Standardized Basic System Software for Automotive Embedded Applications

Thomas M. Galla
Elektrobit Austria GmbH
Stumpergasse 48/28, A-1060 Vienna, Austria
phone: +43 1 59983 15
fax: +43 1 59983 18
email: thomas.galla@elektrobit.com

February 7, 2009

Abstract

The number of electronic systems in cars is continuously growing. Electronic systems, consisting of so-called electronic control units (ECUs) interconnected by a communication network, account for up to 30% of a modern car's worth. Consequently, software plays an ever more important role, both for the implementation of functions and the infrastructure.

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

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. According to the Hansen Report [15], the S-Class Mercedes, for example, utilizes seven communication buses and 72 micro-controllers.

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 [14]. 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 [5] in 1993/94, the HIS ("Herstellerinitiative

[1]Translated into English: "Open Systems and the Corresponding Interfaces for Automotive Electronics"

Software")[2] group [3], the JasPar ("Japan Automotive Software Platform Architecture") consortium [4] in 2004, the EASIS ("Electronic Architecture and System Engineering for Integrated Safety Systems") project consortium [2] in 2003, and last but not least the AUTOSAR ("Automotive Open System Architecture") consortium [11] in 2003.

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 describes the software modules defined by the HIS group. Section 5 illustrates the software modules standardized by the ISO. Section 7 deals with the AUTOSAR initiative. Section 8 focuses on the Japanese counterpart to AUTOSAR named JasPar, and Section 9 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).

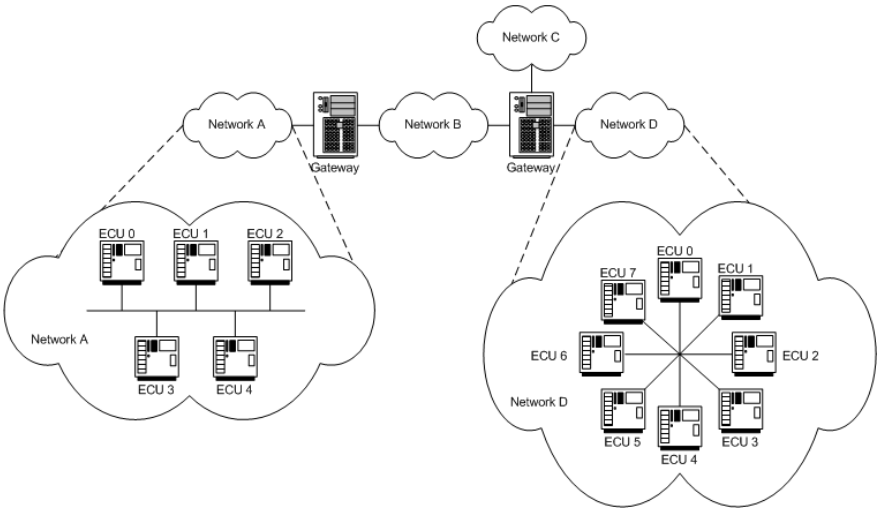


Figure 1: Hardware Architecture – System and Network Level

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 most cases, exactly one MCU and one CC are used to build up an ECU. 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

[2]Translated into English: "Manufacturers' Software Initiative"

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.

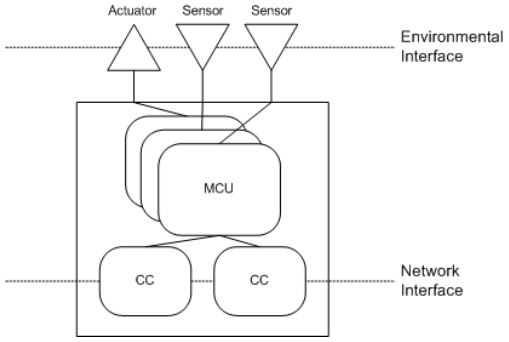


Figure 2: Hardware Architecture – ECU Level

3 OSEK/VDX

The OSEK/VDX standard was the result of the endeavors of major 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 OS is an event-driven operating system intended for hard real-time applications. OSEK provides services for task management, inter-task communication by means of messages, events, resources, alarms, and interrupt handling.

OSEK NM: OSEK NM provides network management facilities which take care of a controlled coordinated startup and shutdown of the communication of multiple ECUs within a network.

OSEK COM: OSEK COM 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).

OSEKtime OS: OSEKtime OS is a time-driven operating system designed for minimal operating system footprint and deployment in safety-related applications.

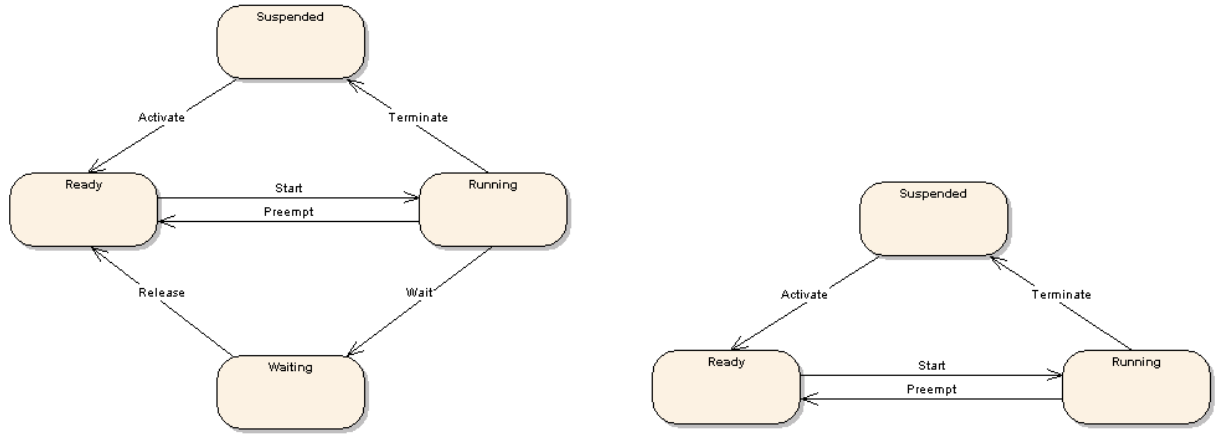
OSEKtime FTCom: OSEKtime FTCom is the fault-tolerant communication layer accompanying OSEKtime OS.

In order to minimize the memory footprint as well as the required central processing unit (CPU) time, each of these components is configured upon design time via a configuration file in OSEK implementation language (OIL) syntax [10]. 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.).

In the following each of the listed OSEK/VDX components is described briefly.

3.1 OSEK OS

OSEK OS [30] is an *event-driven single chip real-time operating system* which is statically configured (as far as available tasks, used resources, etc. are concerned) and thus does not provide dynamic resource allocation. The OS itself defines three different processing levels: *interrupt level*, *logical level* for the scheduler itself, and *task level*. For the task scheduling OSEK OS uses a *priority-based scheduling* strategy. Based on the OS configuration, this scheduling strategy is either *preemptive* or *non-preemptive*. The main OS entities of execution are tasks in OSEK OS. – Hereby OSEK OS distinguishes between *basic tasks* (which do not use any blocking inter process communication (IPC) constructs) and *extended tasks* (which are allowed to use blocking IPC constructs). The second entity of execution in OSEK OS are *interrupt service routines (ISRs)*. – OSEK OS distinguishes between category 1 and category 2 ISRs. Whereas the former are not allowed to call operating system services, the latter are allowed to do so. Depending on the chosen scheduling strategy, an OSEK OS task can be in one of the states depicted in Figure 3.



(a) Extended Task Model

(b) Basic Task Model

Figure 3: OSEK OS Task States

Hereby Figure 3(a) illustrates the task states for the extended task model (which exhibits an additional `Waiting` state) whereas Figure 3(b) depicts the tasks states for basic task model. Synchronization between extended tasks in OSEK OS is performed via *events*. In case extended tasks can wait for an event, they remain in a `Waiting` task state until the event is set by some other extended task. Mutual exclusion between extended tasks is performed by means of *resources*. In order to prevent deadlocks and priority inversion, OSEK OS implements the *priority ceiling protocol* [13] for resources. Communication between tasks in OSEK OS is performed by means of *messages*. – A detailed description of the OSEK communication primitives is provided in the section on OSEK COM (Section 3.3). For cyclic operations, OSEK OS provides the OS entities *counters* and *alarms*. Counters are represented by a counter value. Hereby OSEK OS takes care of periodically incrementing this counter value. In case the counter value reaches a defined threshold, an alarm, which is linked to the counter, can be triggered. – This alarm can, for example, set an event or activate a suspended task. In order to account for different modes of operations of an automotive real-time system, OSEK OS provides different *application modes*. – Each application mode owns its own subset of operating system entities (e.g., tasks, interrupt service routines (ISRs), etc.). – During operation a switch between these application modes is possible.

3.2 OSEK NM

Network management (NM) in the automotive area handles the *controlled coordinated startup and shutdown* of the communication of multiple ECUs within a network. The shutdown of the network (and the accompanying transitions of the ECUs into a low-power or even a power-down mode) is in general done to reduce the network's power consumption in situations when the functions performed by the respective ECUs are not required (e.g., when the car is safely stored in a garage). Additionally OSEK NM [21] provides (weakly-agreed) information on which ECUs within a network are participating in the OSEK NM. – This service is called *node monitoring* in OSEK NM. In order to realize these two services, 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 operation, a transition into the sleep mode is prevented. Wakeup of a sleeping network is triggered by any traffic on the network. Figure 4 depicts the state automaton of OSEK NM.

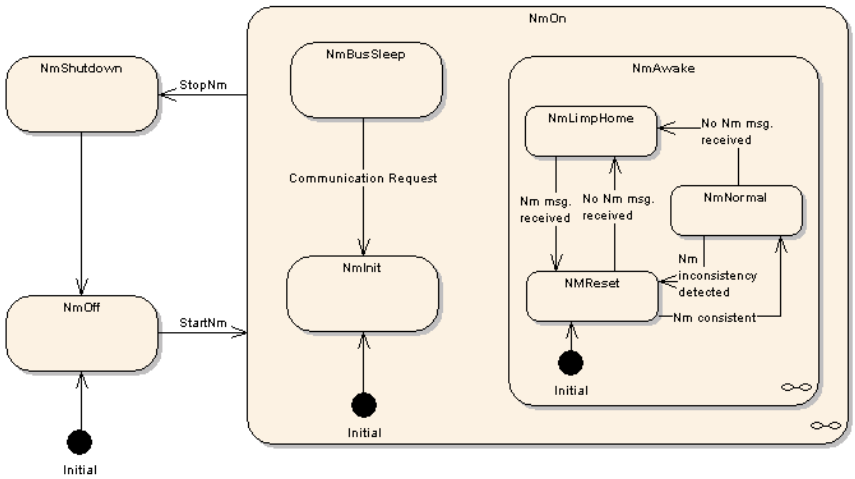


Figure 4: OSEK NM State Automaton

In case an ECU which is currently not a member of the logical ring, wants to join the ring in order to participate in OSEK NM, the ECU's NM module send a so-called *alive message* in order to introduce itself (sub-state `NmReset`). Once the new ECU has joined the ring, it participates in transmission of the ring messages (sub-state `NmNormal`). In case an ECU fails to receive any ring messages while residing in `NmNormal` state, the ECU NM module transits into `NmLimpHome` state where the ECU sends *limp home messages* until ring messages are received again.

Figure 5: OSEK NM Message Format

Figure 5 depicts the message format of OSEK NM's messages. A *source* and a *destination ID* field uniquely identify the sending and the receiving ECU in the network. The *OP code* field defines whether the message is a ring message, a limp home message, or an alive message.

An optional data field may carry additional data from higher layers that is transferred along the ring.

3.3 OSEK COM

OSEK COM [9] provides a uniform communication environment for automotive control unit applications by offering services for data transfer among tasks and/or interrupt service routines. Hereby tasks and ISRs can be located on either the same ECU (internal communication) or on different ECUs (external communication). OSEK COM supports both *cyclic time-driven communication* as well as *event-driven communication*. Additionally OSEK COM provides mechanisms for transmission and reception timeout monitoring (e.g., in case the transmission of a signal triggered by the call of an OSEK COM Application Programming Interface (API) service does not take place within a statically configurable interval a transmission timeout is signaled to the layers located on-top of OSEK COM).

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.

3.4 OSEKtime OS

OSEKtime OS [6] is designed as a statically configured time-driven single chip operating system for distributed embedded systems. Just like OSEK OS OSEKtime OS provides two entities of execution, namely *interrupt service routines* and *tasks*. Interrupt service routines (ISRs) execute interrupt related services. Tasks, however, are started at defined points in time and have one of three different states. In the `Running` state, the CPU is assigned to the task and its commands are executed. Only one task can reside in this state at any point in time while all other states can be adopted simultaneously by several tasks. The `Preempted` state is reached by a task that has been in the `Running` state and was preempted by another task that is to be activated. A task can leave the `Preempted` state only if the preempting task changes into the `Suspended` state. An inactive task that can be activated is in the `Suspended` state. The state changes described so far are depicted in Figure 6.

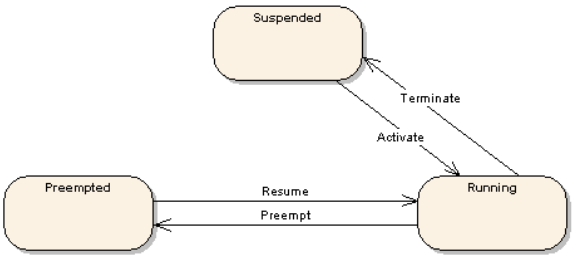


Figure 6: OSEKtime Task States

The activation times of the tasks are stored in a *dispatcher table* before compile time. The dispatcher is the central component of the OSEKtime OS and is responsible for activating the tasks according to their activation times. The processing of the dispatcher table is done

cyclically. A complete cycle of the dispatcher table is called a *dispatcher round*. If a time-driven task is still in the state `Running` when the activation time of another time-driven task is reached, the first task passes over to the `Preempted` state and stays there until the interrupting task is finished. This kind of scheduling is called *stack-based scheduling*; no priorities are assigned to the tasks – only the activation times define the precedence between different tasks.

3.5 OSEKtime FTCom

OSEKtime FTCom [28] is the fault-tolerant real-time communication layer that accompanies OSEKtime OS. – Similar to OSEK COM, OSEKtime FTCom can be used for both external as well as internal communication. – In contrast to OSEK COM, OSEKtime FTCom solely supports *time-driven signal transmission*.

In case of external communication the *interaction sub-layer* of OSEKtime FTCom 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. On the receiver's side a conversion from the network byte order into the byte order of the receiving ECU takes place. Furthermore, the interaction layer packs multiple signals into a single communication frame in order to reduce communication bandwidth consumption. Additionally, 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 sublayer for byte order conversion and packing. Afterwards, the 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 sublayer and reduced to obtain a single application signal which is handed on to the application layer.

4 HIS

The "Hersteller Initiative Software" (HIS) working group has been formed by their members to "bundle their activities for standard software modules, process maturity levels, software test, software tools and programming of control units. The common goal is to achieve and use joint standards". Different from the OSEK/VDX working group, HIS solely includes car manufacturers (OEMs) as members. The components standardized by HIS are listed in the following:

Protected OSEK OS: The protected OSEK OS specification defines additional mechanisms and services (in addition to the conventional OSEK OS) in order to protect separate applications running on the same ECU from each other in the value (memory protection) and in the temporal (execution time monitoring) domain.

IO Library and Drivers: The HIS IO library provides a standardized interface to access the respective low-level drivers.

Flash Driver: The specification of the standardized HIS flash driver describes a standard interface of services required for flash programming.

CAN Driver: The HIS CAN driver provides data-link layer services for the transmission and the reception of frames on CAN [23, 24] networks.

In the following each of the listed HIS components is briefly described.

4.1 Protected OSEK

HIS defines extensions to the OSEK OS specification in order to protect separate application tasks running on the same ECU [18]. – These extensions

- provide means to prevent different application tasks within the same ECU to corrupt each other's data,
- monitor and restrict the execution time of each single application task
- exclusively assign operating system resources to a single application task and prevent all other application tasks from accessing these resources.

In order to fulfill these services, the protected OSEK OS, however, requires hardware support, namely a memory protection unit and the existence of multiple processor modes (e.g., a privileged mode for the OS and a non-privileged mode for the application tasks). For the monitoring and the restriction of the execution time of each task, an OS timer is assigned to each task in order to trace the task's execution time and abort the task in case the task exceeds its statically assigned execution time budget.

4.2 IO Library and Drivers

The HIS Input/Output (IO) library [20] is intended to provide a standard interface to low-level drivers (e.g., for digital IO). The IO library supports both synchronous (i.e., the caller of the library function is blocked till the function completes) as well as asynchronous (i.e., the caller of the library function just triggers the function, and is asynchronously informed once the IO operation has completed) operation. – In case of asynchronous operation, the IO library itself takes care of the required buffer handling to decouple the calling function from the called driver. The IO library and the corresponding drivers provide the following API functions:

Init: This API function is used for the initialization of the controlled device. – The function must be called prior to any other service.

Deinit: This API function is used to de-initialize the controlled device. – The function has to be called upon releasing of the device (e.g., prior to termination of the application).

Read: This API function is used to read data from the controlled device. – The semantics of this function is similar to the UNIX `read(2)` system call.

Write: This functions is used to write data to the controlled device. – The semantics of this function is similar to the UNIX `write(2)` system call.

Ioctl: This function can be used to access other services provided by the respective driver, which cannot be mapped to the semantics of any of the previously described API functions. – The semantics of this function is similar to the UNIX `ioctl(2)` system call.

As far as resource management is concerned, the HIS IO library relies on the services provided by OSEK operating systems. The IO drivers [19] standardized by HIS are drivers for digital input/output, pulse width modulation, pulse width demodulation, analogue/digital converters, watchdog timer, capture compare timers, EEPROM drivers and flash drivers.

4.3 Flash Driver

The HIS standardized flash driver component [16] provides defined services (including a corresponding API) for the initialization of the flash driver component, the de-initialization of the flash driver component, the erasing of the flash memory, and the writing to the flash memory. Hereby, the job of the flash driver is to provide a logical abstraction from the physical organization of the flash memory (e.g., segmentation, banking, a.s.o.). – The HIS flash driver is designed to be used as a driver for the HIS IO library. Conventional application areas for the HIS flash driver are the flash programming services of XCP (see Section 6.1) and ISO diagnostics (see Section 5.2)

4.4 CAN Driver

The HIS CAN driver [17] is an implementation of a data link layer for CAN networks. – The HIS CAN driver provides a uniform interface for the reception and the transmission of frames on the CAN network and thus abstracts from the peculiarities of the underlying buffer configurations of the respective CAN controller (full CAN vs. basic CAN a.s.o.). The main services provided by the CAN driver are the following:

Frame Reception: The CAN driver retrieves received frames from the hardware buffers of the CAN controller and informs upper software layers about the frame reception.

Frame Transmission: On request of higher software layers the CAN driver transfers frames to be transmitted into the hardware buffers of the CAN controller and informs higher software layers about the successful transmission of these frames.

Wakeup Service: If supported by the respective CAN controller, the CAN driver provides services to put the CAN controller in a special low-power mode (the sleep mode). In case a CAN controller transits from sleep mode (due to traffic on the communication media) back into the normal operation mode, the CAN driver informs the higher software layers about this event.

Error Handling: In case fatal errors occur on the CAN communication media, the CAN driver informs the higher software layers about these errors.

Note, however, that the HIS CAN driver does not provide an IO library compliant interface!

5 ISO

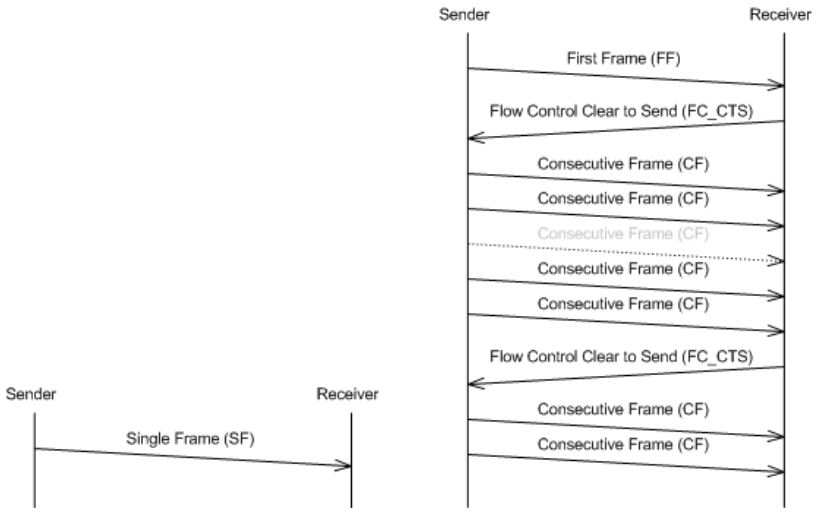
Some of the software layers used for CAN networks in today's cars have been standardized by the International Organization for Standardization (ISO). – The most prominent of these software layers are the following:

Transport Layer (ISO/DIS 15765-2.2): This layer provides *unacknowledged segmented data transfer* to all higher layers and thus facilitates exchange of data amounts larger than the maximum transfer unit (MTU) of the underlying communication system.

Diagnostics – KWP2000 (ISO/DIS 14230-3) and UDS (ISO/DIS 14229-1): Both the Keyword Protocol 2000 (KWP2000) as well as the Unified Diagnostic Services (UDS) define diagnostic services which *allow a tester device to control diagnostic functions* in an ECU via a serial data link.

5.1 Transport Layer (ISO/DIS 15765-2.2)

The ISO transport layer [25] provides unacknowledged segmented data transfer (USDT) services to higher software layers (e.g., ISO diagnostics (see Section 5.2)) 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). 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 7 illustrates the frame exchange in an unsegmented (Figure 7(a)) and a segmented transmission (Figure 7(b)).



(a) Unsegmented Transfer (b) Segmented Transfer

Figure 7: ISO Transport Layer – Transmission Sequence

Hereby the frame format depicted in Figure 8 is used by the ISO transport layer. The *source* and *destination address* fields are used to identify the sender and the receiver(s) of the TP frame. The *address type* field is used to define the semantics of the destination address. Here ISO TP provides support for single ECU addresses (called *physical addresses* by the ISO TP specification) and multicast addresses (called *functional addresses* by the ISO TP

specification). In case of multicast transmissions, only unsegmented data transfer is allowed in ISO TP.

Figure 8: ISO Transport Layer – 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.

5.2 Diagnostics – KWP2000 (ISO/DIS 14230-3)/UDS (ISO/DIS 14229-1)

The Keyword Protocol 2000 (KWP2000) [22] and the Unified Diagnostic Services (UDS) [26] 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, and the encoding of the parameters are standardized by the KWP2000 and the UDS specification. For the exchange of information between the client and the server (and vice versa) both diagnostic protocols use the previously described transport layer. The services provided by KWP2000 and 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 reseting 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. – Two 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.

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 KWP2000 as well as 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.

Describing the format of the diagnostic messages for each value of SID is beyond the scope of this chapter. – Detailed information on this topic can be obtained from the respective ISO specifications [22, 26].

6 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.

6.1 XCP – The Universal Measurement and Calibration Protocol Family

The Universal Measurement and Calibration Protocol Family XCP [29] is used for the following main purposes:

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

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

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, USB, TCP/IP, UDP/IP, and SxI are supported). – Figure 9 illustrates this XCP protocol stack.

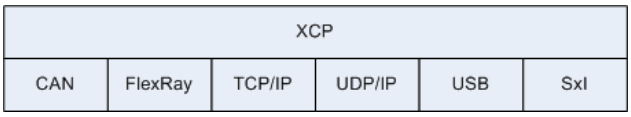


Figure 9: XCP Protocol Stack

6.1.1 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 6.1.3, 6.1.4, and 6.1.5. 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 10 illustrates the structure of an XCP packet.

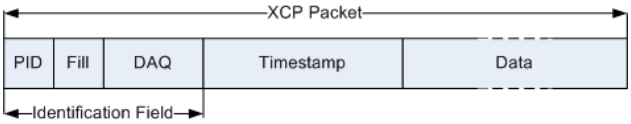


Figure 10: Structure of an XCP Packet

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 subtypes:

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 6.1.3) 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.

6.1.2 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).

6.1.3 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 11).

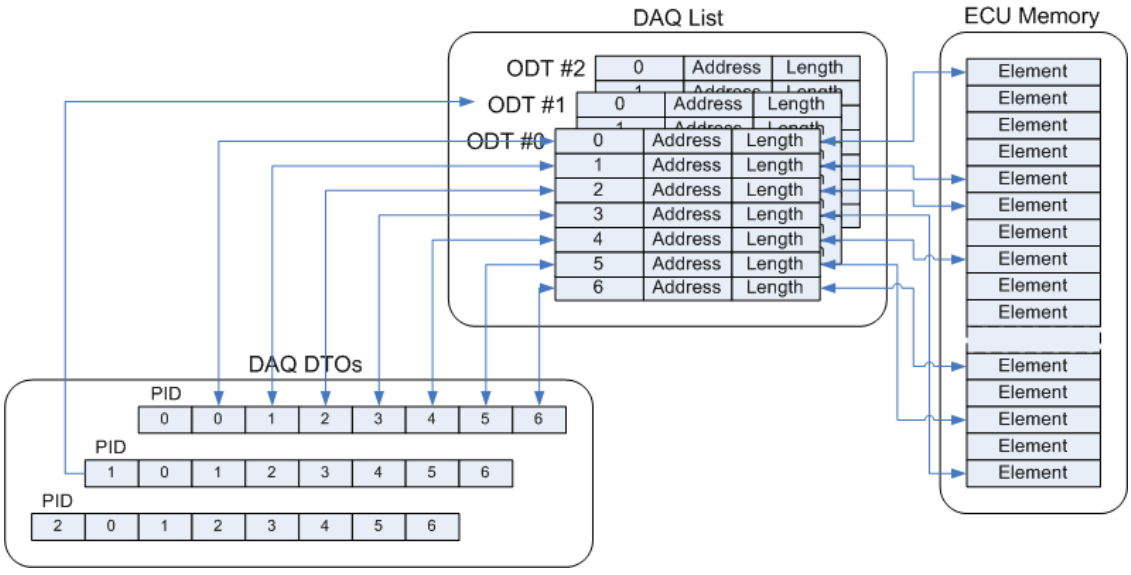


Figure 11: Structure of a DAQ List

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 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.

6.1.4 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.

6.1.5 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 5.2).

7 AUTOSAR

The development partnership AUTomotive Open System ARchitecture (AUTOSAR) is an alliance of OEM manufacturers, major supplier companies, and tool vendors working together to develop and establish an open industry standard for an automotive electronic architecture 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, and 3.0 have been released. For versions 1.0 and 2.0 a dedicated proof-of-concept implementation has been performed and integrated in a sample application.

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 [12, 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.

7.1 Application Software Architecture

Application software in AUTOSAR consists of *application software components*, which are ECU and location independent and *sensor-actuator components* that are dependent on ECU hardware and therefore location dependent. Whereas instances of application software components can easily be deployed to and relocated among different ECUs, instances of sensor-actuator components must be deployed to a specific ECU for performance/efficiency reasons.

Application software components as well as sensor-actuator components are interconnected via so-called connectors. These connectors represent the exchange of signals or the remote method invocations among the connected components.

7.1.1 Runnable Entities

Software components are composed of one or more *runnable entities* which are executed in the context of operating system tasks. These runnable entities are invoked by a dedicated system software module named *Runtime Environment* (`Rte`). Depending on whether or not runnable entities have wait-points, 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 systems (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).

7.1.2 Communication

To enable communication between different software components in AUTOSAR, the components are equipped with a communication interface that consists of several ports. Hereby ports have defined interface types. AUTOSAR distinguishes between *sender-receiver port interfaces* providing message passing semantics and *client-server port interfaces* which provide remote procedure call semantics. AUTOSAR further distinguishes between the following domains of communication: *intra-task communication* (communication between runnable entities of components that are assigned to the same task on the same ECU), *inter-task communication* (communication between runnable entities of components that are assigned to different tasks on the same ECU), and *inter-ECU communication* (communication between runnable entities of components that are assigned to different tasks on different ECUs). Each of the previously described communication semantics is supported by these three communication domains. As far as multiplicity is concerned, AUTOSAR supports 1:1, 1:n, and n:1 communication for sender-receiver port interfaces and 1:1 and n:1 communication (multiple clients, single server) for client-server port interfaces.

7.2 System Software Architecture

In addition to the application software components, AUTOSAR also defines a layered architecture of system software modules [7] which provides the basic platform for the execution of the application software components. Figure 12 provides a coarse grained overview of the major layers of these system software modules.

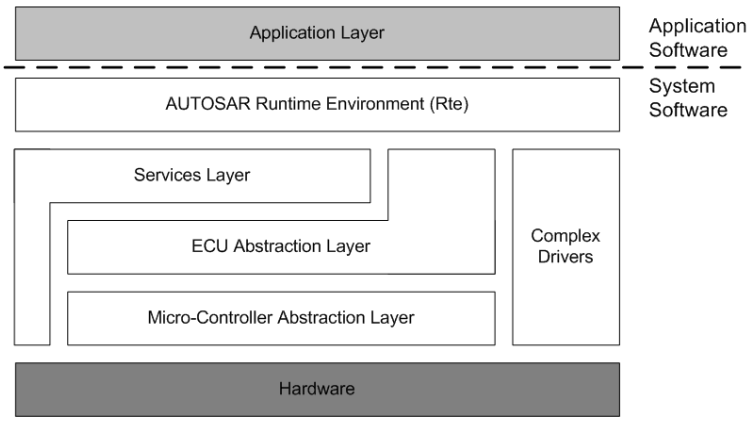


Figure 12: AUTOSAR – System Software Layered Architecture

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 for the application software and other system software modules.

Complex Device Drivers: The concept of complex device 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.

Runtime Environment (`Rte`): The AUTOSAR Runtime Environment provides the interface between application software components and the basic 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 7.1.

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 13.

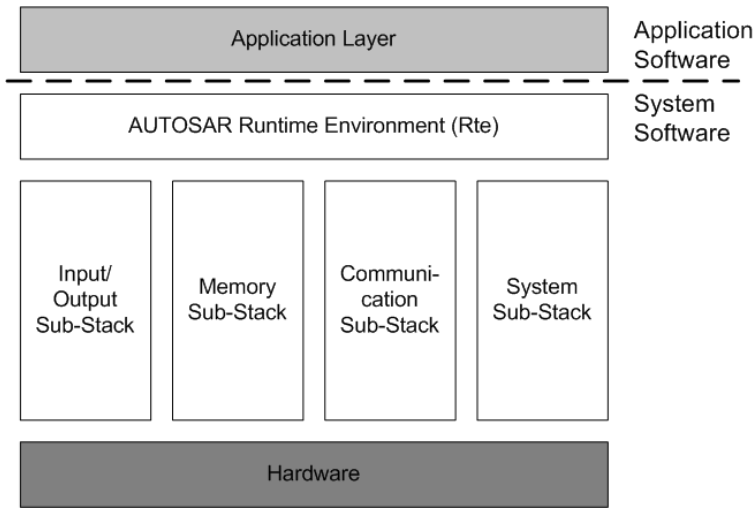


Figure 13: AUTOSAR – System Software Sub-Stacks Overview

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.).

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) [8], the Controller Area Network (CAN) [23, 24], and FlexRay [12, 27]).

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.

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 found 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.

7.2.1 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 protocols (CAN, LIN, and FlexRay). The structure of the communication sub-stack is depicted in Figure 14.

Hereby, `xx` is used as a placeholder for the respective communication protocol (i.e., CAN, LIN, FlexRay). Thus the AUTOSAR communication sub-stack contains communication protocol specific instances of the Transport Protocol (`Tp`), Network Management (`Nm`), Interface (`If`), State Manger (`Sm`), Transceiver Driver (`Trcv`), and Driver (no suffix) modules. In the following, the different modules of the communication sub-stack are described in detail.

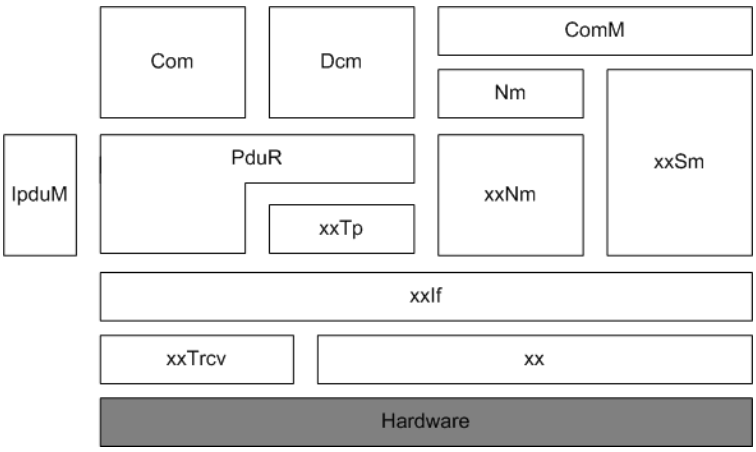


Figure 14: AUTOSAR – Communication Sub-Stack

Driver (`xx`): The Driver module (`Fr`, `Can`, `Lin`) 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.

Transceiver Driver (`xxTrcv`): The various different Transceiver Driver modules (`FrTrcv` and `CanTrcv`) provide API functions for *controlling the transceiver hardware* (i.e., switching the transceivers into special modes (e.g., listen only 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 wakeup events on the bus).

Interface (`xxIf`): Using the frame-based services provided by the Driver module, the Interface module (`FrIf`, `CanIf`, `LinIf`) 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[4]. 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).

The 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

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

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 (`xxTp`): The transport protocol is used to perform *segmentation and reassembly of large protocol data units (PDUs).* On CAN (`CanTp`), this protocol is compatible (in certain configuration settings) to ISO TP (see Section 5.1). Just like ISO TP, the user of the services provided by the transport protocol is the diagnostic layer, called Diagnostic Communication Manager in AUTOSAR (see below).

Network Management (`xxNm`, `Nm`): Similar to OSEK NM (see Section 3.2) 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. AUTOSAR NM is hereby divided into two modules: a communication protocol independent module named *Generic NM* (`Nm`) and communication protocol dependent modules named *FlexRay NM* (`FrNm`) and *CAN NM* (`CanNm`)[5].

State Manager (`xxSm`): The State Manager modules (`CanSm`, `LinSm`, and `FrSm`) facilitate the *state management of the respective communication controllers* with respect to communication system dependent start-up 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`.

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`): The `Com` module 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.

[5]Since LIN is a master-slave bus, no `LinNm` is required. The LIN master dictates when the network shall transit into a low-power mode.

– 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 KWP2000 and UDS (see Section 5.2 for details).

Communication Manager (`ComM`): The Communication Manager is a resource manager which *encapsulates the control of the underlying communication services.* The `ComM` collects bus communication access requests from communication requesters (e.g., `Dcm`) and coordinates these requests by interacting with `Nm` and the respective State Manager 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 communication channel 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 channels) the communication capability of the corresponding channel(s) to `On`, `Silent`, or `Off`.

7.2.2 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 and retrieved data in the ECU's non-volatile memory (e.g., flash EEPROM or EEPROM). The structure of the memory sub-stack is depicted in Figure 15.

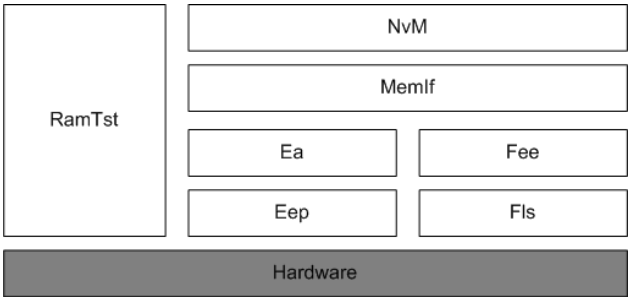


Figure 15: AUTOSAR – Memory Sub-Stack

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.

EEPROM Driver (`Eep`): The EEPROM Driver provides services for *reading from, writing to, and erasing parts of the ECU's EEPROM.* In addition to these basic services, 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.

RAM Test (`RamTst`): The RAM Test module performs *functional tests of the ECU's internal RAM cells. Complete* tests are performed upon ECU start-up 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%.

7.2.3 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 16.

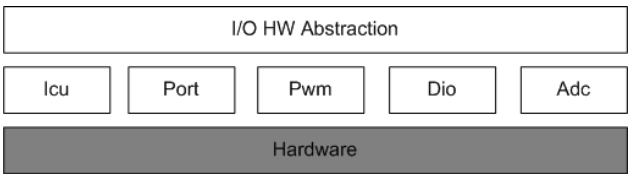


Figure 16: AUTOSAR – Input/Output Sub-Stack

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

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 wakeup interrupts, periodic signal time measurement, edge time stamping (usable for the acquisition of non-periodic signals), and edge counting.

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), 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., `Dio`, `Adc`, ...) rely on the configuration performed by the Port Driver.

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 thus 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.

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` modules 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.

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).

7.2.4 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 library functions (like CRC and interpolation functions). The structure of the system services sub-stack (excluding the library functions) is depicted in Figure 17.

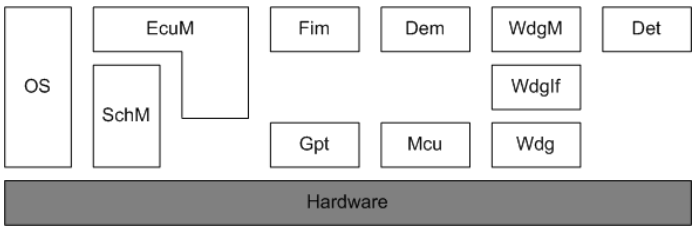


Figure 17: AUTOSAR – System Services Sub-Stack

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 run-time, and is designed to be hostable on low-end controllers.

Similar to the OSEKtime dispatcher tables (see Section 3.4), 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 (similar to the protected OSEK defined by HIS (see Section 4.1)) the memory protection unit of the MCU to provide coarse grained memory protection of the different tasks against each other.

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 in a task context[6].

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 basic software modules including Os and Rte, cooperates with the ComM, and hence indirectly with Nm, to shut down the ECU when needed, manages all wakeup events, and configures the ECU for Sleep state when requested. In order to fulfill all these tasks, the EcuM provides 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 *wakeup validation protocol* to distinguish "real" wakeup events from "erratic" ones, and the *time-triggered increased inoperation protocol* which allows to put the ECU into an increasingly energy saving sleep state.

Function Inhibition Manager (Fim): The Function Inhibition Manager is a sub-module of the AUTOSAR diagnostic module. 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.

Diagnostic Event Manager (Dem): Like the Dcm module and the Fim module, the Diagnostic Event Manager module is a sub-module of the diagnostic module within AUTOSAR. The Dem is responsible for *processing and persistently storing diagnostic events/errors*[7] 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 event 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.

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

[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]UDS and KWP2000 use the term diagnostic trouble codes (DTC) for these events.

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 (periodic) execution* of application software components or other system software modules. Via the services provided by the `WdgIf`, the `WdgM` abstracts from the triggering of hardware watchdog entities and itself offers supervision capabilities to a (theoretically) infinite number of clients. The `WdgM` provides an individual port for each supervised client, where the clients have to indicate their proof of aliveness by updating an alive-counter. Within a cyclic scheduled main processing function of the `WdgM`, the alive-counters of all supervised clients are checked against their own independent timing constraints and a "global supervision status" is derived. 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 and their individual timing constraints is defined by configurable parameters of the `WdgM`.

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 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.

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.

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 ap-

plication related MCU initialization in addition to the start-up code[8]. The services of the 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.

8 JasPar

Founded September 2004 by Toyota, Nissan, Honda, and Toyotsu Electronics, the non profit organization JasPar [4] was created to develop a unified architecture for vehicle networking and software development tailored to the requirements of Japanese customers.

JasPar's goals are focused on the evaluation, the exploitation, and the implementation of existing standards in actual projects. The organization seeks to serve as the Japanese point of contact for automotive-standards organizations in other regions. Based on the results of the AUTOSAR and FlexRay Consortium, JasPar was constructed as complementary organization to unite standardization and product attempts in Japan (see Figure 18) in oder to maintain the possibility of tailored product generation for the Japanese OEM and finally the end customer.

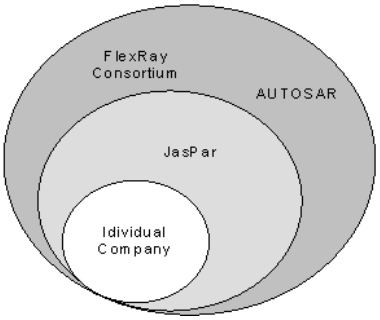


Figure 18: Relationship Autosar, FlexRay, JasPar consortium

Within JasPar, working groups are focusing on developing design tools and prototypes for automotive networks, creating standards for in-vehicle cable routing, writing guidelines for wire-harness design, developing recommended circuits for controllers/transceivers, and planning communication software. Moreover, JasPar is feeding suggestions back to related consortia for new standard requirements.

An example given in the evaluation of existing standards is the standardization of the business process in using FlexRay parameter settings. This standardization shall reduce the risk of failures caused by improper parameter settings and thus shall prevent a reduction of overall quality which is a crucial product criteria on the Japanese market. Further activities of JasPar include interoperability tests between different FlexRay CCs and the evaluation of bus wiring conditions including the observation of the communication behavior of FlexRay at different (currently unspecified) transmission rates.

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

9 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, HIS, JasPar, 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 aims at putting software modules according to the AUTOSAR standard into production vehicles by 2008 (BMW and Honda) [14] 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, HIS CAN driver, 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 plan to release tested and verified AUTOSAR 4.0 specifications together with the corresponding conformance test specifications by September 2009 [1].

References

[1] AUTOSAR – Top Level Schedule Phase II. Consortium Web Page.

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

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

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

[5] OSEK/VDX. Project Web Page.

[6] 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, July 2001.

[7] AUTOSAR Consortium. AUTOSAR – Layered Software Architecture. Technical Report Version 2.2.1, Release 3.0, Rev 0001, AUTOSAR Consortium, February 2008.

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

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

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

[11] 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.

[12] 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.

[13] 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.

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

[15] P. Hansen. New S-Class Mercedes: Pioneering Electronics. *The Hansen Report on Automotive Electronics*, 18(8):1–2, October 2005.

[16] HIS (Hersteller Initiative Software). HIS Functional Specification of a Flash Driver, Version 1.3. Technical report, June 2002.

[17] HIS (Hersteller Initiative Software). HIS CAN Driver Specification, Version 1.0. Technical report, August 2003.

[18] HIS (Hersteller Initiative Software). OSEK OS Extensions for Protected Applications, Version 1.0. Technical report, July 2003.

[19] HIS (Hersteller Initiative Software). API IO Driver, Version 2.1.3. Technical report, April 2004.

[20] HIS (Hersteller Initiative Software). API IO Library, Version 2.0.3. Technical report, March 2004.

[21] 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, July 2004.

[22] ISO. Road Vehicles – Diagnostic Systems – Keyword Protocol 2000 – Part 3: Application Layer. Technical Report ISO/DIS 14230-3, ISO (International Organization for Standardization), 1, rue de Varembe, Case postale 56, CH-1211 Geneva 20, Switzerland, 1999.

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

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

[25] ISO. Road Vehicles – Diagnostics on Controller Area Networks (CAN) – Part 2: Network Layer Services. Technical Report ISO/DIS 15765-2.2, ISO (International Organization for Standardization), 1, rue de Varembe, Case postale 56, CH-1211 Geneva 20, Switzerland, April 2003.

[26] ISO. Road Vehicles – Unified Diagnostic Services (UDS) – Part 1: Specification and Requirements. Technical Report ISO/DIS 14229-1, ISO (International Organization for Standardization), 1, rue de Varembe, Case postale 56, CH-1211 Geneva 20, Switzerland, 2004.

[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] 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, July 2001.

[29] 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 1.0. Technical report, Association for Standardisation of Automation and Measuring Systems (ASAM), April 2003.

[30] 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, February 2005.