Layered Software Architecture

Version 2.0.0
### Document Information and Change History (1)

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changed by</th>
<th>Change Description</th>
</tr>
</thead>
</table>
| 21.03.2006    | 2.0.0   | AUTOSAR Administration| Rework Of:  
- Error Handling  
- Scheduling Mechanisms  
More updates according to architectural decisions in R2.0 |
| 31.05.2005    | 1.0.1   | AUTOSAR Administration| Correct version released                                                          |
| 09.05.2005    | 1.0.0   | AUTOSAR Administration| Initial release                                                                    |
Disclaimer

This specification as released by the AUTOSAR Development Partnership is intended for the purpose of information only. The use of material contained in this specification requires membership within the AUTOSAR Development Partnership or an agreement with the AUTOSAR Development Partnership. The AUTOSAR Development Partnership will not be liable for any use of this Specification.

Following the completion of the development of the AUTOSAR Specifications commercial exploitation licenses will be made available to end users by way of written License Agreement only.

No part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from the publisher.

The word AUTOSAR and the AUTOSAR logo are registered trademarks.

Copyright © 2004-2006 AUTOSAR Development Partnership. All rights reserved.

Advice to users of AUTOSAR Specification Documents:

AUTOSAR Specification Documents may contain exemplary items (exemplary reference models, "use cases", and/or references to exemplary technical solutions, devices, processes or software). Any such exemplary items are contained in the Specification Documents for illustration purposes only, and they themselves are not part of the AUTOSAR Standard. Neither their presence in such Specification Documents, nor any later AUTOSAR compliance certification of products actually implementing such exemplary items, imply that intellectual property rights covering such exemplary items are licensed under the same rules as applicable to the AUTOSAR Standard.
Overview

Part 1 – Introduction, Scope and Limitations
Part 2 – Overview of Software Layers
Part 3 – Contents of Software Layers
Part 4 – Interfaces
   4.1 General Rules
   4.3 Error Handling and Reporting Concept
   4.4 Interaction of Layers – Example “Memory”
   4.5 Interaction of Layers – Example “Communication”
   4.6 Interaction of Layers – Example “ECU State Manager”
   4.7 Reliability Mechanisms Among Communication
Part 5 – Configuration
Part 6 – Scheduling
Part 1 – Introduction, Scope and Limitations
Part 1 – Introduction, Scope and Limitations
ID: 01-01

Purpose of this document
The Layered Software Architecture shall map the identified modules of the Basic Software Module List to software layers and show their relationship. It shall help WP 1.1.2 and following workgroups specifying the interfaces and functionalities of the Basic Software.

This document does not contain requirements. It is a document summarizing architectural decisions and discussions of the work package WP 1.1.2.

This document focuses on static views of a conceptual layered software architecture. This document does not specify a structural software architecture with detailed static and dynamic interface descriptions. This should be done within the WP 4.X working groups responsible for specification and implementation of basic software.

The functionality and requirements of the Basic Software modules are specified in the module specific requirement and specification documents.

Inputs and requirements
This document has been generated based on following documents:
- Basic Software Module List, WP 1.1.2
- Specification of Virtual Functional Bus, WP 1.1.1
- Several views of automotive ECU software architectures
Part 1 – Introduction, Scope and Limitations
ID: 01-02

In Scope:
Automotive ECUs having the following properties:
- Strong interaction with hardware (sensors and actuators)
- Connection to vehicle network via CAN, LIN or FlexRay
- Microcontrollers from 16 to 32 bit with limited resources of Flash and RAM (compared with Enterprise Solutions)
- Real Time Operating System
- Program execution from internal or external flash memory

Not in scope:
High end embedded applications like HMI Head Unit with High end Operating Systems like WinCE, VxWorks, QNX containing
- Middleware concepts like OSGI, CORBA
- Graphics library
- Java Virtual Machine
- E-Mail client
- Communication systems like Bluetooth, USB, Ethernet
- Communication protocols like TCP/IP
- Flash file system
- Dynamic linking and loading of software
- Multi processor support in terms of dynamic load balancing

Extensibility:
- This SW Architecture is a generic approach. Modules can be added or existing ones can be extended in functionality, but their configuration has to be considered in the automatic Basic SW configuration process!
- Complex drivers can easily be added
- Further Layers cannot be added
Current limitations of document version 2.2.3 are:

- **MOST is not supported**
  No seamless integration was possible

- **The Transport Protocol is only used by DCM, not by AUTOSAR COM**
  This basically means that CAN and LIN PDUs that carry application data cannot be longer than 8 bytes.
  From the Layered Architecture this restriction is not visible. A later usage of TP also by AUTOSAR COM will not change this document
Part 2 – Overview of Software Layers
Part 2 – Overview of Software Layers
ID: 02-01 Component View

AUTOSAR Software Component

Application Software Component
AUTOSAR Interface

Actuator Software Component
AUTOSAR Interface

Sensor Software Component
AUTOSAR Interface

AUTOSAR Runtime Environment (RTE)

Sensor Software Component
AUTOSAR Interface

Actuator Software Component
AUTOSAR Interface

Application Software Component
AUTOSAR Interface

ECU-Hardware

AUTOSAR Software

API 0
RTE relevant

API 1
VFB & RTE relevant

API 2
VFB & RTE relevant

Complex Device Drivers

Basic Software

Operating System
Standardized Interface

Services
Standardized Interface

Communication
Standardized Interface

ECU Abstraction
Standardized Interface

Microcontroller Abstraction
Standardized Interface

Standardized Interface

Standardized AUTOSAR Interface

Standardized Interface

Standardized Interface

Standardized Interface

Standardized Interface

Standardized Interface

API 3 Private Interfaces inside Basic Software possible

AUTOSAR Runtime Environment (RTE)

Interface

ECU Firmware

Standard Software

AUTOSAR Interface

AUTOSAR Interface
Part 2 – Overview of Software Layers
ID: 02-02 Layered View: Coarse
Part 2 – Overview of Software Layers
ID: 02-03 Layered View: Detailed

Application Layer

AUTOSAR Runtime Environment (RTE)

- System Services
- Memory Services
- Communication Services
- I/O Hardware Abstraction
- Complex Drivers

- Onboard Device Abstraction
- Memory Hardware Abstraction
- Communication Hardware Abstraction

- Microcontroller Drivers
- Memory Drivers
- Communication Drivers
- I/O Drivers

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

**Task:**
Make higher software layers independent of µC

**Properties:**
Implementation: µC dependent
Upper Interface: standardizable and µC independent
The **ECU Abstraction Layer** interfaces the drivers of the Microcontroller Abstraction Layer. It also contains drivers for external devices. It offers an API for access to peripherals and devices regardless of their location (µC internal/external) and their connection to the µC (port pins, type of interface)

**Task:**
Make higher software layers independent of ECU hardware layout

**Properties:**
Implementation: µC independent, ECU hardware dependent
Upper Interface: µC and ECU hardware independent, dependent on signal type
The **Services Layer** is the highest layer of the Basic Software which also applies for its relevance for the application software: while access to I/O signals is covered by the ECU Abstraction Layer, the Services Layer offers:

- Operating system functionality
- Vehicle network communication and management services
- Memory services (NVRAM management)
- Diagnostic Services (including UDS communication and error memory)
- ECU state management

**Task:**
Provide basic services for application and basic software modules.

**Properties:**
Implementation: partly μC, ECU hardware and application specific
Upper Interface: μC and ECU hardware independent
The **RTE** is a middleware layer providing communication services for the application software (AUTOSAR Software Components and/or AUTOSAR Sensor/Actuator components).

Above the RTE the software architecture style changes from „layered“ to „component style“. The AUTOSAR Software Components communicate with other components (inter and/or intra ECU) and/or services via the RTE.

**Task:**
Make AUTOSAR Software Components independent from the mapping to a specific ECU

**Properties:**
- Implementation: ECU and application specific (generated individually for each ECU)
- Upper Interface: completely ECU independent
The Basic Software can be subdivided into the following types of services:

- **Input/Output (I/O)**
  - Access to sensors, actuators and ECU onboard peripherals

- **Memory**
  - Standardized access to internal/external memory (mainly non volatile memory)

- **Communication**
  - Standardized access to vehicle network systems and ECU onboard communication systems

- **System**
  - Provision of standardizable (operating system, timers, error memory) and ECU specific (ECU state management, watchdog manager) services and library functions
Driver
A driver contains the functionality to control and access an internal or an external device.

Internal devices are located inside the microcontroller. Examples for internal devices are
- Internal EEPROM
- Internal CAN controller
- Internal ADC

A software driver for an internal device is called internal driver and is located in the Microcontroller Abstraction Layer.

External devices are located on the ECU hardware outside the microcontroller. Examples for external devices are
- External EEPROM
- External watchdog
- External flash

A software driver for an external device is called external driver and is located in the ECU Abstraction Layer. It accesses the external device via drivers of the Microcontroller Abstraction Layer.

Example: a driver for an external EEPROM with SPI interface accesses the external EEPROM via the SPIHandlerDriver.

Exception:
The drivers for memory mapped external devices (e.g. external flash memory) may access the microcontroller directly. Those external drivers are located in the Microcontroller Abstraction Layer because they are microcontroller dependent.
Part 2 – Overview of Software Layers
ID: 02-10 Introduction to Basic Software Module Types (2)

Interface
An Interface contains the functionality to abstract the hardware realization of a specific device for upper layers. It provides a generic API to access a specific type of device independent on the number of existing devices of that type and independent on the hardware realization of the different devices.

The interface does not change the content of the data.

In general, interfaces are located in the ECU Abstraction Layer.

Example: an interface for a CAN communication system provides a generic API to access CAN communication networks independent on the number of CAN Controllers within an ECU an independent of the hardware realization (on chip, off chip).

Handler
A handler is a specific interface which controls the concurrent, multiple and asynchronous access of one or multiple clients to one or more drivers. I.e. it performs buffering, queuing, arbitration, multiplexing.

The handler does not change the content of the data.

In AUTOSAR the concept of pure handlers has been abandoned because the growing capabilities of the microcontroller hardware are incorporating handler functionality. It makes no sense to leave the powerful hardware resources unused and implement them in software instead.

This means: handler functionality is often incorporated in the driver (e.g. SPIHandlerDriver, ADC Driver).
Manager
A manager offers specific services for multiple clients. It is needed in all cases where pure handler functionality is not enough for accessing and using drivers.

Besides handler functionality, a manager can evaluate and change or adapt the content of the data.

In general, managers are located in the Services Layer

Example: The NVRAM manager manages the concurrent access to internal and/or external memory devices like flash and EEPROM memory. It also performs management of RAM mirrors, redundant, distributed and reliable data storage, data checking, provision of default values etc. For details refer to the AUTOSAR requirements documents.
Part 3 – Contents of Software Layers
Part 3 – Contents of Software Layers
ID: 03-01 Scope: Microcontroller Abstraction Layer

The µC Abstraction Layer consists of the following module groups:

- **Communication Drivers**
  Drivers for ECU onboard (e.g. SPI) and vehicle communication (e.g. CAN). OSI-Layer: Part of Data Link Layer
- **I/O Drivers**
  Drivers for analog and digital I/O (e.g. ADC, PWM, DIO)
- **Memory Drivers**
  Drivers for on-chip memory devices (e.g. internal Flash, internal EEPROM) and memory mapped external memory devices (e.g. external Flash)
- **Microcontroller Drivers**
  Drivers for internal peripherals (e.g. Watchdog, General Purpose Timer)
  Functions with direct µC access (e.g. Core test)
**Part 3 – Contents of Software Layers**  
**ID: 03-02 Scope: Complex Drivers**

A **Complex Driver** implements complex sensor evaluation and actuator control with direct access to the µC using specific interrupts and/or complex µC peripherals (like PCP, TPU), e.g.

- Injection control
- Electric valve control
- Incremental position detection

**Task:**
Fulfill the special functional and timing requirements for handling complex sensors and actuators

**Properties:**
Implementation: highly µC, ECU and application dependent  
Upper Interface: specified and implemented according to AUTOSAR (AUTOSAR interface)
The **Communication Hardware Abstraction** is a group of modules which abstracts from the location of communication controllers and the ECU hardware layout. For all communication systems a specific Communication Hardware Abstraction is required (e.g. for LIN, CAN, MOST, FlexRay).

**Example:** An ECU has a microcontroller with 2 internal CAN channels and an additional on-board ASIC with 4 CAN controllers. The CAN-ASIC is connected to the microcontroller via SPI. The communication drivers are accessed via bus specific interfaces (e.g. CAN Interface).

**Task:**

Provide equal mechanisms to access a bus channel regardless of it's location (on-chip / on-board)

**Properties:**

Implementation: μC independent, ECU hardware dependent and external device dependent

Upper Interface: bus dependent, μC and ECU hardware independent

**Example:**
This sheet has been inserted for understanding the following sheets: In many ECUs, a lot of onboard hardware devices like external EEPROM, external I/O ASICs, external watchdogs etc. are connected to the microcontroller via SPI.

The SPIHandlerDriver allows concurrent access of several clients to one or more SPI busses.

To abstract all features of a SPI microcontrollers pins dedicated to Chip Select shall directly be handled by the SPIHandlerDriver. That means those pins shall not be available in DIO Driver.

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

The different I/O devices are accessed via an I/O signal interface.

**Task:**
Represent I/O signals as they are connected to the ECU hardware (e.g. current, voltage, frequency).
Hide ECU hardware and layout properties from higher software layers.

**Properties:**
Implementation: µC independent, ECU hardware dependent
Upper Interface: µC and ECU hardware independent, dependent on signal type specified and implemented according to AUTOSAR (AUTOSAR interface)

**Example:**
- **I/O Hardware Abstraction**
- **I/O Signal Interface**
- **Driver for ext. ADC ASIC**
- **Driver for ext. I/O ASIC**
- **COM Drivers**
- **I/O Drivers**
- **SPhandler Driver**
- **DIO Driver**
- **ADC Driver**
- **Microcontroller (µC)**

\[\text{Diagram of AUTOSAR architecture}\]
Part 3 – Contents of Software Layers
ID: 03-05 Scope: Memory Hardware Abstraction

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

The memory drivers are accessed via memory specific abstraction/emulation modules (e.g. EEPROM Abstraction). By emulating an EEPROM interface and Flash hardware units a common access via Memory Abstraction Interface to both types of hardware is enabled.

Task:
Provide equal mechanisms to access internal (on-chip) and external (on-board) memory devices and type of memory hardware (EEPROM, Flash).

Properties:
Implementation: µC independent, external device dependent
Upper Interface: µC, ECU hardware and memory device independent

Example:

![Diagram showing Memory Hardware Abstraction and related components]
Part 3 – Contents of Software Layers  
ID: 03-06 Scope: Onboard Device Abstraction

The **Onboard Device Abstraction** contains drivers for ECU onboard devices which cannot be seen as sensors or actuators like system basic chip, external watchdog etc. Those drivers access the ECU onboard devices via the µC Abstraction Layer.

**Task:**  
Abstract from ECU specific onboard devices.

**Properties:**  
Implementation: µC independent, external device dependent  
Upper Interface: µC independent, partly ECU hardware dependent

**Example:**

- Onboard Device Abstraction
  - Driver for System Basic Chip
  - External Watchdog Driver
- COM Drivers
  - SPI Handler Driver
- I/O Drivers
  - DIO Driver

<table>
<thead>
<tr>
<th>Microcontroller (µC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Onboard Device Abstraction</td>
</tr>
<tr>
<td>Memory Drivers</td>
</tr>
<tr>
<td>Communication Drivers</td>
</tr>
<tr>
<td>I/O Drivers</td>
</tr>
<tr>
<td>Complex Drivers</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>µC</th>
</tr>
</thead>
</table>
Part 3 – Contents of Software Layers
ID: 03-07 Scope: Communication Services – General

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

Task:
1. Provide a uniform interface to the vehicle network for communication between different applications.
2. Provide uniform services for network management
3. Provide uniform interface to the vehicle network for diagnostic communication
4. Hide protocol and message properties from the application.

Properties:
Implementation: µC and ECU HW independent, partly dependent on bus type
Upper Interface: µC, ECU hardware and bus type independent

The communication services will be detailed for each relevant vehicle network system on the following pages.

Color code: Bus specific modules are marked gray.
Note: XCP is currently not discussed within AUTOSAR
The CAN Communication Services are a group of modules for vehicle network communication with the communication system CAN.

Task:
Provide a uniform interface to the CAN network. Hide protocol and message properties from the application.

Properties:
Implementation: µC and ECU HW independent, partly dependent on CAN

AUTOSAR COM and Diagnostic Communication Manager are the same for all vehicle network systems and exist as one instance per ECU. Generic NM is also the same for all vehicle network systems but will be instantiated per vehicle network system. Generic NM interface with CAN via underlying network dependent adapter (CAN NM).

A signal gateway is part of AUTOSAR COM to route signals. PDU based Gateway is part of PDU router.

IPDU multiplexing provides the possibility to add information to enable the multiplexing of I-PDUs (different contents but same IDs).

Upper Interface: µC, ECU hardware and network type independent (goal)

For refinement of GW architecture please refer to slide 04-050.

This is the solution realized in release 2.0.
The solution chosen for future releases is described on slide 03-08.
The **CAN Communication Services** are a group of modules for vehicle network communication with the communication system CAN.

**Task:**
Provide a uniform interface to the CAN network. Hide protocol and message properties from the application.

**Properties:**
Implementation: µC and ECU HW independent, partly dependent on CAN

AUTOSAR COM and Diagnostic Communication Manager are the same for all vehicle network systems and exist as one instance per ECU. Generic NM is also the same for all vehicle network systems but will be instantiated per vehicle network system. Generic NM interface with CAN via underlying network dependent adapter (CAN NM).

A signal gateway is part of AUTOSAR COM to route signals. PDU based Gateway is part of PDU router.
IPDU multiplexing provides the possibility to add information to enable the multiplexing of I-PDUs (different contents but same IDs). Upper Interface: µC, ECU hardware and network type independent (goal)

For refinement of GW architecture please refer to slide 04-050.

---

This is not the solution realized in release 2.0. This slide shows the intended solution for later releases then 2.0.
The LIN Communication Services contain:
- A LIN 2.0 compliant communication stack (LIN Interface) with
  - Scheduler for transmitting LIN frames
  - Diagnostic transport protocol, which can also be used for non
diagnostic purposes.
  - Signal packing and unpacking with a signal based API
  - A WakeUp and Sleep Interface
- An underlying LIN Driver

Note: Integration of LIN into AUTOSAR:
- The Scheduler and its interfaces are still used to decide the point of
time to send a LIN header.
- LIN has to be extended by a LIN NM which controls the
  WakeUp/Sleep API and allows the slaves to keep the bus awake
  (decentralized approach).
- The PDU router accesses the LIN Interface on PDU-Level, not on
  signal level. It is expected that a LIN driver gets rid of the signal
  packing/unpacking code, if no “signals” are configured in the ldf
  (LIN Description File).
- I-PDUs to be transmitted on LIN are requested by the Lin Interface
  at COM via the PDU router at the point in time it requires the data.
- The LIN 2.0 communication stack needs to be extended by a
  indication mechanism for event driven notification of the PDU
  Router when a LIN frame has been received. Otherwise, the PDU
  Router would need to poll the flag interface of the LIN 2.0
  communication stack.
- LIN TP does not support non-diagnostic data transfer \(\Rightarrow\) LIN will not
  support the transmission of AUTOSAR signals > 8 bytes.
- IPDU multiplexing provides the possibility to add information to
  enable the multiplexing of I-PDUs (different contents but same IDs).
The **FlexRay Communication Services** are a group of modules for vehicle network communication with the communication system FlexRay.

**Task:**
Provide a uniform interface to the FlexRay network. Hide protocol and message properties from the application.

**Properties:**
Implementation: μC and ECU HW independent, partly dependent on FlexRay

AUTOSAR COM and Diagnostic Communication Manager are the same for all vehicle network systems and exist as one instance per ECU. Generic NM is also the same for all vehicle network systems but will be instantiated per vehicle network system. The generic NM interfaces with FlexRay via underlying network dependent adapter (FlexRay NM).

A signal Gateway is part of AUTOSAR COM to route signals. PDU based Gateway is part of PDU Router. IPDU multiplexing provides the possibility to add information to enable the multiplexing of I-PDUs (different contents but same IDs).

Upper Interface: μC, ECU hardware and network type independent (goal)
**Part 3 – Contents of Software Layers**  
**ID: 03-10 Scope: Communication Services – LIN Slave**

**LIN Slaves** usually are „intelligent“ actuators and slaves that are seen as black boxes. As they provide very little hardware capabilities and resources it is not intended to shift AUTOSAR SW Components on LIN Slaves.

LIN Slave ECUs can be integrated into the AUTOSAR VFB using their Node Capability Descriptions. They are seen as non-AUTOSAR ECUs. Please reference to the VFB specification.

That means: LIN Slaves can be connected as complete ECUs. But they are not forced to use the AUTOSAR SW Architecture. Perhaps they can use some standard AUTOSAR modules (like EEPROM, DIO).
Reason: LIN slaves usually have very limited memory resources or are ASICs with „hard-coded“ logic.
Part 3 – Contents of Software Layers
ID: 03-12 Scope: Memory Services

The Memory Services consist out of one module, the NVRAM Manager, responsible for the management of non volatile data (read/write from different memory drivers). It expects a RAM mirror as data interface to the application for fast read access.

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

Properties:
Implementation: µC and ECU hardware independent, highly configurable
Upper Interface: µC and ECU hardware independent
    specified and implemented according to AUTOSAR
    (AUTOSAR interface)

Example:
The **System Services** are a group of modules and functions which can be used by modules of all layers. Examples are Real Time Operating System, Error Manager and Library Functions (like CRC, interpolation etc.). Some of these services are μC dependent (like OS), partly ECU hardware and application dependent (like ECU State Manager) or hardware and μC independent.

**Task:**
Provide basic services for application and basic software modules.

**Properties:**
Implementation: partly μC, ECU hardware and application specific
Upper Interface: μC and ECU hardware independent

---

**Example:**

- **System Services**
  - ECU State Manager
  - Communication Manager
  - FIM Function Inhibition Manager
  - DEM Diagnostic Event Manager
  - Watchdog Manager
  - Development Error Tracer
  - Synchronization and Time Service

- **Microcontroller (μC)**
  - Free Run. Timer
  - Interpolation Lib
  - CRC Lib
  - Crypto Lib
  - Bit Lib
  - Other Libs
  - Flash Check

---

**Diagram:**

- **AUTOSAR OS**
- **BSW Scheduler**
- **Complex Drivers**
- **Onboard Dev. Abstraction**
- **Memory Services**
- **Communication Services**
- **COM HW Abstraction**
- **I/O HW Abstraction**
- **I/O Drivers**
- **Memory Drivers**
- **Communication Drivers**
- **Microcontroller Drivers**
**Part 3 – Contents of Software Layers**  
**ID: 03-14 Scope: Sensor/Actuator AUTOSAR Software Components**

The **Sensor/Actuator AUTOSAR Software Component** is a specific type of AUTOSAR Software Component for sensor evaluation and actuator control. Though not belonging to the AUTOSAR Basic Software it is described here due to its strong relationship to local signals. It has been decided to locate the Sensor/Actuator SW Components above the RTE for integration reasons (standardized interface implementation and interface description). Because of their strong interaction with raw local signals they are **not relocatable** in most cases. Tasks and interfaces are similar to that of a Complex Driver. Examples of tasks of a Sensor/Actuator Component are switch debouncing, battery voltage monitoring, DC motor control, lamp control etc.

**Task:**
Abstract from the specific physical properties of sensors and actuators.

**Properties:**
Implementation: µC and ECU HW independent, sensor and actuator dependent

---

**Example:**

- **Application Layer**
  - Actuator Software Component
  - Sensor Software Component

- **Basic Software**
  Interfaces to (e.g.)
  - I/O HW Abstraction (access to I/O signals)
  - Memory Services (access to calibration data)
  - System Services (access to Error Manager)
Part 4 – Interfaces
4.1 General Rules
**Part 4.1 – Interfaces: General Rules**

**ID: 04-002 General Interfacing Rules**

### Horizontal Interfaces

- **Services Layer**: horizontal interfaces are allowed. Example: Error Manager saves fault data using the NVRAM manager.

- **ECU Abstraction Layer**: horizontal interfaces are allowed.

- **A complex driver may use all other BSW modules**

- **μC Abstraction Layer**: horizontal interfaces are not allowed. Exception: configurable notifications are allowed due to performance reasons.

### Vertical Interfaces

- **One Layer may access all interfaces of the SW layer below**

- **Bypassing of one software layer should be avoided**

- **Bypassing of two or more software layers is not allowed**

- **Bypassing of the μC Abstraction Layer is not allowed**

- **A module may access a lower layer module of another layer group (e.g. SPI for external hardware)**

- **All layers may interact with system services.**

---

V2.0.0, Page 39

- AUTOSAR Confidential -
### Part 4.1 – Interfaces: General Rules

#### ID: 04-003 Layer Interaction Matrix

This matrix shows the possible interactions between AUTOSAR Basic Software layers.

- ✓ “is allowed to use”
- ✗ “is not allowed to use”
- △ “restricted use (callback only)”

The matrix is read row-wise:

**Example:** “I/O Drivers are allowed to use System Services and Hardware, but no other layers”.

(Gray background indicates “non-Basic Software” layers)

<table>
<thead>
<tr>
<th>AUTOSAR SW Components / RTE</th>
<th>System Services</th>
<th>Memory Services</th>
<th>Communication Services</th>
<th>Complex Drivers</th>
<th>I/O Hardware Abstraction</th>
<th>Onboard Device Abstraction</th>
<th>Memory Hardware Abstraction</th>
<th>Communication Hardware Abstraction</th>
<th>Microcontroller Drivers</th>
<th>Memory Drivers</th>
<th>Communication Drivers</th>
<th>I/O Drivers</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Services</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Memory Services</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Communication Services</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Complex Drivers</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>I/O Hardware Abstraction</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Onboard Device Abstraction</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Memory Hardware Abstraction</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Communication Hardware Abstraction</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Microcontroller Drivers</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Memory Drivers</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Communication Drivers</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>I/O Drivers</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
</tr>
</tbody>
</table>
Part 4 – Interfaces

4.3 Error Handling and Reporting Concept
Part 4.3 – Interfaces: Error Handling and Reporting Concept
ID: 04-008 References

References
The following specification documents and requirements are referenced / applicable:

AUTOSAR_SRS_General.doc (General SRS)
[BSW00337] Classification of errors (General SRS)

[BSW00338] Reporting of development errors (General SRS)
[BSW00339] Reporting of production relevant errors and exceptions (General SRS)

[BSW00369] Do not return development error codes via API
[BSW00323] API parameter checking
[BSW00331] Separation of error and status values
[BSW00327] Error values naming convention

AUTOSAR_SRS_SPAL.doc (SPAL SRS)
[BSW157] Notification mechanisms of drivers and handlers (SPAL SRS)
Part 4.3 – Interfaces: Error Handling and Reporting Concept
ID: 04-009 Error Classification (1)

Types of errors

Hardware errors / failures
- Root cause: Damage, failure or 'value out of range', detected by software
- Example 1: EEPROM cell is not writable any more
- Example 2: Output voltage of sensor out of specified range

Software errors
- Root cause: Wrong software or system design, because software itself can never fail.
- Example 1: wrong API parameter (EEPROM target address out of range)
- Example 2: Using not initialized data

Software and System exceptions
- Example 1: CAN receive buffer overflow
- Example 2: time-out for receive messages
Part 4.3 – Interfaces: Error Handling and Reporting Concept
ID: 04-010 Error Classification (2)

Time of error occurrence according to product life cycle

Development
Those errors shall be detected and fixed during development phase. In most cases, those errors are software errors. The detection of errors that shall only occur during development can be switched off for production code (by static configuration namely preprocessor switches).

Production / series
Those errors are hardware errors and software exceptions that cannot be avoided and are also expected to occur in production code.

Influence of error on system

Severity of error (impact on the system)
- No influence
- Short misbehavior
- Functional degradation
- Full ECU failure

Failure mode in terms of time
- Permanent errors
- Transient / sporadic errors
Part 4.3 – Interfaces: Error Handling and Reporting Concept
ID: 04-011 Error Reporting – Alternatives

There are several alternatives to report an error:

Via API
Inform the caller about success/failure of an operation.

Via statically definable callback function (notification)
Optional.
Inform the caller about failure of an operation
→ [BSW157] Notification mechanisms of drivers and handlers (SPAL SRS)

Via central Error Hook (Development Error Tracer)
Optional.
For logging and tracing errors during product development. Can be switched off for production code.
→ [BSW00338] Reporting of development errors (General SRS)

Via central Error Function (AUTOSAR Diagnostic Event Manager)
For error reaction and logging in series (production code)
→ [BSW00339] Reporting of production relevant errors and exceptions (General SRS)
Part 4.3 – Interfaces: Error Handling and Reporting Concept
ID: 04-012 Error Reporting – Basic Concept

Each basic software module distinguishes between two types of errors:

1. Development Errors
   The detection and reporting can be statically switched on/off

2. Production relevant errors and exceptions
   This detection is 'hard coded' and always active.

→ [BSW00337] Classification of errors (General SRS)
Error reporting via API
Informs the caller about success/failure of an operation.
→ [BSW00327] Error values naming convention

Basic return type
Success: E_OK (value: 0)
Failure: E_NOT_OK (value: 1)

Specific return type
If different exceptions are required for production code, own return types have to be defined. Different exceptions shall only be used if the caller can really handle the exception. Development errors shall not be returned in detail via the API. They can be reported to the Development Error Tracer (see 04-014).

Example: services of EEPROM driver
Success: EEP_E_OK
General failure (service not accepted): EEP_E_NOT_OK
Busy (service not accepted): EEP_E_BUSY

The specific return values have to be specified within the module’s software specification. For API return, no additional, implementation specific return values are allowed, only the ones contained in the AUTOSAR software specification. Otherwise the calling module depends on the specific implementation of a software module.
Part 4.3 – Interfaces: Error Handling and Reporting Concept
ID: 04-014 Error Reporting – Introduction

Error reporting via Diagnostic Event Manager (DEM)
For reporting production / series errors.
Those errors have a defined reaction within the configuration of this module, e.g.:
- Writing to error memory
- Disabling of ECU functions
- Switching of ECU mode etc.

The Diagnostic Event Manager is a standard AUTOSAR module which is definitely available in production code and whose functionality is specified in the AUTOSAR project.

Error reporting via Development Error Tracer (DET)
For reporting development errors.
This Error tracer is mainly intended for tracing and logging errors during development. Within the Error Tracer many mechanisms are possible, e.g.:
- Count errors
- Write error information to ring buffer in RAM
- Send error information via serial interface to external logger
- Infinite Loop, Breakpoint

The Development Error Tracer is just a help for SW development and integration and is not necessarily contained in the production code. The API is defined, but the functionality can be chosen/implemented by the developer according to his specific needs.
### Development error detection and reporting

- Development error detection and reporting is statically configurable (on/off) per module.

#### API and Interface

- **Dem_ReportErrorStatus** (statically configured Error-ID, Status)
- **Det_ReportError** (ModuleId, ApiId, ErrorId)

#### Diagram

- **Diagnostic Event Manager**
- **Development Error Tracer**
- **Software Module**
### Part 4.3 – Interfaces: Error Handling and Reporting Concept

**ID: 04-016 Error Reporting – Example for Development Errors**

#### Caller

**Det_ReportError(ModuleId, ApiId, ErrorId)**

- **API call with wrong parameters**
  - Return (E_NOT_OK)

#### Software Module

#### Development Error Tracer

- **API call with correct parameters**
  - Return (E_OK)

- **Det_ReportError(ModuleId, ApiId, ErrorId)**
Part 4.3 – Interfaces: Error Handling and Reporting Concept
ID: 04-017 Error Reporting – Diagnostic Event Manager

API
The Diagnostic Event Manager has semantically the following API:

Dem_ReportErrorStatus(EventId, EventStatus)

Problem: the error IDs passed with this API have to be ECU wide defined, have to be statically defined and have to occupy a compact range of values for efficiency reasons. Reason: The Diagnostic Event Manager uses this ID as index for accessing ROM arrays.

Error numbering concept: XML based error number generation

Properties:
- Source and object code compatible
- Single name space for all production relevant errors
- Tool support required
- Consecutive error numbers → Error manager can easily access ROM arrays where handling and reaction of errors is defined

Process:
1. Each BSW Module declares all production code relevant error variables it needs as “extern”
2. Each BSW Module stores all error variables that it needs in the ECU configuration description (e.g. CAN_E_BUS_OFF)
3. The configuration tool of the Diagnostic Event Manager parses the ECU configuration description and generates a single file with global constant variables that are expected by the SW modules (e.g. const Dem_EventIdType CAN_E_BUS_OFF=7; or #define CAN_E_BUS_OFF ((Dem_EventIdType)7))
4. The reaction to the errors is also defined in the Error Manager configuration tool. This configuration is project specific.
API
The Development Error Tracer has syntactically the following API:
\[ \text{Det\_ReportError}(\text{uint16 ModuleId}, \text{uint8 ApiId}, \text{uint8 ErrorId}) \]

Error numbering concept
ModuleId (uint16)
The ModuleId contains the AUTOSAR module ID from the Basic Software Module List. As the range is 16 Bit, future extensions for development error reporting of application SW-C are possible. The Basic SW uses only the range from 0..255.

ApiId (uint8)
The API-IDs are specified within the software specifications of the BSW modules. They can be #defines or constants defined in the module starting with 0.

ErrorId (uint8)
The Error-IDs are specified within the software specifications of the BSW modules. They can be #defines defined in the module’s header file.

If there are more errors detected by a particular software module which are not specified within the AUTOSAR module software specification, they have to be documented in the module documentation.

All Error-IDs have to be specified in the ECU configuration description.
Part 4.3 – Interfaces: Error Handling and Reporting Concept
ID: 04-019 Error Reporting – Development Error Tracer (2)

Error detection

The detection and reporting of development errors to the Development Error Tracer can be statically switched on/off per module (preprocessor switch or two different object code builds of the module).
Part 4 – Interfaces
4.4 Interaction of Layers – Example “Memory”
The following pages explain using the example „memory“:

➢ How do the software layers interact?
➢ How do the software interfaces look like?
➢ What is inside the ECU Abstraction Layer?
➢ How can abstraction layers be implemented efficiently?
This example shows how the NVRAM Manager and the Watchdog Manager interact with drivers on an assumed hardware configuration.

Let the ECU hardware have an external EEPROM (ST16RF42) and an external watchdog connected to the microcontroller via the same SPI.

The SPIHandlerDriver controls the concurrent access to the SPI hardware and has to give the watchdog access a higher priority than the EEPROM access.

The microcontroller is assumed to have also an internal flash which is used in parallel to the external EEPROM. The EEPROM Abstraction and the Flash EEPROM Emulation have an API that is semantically identical.

The Memory Abstraction Interface can be realized in the following ways:
- routing during runtime based on device index (int/ext)
- routing during runtime based on the block index (e.g. > 0x01FF = external EEPROM)
- routing during configuration time via ROM tables with function pointers inside the NVRAM Manager (in this case the Memory Abstraction Interface only exists „virtually“)
Part 4.4 – Interfaces: Interaction of Layers – Example “Memory”
ID: 04-023 Closer Look at Memory Hardware Abstraction

Architecture Description
The NVRAM Manager accesses drivers via the Memory Abstraction Interface. It addresses different memory devices using a device index.

Interface Description
The Memory Abstraction Interface could have the following interface (e.g. for the write function):

```
Std_ReturnType MemIf_Write
{
    uint8 DeviceIndex,
    uint8 BlockNumber,
    uint8 *DataBufferPtr
}
```

The EEPROM Abstraction as well as the Flash EEPROM Emulation could have the following interface (e.g. for the write function):

```
Std_ReturnValue Ea_Write
{
    uint8 BlockNumber,
    uint8 *DataBufferPtr
}
```
Part 4.4 – Interfaces: Interaction of Layers – Example “Memory”
ID: 04-024 Implementation of Memory Abstraction Interface (1)

**Situation 1: only one NV device type used**
This is the usual use case. In this situation, the Memory Abstraction could be implemented as a simple macro which neglects the DeviceIndex parameter. The following example shows the write function only:

File MemIf.h:
```c
#include "Ea.h"; /* for providing access to the EEPROM Abstraction */
...
#define MemIf_Write(DeviceIndex, BlockStartAddress, DataBufferPtr) \
    Ea_Write(BlockStartAddress, DataBufferPtr)
```

File MemIf.c:
Does not exist

**Result:**
No additional code at runtime, the NVRAM Manager virtually accesses the EEPROM Abstraction or the Flash Emulation directly.
Situation 2: **two or more** different types of NV devices used
In this case the DeviceIndex has to be used for selecting the correct NV device. The implementation can also be very efficient by using arrays of pointer to function. The following example shows the write function only:

**File EepIf.h:**
```c
typedef uint32 MemIf_AddressType;
...

extern const WriteFctPtrType WriteFctPtr[2];
```

```c
#define MemIf_Write(DeviceIndex, BlockStartAddress, DataBufferPtr) 
  WriteFctPtr[DeviceIndex](BlockStartAddress, DataBufferPtr)
```

**File MemIf.c:**
```c
#include „Ea.h“; /* for getting the API function addresses */
#include „Fee.h“; /* for getting the API function addresses */
#include „MemIf.h“; /* for getting the WriteFctPtrType */

const WriteFctPtrType WriteFctPtr[2] = {Ea_Write, Fee_Write};
```

**Result:**
The same code and runtime is needed as if the function pointer tables would be inside the NVRAM Manager. The Memory Abstraction Interface causes no overhead.
Part 4.4 – Interfaces: Interaction of Layers – Example “Memory”
ID: 04-026 Conclusion

Conclusions:

- Abstraction Layers can be implemented very efficiently
- Abstraction Layers can be scalable
- The Memory Abstraction Interface eases the access of the NVRAM Manager to one or more EEPROM and Flash devices
- The architectural targets and requirements are fulfilled
- The EEPROM Abstraction Interface contains functionality which could not be replaced very easy by Macros, therefore this layer cannot be substituted.
Part 4 – Interfaces
4.5 Interaction of Layers – Example “Communication”
**Part 4.4 – Interfaces: Interaction of Layers – Example “Communication”**

**ID: 04-51 PDU Flow through the Layered Architecture**

Explanation of terms:

- **SDU**
  SDU is the abbreviation of “Service Data Unit”. It is the data passed by an upper layer, with the request to transmit the data. It is as well the data which is extracted after reception by the lower layer and passed to the upper layer. A SDU is part of a PDU.

- **PCI**
  PCI is the abbreviation of “Protocol Control Information”. This Information is needed to pass a SDU from one instance of a specific protocol layer to another instance. E.g. it contains source and target information. The PCI is added by a protocol layer on the transmission side and is removed again on the receiving side.

- **PDU**
  PDU is the abbreviation of “Protocol Data Unit”. The PDU contains SDU and PCI. On the transmission side the PDU is passed from the upper layer to the lower layer, which interprets this PDU as its SDU.
ID: 04-52 SDU and PDU Naming Conventions

Naming of PDUs and SDUs respects the following rules:

For PDU:

<bus prefix> <layer prefix> - PDU

For SDU:

<bus prefix> <layer prefix> - SDU

The bus prefix and layer prefix are described in the following table:

<table>
<thead>
<tr>
<th>ISO Layer</th>
<th>Layer Prefix</th>
<th>AUTOSAR Modules</th>
<th>PDU Name</th>
<th>CAN prefix</th>
<th>LIN prefix</th>
<th>FlexRay prefix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Presentation</td>
<td>I</td>
<td>COM, DCM</td>
<td>I-PDU</td>
<td>N/A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Interaction</td>
<td>I</td>
<td>PDU router, PDU multiplexer</td>
<td>I-PDU</td>
<td>N/A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Network</td>
<td>N</td>
<td>TP Layer</td>
<td>N-PDU</td>
<td>CAN SF</td>
<td>LIN SF</td>
<td>FR SF</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>CAN FF</td>
<td>LIN FF</td>
<td>FR FF</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>CAN CF</td>
<td>LIN CF</td>
<td>FR CF</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>CAN FC</td>
<td>LIN FC</td>
<td>FR FC</td>
</tr>
<tr>
<td>Data link</td>
<td>L</td>
<td>Driver, Interface</td>
<td>L-PDU</td>
<td>CAN</td>
<td>LIN</td>
<td>FR</td>
</tr>
</tbody>
</table>

Examples:
- I-PDU or I-SDU
- CAN FF N-PDU or FR CF N-SDU
- LIN L-PDU or FR L-SDU
Routing Components

- **PDU Router1**
  - Provides routing of PDUs between different abstract communication controllers and upper layers
  - Scale of the Router is ECU specific (down to no size if e.g. only one communication controller exists)
  - Provides TP routing on-the-fly. Transfer of TP data is started before full TP data is buffered.

- **COM**
  - Provides routing of individual signals or groups of signals between different I-PDUs.

- **NM synchronization**
  - Synchronization of Network States of different communication channels connected to an ECU via the network managements handled by the NM Gateway

1 The Interface between PduR and Tp differs significantly compared to the interface between PduR and the Ifs.i In case of TP involvement a handshake mechanism is implemented allowing the transmission of I-Pdus > Frae size.
Part 4 – Interfaces
4.6 Interaction of Layers – Example “ECU State Manager”
Part 4.6 – Interfaces: Interaction of Layers – Example “ECU State Manager”
ID: 04-071 Interaction with ECU State Manager

This figure does not show all interactions between all modules. It is a discussion base only.
Part 4 – Interfaces
4.7 Reliability Mechanisms Among Communication
## Part 4.7 – Interfaces: Reliability Mechanisms Among Communication

**ID: 04-061 Overview and Assignment to SW Layers (1)**

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Use Case</th>
<th>Benefit</th>
<th>SW Layer / Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bus frame checksum protection</td>
<td>State of the art protocol features of CAN, FlexRay, LIN</td>
<td>Data integrity</td>
<td>Bus hardware (CAN, FlexRay) or bus driver (LIN)</td>
</tr>
<tr>
<td>AUTOSAR Signal checksum protection</td>
<td>Additional security for safety related data within a PDU.</td>
<td>Data integrity</td>
<td>Application (see BSW Release Planning)</td>
</tr>
<tr>
<td></td>
<td>Data inconsistencies occurred during writing to/reading from communication data buffers can be detected.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>To achieve the best protection, this mechanism should be located rather in a high software layer.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AUTOSAR Signal sequence counter</td>
<td>Allows the receiver of a periodic AUTOSAR signal to detect if it has missed to receive one signal.</td>
<td>Data integrity Detection of message loss Sender „alive“ detection</td>
<td>Application (see BSW Release Planning)</td>
</tr>
<tr>
<td></td>
<td>To achieve the best protection, this mechanism should be located rather high in the software (example: a sequence counter added in the bus driver will not allow to detect that a transmission request has been blocked in a higher layer, e.g. AUTOSAR COM).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AUTOSAR Signal receive timeout supervision</td>
<td>State of the art feature of all OEM’s COM implementations (also OSEK COM 3.0.2).</td>
<td>Detection of sender failure and/or message loss</td>
<td>COM</td>
</tr>
<tr>
<td></td>
<td>Works only for signals contained in periodic bus frames.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Part 4.7 – Interfaces: Reliability Mechanisms Among Communication

**ID: 04-062 Overview and Assignment to SW Layers (2)**

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Use Case</th>
<th>Benefit</th>
<th>SW Layer / Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiple (repeated) transmission of event triggered bus frames</td>
<td>Allows to „force through“ a bus frame. Used on event triggered, arbitrating network systems like CAN. Example: transmission of window lift switch activation/deactivation event via CAN. Often used when transmission problems occur on an overloaded or not well scheduled CAN bus.</td>
<td>Higher probability that the bus frame reaches the receiving ECU and it’s application</td>
<td>COM</td>
</tr>
</tbody>
</table>
| Redundant transmission and reception                                       | a) Transmission within different frames on the same bus channel  
 b) Transmission via different physical busses (e.g. 2 physical FlexRay Channels)  
 Allows the detection of data inconsistencies. Additionally, b) has true fail safe capability. | Data integrity  
 Fail safe capability                                                                 | RTE or application                     |
| Acknowledgement                                                          | Allows the detection of communication failures on sender side. Applicable only in 1:1 communication (1:n would require collecting acknowledgements ...)  | Assurance that the receiver has received the data.                                          | Application (see BSW Release Planning) |
Part 5 – Configuration
Part 5 – Configuration
ID: 05-000 Overview

The AUTOSAR Basic Software supports the following configuration classes:

1. **Pre compile time**
   - Preprocessor instructions
   - Code generation (selection or synthetization)

2. **Link time**
   - Constant data outside the module; the data can be configured after the module has been compiled

3. **Post build time**
   - Loadable constant data outside the module. Very similar to [2], but the data is located in a specific memory segment that allows reloading (e.g. reflashing in ECU production line)

In many cases, the configuration parameters of one module will be of different configuration classes. Example: a module providing post build time configuration parameters will still have some parameters that are pre compile time configurable.
Use cases
Pre compile time configuration would be chosen for
- Enabling/disabling optional functionality
  This allows to exclude parts of the source code that are not needed
- Optimization of performance and code size
  Using #defines results in most cases in more efficient code than access to constants or even access to constants via pointers.
  Generated code avoids code and runtime overhead.

Restrictions
- The module must be available as source code
- The configuration is static. To change the configuration, the module has to be recompiled

Required implementation
Pre compile time configuration shall be done via the module's configuration file (*_Cfg.h) or by code generation:
Example 1: Enabling/disabling functionality
File Spi_Cfg.h:
#define SPI_DEV_ERROR_DETECT ON

File Spi.c (available as source code):
#include "Spi_Cfg.h"  /* for getting the configuration parameters */

#if (SPI_DEV_ERROR_DETECT == ON)
Det_ReportError(Spi_ModuleId, 3, SPI_E_PARAM_LENGTH);
#endif
**Example 2: Event IDs reported to the DEM**

File NvM.xml

Specifies that it needs the event symbol **NVM_E_WRITE_ALL_FAILED** for production error reporting.

File Dem_Cfg.h (generated by DEM configuration tool):

```c
typedef uint8 Dem_EventIdType; /* total number of events = 46 => uint8 sufficient */

#define FLS_E_ERASE_FAILED 1
#define FLS_E_WRITE_FAILED 1
#define FLS_E_READ_FAILED 1
#define FLS_E_UNEXPECTED_FLASH_ID 2
#define NVM_E_WRITE_ALL_FAILED 3
#define CAN_E_TIMEOUT 4
...```

File Dem.h:

```c
#include "Dem_Cfg.h" /* for providing access to event symbols */
```

File NvM.c (available as source code):

```c
#include "Dem.h" /* for reporting production errors */

Dem_SetEventStatus(NVM_E_WRITE_ALL_FAILED, DEM_EVENT_STATUS_PASSED);
```
Part 5 – Configuration  
ID: 05-003 Link Time

Use cases
Link time configuration would be chosen for
- Configuration of modules that are only available as object code (e.g. due to safety, IP protection or warranty reasons)
- Selection of configuration set during link time

Required implementation
1. One configuration set, no runtime selection
Configuration data shall be captured in external constants. These external constants are located in a separate file. The module has direct access to these external constants.
Example 1: Event IDs reported to the DEM by a module (CAN Interface) that is available as object code only

File CanIf.xml
Specifies that it needs the event symbol CANIF_E_INVALID_TXPDUID for production error reporting.

File Dem_Cfg.h (generated by DEM configuration tool):
typedef uint16 Dem_EventIdType; /* total number of events = 380 => uint16 required */

#define FLS_E_UNEXPECTED_FLASH_ID    1
#define NVM_E_WRITE_ALL_FAILED       2
#define CAN_E_TIMEOUT                3
#define CANIF_E_INVALID_TXPDUID      4
...

File CanIf_Cfg.c:
const Dem_EventIdType CanIf_InvalidTxPduId = CANIF_E_INVALID_TXPDUID;

File CanIf.c (available as object code):
#include „Dem.h“       /* for reporting production errors       */

Dem_SetEventStatus(CanIf_InvalidTxPduId, DEM_EVENT_STATUS_FAILED);

Note: the complete include file structure with all forward declarations is not shown here to keep the example simple.
Example 1: Event IDs reported to the DEM by a module (CAN Interface) that is available as object code only

**Problem**

Dem_EventIdType is also generated depending of the total number of event IDs on this ECU. In this example it is represented as uint16. The Can Interface uses this type, but is only available as object code.

**Solution**

In the contract phase of the ECU development, a bunch of variable types (including Dem_EventIdType) have to be fixed and distributed for each ECU. The object code suppliers have to use those types for their compilation and deliver the object code using the correct types.

In a multi software supplier project these type negotiations will cause a complex process. This process has to be supported by the configuration process. That is the price you have to pay using object code delivery.
Part 5 – Configuration
ID: 05-007 Post Build Time

Use cases
Post build time configuration would be chosen for
- Configuration of data that is not known during ECU build time (e.g. communication matrix)
- Configuration of data that is likely to change or has to be adapted after ECU build time (e.g. end of line, during test & calibration)
- Reusability of ECUs across different product lines (same code, different configuration data)

Restrictions
- Implementation requires dereferencing which will have impact on performance and code size

Required implementation
1. One configuration set, no runtime selection (loadable)
   Configuration data shall be captured in external constant structs. These external structs are located in a separate memory segment that can be individually reloaded.

1. 1..n configuration sets, no runtime selection possible (selectable)
   Configuration data shall be captured within external constant structs. These configuration structures are located in one separate file. The module gets a pointer to one of those structs at initialization time. The struct can be selected at each initialization.
Example 1
If the configuration data is fixed in memory size and position, the module has direct access to these external structs.
If the configuration can vary in memory position and/or size, the module gets a pointer to the configuration struct at initialization.
Required implementation 2: Configuration of CAN Driver that is available as object code only; multiple configuration sets can be selected during runtime.

File Can_Cfg.c:

```c
#include "Can.h" /* for getting Can_ConfigType */
const Can_ConfigType MyFirstStupidCanConfig =
{
    Can_BitTiming = 0xDF,
    Can_AcceptanceMask1 = 0xFFFFFFFF,
    Can_AcceptanceMask2 = 0xFFFFFFFF,
    Can_AcceptanceMask3 = 0x00034DFF,
    Can_AcceptanceMask4 = 0x00FF0000
};
```

File EcuFirmware.c:

```c
#include "Can.h"     /* for initializing the CAN Driver */
Can_Init(&MyFirstStupidCanConfig);
```

File Can.c (available as object code):

```c
#include "Can.h"     /* for getting Can_ConfigType */
void Can_Init(Can_ConfigType* Config)
{
    /* write the init data to the CAN HW */
}
```
Part 5 – Configuration
ID: 05-009 Variants

Different use cases require different kinds of configurability:

**Example use cases:**
- Reprogrammable PDU routing tables in gateway (post build time configurable PDU Router required)
- Statically configured PDU routing with no overhead (Pre-Compile time configuration of PDU Router required)

To allow the implementation of such different use cases in each BSW module for each module up to 3 variants can be specified:
- A variant is a dedicated assignment of the configuration parameters of a module to configuration classes
- Within a variant a configuration parameter can be assigned to only ONE configuration class
- Within a variant a configuration classes for different configuration parameters can be different (e.g. Pre-Compile for development error detection and post-build for reprogrammable PDU routing tables)
- It is possible and intended that specific configuration parameters are assigned to the same configuration class for all variants (e.g. development error detection is in general pre-compile time configurable).
Part 6 – Scheduling
Part 6 – Scheduling  
ID: 06-001 Basic Scheduling Concepts of the BSW

- BSW Scheduling shall assure
  - Correct timing behavior of the BSW, i.e., correct interaction of all BSW modules with respect to time
  - Data consistency

- Single BSW modules do not know about
  - ECU wide dependencies
  - Scheduling implications

- Centralize the BSW schedule in the BSW Scheduler implemented by the ECU/BSW integrator
  - Eases the integration task
  - Enables to apply different scheduling strategies
    - Preemptive, non-preemptive, ...
  - Enables to apply different data consistency mechanisms
  - Enables to reduce resources (e.g., minimize the number of tasks)

- Restrict the usage of OS functionality
  - Only the Schedule Module shall use OS objects or OS services (exceptions: EcuM, ...)
  - Rationale:
    - Scheduling of the BSW shall be transparent to the system (integrator)
    - Enables to reduce the usage of OS resources (Tasks, Resources, ...)
    - Enables to re-use modules in different environments
Part 6 – Scheduling
ID: 06-002 Scheduling Objects ...

BSW Scheduling objects
- Main functions
  - n per module
  - located in all layers
- Interrupt Service Routines (ISR cat. II)
  - located only in MCA Layer
- Runnable Entities
  - located only in modules with AUTOSAR interfaces (Service Layer or ECU Abstraction Layer)
Part 6 – Scheduling
ID: 06-003 ... and Triggers

BSW Events
- TimingEvent
- SporadicEvent
- InterruptEvent

RTE Events
- DataReceivedEvent
- OperationInvokedEvent

BSW Scheduling objects
- Main functions
  - can be triggered in all layers by
    - TimingEvents
    - SporadicEvents
- Interrupt Service Routines (ISR cat. II)
  - can be triggered only in MCA Layer by
    - InterruptEvent
- Runnable Entities
  - triggered by data-based RTEEvents
## Part 6 – Scheduling

### ID: 06-004 Transformation Process (1)

<table>
<thead>
<tr>
<th>Logical Architecture (Model)</th>
<th>Technical Architecture (Implementation)</th>
</tr>
</thead>
<tbody>
<tr>
<td>➢ Ideal concurrency</td>
<td>➢ Restricted concurrency</td>
</tr>
<tr>
<td>➢ Unrestricted resources</td>
<td>➢ Restricted resources</td>
</tr>
<tr>
<td>➢ Only real data dependencies</td>
<td>➢ Real data dependencies</td>
</tr>
<tr>
<td>➢ Dependencies given by restrictions</td>
<td></td>
</tr>
</tbody>
</table>

- Scheduling objects
- Trigger
  - BSW events
  - RTE events
- Sequences of scheduling objects
- ...

- OS objects
  - Tasks
  - ISRs
  - Alarms
  - Resources
  - OS services
- Sequences of scheduling objects within tasks
- Sequences of tasks
- ...

### Transformation

- Mapping of scheduling objects to OS Tasks
- Specification of sequences of scheduling objects within tasks
- Specification of task sequences
- Specification of a scheduling strategy
- ...

- AUTOSAR Confidential -
Part 6 – Scheduling
ID: 06-005 Transformation Process (2)

Logical Architecture (Model)

➢ BSW Module Description Template

Technical Architecture (Implementation)

➢ ECU Configuration Template
➢ Programming Language „C“

 Transformation

➢ ECU Configuration Template
Part 6 – Scheduling
ID: 06-006 Transformation Process – Example 1

Logical Architecture (Model)  

Task1 {
  Zzz_MainFunction_Bbb();
  Yyy_MainFunction_Aaa();
  Xxx_MainFunction_Aaa();
}

Technical Architecture (Schedule Module ScM)

Task1 {
  Zzz_MainFunction_Bbb();
  Yyy_MainFunction_Aaa();
  Xxx_MainFunction_Aaa();
  ...
}

- Mapping of scheduling objects to OS Tasks
- Specification of sequences of scheduling objects within tasks
Part 6 – Scheduling
ID: 06-007 Transformation Process – Example 2

Logical Architecture (Model)

- Xxx_MainFunction_Bbb();
- Yyy_MainFunction_Bbb();

Technical Architecture (Schedule Module ScM)

- Task2 {
  ...
  Xxx_MainFunction_Bbb();
  ...
}

- Task3 {
  ...
  Yyy_MainFunction_Bbb();
  ...
}

- Mapping of scheduling objects to OS Tasks
Part 6 – Scheduling
ID: 06-008 Data Consistency – Motivation

- Access to resources by different and concurrent entities of the implemented technical architecture (e.g., main functions and/or other functions of the same module out of different task contexts)

Logical Architecture (Model)

- XXX_Module
- XXX_MainFunction();
- YYY_AccessResource();
- Resource
- YYY_MainFunction();
- YYY_Module

Technical Architecture (Schedule Module ScM)

> Data consistency strategy to be used
- Sequence
- Interrupt blocking
- Cooperative Behavior
- Semaphores (OSEK Resources)
- Copies of ...
- ...

- AUTOSAR Confidential -
Data consistency is ensured by interrupt blocking.

```c
#define ScM_EnterCS_Resource DisableAllInterrupts
#define ScM_ExitCS_Resource EnableAllInterrupts

Yyy_AccessResource() {
    ...
    ScM_EnterCS_Resource
    <access_to_shared_resource>
    ScM_ExitCS_Resource
    ...
}

Yyy_MainFunction() {
    ...
    ScM_EnterCS_Resource
    <access_to_shared_resource>
    ScM_ExitCS_Resource
    ...
}
```
Logical Architecture (Model) / Technical Architecture (Schedule Module ScM)

Implementation of Schedule Module ScM

#define ScM_EnterCS_Resource
#define ScM_ExitCS_Resource

Yyy_AccessResource() {
    ...
    ScM_EnterCS_Resource
    <access_to_shared_resource>
    ScM_ExitCS_Resource
    ...
}

Yyy_MainFunction() {
    ...
    ScM_EnterCS_Resource
    <access_to_shared_resource>
    ScM_ExitCS_Resource
    ...
}

Data consistency is given by
- Sequence