics. Technical Report J1939/73, Society of Automotive Engineers (SAE) International, February 2010.

[29] Society of Automotive Engineers (SAE) International. Recommended Practice for a Serial Control and Communications Vehicle Network – Data Link Layer. Technical Report J1939/21, Society of Automotive Engineers (SAE) International, December 2010.

[30] Society of Automotive Engineers (SAE) International. Recommended Practice for a Serial Control and Communications Vehicle Network – Network Management. Technical Report J1939/81, Society of Automotive Engineers (SAE) International, June 2011.

[31] A. Schedl, E. Dilger, T. Führer, B. Hedenetz, J. Ruh, M. Kühlewein, E. Fuchs, T. M. Galla, Y. Domaratsky, A. Krüger, P. Pelcat, M. Taï-Leung, M. Glück, S. Poledna, T. Ringler, B. Nash, and T. Curtis. OSEK/VDX – Fault-Tolerant Communication, Version 1.0. Technical report, OSEK, July 2001.

[32] R. Schuermans, R. Zaiser, F. Hepperle, H. Schröter, R. Motz, A. Aberfeld, H.-G. Kunz, T. Tyl, R. Leinfellner, H. Amsbeck, H. Styrsky, B. Ruoff, and L. Wahlmann. XCP – The Universal Measurement and Calibration Protocol Family, Version 1.1.0. Technical report, Association for Standardisation of Automation and Measuring Systems (ASAM), March 2008.

[33] T. Wollstadt, W. Kremer, J. Spohr, S. Steinhauer, T. Thurner, K. J. Neumann, H. Kuder, F. Mosnier, D. Schäfer-Siebert, J. Schiemann, R. John, S. Parisi, A. Zahir, J. Söderberg, P. Mortara, B. France, K. Suganuma, S. Poledna, G. Göser, G. Weil, A. Calvy, K. Westerholz, J. Meyer, A. Maisch, M. Geischeder, K. Gresser, A. Jankowiak, M. Schwab, E. Svenske, M. Tchervinsky, K. Tindell, G. Göser, C. Thierer, W. Janz, and V. Barthelmann. OSEK/VDX - Operating System, Version 2.2.3. Technical report, OSEK, February 2005.